48th IEEE/IFIP International Conference on Dependable Systems and Networks
Luxembourg City, Luxembourg, (June 2018).

Fast Hypervisor Recovery Without Reboot

Diyu Zhou and Yuval Tamir
Computer Science Department, UCLA
Email: {zhoudiyu,tamir}@cs.ucla.edu

***Abstract*—System recovery latency is decreased by using *microreboot* to reboot only the failed component instead of the entire system. For large, complex components, such as hypervisors, even the latency of microreboot is unacceptably high in important deployment scenarios. We investigate an alternative component-level recovery mechanism, which we call *microreset*, that can achieve dramatically lower recovery latency for some such components. Instead of component reboot, microreset quickly resets the component to a quiescent state that is highly likely to be valid and where the component is ready to handle new or retried interactions with the rest of the system. We present a recovery mechanism for the Xen hypervisor, called *NiLiHype*, based on microreset. We show that, compared to microreboot-based hypervisor recovery, NiLiHype achieves nearly the same recovery success rate but with a recovery latency that is shorter by a factor of over 30.**

I. Introduction

System-level virtualization [27] is widely used in servers and datacenters of all sizes. Errors that occur during the execution of one of the virtual machines (VMs) due to a transient hardware fault or software fault are highly likely to be confined within that particular VM. Hence, other VMs are not affected. However, if such errors occur during the execution of hypervisor code, the resulting hypervisor failure leads to the failure of all the VMs on the host, and thus, potentially, to a significant impact on datacenter operation. Thus, as explained further below, there is strong motivation to develop fault tolerance mechanisms that allow the VMs to survive across hypervisor failure [19], [21], [28].

VM replication [8] is often used to provide fault tolerance with respect to VM failures, including with commercial products, such as VMware's vLockstep [29]. If the replicas are running on different hosts, failures during the execution of the hypervisor are also covered. Obviously, the resource overhead during normal operation is high; too high for many deployment scenarios. A decision *not* to use VM replication implies that the loss of any one specific VM is tolerable. For example, this might be the case when a VM is providing web service [33]. However, the loss of all the VMs on a host has more impact since it leads to the unavailability of a larger fraction of datacenter capacity, which is particularly significant in a small datacenter. Furthermore, even if VM replication is used, there are performance benefits as well as greater flexibility in VM placement, if the replicas are on the same host [26]. Placing replicas on the same host can only be considered if an error during hypervisor execution is unlikely to lead to a hypervisor crash.

In order to prevent the hypervisor from being a single point of failure, the system must support recovery from hypervisor failure while preserving the hosted VMs. This has been done in *ReHype* [19], [21]. With ReHype, upon detection of an error in the hypervisor, *microreboot* [7] of the hypervisor is performed, while allowing other system components to maintain their states. The overhead during normal operation is small and essentially no work is lost when recovery is performed.

ReHype involves booting a new hypervisor instance. The state of the hypervisor is then updated to make it consistent with the states of the other system components (e.g., the guest VMs) [19], [21]. This requires reusing a significant amount of state from the previous (failed) hypervisor instance. Obviously, this reused state is potentially corrupted. However, it has been shown that recovery success rates above 85% are achieved. Achieving such recovery rates despite reusing state from the failed instance is an indication that there is a relatively low probability of a fault corrupting state critical to the survival of the entire system.

Fault injection studies of the Linux kernel [31], [32] have shown that the errors caused by most faults affect "process local" state as opposed to state that can impact the kernel itself or other processes. Based on this motivation, recovery of the Linux kernel without reboot was proposed. For a subset of possible faults (those leading to kernel *oops*), a simple recovery scheme was proposed that involves aborting the faulting process. This yielded a recovery rate of approximately 60%. These results for the Linux kernel together with the ReHype results discussed in the previous paragraph raise the possibility that recovery from hypervisor failure may not require booting a new hypervisor instance.

For many important applications, the latency of the reboot step of microreboot results in unacceptably long service interruption. This paper investigates component-level recovery of a hypervisor without reboot. Using a mechanism which we call *microreset*, a failed component is reset to a quiescent state that is highly likely to be valid and where the component is ready to handle new or retried interactions with the rest of the system. By avoiding the reboot step of microreboot, the recovery latency is dramatically reduced.

We implement and evaluate microreset for the Xen hypervisor [3], using a mechanism which we call *NiLiHype* (Nine Lives Hypervisor). We show that NiLiHype and ReHype have the same relatively small overhead during normal operation. NiLiHype achieves a successful recovery rate of over 88%, slightly lower than ReHype's rate of over 90%. However, NiLiHype performs the recovery more than 30 times faster than ReHype. NiLiHype's recovery latency is low enough (22ms) that service interruption is negligible in most deploy-

ment scenarios. Considering this recovery rate vs. recovery latency tradeoff, NiLiHype is an attractive point in the design space.

We make the following contributions: 1) present the design of microreset-based component-level recovery as an alternative to microreboot; 2) describe the implementation issues involved in converting the microreboot-based ReHype to the microreset-based NiLiHype; 3) use fault injection to evaluate the recovery rate of NiLiHype for different fault types and workloads; 4) evaluate the hypervisor processing overhead of NiLiHype during normal operation, NiLiHype's recovery latency, and its implementation complexity.

The next section presents microreset and compares it to microreboot as a technique for component-level recovery. The basic operation of Xen hypervisor recovery using ReHype and NiLiHype is described in Section III. The starting point for our implementation was the ReHype source code [19], [21]. Section IV describes the porting of ReHype to a more modern platform and key enhancements that improve its recovery rate. The implementation of NiLiHype is described in Section V. The experimental setup and evaluation results are presented in sections VI and VII, respectively. Related work is summarized in Section VIII.

II. Component-Level Recovery

The key idea in component-level recovery (CLR) is to reduce recovery latency by limiting recovery to the failed component instead of involving the entire system [7]. This section presents the key challenges in CLR as well as two alternative CLR mechanisms that involve low overhead during normal operation: microreboot and microreset, in Subsections II-A and II-B, respectively.

Three main challenges must be overcome in any CLR implementation: (1) preventing the process of recovering the failed component from harming the rest of the system while also preventing the rest of the system from interfering with the recovery process; (2) restoring the failed component to a valid state; and (3) ensuring that the state of the recovered component is consistent with the state of the rest of the system.

One example of interference between CLR and the rest of the system is when the component being recovered is an OS kernel [10]. If an I/O interrupt is generated during recovery, when the kernel is not in a valid state to handle the interrupt, the result is likely to be recovery failure. The simple step of disabling interrupts during part of the recovery process is an obvious solution for this example.

As an example of challenge (2) above, consider the case where recovery involves stopping further execution of the failed component. At that point different parts of the component state may be inconsistent with each other. For instance, a lock may have been acquired by an execution thread within the component but that execution thread is abandoned [19], [21]. Hence, component recovery must involve one, or a combination of, booting a new instance, rollback to a known valid checkpoint, and/or a roll forward procedure to fix the invalid state.

Ensuring that the state of the recovered component is consistent with the rest of the system is the most difficult challenge for CLR (challenge (3) above). Again we use an example where the component being recovered is an OS kernel [10]. If recovery involves booting a new kernel, the new kernel instance will not have the up-to-date state of the data structures that maintain information regarding the user processes that were running prior to the failure. Hence, there is no way to continue running these processes following recovery. To resolve this issue, the recovery mechanisms must reuse part of the state from the previous instance. This introduces an obvious vulnerability since, when recovery is triggered, the state of the previous instance is, by definition, invalid. Recovery can be successful only if the error is not propagated to the reused part of the state or the recovery mechanism can somehow fix corrupted reused state.

A. Microreboot: Component-Level Recovery with Reboot

Microreboot is a CLR that involves booting a new component instance. As discussed in connection with the third challenge above, this requires reuse of part of the state of the previous instance. Therefore microreboot needs to preserve part of the state of the failed instance across the reboot and then reintegrate it with the state of the newly booted instance [10], [19], [20], [21]. The state of the component after recovery is thus a combination of the state of the initial boot and the reused state from the failed instance.

B. Microreset: Component-Level Recovery without Reboot

A significant drawback of microreboot is recovery latency. This latency includes the time to reboot plus the time to re-integrate the state from the previous instance. For large, complex components, such as kernels and hypervisors, this latency can be from multiple hundreds of milliseconds [21] to tens of seconds [10]. It is possible to reduce part of the reboot time by replacing the reboot with a rollback to a checkpoint saved right after a previous reboot [30]. However, even in this case, there would be significant latency for reintegrating state from the previous instance. For example, with a mechanism similar to ReHype, this latency would be multiple hundreds of milliseconds even for a very small workload of three simple application VMs [21].

With the goal of reducing the recovery latency, this paper investigates an alternative to microreboot, which we call *microreset*. Microreset is suitable for large, complex components that process requests from the rest of the system. OS kernels and hypervisors are examples of such components. Specifically, a hypervisor receives requests in the form of hypercalls or traps from the VMs as well as interrupts from hardware timers and potentially other devices. Multiple requests may be processed simultaneously by separate execution threads.

With microreset, upon error detection, the processing of all current requests is abandoned. This resets the component to a quiescent state. At that point, the microreset mechanism must perform additional operations to deal with the last two CLR challenges discussed above. Specifically, there is a need to

perform roll forward operations to fix any corruptions in the component state as well as inconsistencies among different parts of the component state. Next, inconsistencies between the recovering component state and the states of other components in the system must be resolved. For example, this may require retrying requests from other components that were abandoned when the error was detected.

With microreset, only a small fraction of the component state is discarded during recovery. Specifically, it is just the "local" states of the abandoned execution threads (e.g., variables on the stacks). The entire remaining state is kept in place and reused. On the other hand, with microreboot, only part of the global state from the failed instance is reused and the rest of the state is restored to its initial values by the reboot. Microreset's reuse of a larger fraction of the pre-recovery component state increases the probability that the post-recovery state is invalid. Hence, there is a reason to expect some reduction in recovery rate with microreset compared to microreboot.

III. Implementing CLR of a Hypervisor

Section I presented the motivation for a virtualized system to be able to recover from hypervisor failure while preserving and then continuing the execution of application VMs. This section discusses how component-level recovery (CLR) can be applied to the Xen [3] hypervisor, based on microreboot and microreset. Subsection III-A is a brief overview of the Xen virtualization platform and examples of CLR challenges (Section II) applicable to this particular component. Subsection III-B provides a high-level description of how microreboot has been applied to Xen with *ReHype* [19], [21]. Subsection III-C provides a high-level description of how we apply microreset to Xen with *NiLiHype*.

A. The Xen Virtualization Platform

The Xen virtualization platform consists of two components: the hypervisor and the privileged VM (PrivVM, also known as Dom0). The hypervisor provides the core functionality of the virtualization platform, such as memory management and scheduling of the VMs. The PrivVM performs management operations, such as creating, checkpointing, and destroying VMs. The PrivVM may also host the device drivers for the I/O devices in the system and facilitates their sharing among the application VMs (AppVMs) [3].

There are three ways for control to transfer from the VMs to the hypervisor: hypercalls, exceptions and hardware interrupts. VMs issue hypercalls to the hypervisor to request service from the hypervisor. An example of a hypercall is a request by a VM for the hypervisor to update page table entries. This particular example is relevant for *paravirtualized* VMs, (PVMs) [27], [3] and it should be noted that the PrivVM is a PVM. Exceptions occur when VMs execute privileged or illegal instructions. An interrupt is triggered by hardware and causes a trap to a handler in the hypervisor, from which it is sometimes forwarded to some VM.

The ability to recover from a failure during the execution of any part of the virtualization platform requires dealing with the PrivVM as well as the hypervisor. This issue has been addressed in previous work [20], [21] and is not discussed in this paper.

The challenges faced by any CLR, discussed in Section II, must, of course, be handled by CLR of the Xen hypervisor. For example (challenge (1)), if VMs continue to execute during recovery, a trap from a VM may cause the hypervisor to fail since it is not in a proper state for handling such events. An example of challenge (2) from Section II is that internal hypervisor data structures, such as timer heap, may be corrupted by the fault or left in an inconsistent state. An example of challenge (3) is that the hypervisor may be in the middle of handling a hypercall when an error is detected. The failure to complete this hypercall may cause the initiating VM to fail following hypervisor recovery.

B. ReHype: Microreboot of the Xen Hypervisor

ReHype [19], [21] recovers from failures of the Xen hypervisor using microreboot. When an error is detected, a recovery handler is invoked. The first few steps cause all the CPUs to disable interrupts and all but one to halt. The CPU that does not halt handles most of the rest of the recovery process. These initial steps prevent interference between the recovery process and the rest of the system. The handler then saves a copy of the data in the static data segments to a memory location where it will not be overwritten by the new hypervisor instance. The next step is to boot a new hypervisor instance. During reboot, parts of the preserved static data segments are used to overwrite some of the values initialized earlier in the boot process. The non-free heap pages in the pre-recovery hypervisor instance are preserved and re-integrated in the new heap. Pre-recovery page tables are restored. The final step of the basic scheme is to wake up all the CPUs and resume normal operation.

Enhancements of the basic mechanism outlined above are used to achieve a high recovery rate [19], [21]. These enhancements deal with the last two CLR challenges discussed in Section II. For example, the reused state from the previous hypervisor instance includes locks. In order to make the new hypervisor state self-consistent, all of these locks are released. In order to resolve inconsistencies with respect to the VMs, for any partially executed hypercall, the VM state of the corresponding VM is set up so that the hypercall is retried once VM execution is resumed. To partially resolve inconsistencies with respect to the hardware, all pending and in-service interrupts are acknowledged.

C. NiLiHype: Microreset of the Xen Hypervisor

NiLiHype is essentially ReHype without reboot. Many of the basic operations as well as enhancements that are performed by NiLiHype are identical or similar to those in ReHype. For example, both have to ensure that partially executed hypercalls are retried following recovery. Similarly,

as part of recovery, both have to release locks used by the hypervisor.

Since NiLiHype does not include reboot, some complex operations required by ReHype are not needed. An example of this is the ReHype step of rebuilding the heap to re-integrate the data reused from the previous instance. On the other hand, the reboot in ReHype does help in producing a hypervisor state that is self-consistent. Thus, NiLiHype requires additional enhancements to overcome some CLR challenges (challenge 2 in Section II) that ReHype overcomes by performing a reboot.

When an error is detected, the recovery handler of NiLiHype is invoked on the CPU where the error is detected. The handler disables interrupts on its own CPU and interrupts all the other CPUs, which then disable interrupts. Each of the CPUs discards its execution thread within the hypervisor by discarding the hypervisor stack (resetting the stack pointer). All the CPUs, except for the one that detected the error, then enter busy waits. At this point, enhancements to increase the recovery rate should be performed by the CPU that detected the error (see below). The above initial steps prevent interference between the recovery process and the rest of the system. The final step of the basic scheme is to allow all the CPUs to exit their busy waits and resume normal operation.

With the basic mechanism described above, recovery always fails. Enhancements that deal with the last two CLR challenges discussed in Section II are necessary to achieve a high recovery rate. These enhancements are described in Subsection V-A.

With NiLiHype, all threads of execution within the hypervisor are discarded. A possible alternative design choice would be to discard only the execution thread of the CPU that detects the error. The choice made in NiLiHype makes it more similar to ReHype, where the reboot effectively discards all the execution threads. It is expected that the alternative choice would be more complex to implement and result in lower recovery rate. The reasons for this are interactions among hypervisor threads of execution as well as interactions between these threads and the recovery process itself. An example of this is if CPU1 sends an interprocessor interrupt (IPI) to CPU2 and waits for a response. An error may be detected on CPU2 after receiving the IPI but before responding. Since CPU2 discards its execution thread, CPU1 may be blocked forever. A second example is related to the impact of changes to the global state by the recovery process. These changes may cause non-discarded threads to encounter unexpected state changes that lead to their failure.

IV. Porting and Enhancing ReHype

Due to the similarity between NiLiHype and ReHype, the starting point for the NiLiHype implementation was the ReHype source code [19], [21]. Our first step was to port this implementation to the x86-64 ISA (from x86-32), Xen version 4.3.2 (from 3.3.0), with all VMs running Linux 3.16.1 kernels. As described in the rest of this section, we then implemented enhancements to improve the recovery rate.

Our initial port mostly resolved the expected porting issues caused by the evolution of Xen code, such as changes in function/variable/macro names. We then used fault injection to evaluate the port and guide further enhancements. This was based on running a simple workload (one AppVM) and injecting fail-stop faults. After the initial port, the recovery rate was 65%. Three enhancements were required due to platform changes while the fourth would also have been useful for the older platform. Together, these enhancement increased the recovery rate to 96%.

Syscall retry. With the x86-32 ISA, system calls from the VM processes directly trap into the VM kernel. However, with the x86-64 ISA, system calls from the VM processes trap into the hypervisor which then forwards them to the appropriate kernel. In order to handle the possibility that an error is detected when the hypervisor is forwarding a system call, ReHype had to be enhanced to ensure that such system calls are retried following recovery. The implementation is similar to hypercall retry.

Fine-granularity batched hypercall retry. With the new Xen platform, in order to reduce the virtualization overhead, several hypercalls may be batched into one hypercall. In order to better handle such batched hypercalls, the hypervisor logs the completion of each hypercall within a batch as it completes. If, following recovery, the batched hypercall is retried, those component hypercalls that completed earlier are skipped.

Save FS/GS. Xen on x86-64 doesn't use the FS and GS registers and thus does not save them when the hypervisor is entered. Xen on x86-32 does save these registers. Hence, with the initial ReHype port, these registers are lost following recovery. The fix is for the hypervisor to save these registers when an error is detected.

Mechanisms to mitigate hypercall retry failure. With the above three enhancements, the recovery rate is 84%. The remaining recovery failures are largely caused by re-executing non-idempotent hypercalls. For example, several hypercalls increase or decrease a reference counter in the page frame descriptor by one. If an error is detected after a hypercall updates the counter but before completion, the re-execution results in an inconsistent state.

A comprehensive solution to the above problem would be to transactionalize all the non-idempotent hypercalls. Doing that would require major changes to the code and/or significant overhead. Instead, we used fault injection to identify problem cases and resolve them using lightweight logging and code reordering. A downside of our approach is that we have not tested all hypercall handlers. Thus, there are likely to be several infrequently-used non-idempotent hypercall handlers that we have not properly enhanced. Furthermore, even for the handlers that have been modified, the changes do not resolve 100% of the problem. However, the changes do significantly reduce the window of vulnerability and minimize the probability of recovery failure.

Logging enables undoing changes performed by partially executed hypercalls. Changes to critical variables are logged and the changes are undone following recovery, before a retried hypercall reads or modifies these variables. For some

hypercalls, it was possible to reduce the window of vulnerability by simply reordering the code, without changing the functionality or incurring overhead. An example of this is moving modifications of critical variables to the end of the hypercall so that there is minimal code to execute between the state changes and the completion of the hypercall. Altogether, these changes for handling non-idempotent hypercalls increase the recovery rate from 84% to 96%.

V. NiLiHype Implementation

The starting point for the NiLiHype implementation is the enhanced ReHype implementation described in Section IV. Some of the features that are common to NiLiHype and ReHype are: 1) VMs are suspended and interrupts are disabled during recovery; 2) almost all the ReHype enhancements described in [19], [21]; 3) all the enhancements described in Section IV.

As mentioned in Subsection III-C, since NiLiHype does not involve reboot, some of the most complex and time-consuming operations required by ReHype are not needed. Specifically, these include hardware initialization as well as operations to preserve and later re-integrate state from the pre-recovery hypervisor instance. These operations are described in detail in Section 3 of [19] and their latencies are presented in Section 10 of [21]. Similar latency measurements, for our enhanced ReHype implementation, are presented in Subsection VII-C.

This section focuses on additional enhancements needed by NiLiHype to overcome CLR challenges that are resolved by the reboot in ReHype. Subsection V-A presents the enhancements. Section V-B presents the measurement-based incremental development of the NiLiHype-specific enhancements.

A. Enhancements Required by NiLiHype

With ReHype, a very low recovery rate (5.6%) is achieved without any enhancements [19]. That recovery is even possible with the basic scheme is due to the operations performed by the reboot, that include re-initializing the hardware and initializing a new, valid hypervisor memory state. As mentioned in Subsection III-C, with just the basic NiLiHype mechanism (discard all hypervisor threads of execution), recovery never succeeds. Hence, the enhancements of the basic scheme, that resolve the CLR challenges (Section II), are even more critical in NiLiHype.

One of the enhancements developed for NiLiHype is needed to bring the hardware to a consistent state: *reprogram hardware timer*. Four additional enhancements deal with hypervisor memory state. All of these enhancements are described below.

Reprogram hardware timer. Xen relies on the hardware timer in the interrupt controller (APIC) to trigger the examination of the software timer heap. The handler reprograms the APIC timer to fire again at a time determined by the top node of the timer heap. If the fault occurs after the APIC timer has fired but before Xen reprograms it, without additional mechanisms, the APIC timer will never fire again after recovery. NiLiHype handles this issue by ensuring that each CPU reprograms its APIC timer before resuming normal operation.

Clear IRQ count. In Xen, each CPU maintains a per-CPU variable named *local_irq_count* that records the nesting level of interrupts. When the CPU enters or leaves an interrupt handler, *local_irq_count* is, respectively, incremented or decremented. The *local_irq_count* value is used in hypervisor assertions to check whether the CPU is currently servicing an interrupt. As NiLiHype discards all the execution threads in the hypervisor, the *local_irq_count* variables of all the CPUs are set to zero during the recovery.

Ensure consistency within scheduling metadata. Xen maintains scheduling metadata that includes: (1) the *runqueue* of each CPU, which is a linked list of vCPUs; (2) per-CPU variables indicating the current executing vCPU; and (3) per-vCPU variables representing the execution states of the vCPUs. Hypervisor failure followed by recovery can easily leave this scheduling metadata in an inconsistent state. Inconsistencies within the scheduling metadata can cause the hypervisor to incorrectly restore the register context of one vCPU when another vCPU is scheduled to run. Such inconsistencies can also result in the failure of assertions in the scheduling routine, leading to hypervisor failure.

Resolving potential scheduling metadata inconsistencies is done based on two key ideas: 1) where possible, initialize the data to a fixed valid value instead of relying on the existing value; 2) if it is necessary to use existing data, pick the most reliable source and make the rest of the metadata consistent with that.

With NiLiHype, we encountered a particularly critical problem related to scheduling metadata inconsistencies. The information regarding which vCPU is currently running on each CPU is stored redundantly in multiple places. Specifically, it is stored in the per-CPU structures as well as two different locations in the per-vCPU structures. To resolve the inconsistencies, the information in the per-CPU structures is used to set the information in all the per-vCPU structures.

Unlock static locks. Since both ReHype and NiLiHype effectively discard all threads of execution in the hypervisor, all locks should be in their unlocked state following recovery. ReHype includes a mechanism to release all the locks stored in the heap. NiLiHype uses the same mechanism. With ReHype, locks in the static data segment ("static locks") are initialized to their unlocked state during boot. NiLiHype requires an additional mechanism to release all such locks.

NiLiHype avoids a complex mechanism for tracking the static locks. Instead, NiLiHype takes advantage of the fact that, in Xen, all the static locks are defined using a macro. We modified the linker script used to build the Xen image and the macro defining locks to put all the static locks in a separate segment in the Xen image, effectively placing them all in one array. During the recovery process, before multiple CPUs are allowed to execute, the CPU that detects the error iterates over all the locks in the segment and unlocks any locked locks.

Reactivate recurring timer events. Xen uses several recurring timer events. These include events to synchronize system

TABLE I
SUMMARY OF MECHANISMS TO ENHANCE NILIHYPE

Mechanism	Successful Recovery Rate
Basic	0%
+ Clear IRQ count	16.0% ± 2.3%
+ Enhanced with ReHype mechanisms	51.8% ± 3.1%
+ Ensure consistency within scheduling metadata	82.2% ± 2.4%
+ Reprogram hardware timer	95.0% ± 1.4%
+ Unlock static locks + Reactivate recurring timer events	96.1% ± 1.2%

time among CPUs and to update the execution time of vCPUs for the scheduler. For each of these timers, the handler re-activates the timer each time it is fired. If a fault occurs while a CPU is executing one of these timer handlers, before the handler successfully re-activates the timer, the recurring timer is lost.

This problem is very similar to the problem described above that motivates the *reprogram hardware timer* enhancement. To solve the problem discussed here, for each relevant timer event, there is a flag that is set on entry to the handler and cleared upon exit. As part of the recovery process, for any of these flags that is set, the corresponding timer is re-activated.

B. Incremental Development of NiLiHype Enhancements

To identify the need for the enhancements discussed in Subsection V-A, we use the same incremental procedure, based on fault injection, used in multiple previous works [24], [19]. Results from fault injections are analyzed to identify the cause of the plurality of recovery failures. A mechanism is developed to handle that particular problem. The process is repeated with new fault injections in each iteration.

Table I summarizes all the mechanisms we use to enhance NiLiHype with their impact on the recovery rate. The fault injection setup is the same one used in Section IV. Specifically, we use the 1AppVM workload with the *UnixBench* benchmark (Subsection VI-A). 1000 fail-stop faults are injected in each iteration.

Table I shows that the *Clear IRQ count* enhancement is mandatory in order for NiLiHype recovery to succeed. This is because the CPU that detects the error sends IPIs to other CPUs to initiate the recovery. Each of the CPUs receiving the IPI increments its *local_irq_count*. Since the CPU then discards its thread of execution (discards its stack), it never returns from the IPI. As a result, *local_irq_count* ends up with an inconsistent value (not 0). This later causes hypervisor failure as a result of the failure of assertions in several critical routines that check whether the CPU is in interrupt context.

Table I also shows that, with all the enhancements, for this particular setup, NiLiHype achieves the same recovery rate as ReHype (Section IV).

VI. EXPERIMENTAL SETUP

This section presents the experimental setup used to evaluate NiLiHype and ReHype. This setup is very similar to the one used in [19], [21]. The key difference is the use of a more modern platform (ISA and Xen/Linux versions). The configurations of the systems evaluated (target systems) are described in Subsection VI-A. Subsection VI-B presents the error detection mechanisms used in the target systems. Details regarding the fault injection are described in Subsection VI-C.

A. Target System Configurations

We evaluate virtualized systems running synthetic benchmarks designed to stress different aspects of the system. The hypervisor is Xen 4.2.3. The system includes the privileged VM (PrivVM) and either one application VM (AppVM), in the *1AppVM* configuration, or three AppVMs, in the *3AppVM* configurations. Each VM consists of one vCPU (virtual CPU) and each of the vCPU is pinned to a different physical CPU. The physical machines used for all the experiments are 8-core systems based on Intel Nehalem CPUs.

All the AppVMs are paravirtualized VMs (PVMs). It should be noted that previous work has shown that fault injection results obtained with AppVM supported by full hardware virtualization (HVMs) are very similar to those obtained with paravirtualized AppVMs [21].

Three synthetic benchmarks are used: *BlkBench*, *UnixBench*, and *NetBench*. *BlkBench* focuses on the interface to block devices (disk). It creates, copies, reads, writes and removes multiple 1MB files containing random content. To ensure that the device is actually accessed, requiring hypervisor activity, caching of block and file system data in the AppVM is turned off. Without this setting, caching within the AppVM would minimize the chances for exposing recovery failure. *UnixBench* is a collection of programs designed to stress different aspects of the system [2]. We use a subset of the programs in the original UnixBench. The programs were selected for their ability to stress the hypervisor's handling of hypercalls, especially those related to virtual memory management.

For *BlkBench* and *UnixBench*, the execution is considered as failed if 1) one or more files produced by the benchmark are different from the ones in a golden copy, or 2) logging messages from the benchmarks indicate that one or more than one system calls to the OS of the AppVM failed.

NetBench is a user-level network ping program. It is used to exercise the interface to the network as well as to evaluate the recovery latency. It involves two processes: the *receiver* runs in an AppVM in the target system, and the *sender* runs on a separate physical host. The *sender* sends a UDP packet to the *receiver* every 1ms. Upon receiving a packet, the *receiver* sends a reply back to the *sender*. *NetBench* execution is considered as failed if the packet reception rate of the *sender* drops by more than 10% compared to its reception rate during normal execution in any one-second interval.

In the *1AppVM* setup, the AppVM runs either *BlkBench* or *UnixBench*. Each one of the benchmarks is configured to run

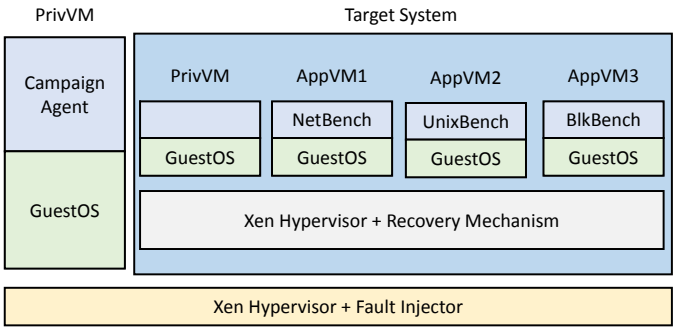

Fig. 1. Fault injection setup with the 3AppVM workload. The entire target system is in a VM. The injector is in the outside hypervisor

for around 10 seconds. This setup is mostly used to guide the measurement-based incremental development of recovery enhancement mechanisms.

In the *3AppVM* setup, the system initially runs two AppVM: one with *UnixBench* and the other with *NetBench*. Following recovery, a third AppVM is created and it runs *BlkBench*. The third AppVM is used to check whether the hypervisor still maintains its ability to create and host newly created VMs after recovery. The first two AppVMs are configured to run for approximately 24 seconds.

B. Error Detection

The focus of this paper is on error recovery, not error detection. However, an error detection mechanism is necessary for the experimental evaluation since such a mechanism is responsible for initiating recovery.

We rely on the built-in panic and hang detectors in Xen to detect errors. A panic is detected when a fatal hardware exception occurs or a software assertion fails. The hang detector is based on a watchdog timer. It is implemented based on hardware performance counters and software timer events. A performance counter on each CPU is used to generate an NMI every 100ms of unhalted CPU cycles. There is also a recurring software timer event that increments a counter every 100ms. The handler of the performance counter checks for changes in the counter. If the counter is not incremented for three consecutive invocations of the performance counter handler, a hang is detected.

C. Fault Injection

Software-implemented fault injection is used to determine the recovery rate of NiLiHype and ReHype. A fault injection *run* consists of booting the target system, starting the benchmarks in the AppVMs, injecting one fault, and collecting logs that allow the results to be analyzed. For each fault type and system configuration there is an injection *campaign* that consists of multiple *runs*.

We ported the Gigan fault injector [22] to a modern platform (ISA and Xen/Linux version). To minimize intrusion by the injector and simplify campaign setups, we use Gigan in a configuration based on two-level nested virtualization. Specifically, the entire virtualized target system runs in a VM supported by full hardware virtualization (i.e., an HVM). It has been shown that fault injection and recovery results obtained with this setup are similar to those obtained when the target system runs on bare hardware [21], [22].

The injector runs outside the target system, in the hypervisor (the "outside" hypervisor) that hosts the HVM with the target system. A user-level campaign script, the *Campaign Agent*, runs in the PrivVM of the outside hypervisor. The *Campaign Agent* creates the VM with the target system, configures the fault injector, and collects logs and output from each run. The fault injection setup with the 3AppVM workload is shown in Figure 1.

We inject three types of faults: *Failstop*, transient *Register* faults, and *Code* faults. *Failstop* faults are injected by changing the value of the program counter to 0. *Register* faults are injected by flipping a random bit in a random register selected from the 16 general-purpose registers, the stack pointer, the flag register, and the program counter. *Code* faults are injected by flipping a random bit in a random byte chosen within the 15-byte range (the maximum length of an x86-64 instruction) starting from the current value of the program counter. When/if the *Code* fault causes an error that is detected, the injector "repairs" the fault. Thus, the effects of a *Code* fault do not persist during the recovery process. Hence, the effects of a *Code* fault are almost the same as if it was a transient fault.

The injected faults do not cover all possible faults. NiLiHype is designed to recover from transient hardware faults as well as rare software bugs (Heisenbugs) [11], that occur only under particular timing and ordering of asynchronous events in the system. Transient hardware faults in the CPU datapath are likely to be manifested as erroneous values in registers. Hence, register bit flips can be expected to be reasonably representative of such faults. Injection of bit flips in the code attempts to partially represent faults in the instruction fetch and decode hardware. Similar injection campaigns have been widely used in prior works [9], [14], [5]. As discussed in Subsection VII-A, results from the injection of failstop faults, together with other results, help with the understanding of the tradeoffs between microreboot and microreset.

All the faults are injected by using a two-level chained trigger. When the first-level trigger fires, it triggers the second-level, which, when fired, triggers the fault injection. The first-level trigger is a timer that fires after a specified amount of time has elapsed. It is configured differently in the 1AppVM and 3AppVM setups. With the 1AppVM setup, it is set to fire at a random time after the initial 10% and before the final 10% of the benchmark execution time. With the 3AppVM setup, the first-level trigger is configured to fire at a random time between 500ms and 6 seconds. This is well past the start of the *UnixBench* and *NetBench* AppVMs while leaving most of their 24 seconds execution to occur after recovery.

The second-level trigger fires after a random number of instructions between 0 to 20000 have been executed in the target hypervisor. This trigger ensures that faults are injected only while the CPU is executing code of the target hypervisor.

VII. Evaluation

This section presents an evaluation of NiLiHype using the experimental setup described in Section VI. Along every axis, NiLiHype is compared to ReHype. The recovery rate, recovery latency, hypervisor processing overhead during normal operation, and implementation complexity are presented in Subsections VII-A, VII-B, VII-C, and VII-D, respectively.

A. Successful Recovery Rate

It has been shown that, for typical deployments of virtualization, less than 5% of CPU cycles are spent executing hypervisor code [6], [4]. Hence, a random transient fault is much more likely to occur when executing code in a VM than when executing hypervisor code. Thus, a random transient fault is highly likely to affect *one* of the VMs, possibly causing it to fail, even if the virtualization platform is completely immune to all faults. Whether a fault in the hypervisor can affect a single VM becomes relevant only if mechanisms implemented strictly within the VM itself allow it to mask or recover from the overwhelming majority of faults that may occur during the execution of the VM. Taking this into account, it is not meaningful to evaluate any hypervisor resilience mechanism based on a criterion that a manifested fault in the hypervisor should not affect even *one* VM.

Without any resilience mechanisms, a single transient fault can cause the hypervisor to fail, taking down all the VMs it hosts. Based on the discussion above, we can define a reasonable goal for a hypervisor resilience mechanism. Specifically, taking into account only transient faults (including Heisenbugs [11]), running multiple VMs on a single host should not be worse than running them without virtualization on separate physical machines [18]. In practice, this goal cannot really be met due to practical issues, such as power supplies, network connections, etc. However, this forms the basis for our definition of "successful recovery" from hypervisor failure.

We define recovery from hypervisor failure to be "successful" if no more than one AppVM is affected by the fault and, after recovery, the hypervisor continues to operate correctly (new VMs can be created, etc). The 3AppVM setup is designed to allow evaluation based on this definition. Specifically, the setup includes creating a new AppVM (*BlkBench*) after recovery, and an ability to verify that *BlkBench* runs correctly to completion. Note that, for the 1AppVM setup, we define "recovery success" to mean that no VM is affected.

The recovery rate is evaluated with the 3AppVM setup. Separate campaigns are run with the three fault types: *Failstop*, *Register*, and *Code*. For each fault injection run, the outcome can be classified into three categories: non-manifested, silent data corruption (SDC), and detected. Non-manifested means that the injected fault does not cause any observable abnormal behavior: the benchmarks finish successfully (produce the correct outputs) and the detection mechanisms are not triggered. SDC means that the detection mechanisms are not triggered but at least one of the benchmarks fails to produce the expected outputs. Detected means that one of the detection mechanisms is triggered. Obviously, the recovery mechanism is triggered only for fault outcomes in the last category.

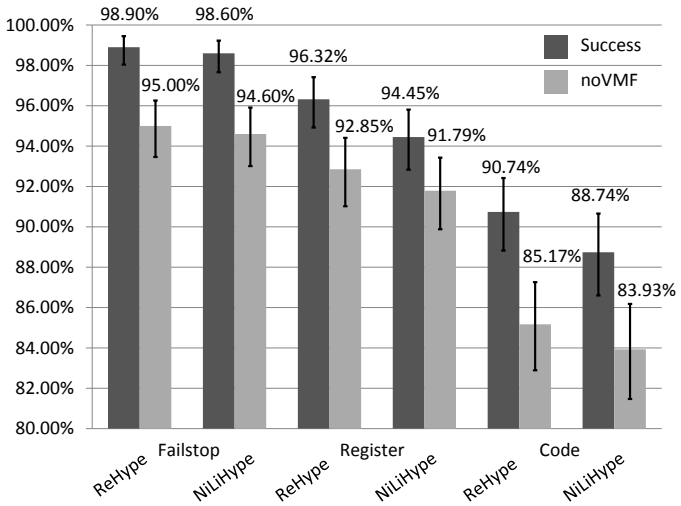

Fig. 2. Successful recovery rates of NiLiHype and ReHype for different fault types with the 3AppVM setup. Error bars show the 95% confidence intervals. noVMF stands for no AppVM failure cases.

The breakdown of injection outcomes varies with fault type. Obviously, all *Failstop* faults are detected. For *Register* faults, the breakdown in our campaign is: 74.8% non-manifested, 5.6% SDC, and 19.6% detected. For *Code* faults, the breakdown is: 35.0% non-manifested, 12.1% SDC, and 52.9% detected.

The fault injection campaigns for evaluating the recovery rate include 1000 *Failstop* faults, 5000 *Register* faults, and 2000 *Code* faults. In each case, the number of the injected faults was chosen so that, for both NiLiHype and ReHype, the 95% confidence interval for the recovery rate was within $\pm 2\%$.

Figure 2 presents the successful recovery rate of NiLiHype and ReHype with the 3AppVM setup. For the completeness, we also report the portion of detected errors (recovery initiations) that resulted in no AppVM failures (*noVMF* — no VM failures). The differences between *Success* and *noVMF* are due to injection runs where the only impact of the fault is to cause one of the first two initial AppVMs to fail.

Figure 2 shows that NiLiHype and ReHype achieve essentially identical recovery rates for *Failstop* faults, but not for the other fault types. *Failstop* faults can only result in inconsistencies within the hypervisor state or between the hypervisor state and the states of other hardware and software components. The other fault types can result in the same inconsistencies but also in state corruption. It is likely that, for *Register* and *Code* faults, ReHype's small advantage is due to cases where the fault corrupted part of the hypervisor state that is discarded and re-initialized by the reboot.

To understand the reasons for recovery failures, we analyzed the recovery failure cases when *Register* faults are injected. ReHype resulted in 35 such cases and NiLiHype in 54. The reasons for recovery failures with ReHype and NiLiHype are similar. The top three reasons are: 1) the recovery routine fails

TABLE II
RECOVERY LATENCY BREAKDOWN OF REHYPE

Operations	Time
Hardware initialization:	412ms
- Early initialize of the boot CPU	12ms
- Initialize and wait for other CPUs to come online	150ms
- Verify, connect and setup local APIC and setup IO ACPI	200ms
- Initialize and calibrate TSC timer	50ms
Memory initialization	266ms
- Record allocated pages of old heap (Use to preserve content of old heap)	21ms
- Restore and check consistency of page frame entries	21ms
- Re-initialize the page frame descriptor for un-preserved pages	13ms
- Recreate the new heap	211ms
Misc	35ms
- SMP initialization	20ms
- Identify valid page frame, relocate boot up modules	2ms
- Others	13ms
Total	713ms

TABLE III
RECOVERY LATENCY BREAKDOWN OF NILIHYPE

Operations	Time
- Restore and check consistency of page frame entries	21ms
- Others	1ms
Total	22ms

to be invoked due to the corrupted hypervisor state, 2) the PrivVM fails, and 3) the error causes a data structure in the hypervisor, typically a linked list or the heap, to be corrupted or left in an inconsistent state.

Figure 2 shows that *Code* faults result in the lowest recovery rate. This is likely due to the significantly longer detection latency of these faults [22], providing more time for errors to propagate and cause greater state corruption.

B. Recovery Latency

When hypervisor recovery is in progress, all the AppVMs are paused. Hence, it is easy to measure the recovery latency by measuring the service interruption of a service executing in an AppVM in the target system. Latency measurements may be distorted when the system is deployed in a nested virtualization configuration. Hence, in order to obtain accurate latency results, the target system runs on bare hardware. As a service, we use *NetBench* in the 1AppVM setup. The service interruption is measured at the *sender*, that runs on a separate physical host (Subsection VI-A).

For NiLiHype, we measured a recovery latency of 22ms. For ReHype, with all the recovery latency optimizations discussed in [21], we measured a recovery latency of 713ms. Repeating each experiment five times, the latencies varied by no more than 1ms for NiLiHype and 10ms for ReHype.

To determine how much time the different operations involved in recovery contribute to the overall recovery latency, we added code in the recovery code of NiLiHype and ReHype to record the value of the time stamp counter (TSC) after each major recovery step is completed. The results for ReHype and NiLiHype are shown in Table II and Table III, respectively. These tables list every step that takes more than 1ms.

Most of NiLiHype's recovery latency is due to an operation done to ensure consistency of the page frame descriptors. This is an operation that ReHype also performs [19], [21]. The problem that this operation solves is that, following recovery, there are two components in each page frame descriptor that may be left in inconsistent states: the *validation bit* and the *page use counter*. This can cause the hypervisor to hang following recovery. To fix this issue, the recovery routine iterates over all the page frame descriptors in the hypervisor to check for the inconsistency and update as required to restore consistency.

The latency of the operation described above is proportional to the size of the host memory (and thus the number of page frame descriptors). In our system, 8GB of physical memory results in a latency of 21ms. Obviously, this would be a problem in a large system with tens or hundreds of GB of memory. The problem could be mitigated by exploiting parallelism. For example, use multiple cores to perform the operation. Another option is to not perform this recovery step. This option has the disadvantage that it results in a reduction of 4% in the recovery rate [19].

C. Hypervisor Processing Overhead in Normal Operation

A key question regarding any resilience mechanism is how much performance overhead during normal operation it incurs. With NiLiHype, this translates to the extent to which, for a fixed workload, the count of CPU cycles spent executing hypervisor code is higher with the NiLiHype modifications compared to with stock Xen.

To get accurate results, the hypervisor processing overhead is measured with the target system running on bare hardware. In each one of the CPUs (including the one running the PrivVM), a hardware performance counter is used to count the unhalted cycles spent in the hypervisor. For repeatable results, the precise points in time for starting and ending the measurement are carefully controlled. All the benchmarks running in all the AppVMs are synchronized. Measurement starts when all the benchmarks are ready to begin executing. Measurement ends when all the benchmarks complete. The benchmarks "inform" the measurement code when they begin and end using traps (the CPUID instruction). All the benchmarks execute for approximately the same amount of time (21s).

We define the hypervisor processing overhead as the percent increase in the unhalted cycle count in the hypervisor with NiLiHype relative to that same count with stock Xen.

We used four target system configurations. The first three use the 1AppVM setup with the three benchmarks: *BlkBench*,

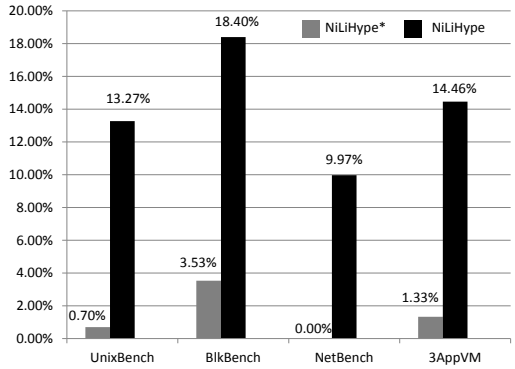

Fig. 3. Hypervisor processing overhead (based on CPU cycles) of NiLiHype during normal execution. NiLiHype* stands for NiLiHype without logging to mitigate hypercall retry failure.

UnixBench, and *NetBench*. Obviously, for these three configurations, synchronizing the execution of the benchmarks is not relevant. The fourth configuration is a slightly modified version of the 3AppVM setup. Since recovery is not actually done in these measurements, all three of the AppVMs are created at the same time and they all run their benchmarks throughout the experiment. For each configuration, we repeated the measurements five times and found that the differences in the measurement results were all less than 1%.

Figure 3 shows the hypervisor processing overhead for NiLiHype for all the configurations. Most of this overhead is due to logging used to mitigate recovery failures due to retries of non-idempotent hypercalls. To show that, the figure also shows the overhead of NiLiHype without this logging (NiLiHype*).

We have also measured the hypervisor processing overhead of ReHype and found it to be the same as for NiLiHype. This is not surprising since the logging in NiLiHype and ReHype are almost identical,

As mentioned earlier (Subsection VII-A), it has been shown that, for typical deployments of virtualization, less than 5% of CPU cycles are spent executing hypervisor code [6], [4]. Hence, even with logging, the actual impact of the hypervisor processing overhead is negligible. Specifically, even in the worst case (*BlkBench*), the overhead in terms of total CPU cycles can be expected to be less than 1%. If this overhead is not acceptable, there is the option of turning the logging off. As discussed in Section IV, this will reduce the recovery rate by approximately 12%.

D. Implementation Complexity

As a measure of the implementation complexity of NiLiHype and ReHype, we use the total number of lines of code (LOC) added and modified starting with the source code of the stock Xen hypervisor. We use the code line count tool CLOC [1] to measure the LOC.

We partition the LOC added and modified into two categories: (1) code that executes during normal operation to enable or enhance NiLiHype/ReHype functionality, and (2) code that executes only during recovery. Table IV presents the results.

TABLE IV
IMPLEMENTATION COMPLEXITY OF NILIHYPE AND REHYPE

Type	Mechanisms	NiLiHype	ReHype
Normal operation	Mitigating hypercall retry failure	991	991
	Other logging	532	594
Recovery routine	Shared recovery mechanism	543	543
	Specific recovery mechanism	113	642
Total	Total	2179	2770

For both NiLiHype and ReHype, most of the added/modified code in category (1) is related to mitigating hypercall retry failure (Section IV). For all the code in category (1), NiLiHype requires slightly less code. This is due to two types of logging that are not needed for NiLiHype. Firstly, ReHype needs to log changes to the I/O APIC registers during the normal execution. This is because ReHype recovery re-initializes these registers as part of the boot process. However, for recovery to succeed, these registers must be restored to their pre-recovery values. Secondly, ReHype needs to log the values of the boot line options during the normal boot up. These values are used by the hypervisor to correctly initialize the system during boot. Hence, ReHype recovery needs to reuse the previously logged boot line options to correctly boot up the hypervisor.

The main difference in the implementation complexity between NiLiHype and ReHype is in code that executes during recovery. ReHype has significantly more code needed to preserve and re-integrate the failed hypervisor state back to the new hypervisor instance.

VIII. RELATED WORK

NiLiHype is related to numerous works aimed at increasing the resilience to errors in OS kernels and hypervisors. Most of the work on OS kernels has focused on ways to partition the kernel, isolate the partitions (fault domains) from each other, and recover failed partitions without requiring full system reboot [5], [9], [12], [23], [25]. In many cases this is facilitated by an underlying design, based on a small microkernel and a collection of drivers and servers isolated from each other by the memory-management hardware [5], [9], [12]. On the other hand, microreset, and thus NiLiHype, are aimed at monolithic kernels and hypervisors.

VirtuOS [25] proposes *vertical slicing* of the Linux kernel to service domains. The encapsulation of the slices is performed using virtualization based on Xen. Requests from user processes interact directly with the appropriate service domain, which is isolated from the rest of the kernel. Since VirtuOS requires virtualization, applying the idea to a hypervisor would require nested virtualization with its associated overheads.

Akeso [23] dynamically partitions the Linux kernel into request-oriented recovery domains. A recovery domain is

formed by the execution thread that handles a request, such as a system call or interrupt. A modified compiler is used to instrument the code to track state changes caused by the domain as well as dependencies among domains. When an error is detected, the affected domain and dependent domains are rolled back. Due to the code instrumentation, the performance overhead of Akeso is between 8% and 560%.

NiLiHype can be viewed as a lightweight version of Akeso [23]. As with Akeso, upon the detection of an error, the affected execution thread handling a request is abandoned. Instead of tracking dependencies among execution threads, NiLiHype simply abandons all current execution threads. With Akeso, the instrumentation added by the modified compiler logs state that allows domains to be rolled back. NiLiHype does not require a modified compiler. As described in Section IV, NiLiHype relies on manual modification of the Xen code to facilitate successful retry of non-idempotent hypercalls. Since NiLiHype does not involve the extensive code instrumentation of Akeso, the performance overhead is likely to be significantly lower. However, a direct comparison would require applying Akeso with Xen and performing the measurements with identical workloads. Since NiLiHype does not include the comprehensive compiler-implemented tracking of state changes of Akeso, NiLiHype's recovery rate is likely to be lower.

As discussed in Section I, Yoshimura et al. [31], [32] proposed the idea that it is possible to recover from some errors in the Linux kernel without reboot. Their recovery scheme always involves killing a running process. They achieved a recovery success rate of 60%. In contrast, with NiLiHype the recovery rate is over 88% and in over 83% of the cases no AppVM is lost.

Otherworld [10] allows a Linux kernel to be recovered from failures by a full reboot of the kernel, while preserving in place the states of the running processes. Since the "system" consists of the kernel as well as all the processes, this can be viewed as a *microreboot* [7] of the kernel. Otherworld rebuilds many kernel data structures associated with each process, such as the process descriptor, the file descriptor table, and signal handler descriptors. Restoration of kernel components requires traversing many complex data structures in a possibly corrupted kernel, increasing the chance of failed recoveries. In many cases, user-level processes require custom crash procedures in order to properly resume execution.

A hypervisor maintains less state for each VM compared to the state for each process maintained by the kernel. This makes the microreboot of a hypervisor simpler than of kernel and increases the chance of a successful recovery [19].

RootHammer [16] uses microreboot to rejuvenate virtualized systems based on the Xen. It reduces the time for this rejuvenation by rebooting only the Xen hypervisor and the PrivVM, while preserving in memory the states of VMs and their configurations. During rejuvenation, the PrivVM is properly shut down and the VMs suspend themselves cleanly. Hence, RootHammer operates within a healthy and functioning system. RootHammer is not designed to recover from failure and thus does not deal with possible arbitrary corruptions and inconsistencies in the system. On the other hand, NiLiHype reuses almost the entire hypervisor state and is thus not useful for rejuvenation [13].

TinyChecker [28] proposes the use of nested virtualization to manage hardware enforced protection domains during hypervisor execution. TinyChecker is a small hypervisor that runs underneath the commodity hypervisor, monitors transitions between the commodity hypervisor and the VMs it hosts, limits the memory regions writable during hypervisor execution, and attempts to detect accesses that are possibly erroneous and take checkpoints to facilitate recovery. TinyChecker has never been fully implemented or evaluated. In a full implementation, the nested virtualization mechanism is expected to involve significant overhead.

This work is most closely related to ReHype [19], [21], that uses microreboot for recovery from hypervisor failure. ReHype and comparisons between ReHype and NiLiHype are extensively discussed throughout this paper. There are works that use virtualization to provide resilience to device driver failures [15], [17] and the ability to recover from PrivVM failures [20]. The hypervisor, device drivers, and PrivVM together form the *virtualization infrastructure* (VI). The resilient VI can be combined with appropriate middleware to provide, on a virtualized system, a resilient platform for applications and services [18], [21].

IX. Conclusions and Future Work

There are compelling reasons to enhance hypervisors with the ability to recover from failures while allowing the VMs they host to resume normal operation without loss of work. With such capability, the fraction of datacenter capacity unavailable due to a single fault is reduced, there is greater flexibility in assigning VMs to hosts, and VM replication with both replicas running on a single host becomes an attractive point in the design space in some deployments.

ReHype, which is based on microreboot, has been previously presented as a mechanism for providing the above hypervisor recovery capability. This paper investigated an alternative to microreboot, which we call microreset, that allows component-level recovery without reboot. Instead of rebooting a new instance, microreset resets the component to a quiescent state that is highly likely to be valid. Microreset is suitable for large, complex components that process requests from the rest of the system. The state reset it performs involves discarding all threads of execution within the component. By avoiding component reboot, microreset has the potential to achieve significantly lower recovery latencies than microreboot.

NiLiHype utilizes microreset to implement recovery from hypervisor failures. To achieve a high recovery rate, NiLiHype includes numerous enhancements needed to restore the hypervisor to a valid consistent state. Thus, the idea of microreset, by itself, is not sufficient for building an effective recovery scheme.

We have implemented NiLiHype and evaluated it in terms of recovery rate, recovery latency, hypervisor processing over-

head during normal operation, and implementation complexity. We have shown that NiLiHype achieves a recovery rate of over 88%, only 2% lower than the rate achieved by ReHype. In return, NiLiHype's recovery latency is over a factor of 30 lower, at 22ms. With this low recovery latency, for many important application, service interruption would not be noticeable. Based on our measurements, the performance overhead of NiLiHype during normal operation is expected to be under 1%. NiLiHype's implementation required adding or modifying less than 2200 lines in the Xen hypervisor.

One of the remaining questions regarding microreset-based recovery, is the extent to which it is applicable to components other than OS kernels and hypervisors. Investigating this question is part of our future work.

Future work on NiLiHype will involve evaluation with more complex configurations, that include multiple vCPUs per CPU as well as a variety of workloads running in the VMs. We also plan to evaluate NiLiHype's effectiveness under additional fault types.

It is clear from prior work on ReHype as well as our work on NiLiHype that the development of enhancements necessary for achieving a high recovery rate is a difficult process. The changes to mitigate recovery failures due to non-idempotent hypercalls were particularly tough to develop. Another part of our future work is to investigate more systematic techniques for performing such enhancements.

REFERENCES

[1] "Cloc – count lines of code," http://cloc.sourceforge.net/, accessed: 2017-11-13.

[2] "Unixbench," https://github.com/kdlucas/byte-unixbench, accessed: 2017-10-12.

[3] 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," in *19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003, pp. 164–177.

[4] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The Turtles project: Design and implementation of nested virtualization," in *9th USENIX Conference on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010, pp. 423–436.

[5] K. Bhat, D. Vogt, E. van der Kouwe, B. Gras, L. Sambuc, A. S. Tanenbaum, H. Bos, and C. Giuffrida, "OSIRIS: efficient and consistent recovery of compartmentalized operating systems," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Toulouse, France, Jun. 2016, pp. 25–36.

[6] J. Buell, D. Hecht, J. Heo, K. Saladi, and H. R. Taheri, "Methodology for performance analysis of VMware vSphere under tier-1 applications," *VMware Technical Journal*, vol. 2, no. 1, pp. 19–28, Jun. 2013.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot — a technique for cheap recovery," in *6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004, pp. 31–44.

[8] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *5th USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, CA, Apr. 2008, pp. 161–174.

[9] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "CuriOS: improving reliability through operating system structure," in *8th USENIX Conference on Operating Systems Design and Implementation*, San Diego, California, Dec. 2008, pp. 59–72.

[10] A. Depoutovitch and M. Stumm, "Otherworld: Giving applications a chance to survive os kernel crashes," in *5th European conference on Computer systems*, Paris, France, Apr. 2010, pp. 181–194.

[11] J. Gray, "Why do computers stop and what can be done about it?" in *5th Symposium on Reliability in Distributed Software and Database Systems*, Jan. 1986, pp. 3–12.

[12] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a highly dependable operating system," in *Sixth European Dependable Computing Conference*, Coimbra, Portugal, Oct. 2006.

[13] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *25th Fault-Tolerant Computing Symposium*, Pasadena, CA, Jun. 1995, pp. 381–390.

[14] C. M. Jeffery and R. J. Figueiredo, "A flexible approach to improving system reliability with virtual lockstep," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 1, pp. 2–15, Jan. 2012.

[15] H. Jo, H. Kim, J.-W. 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.

[16] K. Kourai and S. Chiba, "A fast rejuvenation technique for server consolidation with virtual machines," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Edinburgh, UK, Jun. 2007, pp. 245–255.

[17] M. Le, A. Gallagher, Y. Tamir, and Y. Turner, "Maintaining network QoS across NIC device driver failures using virtualization," in *8th IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, Jul. 2009, pp. 195–202.

[18] M. Le, I. Hsu, and Y. Tamir, "Resilient virtual clusters," in *17th IEEE Pacific Rim International Symposium on Dependable Computing*, Pasadena, CA, Dec. 2011, pp. 214–223.

[19] M. Le and Y. Tamir, "ReHype: enabling VM survival across hypervisor failures," in *7th ACM International Conference on Virtual Execution Environments*, Newport Beach, CA, Mar. 2011, pp. 63–74.

[20] ——, "Applying microreboot to system software," in *IEEE International Conference on Software Security and Reliability*, Washington, D.C., Jun. 2012, pp. 11–20.

[21] ——, "Resilient virtualized systems using ReHype," UCLA Computer Science Department Technical Report #140019, Oct. 2014.

[22] ——, "Fault injection in virtualized systems – challenges and applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 3, pp. 284–297, May 2015.

[23] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: An organizing principle for recoverable operating systems," in *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, USA, Mar. 2009, pp. 49–60.

[24] W. T. Ng and P. M. Chen, "The systematic improvement of fault tolerance in the Rio file cache," in *29th Annual International Symposium on Fault-Tolerant Computing*, Madison, WI, Jun. 1999, pp. 76–83.

[25] R. Nikolaev and G. Back, "VirtuOS: an operating system with kernel virtualization," in *24th ACM Symposium on Operating Systems Principles*, Farmington, PA, Nov. 2013, pp. 116–132.

[26] H. P. Reiser, F. J. Hauck, R. Kapitza, and W. Schroder-Preikschat, "Hypervisor-based redundant execution on a single physical host," in *6th European Dependable Computing Conference, Supplemental Volume*, Coimbra, Portugal, Oct. 2006, pp. 67–68.

[27] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *IEEE Computer*, vol. 38, no. 5, pp. 39–47, May 2005.

[28] C. Tan, Y. Xia, H. Chen, and B. Zang, "TinyChecker: transparent protection of vms against hypervisor failures with nested virtualization," in *2nd International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology*, Boston, MA, Jun. 2012.

[29] VMware, "Providing fault tolerance for virtual machines," https://pubs.vmware.com/vsphere-4-esx-vcenter/topic/com.vmware.vsphere.availability.doc_41/c_ft.html, accessed: 2017-12-01.

[30] K. Yamakita, H. Yamada, and K. Kono, "Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery," in *41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Hong Kong, China, Jun. 2011, pp. 169–180.

[31] T. Yoshimura, H. Yamada, and K. Kono, "Can Linux be rejuvenated without reboots?" in *IEEE Third International Workshop on Software Aging and Rejuvenation*, Hiroshima, Japan, Nov. 2011, pp. 50–55.

[32] ——, "Is Linux kernel Oops useful or not?" in *Eighth USENIX Workshop on Hot Topics in System Dependability*, Hollywood, CA, Oct. 2012, pp. 1–6.

[33] W. Zhang and W. Zhang, "Linux virtual server clusters," *Linux Magazine*, vol. 5, no. 11, Nov. 2003.