7th ACM International Conference on Virtual Execution Environments
Newport Beach, California, pp. 63-74, March 2011.

ReHype: Enabling VM Survival Across Hypervisor Failures

Michael Le   Yuval Tamir
Concurrent Systems Laboratory
UCLA Computer Science Department
{mvle,tamir}@cs.ucla.edu

Abstract

With existing virtualized systems, hypervisor failures lead to overall system failure and the loss of all the work in progress of virtual machines (VMs) running on the system. We introduce ReHype, a mechanism for recovery from hypervisor failures by booting a new instance of the hypervisor while preserving the state of running VMs. VMs are stalled during the hypervisor reboot and resume normal execution once the new hypervisor instance is running. Hypervisor failures can lead to arbitrary state corruption and inconsistencies throughout the system. ReHype deals with the challenge of protecting the recovered hypervisor instance from such corrupted state and resolving inconsistencies between different parts of hypervisor state as well as between the hypervisor and VMs and between the hypervisor and the hardware. We have implemented ReHype for the Xen hypervisor. The implementation was done incrementally, using results from fault injection experiments to identify the sources of dangerous state corruption and inconsistencies. The implementation of ReHype involved only 880 LOC added or modified in Xen. The memory space overhead of ReHype is only 2.1MB for a pristine copy of the hypervisor code and static data plus a small reserved memory area. The fault injection campaigns used to evaluate the effectiveness of ReHype involved a system with multiple VMs running I/O and hypercall-intensive benchmarks. Our experimental results show that the ReHype prototype can successfully recover from over 90% of detected hypervisor failures.

Categories and Subject Descriptions

D.4.5 [Operating Systems]: Reliability - Fault-tolerance

General Terms

Reliability, Experimentation, Performance

Keywords

Virtualization, Reliability, Recovery, VMM, Microreboot, Xen

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'11, March 9-11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0501-3/11/03...$10.00.

1. Introduction

System virtualization [19] enables server consolidation by allowing multiple virtual machines (VMs) to run on a single physical host while providing workload isolation and flexible resource management. The hypervisor manages the access of the VMs to physical resources and is critical to the operation of the entire system. Failure of the hypervisor due to software bugs or transient hardware faults generally results in the failure of *all* the system's VMs. Recovery from such a failure typically involves rebooting the entire system, resulting in loss of the work in progress in all the VMs. This problem can be mitigated through the use of periodic checkpointing of all the VMs and restoration of all the VMs to their last checkpoint upon reboot. However, this involves performance overhead for checkpointing during normal operation as well as loss upon recovery of work done since the last checkpoint.

This paper introduces and evaluates a mechanism for recovery from hypervisor failures called *ReHype*. To the best of our knowledge, ReHype is the first mechanism that allows VMs to survive hypervisor failure without any loss of work in progress and without any performance overhead during normal operation. Upon hypervisor failure, ReHype boots a new hypervisor instance while preserving the state of running VMs. VMs are stalled for a short duration during the hypervisor reboot. After a new hypervisor is booted, ReHype integrates the preserved VM states with the new hypervisor to allow the VMs to continue normal execution.

Hypervisor failure almost always involves state corruption. Corruptions can occur in the hypervisor state as well as in VM state. Hence, no recovery mechanism that relies on system state at the time a failure is detected can be 100% guaranteed to restore all system components to valid states. Furthermore, since ReHype involves reinitializing part of the hypervisor state while preserving the rest of the state, the result of recovery may include inconsistencies in the hypervisor state, between hypervisor and VM states, and between the hypervisor and hardware states. For example, hypervisor failure can occur in the middle of handling a hypercall from a VM or before acknowledging an interrupt from a device controller.

A key contribution of our work is to identify the specific sources of state corruptions and inconsistencies, determine which of those are most likely to prevent successful recovery, and devise mechanisms to overcome these problems. We have implemented and tested ReHype with the Xen [2] hypervisor and VMs running Linux. We use the results of fault injection to incrementally enhance [17] an initial basic version of ReHype. These incremental steps improve the rate of successful recovery from an initial 5.6% of detected faults to over 90% of detected faults. Our evaluation of the final scheme points to ways in which the success rate can be further improved.

ReHype builds upon the Otherworld [4] mechanism for microrebooting the Linux kernel while preserving process states and the RootHammer [12] mechanism for rejuvenation of the Xen hypervisor through a reboot while maintaining VM state without the overhead of saving VM memory images to persistent storage. Compared to Otherworld, the memory overhead for ReHype is significantly lower. Furthermore, while ReHype does not involve any changes to the VMs or the applications in the VMs, Otherworld requires modifications for system call retries and, in many cases, application-level ‘‘crash procedures’’ that are invoked upon recovery. Finally, we design and evaluate enhancements to the basic recovery mechanism in the context of a hypervisor, while Otherworld is focused on microrebooting of an OS kernel. RootHammer does deal with the Xen hypervisor and provides proactive rejuvenation. However, proactive rejuvenation is simpler since it does not address recovery from failure and thus does not deal with possible arbitrary corruptions and inconsistencies throughout the system.

The following section discusses the requirements and approaches to recover from a failed hypervisor. Section 3, describes the implementation of a version of ReHype that provides basic transparent hypervisor recovery but does not deal with many problems caused by state corruptions and inconsistencies. Incremental improvements to ReHype, based on fault injection results, are described in Section 4. The details of the experimental setup are presented in Section 5. Section 6 presents an analysis of the results for the final version of ReHype. Related work is discussed in Section 7.

2. Tolerating VMM Failure

Hardware faults or software bugs in the virtual machine monitor (VMM†) can cause the corruption of VMM state or the state of VMs. As a result, the VMM or individual VMs may crash, hang, or perform erroneous actions. The safest way to recover the system is to reboot the VMM and also reboot each of the VMs. However, this requires a lengthy recovery process and involves loss of the work in progress of applications running in the VMs. Periodic checkpointing of VMs can reduce the amount of lost work upon recovery but the work done since the last checkpoint is lost and there is performance overhead during normal operation for checkpointing. Alternative mechanisms involve less overhead and lost work but may result in recovery of only parts of the system or even a complete failure to recover a working system. This section discusses the basic design alternatives for mechanisms that can recover from VMM failure.

Virtualization is often used to consolidate the workloads of multiple physical servers on a single physical host. With multiple physical servers, a single software or transient hardware fault may cause the failure of one of the servers. An aggressive reliability goal for a virtualized system is to do no worse than a cluster of physical servers. Hence, if a transient hardware fault or a software fault in any component (including the VMM) affects only one VM running applications, the goal is met. Recovery from VMM failure that avoids losing work in progress in the VMs necessarily relies on utilizing the VM states at the time of failure detection. One or more of those VM states may be corrupted, resulting in the failure of those VMs even if the rest of the system is restored to correct operation. Based on the reliability goal above, we define recovery from VMM failure to be successful if no more than one of the existing VMs running applications fails *and* the recovered VMM maintains its ability to host the other existing VMs as well as create and host new VMs.

Successfully ‘‘tolerating’’ VMM failure requires detection of such failures and successfully recovering from them, as defined above. To accomplish this goal, mechanisms must exist to: (1) detect VMM failure, (2) repair VMM corruption, and (3) resolve inconsistencies within the VMM, between the VMM and VMs, and between the VMM and the hardware. Detecting VMM failure boils down to being able to detect a VMM crash, hang, or silent data corruption, as described in Subsection 2.1. Subsection 2.2 discusses different approaches to repairing VMM corruption and the tradeoffs among them in terms of implementation complexity and expected success rates. Inconsistencies among the states of different components following recovery may be resolved entirely in the VMM or may require VM modifications. Details of the sources of inconsistencies and techniques for resolving inconsistencies are described in Subsection 2.3.

2.1. Detection

VMM failures can be manifested as VMM crashes, hangs, or silent corruption (arbitrary incorrect actions). Crashes can be detected using existing VMM panic and exception handlers. If the VMM panics, then a crash has occurred. Detecting VMM hangs requires external hardware. A typical hang detector, such as the one implemented in the Xen VMM, uses a watchdog timer that sends periodic interrupts to the VMM. The interrupt handler checks whether the VMM has performed certain actions since the last time the handler was invoked. If it has not, the handler signals a hang.

Silent VMM corruption is more difficult to detect. Detection mechanisms involve redundant (e.g., replicated) data structures and redundant computations (e.g., performing sanity checks). Fortunately, our fault injection results (Section 6) indicate that the majority of VMM failures (80%) are manifested as crashes and hangs and are thus detectable using the simple mechanisms discussed above. More extensive fault injection campaigns in the Linux kernel [7, 8] indicate that the fraction of VMM failures manifested as crashes or hangs is likely to be significantly higher.

2.2. Repairing VMM Corruption

Repair is initiated when the detection mechanism invokes a failure handler. Corrupted VMM state can then be repaired by either identifying and fixing the specific parts of the VMM state that are corrupted or simply booting a new VMM instance. A major difficulty with the first alternative is the requirement to identify which parts of the state are erroneous. This is likely to require significant overhead for maintaining redundant data structures. Furthermore, complex repair operations performed in the context of a failed VMM can increase the chance of failed recoveries [20]. Hence, we focus on the latter alternative.

Normally, a system reboot would cause the loss of all VM states. The simplest approach to preserve all VM states across a

† The terms *hypervisor* and *VMM* are used interchangeably.

reboot is to checkpoint the VMs to stable storage in the failure handler. Once a fresh system boots up, the VMs can be restored. However, checkpointing VM state in the context of a failed VMM can increase the chance of failed recoveries since the VMM must perform I/O and access possibly corrupted structures that hold VM state. In addition, the space and time overhead of checkpointing can be large as the state of all VMs must be copied to stable storage.

An alternative approach to a system-wide reboot, is to microreboot [3] the VMM. In this approach, VM states are preserved in memory across the reboot. This avoids the space and time overhead of checkpointing VM states to stable storage. Once the new VMM has been booted, it must be re-integrated with the preserved VMs. This re-integration can be done by either recreating the VMM structures used to manage the VMs or reusing VMM structures preserved from the old VMM. Either way, some amount of VMM data needs to be preserved across a VMM reboot for the re-integration process.

Variations of the microreboot approach can be categorized based on whether the new VMM is rebooted in place or in a reserved memory area, and, regarding VMM structures for managing VMs, whether to reuse these structures from the old VMM instance or create new instances of these structures. The following paragraphs discuss these variations, their effect on the ability to successfully recover a VMM, and the implementation complexity involved.

The choices of whether to reboot the VMM in place or in a reserved memory area affects the operations that must be done in the failure handler. This, in turn, can affect the chances of a successful recovery. When the new VMM is booted in place, the failure handler must perform two operations: 1) preserve VMM state (data structures) needed for re-integration by copying it to a reserved memory location, and 2) overwrite the existing VMM image in memory with a new image. These two operations do not have to be performed if the new VMM is booted into a reserved memory area. There is no VMM state to copy since the new VMM is confined to the reserved memory area on boot so no old VMM memory state is lost. In addition, the new VMM image can be preloaded into the reserved memory area without affecting the operation of the current VMM. Performing fewer operations in the failure handler, as is the case when rebooting into a reserved area, can increase the chance of a successful recovery. However, as discussed below, the choice of booting the new VMM instance in a reserved memory area requires complex operations for the re-integration of the new VMM instance with the preserved VMs.

Since the state of the old VMM instance may be corrupted, the ability to successfully recover is directly related to the amount of data reused from the old VMM instance. In some cases, data structures in the new VMM instance can be re-initialized to static values (e.g., clearing all locks) or reconstructed from sources that are unlikely to be corrupted (e.g., obtaining the CPUID of a core from the hardware). However, some data structures are dynamically updated based on the activity of the system and cannot be re-initialized with static or ‘‘safe’’ values. For example, a corrupted VM page table can allow a VM access to the VMM or another VM’s memory space.

Re-integrating the new VMM state with preserved VM states involves creating VMM structures that can manage the VMs. This can be done by either directly reusing the preserved VMM structures from the old VMM instance or creating new instances of the structures and populating them with values from the old VMM instance. The first alternative is simpler to implement as only pointers to the preserved structures need to be restored in the new VMM instance. Implementation of the second alternative is more complex, as it requires deep copy of all the required structures from the old VMM and updating all pointers within those structures. With either of these alternatives there is a possibility of ending up with corrupted values in the new VMM’s data structures. However, if the preserved structures are reused, there is a greater risk of also introducing into the new VMM instance corrupted pointers, which may lead to further corruption.

There is no perfect solution to the problem of ending up with corrupted values in the new VMM’s data structures. As discussed above, some structures can be initialized with static or ‘‘safe’’ values. In other cases, data structures may be populated based on other data structures from the failed VMM instance, at least ensuring consistency (see Section 2.3). In general, the recovery process needs to ensure that all values that will be used by the new VMM instance are safe — will not lead to subsequent VMM failure. This may involve performing integrity checks, possibly requiring maintaining redundant information, such as logs, during normal operation.

Given the tradeoffs presented in this subsection, ReHype uses the microreboot approach and opts for a simple implementation that does not require major modifications to the VMM. Hence, ReHype preserves and reuses almost all of the VMM’s dynamic memory but updates a few key data structures with ‘‘safe’’ values, as described in sections 3 and 4. The benefit of reusing most of the VMM’s data structures is that it allowed ReHype to be easily integrated into a VMM (Xen in our case) with minor (880 LOC added/modified) modifications. Despite the reuse of old VMM data in ReHype, fault injection results show a high rate of successful recoveries (Section 6).

2.3. Resolving Inconsistencies

By design, the VMM is assumed to be reliable by the VMs and hardware. Typically, the VMM itself is, for the most part, implemented with the assumption that there are no hardware faults and no software faults in the VMM code. These assumptions are violated when the VMM fails. Thus, after recovery from a failed VMM, even if none of the states of the system components are corrupted, these states may be *inconsistent*, preventing the system from operating correctly.

The VMM executes some operations in critical sections to ensure atomicity, e.g. update a VM grant table. Atomicity can be violated when a VMM failure occurs in the middle of such critical sections. In such cases, some data structures may be partially updated, leading to inconsistencies within the VMM (VMM/VMM). The VMs expect the VMM to provide a monotonically increasing system time, handle hypercalls, and deliver interrupts. The hardware expects the VMM to acknowledge all interrupts it receives. When a VMM failure occurs, the assumption of a reliable VMM is violated and this can lead to inconsistencies between the VMM and VMs (VMM/VM) and between the VMM and hardware (VMM/hardware). The recovery process must resolve these inconsistencies so that the

virtualized system can continue to operate correctly. The following paragraphs discuss in more detail these inconsistencies and the techniques for resolving them.

Sources of VMM/VMM inconsistencies include partially updated structures, unreleased locks, and memory leaks. The options for resolving these inconsistencies are, essentially, special cases of the options for dealing with state corruption, discussed in the previous subsection. Resolving inconsistencies caused by partially updated structures requires either reconstructing the data structures using information from the failed VMM or using redundant information logged prior to failure to fix the new VMM's state. A scheduler's run queue is an example of a data structure for which the former technique can be used. Inconsistency can occur if a VCPU becomes runnable but the VMM fails before inserting it into the run queue. Resolving this inconsistency requires re-initializing the run queue to empty upon bootup and re-inserting all runnable VCPUs (obtained from the failed VMM) into the run queue. For other data structures, such as the ones that track memory page usage information, reconstruction is more difficult so the latter technique may be preferable. For instance, an inconsistency can occur if a failure happens right when a page use counter has been updated but before that page has been added to a page table. Resolving this inconsistency by traversing all page table entries to count the actual mappings to that page can be done but is complex and slow. Instead, the entire mapping operation can be made atomic with respect to failure using *write-ahead logging*, involving a small overhead during normal operation and simple, fast correction of any inconsistencies upon recovery.

Locks and semaphores that are acquired prior to VMM failure must be released (re-initialized to a static value) upon recovery to allow the system to reacquire them when needed. In order to do so, all locks and semaphores must be tracked and re-initialized in data structures that are reused or copied from the failed VMM.

A memory leak can occur if a failure happens between allocation and freeing of a memory region in the VMM. Such a memory leak is benign if the leaked region is small. After VMM recovery, the system can be scheduled to be rebooted to reclaim leaked memory. Alternatively, leveraging work by Kourai and Chiba[13], after recovery, the virtualized system can be quickly rejuvenated to get rid of any memory leaks.

Sources of VMM/VM inconsistency include non-monotonically increasing system time, partially executed hypercalls, and undelivered virtual interrupts. The correct operation of many VMs depends on a monotonically increasing system time. In a virtualized system, the VMs' source of time is the VMM. When a VMM is rebooted, its time keeping structures are reset, potentially resulting in a time source for VMs that is not monotonically increasing. In addition, such a reset can result in timer events set using time relative to the VMM's system time prior to recovery to be delayed. One technique for resolving this inconsistency is to simply save the VMM time structures upon failure and restore those structures after the VMM reboot and before the VMs are scheduled to run. This allows time to continue moving forward with no interruption visible by the VMs.

When the VMM recovers from a failure, partially executed hypercalls must be re-executed. Our experimental results show that, at least for the hypercalls that were exercised by our target system, simply retrying hypercalls does work most of the time and allows VMs to continue to operate. In general, hypercalls that are not idempotent may fail on a retry, in which case the VM executing the hypercall may also fail.

Hypercall retry can be implemented by modifying the VM to add a "wrapper" around hypercall invokation that will re-invoke the hypercall if a retry value is returned by the VMM. The VMM must also be modified to return a retry value indicating a partially executed hypercall. This approach allows the VMs more control over which hypercalls to retry and allows the VMs to gracefully fail if a hypercall retry is unsuccessful.

Hypercall retry can be implemented without modifying the VMs. To force re-execution of a hypercall after recovery, the VMM adjusts the VM's instruction pointer to point back to the hypercall instruction (usually a trapping instruction). When the VM is scheduled to run, the very next instruction it executes will be the hypercall. This mechanism is already used in the Xen [2] VMM to allow the preemption of long running hypercalls transparently to the VMs.

The VMM is responsible for delivering interrupts from hardware and event signals from other VMs as virtual interrupts to the destination VM. These virtual interrupts may be lost if the VMM fails. Some inconsistencies of this type can be resolved without any modifications to the system by relying on existing timeout mechanisms that are implemented in the kernels and device drivers of the VMs. A timeout handler can resend commands to a device or resignal another VM if an expected interrupt does not arrive within a specified period of time. We have verified that timeout mechanisms exist for the Linux SCSI block driver (used for SATA disks) and the Intel E1000 NIC driver, representing the most important devices for servers (storage and network controllers). Obviously, such timeout mechanisms do not deal with lost interrupts from unsolicited sources, such as packet reception from a network device. However, at least for network devices, the problem is ultimately resolved by existing higher-level end-to-end protocols (e.g., TCP).

A source of VMM/hardware inconsistency is unacknowledged interrupts. Interrupt controllers will block an interrupt source until the previous interrupt from that source has been acknowledged by the processor. Since VMM failure can occur at any time, pending interrupts may never get acknowledged, thus blocking the interrupt source indefinitely. If VMM recovery is done without performing a hardware reset, a mechanism is needed to either reset the I/O controller or to acknowledge all pending interrupts during recovery. In the case of acknowledging pending interrupts, the interrupt source must be blocked at the interrupt controller before the interrupt is acknowledged to prevent another interrupt from slipping by before the VMM is ready to handle the interrupt.

3. Transparent VMM Microreboot

We have implemented a ReHype prototype for version 3.3.0 of the Xen [2] VMM. This section describes the implementation of a version of ReHype that provides the basic capability to microreboot the VMM while preserving the running VMs and allowing them to resume normal execution following the

microreboot. Improvements to the basic scheme that enhance recovery success rates are discussed in Section 4. So far, ReHype has been evaluated and validated only with paravirtualized VMs. However, based on preliminary experimentation, we expect the current ReHype implementation, with few or no modifications, to operate successfully with fully-virtualized VMs.

To microreboot the VMM, ReHype uses the existing Xen port of the Kdump [6] tool. Kdump is a kernel debugging tool that provides facilities to allow a crashed system to load a pristine kernel image, in this case the VMM image, on top of the existing image and directly transfer control to it. The Kdump tool by itself, however, does not provide any facilities to preserve parts of memory, such as those holding VM states. The burden of memory preservation is on the kernel or VMM being booted.

A VMM microreboot is differentiated from a normal VMM boot by the value of a global flag added to the initialized data segment of Xen. The flag is clear in the original VMM image on disk but is set after loading the recovery image to memory using the Kdump tool upon initial system boot. All the modifications to the bootup process described henceforth refer to actions performed when the flag is set (the microreboot path).

On boot, the stock Xen VMM initializes the entire system memory and allocates memory for its own use. The modifications for ReHype must ensure that, upon recovery, the new VMM instance preserves the memory used by VMs and memory that was used by the previous VMM instance to hold state needed for managing the VMs. Hence, as described in Subsection 3.1, the ReHype version of the Xen VMM allocates "around" the preserved memory regions during a VMM microreboot. When the new VMM instance is booted and initialized, it does not contain information about the running VMs, and thus has no way to run and manage them. Subsection 3.2 describes how the VMM and VM states preserved during VMM recovery are re-integrated with the new VMM instance.

3.1. Preserving VMM and VM States

The state that must be preserved across a VMM microreboot includes information in the VMM's static data segments, the VMM's heap, the structure in the VMM that holds information about each machine page, and special segments that are normally used only during VMM bootup.

VMM microreboot involves overwriting the existing VMM image (code, initialized data, and bss) with a pristine image. The VMM's static data segments (initialized data and bss) contain critical information needed for the operation of the system following bootup of the new VMM instance. For example, this includes interrupt descriptors and pointers to structures on the Xen heap, such as Domain 0's VM structure and the list of domains. Hence, some of the information in the old static data segments must be preserved. While it is only necessary to preserve a subset of the static data segments, since they are relatively small, we reduce the implementation complexity by preserving the segments in their entirety.

Figure 1 shows the memory layout for Xen, as modified for ReHype. In particular, the bss segment is extended by approximately 1MB — sufficient space to hold complete copies of the original bss and initialized data segments. This is done by changing the linker script used for creation of the Xen image. Since the area reserved is in the bss segment, no extra disk space is taken up for the new VMM image. The bss segment only takes up space when it is loaded into memory. Upon failure detection, before initiating a VMM microreboot, the failure handler copies the initialized data and (unextended) bss segments to this new reserved memory.

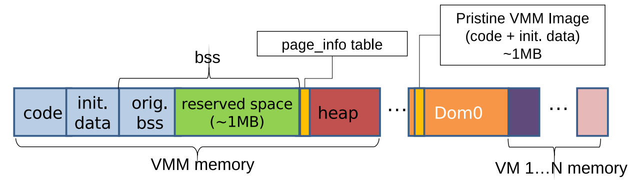

Figure 1: Layout of system memory for ReHype.

As discussed above, for each VM, the state that must be preserved includes both the VM's memory image and parts of the VMM state used to manage the VM. Since this information is maintained on the VMM's heap, the VMM's heap must be preserved. Preserving the VMM's heap requires modifications of the VMM's bootup code. During VMM initialization, before the heap is created, the old heap (from the previous VMM instance) is walked to identify all the free pages. When the new heap is created and populated with free pages, only pages that are free in the old heap are added to the new heap. This ensures that the new VMM will not allocate (overwrite) any pages that are still in use. To access the old heap, the page table from the old VMM must be restored. This requires copying the old page directory from the old bss segment, preserved as discussed above, to the new bss segment.

The Xen VMM maintains a structure (page_info) that holds information about each machine page, such as the page ownership and use count. For all the pages that are preserved across a VMM microreboot, the information in this structure must be preserved. This structure is allocated in a special memory area, between the bss segment and the heap. The VMM bootup code was modified to avoid initializing the page_info entries for pages that are not free.

The stock Xen VMM image includes two static segments (init.text and init.data) that are normally used during bootup and then freed to the heap so that they can be used for other purposes. Hence, with stock Xen, a microreboot would overwrite these segments, potentially corrupting data in pages that had been reallocated. To prevent this problem, the bootup code (normal and microreboot) has been modified to avoid freeing these pages. This results is an extra 100KB memory overhead.

Preserving the heap and static data segments of a failed VMM is unsafe — it can result in recovery failure if those preserved values are corrupted by the failed VMM. Section 4 will discuss mechanisms that dramatically improve the chances of successful recoveries despite re-using the preserved heap and static data segments.

3.2. Re-integrating VMM and VM States

Following a VMM microreboot, the VMM does not have the information required to resume execution and manage the VMs that were running at the time of failure. The missing system state

includes the list of VMs it was managing, the system time that was provided to the VMs, information for interrupt routing (to processors) and forwarding (to VMs), and timer events that were scheduled by the VMs. To allow the virtualized system to continue running, these components of the system state must be restored. As discussed in the previous subsection, all the required state is preserved across a VMM microreboot. Hence, all that is needed is to re-integrate the preserved information with the new VMM instance. This re-integration is accomplished by copying a few key values from the old static data segments to the new static data segments. The restoration is done before the VMs can be scheduled to run. The following structures must be restored:

- Pointer to xmalloc free list: prevent memory leaks.
- Pointers to the domain list and hash table: allow Xen to access the state of the running VMs.
- Pointer to the Domain 0 descriptor: since Domain 0 is not rebooted as part of recovery, the pointer to it must be restored to allow Xen access to the Domain 0 structure.
- Pointers to timer event objects: restore pending timer events on the old timer heap to the new timer heap.
- Pointer to the machine-to-physical (m2p) mapping table: make available mapping of machine frame numbers to physical frame numbers.
- System time variables: maintain monotonically increasing time. The time-stamp counter (TSC) must not be reset.
- IRQ descriptor table and IO-APIC entries, including correct IRQ routing affinity and mask: allow VMs to continue to receive interrupts from their devices.
- Structures for tracking the mappings of pages shared between VMs and the VMM: prevents overwriting mappings that are still in use.

There are two additional differences between the VMM microreboot path and a normal VMM boot: Domain 0 is not created and VMs are re-associated with the scheduler. The bootup code has been modified to skip Domain 0 creation and to restore the Domain 0's global pointer so that the new Xen can access Domain 0's state. VMs are re-associated with the VMM's scheduler by invoking the scheduling initialization routines for each VM and inserting runnable VCPUs into the new run queue.

Some of the structures needed by the VMM are re-created during a microreboot. These include the idle domain as well as structures holding hardware information such as the model and type of the CPU and the amount of memory available. For structures that are re-created on the heap, ReHype prevents a memory leak by first freeing the old structures.

4. Improving Recovery

The scheme presented in the previous section provides basic capabilities for VMM microreboot. However, as explained below, with this basic mechanism the probability of successful recovery is very low. The section starts with the basic scheme and incrementally improves on that scheme to achieve higher recovery success rates. Table 1 shows the mechanisms used to improve the basic recovery scheme. As in [17], the choice of improvements is guided by results from fault injection experiments.

We used software-implemented fault injection to introduce errors into CPU registers when the VMM is executing. The goal of the injection was to cause arbitrary failures in the VMM and evaluate the effectiveness of different recovery mechanisms. Two system setups were used: 1AppVM and 3AppVM. The 1AppVM setup was comprised of two VMs: Domain 0, the *Privileged VM* (PrivVM), and an Application VM (AppVM) running a disk I/O (block) benchmark. This setup was used to quickly identify major short comings with the recovery mechanisms. The more complex 3AppVM system setup, shown in Figure 2, was used to further stress the virtualized system, once the majority of sources of failed recoveries had been fixed. In this setup, AppVM2 is designed to boot after VMM recovery. This is a check of whether the recovered virtualized system maintains the ability to create new VMs.

Table 1. Improvements over the basic ReHype recovery.

Mechanism	Description
NMI IPI	Use NMI IPIs in failure handler. Avoid IPI blocking by failed VMM.
Acknowledge interrupts	Acknowledges all pending interrupts in all processors to avoid blocked interrupts after recovery.
Hypercall retry	All partially executed hypercalls are retried transparently to the VMs.
FixSP	Stack pointer set to "safe" value in failure handler.
NMI "ack"	Execute iret to "ack" NMI when hang detected on non-CPU0.
Reinitialize locks	Dynamically allocated spin locks and non-spin locks are unlocked.
Reset page counter	Reset page use counter based on page validation bit.

Table 2. Injection outcomes.

Outcome	Description
Detected VMM failure	Crash: VMM panics due to unrecoverable exceptions Hang: VMM no longer makes observable progress
Silent failure	Undetected failure: No VMM crash/hang detected but applications in one or more VMs fail to complete successfully
Non-manifested	No errors observed

Table 2 summarizes the possible outcomes from an injected fault (injection experiment). Only detected VMM failures lead to VMM recoveries. With the 1AppVM setup, a recovery is considered successful if the benchmark in the AppVM completes correctly. With the 3AppVM setup, following the explanation in Section 2, a recovery is considered successful if either AppVM0 or AppVM1 completes its benchmark correctly and AppVM2 is able to boot and run its benchmark to completion without errors. Silent failures, discussed in Section 6, do not trigger VMM recovery and are thus excluded from further discussion in this section. Details of the experimental setup, fault injection campaign, and failure detection mechanisms are in Section 5.

In the rest of this section, each version of the recovery

Table 3. Fault injection results for 1AppVM system setup. Percentage of successful recoveries out of detected VMM failures (VMM crash/hang).

Mechanism	Successful Recovery Rate
Basic	5.6%
+ NMI IPI	17.6%
+ Ack interrupts	48.6%
+ Hypercall retry	62.6%
+ FixSP+NMI "ack"	77.0%
+ Reinitialize locks	95.8%

scheme is described, and fault injection results are presented. This is followed by an analysis of the main cause of failed recoveries, motivating the next version of the recovery scheme. At each step, only the problem that leads to the plurality of failed recoveries is analyzed and fixed. Table 3 summarizes the rate of successful recoveries with the basic ReHype scheme and the various incremental improvements that are made.

Basic: As shown in Table 3, with the Basic recovery scheme (Section 3), the successful recovery rate is only 5.6%. A large fraction of recovery failures (44%) occur because the failure handler is unable to initiate the VMM microreboot. Normally, the failure handler relies on interprocessor interrupts (IPIs) to force all processors to save VM CPU state and halt execution before microrebooting the VMM. Microrebooting the VMM cannot proceed until all processors execute the IPI handler. Therefore, the failure handler is stuck if a processor is unable to execute the IPI handler due to a blocked IPI or memory corruption.

NMI IPI: To get around the above problem, NMI IPIs can be used. In addition, a spin lock protecting a structure used to set up an IPI function call must be busted to prevent the failure handler from getting stuck.

Table 3 shows an increase in recovery success rate to 17.6% when these fixes are used. Only 8.2% of the failures are now caused by an inability to initiate the VMM microreboot. The plurality of the remaining failures (45%) are due to interrupts from the block device not getting delivered to the PrivVM. This causes the block device driver in the PrivVM to time out, thus leading to the failure of block requests from the AppVM.

The block device uses level-triggered interrupts. For such interrupts, the I/O controller blocks further interrupts until an acknowledgment from the processor arrives. If the VMM fails before acknowledging pending interrupts, those level-triggered interrupts remain blocked after recovery.

Acknowledge interrupts: To prevent level-triggered interrupts from being blocked, the failure handler must acknowledge all pending interrupts on all processors. This must be done in the failure handler since information about pending interrupts are cleared after a CPU reset during a VMM reboot.

Table 3 shows that when this mechanism is added, the successful recovery rate jumps to 48.6%. Of the remaining unsuccessful recoveries, 52.8% are caused by a crashed AppVM or PrivVM after recovery. The crashes are caused by bad return values from hypercalls. Since VMM failures can occur in the middle of a hypercall, it is necessary to be able to transparently continue the hypercall after recovery. Without mechanisms to do this, after recovery, the VM starts executing right after the hypercall, using whatever is currently in the EAX register as the return value.

Hypercall retry: The ability to restart a hypercall is already provided in Xen. The mechanism involves changing the program counter (PC) of the VCPU to the address of the instruction that invokes the hypercall. For each VM, the VMM determines whether a hypercall retry is needed after the VMM microreboot, before loading the VM state. Specifically, for each VCPU, the VMM checks if the VCPU's PC is within the VM's hypercall page. If so, the VMM updates the VCPU's PC. Arguments to the hypercall are already preserved in the VM VCPU state.

Table 3 shows that, with hypercall retry, the successful recovery rate is 62.6%. Out of the remaining unsuccessful recoveries, 41% are caused by the same symptom encountered and partially solved with the Basic scheme — the inability of the failure handler to initiate the VMM microreboot. With the improved recovery rate, the causes of this symptom not previously resolved are now responsible for the plurality of failed recoveries.

The experimental results show two causes for the symptom above: 1) NMI IPIs sent to the wrong destination CPU due to stack pointer corruption and 2) NMIs are blocked due to NMI-based watchdog hang detection. Problem (1) occurs because a corrupted stack pointer is used to obtain the CPUID of the currently running processor. The obtained CPUID is incorrect and is, in turn, used to create a CPU destination mask for the NMI IPI. This mask can end up containing the sending processor as one of the destination CPUs. The result of this is that an IPI is incorrectly sent to the sending processor. This IPI is dropped and the sender waits forever for the completion of the IPI handling.

Problem (2) is due to the fact that NMI delivery is blocked if a CPU is in the middle of handling a previous NMI — an *iret* instruction matching a previous NMI has not been executed [10]. The Xen hang detector is based on periodic NMIs from a watchdog timer. If a hang is detected on a processor, that processor immediately executes the panic handler and never executes an *iret* instruction. This prevents the processor from getting an NMI IPI from the boot processor to initiate recovery.

FixSP+NMI "ack": Problem (1) above can be fixed by not relying on the stack pointer to obtain the CPUID during failure handling. Instead the CPUID can be obtained by first reading the APICID from the CPU and then converting the APICID to CPUID by using an existing APICID to CPUID mapping structure stored in the static data segment of Xen. With this technique, the VMM has a chance to continue with the recovery despite a corrupted stack pointer. However, the corrupted stack pointer can cause critical problems that are unrelated to the CPUID. Specifically, the handler invoked when VMM failure is detected must save VCPU registers (located on the stack) into preserved VMM state. A corrupted stack pointer leads to saving the contents of a random region in memory as the saved VCPU register values. This can cause the VMM to crash after recovery when trying to restore the corrupted (incorrect) values to VCPU registers. Specifically, when attempting to load the saved VCPU registers after recovery, the VMM may try to restore a corrupted value as the VCPU code segment register. This may cause the VMM to continue executing with VMM privilege using corrupted (incorrect) register values.

ReHype implements a solution to Problem (1) above that avoids the deficiency described in the previous paragraph. Specifically, the failure handler, invoked upon VMM failure, sets the stack pointer to a ''safe'' value. This can be done based on the observation that the failure handler never returns, and therefore, the stack pointer can be reset to any valid stack location. The address of the bottom of the stack is kept by Xen in a static data area. The stack pointer is set to that value minus sufficient space for local variables used by the failure handler.

Problem (2) above is resolved by forcing the execution of *iret* in the failure handler. The stack is manipulated so that the *iret* instruction returns directly to the failure handler code.

With the two improvements above, the rate of successful recoveries is 77.0%. The majority of the increase is due to fixing the stack pointer. Since hangs are responsible for only a small fraction (7.1%) of detected VMM failures, the impact of fixing problem (2) on the overall recovery success rate is small. However, with this fix, there was successful recovery from all hangs detected in this experiment.

Out of the remaining unsuccessful recoveries, 82.8% are due to spin locks being held after recovery. Spin locks that are statically allocated are re-initialized on boot but locks that are on the heap are not. This causes the VMM to hang immediately after recovery.

Reinitialize locks: Re-initializing dynamically-allocated spin locks requires tracking the allocation and de-allocation of these locks. All locks that are still allocated upon recovery are initialized to unlocked state. This tracking of spin locks is the only extra work that ReHype must perform during normal operation. The associated performance overhead is negligible since the allocation and de-allocation of spin locks is normally done only as part of VM creation and destruction. Furthermore, there are only about 20 spin locks that are tracked per VM.

Locking mechanisms that are not spin locks must also be re-initialized to their free states. A key example of this are the page lock bits used to protect access to bookkeeping information of pages. With the previous version of the recovery scheme, not initializing these bits resulted in 10% of unsuccessful recoveries.

As shown in Table 3, re-initializing locks increases successful recovery rate to 95.8%. For the remaining recovery failures there is not one dominant cause.

While the 1AppVM system setup is useful for uncovering the main problems with the Basic ReHype recovery, it is very simple, thus potentially hiding important additional problems. To better stress the virtualized system, the rest of the experiments in this section use the 3AppVM setup. The results with this setup are summarized in Table 4.

As shown in Table 4, with the 3AppVM setup, the reinitialize locks mechanism results in a recovery success rate of 90.2%. Hence, there is a small decrease in the success rate compared to the 1AppVM setup. Out of the remaining recovery failures, 25% are due to the VMM hanging immediately after recovery. This problem is caused by a data inconsistency resulting from a VMM failure while in the middle of handling a page table update hypercall. This hypercall promotes an unused VM page frame into a page table type by incrementing a page type use counter and performing validity check on the page frame. After the validity check, a validity bit is set to indicate that the page can be used as a page table for the VM. Inconsistency arises when a VMM failure occurs before the validity check is completed but after the page type use counter has been incremented. When the hypercall is retried after recovery, since the page use counter is not zero and the validity bit is not set, it assumes validation is in progress and waits by spinning. Of course, there is no validation taking place, and the CPU is declared hung by the hang detector.

Table 4. Fault injection results using 3AppVM system setup. Percentage of successful recoveries out of detected VMM failures (VMM crash/hang).

Mechanisms	Successful Recovery Rate
Reinitialize locks	90.2%
+ Reset page counter	94.3%

Reset page counter: To fix the above problem, code is added during VMM bootup to check the consistency between the validity bit and page use counter. If the page use counter is non-zero but the validity bit is not set, then the page use counter is set to zero.

With the page counter fix employed, recovery success rate improves to 94.3% The remaining causes of failed recoveries vary widely and are discussed in more detail in Section 6.

5. Experimental Setup

This section presents the experimental setup used to evaluate the ReHype prototype. It discusses details of the target virtualized system, the benchmarks running in the Application VMs (AppVMs), the VMM failure detection mechanisms (that trigger recovery), and the fault injection campaign.

The evaluation of ReHype was done on a system comprising of the Xen VMM, augmented with ReHype, hosting multiple paravirtualized VMs. To simplify the setup for software-implemented fault injection, the target system was run inside a fully-virtualized (FV) VM [14]. This made it easy to restart the target system and refresh its disk images after each injection run to isolate the effects of faults injected in different runs.

As mentioned in Section 4, two system configurations were used: 1AppVM and 3AppVM. With the 1AppVM configuration, the system hosts two VMs: a PrivVM (Domain 0) and a single AppVM. The AppVM runs the *blkbench* benchmark, described below, which continuously performs disk I/O (block) operations. The PrivVM hosts the block backend for the AppVM. Each of the VMs consists of one virtual CPU (VCPU) that is pinned to its own physical CPU (PCPU).

The 3AppVM configuration is shown in Figure 2. In this configuration, the PrivVM's root filesystem is in memory and the PrivVM does not access any devices. A separate Driver VM (DVM)[15] has direct access to the network and SCSI controllers and serves as the network backend for AppVM0 and block backend for AppVM2. AppVM1 has direct access to the IDE controller, and thus does not rely on the DVM for any device access. Each VM consists of one VCPU which is pinned to its own PCPU. To check whether the recovered system maintains its

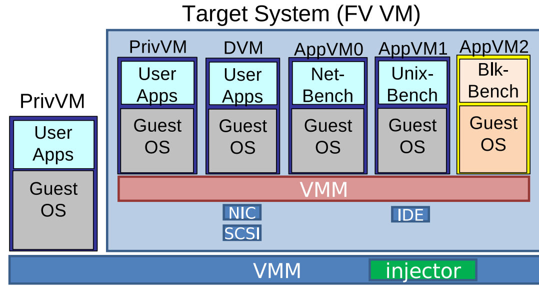

Figure 2: System configuration for 3AppVM. AppVM2 created after recovery.

ability to create new VMs, AppVM2 is designed to boot up after a possible VMM recovery and run the *blkbench* benchmark.

Three application benchmarks are used in our experiments: *netbench*, *blkbench*, and *UnixBench*. *Netbench* is a user-level *ping* program that exercises the interface to the network. It consists of two processes: one running inside an AppVM (the *receiving host*), and another running on a separate physical machine (the *sending host*). Every 1ms the sending host transmits a UDP packet to the receiving host, which, upon receiving this packet, transmits UDP packet back to the sending host. Unsuccessful application completion occurs when the packet reception rate on the *sending host* drops below a threshold (450 packets per second).

Blkbench stresses the interface to the block device (disk) by creating directories and creating, removing, and copying 1MB files. To ensure block activity, this benchmark prevents caching of block and filesystem data inside the AppVM's OS. Unsuccessful application completion occurs when the application reports an error (failed system call) or the resulting disk image when the benchmark completes differs from a reference image.

The third benchmark (*UnixBench*) is a subset of the set of programs in UnixBench [1] with minor modifications to improve logging and failure detection. The selected programs were chosen for their abilities to stress the VMM's handling of hypercalls such as virtual memory management and process scheduling. Unsuccessful application completion occurs when one or more programs in UnixBench terminate prematurely due to failed system calls or the resulting program output differs from a reference output.

As summarized in Table 2, the outcome of each injection run is classified as: detected (VMM crash/hang), silent (undetected failure), or non-manifested. A crash occurs when the VMM panics due to unrecoverable exceptions. Hangs are detected using a watchdog mechanism built into Xen. Specifically, Xen maintains a watchdog counter that is supposed to be incremented by a normal timer event every 500ms. A watchdog NMI is generated every 500ms of unhalted CPU cycles. If the watchdog NMI handler detects that the watchdog counter has not been incremented for 2s, the system is declared hung.

A silent VMM failure occurs when no VMM hang or crash is detected but the applications (benchmarks) in one or more AppVMs fail to complete successfully. What constitutes unsuccessful completion is application specific and is discussed above. Non-manifested means that no errors are observed.

We used the UCLA *Gigan* fault injector [14, 9] to evaluate ReHype. Gigan can reside in the VMM and inject many types of faults into the VMs and the VMM. Injection into VMs can be done without any modifications to the VMs. With the ReHype evaluation setup, the entire target system is in an FV VM so the injection does not require any modifications (intrusion) of the target system. Single bit-flip faults were injected into the registers of the processors during the execution of VMM code. While these injected faults do not accurately represent all possible faults, they are a good choice since transient hardware faults in CPU logic and memory are likely to be manifested as erroneous values in registers. Furthermore, these faults can cause arbitrary corruptions in the entire system. Hence, this limited fault injection satisfies the main goal of the evaluation, which is to "stress" the recovery mechanisms in order to identify problem areas. There is a possibility that some problem areas remain and will be uncovered only by a more comprehensive fault injection campaign. This will be investigated in future work.

A fault injection campaign consists of many fault injection runs. A single fault injection run that uses the 1AppVM system configuration consists of first booting the VMM along with the PrivVM and AppVM. The AppVM begins running the blkbench benchmark and a fault is injected into the VMM. The injection campaign infrastructure allows the target system sufficient time for the VMM to recover and for the benchmark to complete. If the benchmark does not complete, a timeout mechanism identifies system failure. At the end of each run, fault injection logs and benchmark output are retrieved and stored for analysis.

An injection run using the 3AppVM configuration is similar to the 1AppVM configuration except that an injection is performed only after the VMM, PrivVM, DVM, AppVM0, and AppVM1 have been booted and the two AppVMs have started running their respective benchmarks. 9s after the two AppVMs begin running their benchmarks, AppVM2 is booted to run its own benchmark. The injection run ends when all three AppVMs complete their benchmark runs or a timeout occurs.

An injection is triggered after a random time period between 500ms to 6.5s after the AppVMs begin running their benchmarks. To ensure that the injection occurs only when the VMM is executing, a fault is only injected after the designated time has elapsed and 0 to 20,000 VMM instructions, chosen at random, have been executed. The injection is a single bit-flip into a randomly selected bit of a randomly selected register. The target registers include general purpose registers, instruction and stack pointer registers, and the system flags register. Each injection selects randomly among the VCPUS of the target system.

6. Analysis

This section analyzes fault injection results for the final version of ReHype, with all the recovery improvements from Section 4. We discuss: I) the impact of the distribution of injections across CPUs, II) the causes of AppVM failure in the successful VMM recoveries that result in single AppVM failure, and III) the causes of VMM recovery failures and silent system failures.

It is expected that faults in a CPU that rarely executes VMM code are less likely to lead to VMM failures than faults in a CPU that executes a larger fraction of VMM code. Due to different activities on different VMs, the execution time of VMM code is

not evenly distributed across the CPUs hosting these VMs. However, for the fault injection results presented so far, injections were uniformly distributed across CPUs. Hence, it is critical to evaluate whether the results are qualitatively different if the distribution of fault injections is adjusted to match the fraction of time each CPU spends executing VMM code.

We use the Xenoprof profiler [16] to measure the distribution of Xen execution time across CPUs when running the 3AppVM setup. As explained in Section 5, each VM is pinned to a CPU. The results are as follows: 7.5% PrivVM CPU, 11.7% DVM CPU, 5.1% AppVM0 (NetBench) CPU, and 75.6% AppVM1 (UnixBench) CPU. The AppVM2 CPU is not included in the profile because it is created after the injection.

Table 5. Fault injection results for the final version of ReHype with injections distributed across CPUs uniformly or weighted by VMM execution time. Percentage of successful recoveries are out of detected VMM failures. Percentage of silent failures are out of manifested faults.

Distribution of Injections Across CPUs	Manifested		
	Detected: Successful Recovery Rate	Silent Single AppVM Failure	Silent System Failure
Uniform	94.3%	6.0%	14.0%
Weighted	94.5%	6.6%	11.3%

Table 5 shows the injection results using the uniform distribution across CPUs and when injection distribution is weighted VMM execution time. The results are very similar. This is due to a combination of two factors. First, the fraction of injected faults leading to detected VMM failures is approximately the same across the four CPUs (24%-27%). Second, under uniform distribution of injections, 5.7% of detected VMM failures result in unsuccessful recoveries. Considering only injections into AppVM1, the corresponding number is almost the same — 5.2%. Hence, with the weighted distribution of injections, most of the injections are applied to a CPU whose behavior with respect to injection closely matches the behavior of the overall system under uniform distribution of injections. Since results from the two injection distributions are very similar, the rest of the analysis in this section is based on the uniform distribution results.

As explained in Section 2, VMM recovery that leads to the failure of only a single AppVM is considered successful. However, it is clearly preferable for none of the AppVMs to fail. In our experiments, a single AppVM fails to correctly execute its application in 32.6% of successful recoveries. The vast majority of such cases are due to the failure of netbench in AppVM0. The failure of netbench is caused by blocked network interrupts at the I/O controller after recovery, preventing netbench from receiving additional packets from the sender host. Unfortunately, the *acknowledge interrupts* mechanism discussed in Section 4 cannot be used to solve this problem. To acknowledge an interrupt, the CPU must be currently servicing that interrupt. However, a VMM failure can occur after an I/O controller delivers an interrupt to the CPU but before the CPU begins servicing the interrupt. Thus, upon recovery, the CPU cannot acknowledge the interrupt, leaving the interrupt blocked at the I/O controller. To resolve this problem, there needs to be a way to clear pending interrupt states in the I/O controller without performing a full hardware reset. Based on code in the Linux kernel and Xen, there is a way to do this (simulating an interrupt acknowledgment by setting the interrupt trigger mode to edge then back to level). However, this has not worked with our experimental setup, where the entire target system is running in an FV VM. We plan to further investigate this approach in future work.

ReHype failed to recover the VMM in only 5.7% of detected failures. Approximately 50% of failed recoveries are caused by three problems: (1) failure of the PrivVM or DVM, preventing successful completion of applications in more than one AppVM, (2) a triple fault exception generated during the execution of the failure handler triggering a hardware system reset, and (3) a combination of problems causing the failure of both the unixbench and netbench to complete successfully. The following paragraphs discuss these three problems in more detail.

Failures of the PrivVM and DVM in problem (1) are due to kernel panics in the VMs caused by state corruption and error return values from hypercalls. Problem (1) can be partly resolved by providing mechanisms to recover a PrivVM or DVM from failures. DVM recovery has been previously explored in non-VMM failure context [5, 15, 11]. A simple DVM recovery scheme would include destroying the failed DVM, booting a new DVM in its place, and restoring device access to the AppVMs. PrivVM recovery can potentially be done in a similar way. However, unlike DVM recovery, recovering the PrivVM would require preserving configurations of running VMs to allow the PrivVM the ability to continue managing the VMs after recovery.

A double fault exception is generated if a fault is triggered while trying to invoke an exception handler. A triple fault exception is generated if a fault is triggered while trying to invoke the double fault handler. Problem (2) above prevents the failure handler from completing because a triple fault exception is generated. Normally, this causes the hardware to perform a system reset. With our setup, the VMM hosting our target system in an FV VM performs an FV VM reset. This problem may be caused by corruption of VMM state while executing the failure handler. Corruptions can affect the interrupt descriptor table or the page directory, which can lead to a triple fault exception. There is evidence that the frequency of this problem could be reduced by simplifying the failure handler, possibly including the elimination of output of debugging information.

Problem (3) is caused by two independent problems that prevented AppVM0 (netbench) and AppVM1 (unixbench) from finishing their respective applications. The first problem is caused by the blocking of network interrupts at the I/O controller after recovery, preventing the netbench from continuing correctly. This is the same problem described above with respect to successful recoveries resulting in one failed AppVM. The second problem is caused by a panic in AppVM1's kernel after receiving an error return value from a hypercall, preventing the unixbench from completing correctly. The error return value may be caused by inconsistencies within the VMM, and as discussed in Subsection 2.3, may require maintaining redundant information during normal operations to resolve. Fixing either problem should improve overall successful recoveries.

The remaining recovery failures are caused by various VMM corruptions and inconsistencies. Some of the causes of these failures include: (1) corruption of VM's VCPU registers causing the new VMM to crash after recovery when restoring the VCPU, (2) corruption of the timer heap which leads to a page fault in the VMM when the old timer heap is walked to restore timer events, and (3) page table corruption that causes the new VMM to page fault early in the boot code. These problems require mechanisms to check and ensure these data are ''safe'' to use as discussed in Subsection 2.2. Checks are needed to ensure that critical VCPU register values are consistent, e.g. code segment contains the correct privilege, pointers in the timer heap are valid, and page tables contain well formed entries before using or restoring them.

VMM corruptions can lead to failures that are not detectable by simple crash and hang detectors. These silent failures can manifest as failures of one or more VMs and/or failure of the VMM to host additional VMs. As discussed in Section 2, the reliability goal of ReHype is met if no more than one AppVM fails due to a fault and the VMM can still host existing VMs and create new VMs. In the case of a silent single AppVM failure, the reliability goal is met. However, silent system failures, which are silent VMM failures that result in more than one VM failure and/or the failure of the VMM to host additional VMs, reduce the reliability of the virtualized system. Hence, the rest of this section discusses the causes of these failures.

In our experiments, silent VMM failures are roughly 20% of all manifested failures. However, only 14% cause system failures (Table 5). The remaining 6% cause a benchmark (netbench or unixbench) in a single AppVM to complete incorrectly. This can be caused by a failed hypercall causing a VM kernel panic or a blocked interrupt. Roughly 60% of silent system failures are caused by a hardware system reset due to a triple fault exception. Unlike the triple fault exceptions discussed above that occurred during the execution of the failure handler, in these cases, there are no clear indications whether the failure handler ever executed. Simplifying the failure handler, as described above, should allow for a better understanding of this problem.

35% of silent system failures may be artifacts of the fault injection setup. Specifically, in 20% of failures the host VMM crashes the FV VM running the target system. This can happen if the VMM attempts to access invalid state in the FV VM while performing some operations on its behalf. For example, as part of handling paging mode updates (writes to CR4 register) from an FV VM using hardware-assisted paging, the VMM may map in the page pointed to by the FV VM's CR3 register. If the mapping fails (no valid page) due to a corrupted CR3, the VMM will crash the FV VM. With ReHype running directly on hardware, such a scenario would likely result in a detected VMM failures, allowing recovery to be attempted. 15% of silent system failures are caused by communication failures (dropped event signals) between the fault injection campaign on the host VMM and campaign coordination code in the target system. In these cases, the host campaign times out and records a target system failure when it fails to receive a signal from the target system after an injection. In an actual deployment of ReHype, the same fault may not be manifested or may be manifested in a different way, possibly allowing recovery to be attempted.

The remaining 5% of silent system failures are cases in which the kernel in the PrivVM or DVM crashed because of memory corruption or hypercalls returning errors. This caused more than one AppVM to fail completing its benchmark. Mechanisms to recover the PrivVM and DVM should reduce this type silent system failures.

7. Related Work

Many researchers have proposed rebooting subcomponents in application software systems, operating systems, and virtualized systems to increase system reliability and availability [3, 21, 11, 15]. These works, however, have not addressed how to preserve the subcomponents while rebooting the underlying system. The two works that are most closely related to ReHype are RootHammer [13] and Otherworld [4].

RootHammer reduces the time to reboot (rejuvenate) a virtualized system by rebooting the Xen VMM and Domain 0 while preserving in memory the states of VMs and their configurations. During rejuvenation, Domain 0 is properly shut down and the VMs suspend themselves cleanly. Kexec [18] is used to quickly reboot the Xen VMM and Domain 0, similar to ReHype. After a reboot, Domain 0 must re-instantiate and resume all the VMs. This requires modifications to tools in Domain 0 to access VM configurations and state already resident in memory.

RootHammer operates within a healthy and functioning virtualized system. Hence, there is no concern for the safety of the VMM due to corrupted VM states during VM re-integration or the need to resolve inconsistencies, such as acquired locks and interrupted hypercalls. On the other hand, ReHype aims to recover a failed VMM that can be corrupted and may have inconsistencies within the VMM state, between the VMM and VMs, and between the VMM and hardware.

Unlike RootHammer, ReHype preserves Domain 0 and management structures for VMs across a VMM failure. As discussed in Subsection 2.2, this can be unsafe as states can become corrupted. However, preserving Domain 0 allows recovery to occur without any modifications to the VMs as is needed with RootHammer. In addition, without tying Domain 0 recovery to the VMM recovery, recovery latency can be reduced as VMs can continue to operate as soon as the VMM is booted without having to also wait for Domain 0 to boot. A possible extension of ReHype is to follow the microreboot of the VMM with a subsequent proactive rejuvenation, scheduled at a convenient time, involving recovery of Domain 0 and re-creation of the VM structures in the VMM.

Otherworld [4] allows a Linux kernel to be recovered from failures while preserving the state of the running processes. KDump [6] is used to load and boot a new kernel. The new kernel boots within a reserved memory space. Hence, the memory contents of the failed system are preserved. In ReHype, the VMM is booted with access to the entire system memory and does not need a large preserved memory region (64MB used in Otherworld). With Otherworld, processes are restored by recreating the process descriptors and copying the process memory from the old memory region. ReHype reuses the VM descriptors and does not need to copy the VM memory. Both approaches require mechanisms to ensure the safety of the reused data (see Section 2.2).

With Otherworld, restoration of kernel components requires traversing many complex data structures in a possibly corrupted kernel. This increases the chance of failed recoveries. ReHype benefits from the simplicity of the state that the VMM keeps for the VMs, enabling a simpler recovery process and increasing the chance of a successful recovery. Specifically, the number of components that must be restored in a VMM for each VM is small, as discussed in Section 3.

Otherworld must individually restore kernel resources that are used by the processes, such as open files, signal handlers, and shared memory IPC. The network stack and pipes cannot yet be restored. Applications that use such kernel resources need to have custom crash procedures to perform application specific recovery tasks, such as re-opening sockets or restarting the application. With ReHype, all the states of the applications are maintained within the VM. Hence, application failure handlers or any other application modifications are not needed. VM failure handlers could be useful for performing data integrity checks in the VM using VM-specific knowledge. Since there are fewer types of VMs than there are applications, if VM failure handlers are needed, fewer have to be written.

8. Conclusions and Future Work

We have developed the ReHype mechanism that recovers from hypervisor failure, using microreboot, while preserving the state of running VMs. The basic version of ReHype recovered successfully from only 5.6% of detected hypervisor failures. We used fault injection results to guide incremental improvements of ReHype, leading to a success rate of over 90%. The incremental improvements involved a combination of mechanisms to repair VMM corruption and resolve inconsistencies within the VMM, between the VMM and VMs, and between the VMM and the hardware. Our results indicate that almost half of the remaining failed recoveries (3% of detected failures) may be resolved by performing PrivVM or DVM recovery, simplifying the failure handler, and clearing pending interrupts in the I/O controller. 14% of manifested faults lead to undetected VMM failures that result in system failures. 60% of these failures are caused by a single problem — triple fault exception leading to a system reset.

In future work, we will add PrivVM and DVM recovery to enhance overall system reliability. We also plan to evaluate ReHype on bare hardware to check whether any of our results are significantly skewed by our current experimental setup, where the target system is in an FV VM. Additional areas of interest are: evaluation and optimization of recovery latency, preserving FV VMs across a VMM microreboot, and additional stressing of ReHype using, for example, injected software errors.

Acknowledgments

This work was supported, in part, by a donation from the Xerox Foundation University Affairs Committee.

References

[1] ‘‘UnixBench,’’ *www.tux.org/pub/tux/benchmarks/System/unixbench*.

[2] 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,’’ *Nineteenth ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, pp. 164-177 (October 2003).

[3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, ‘‘Microreboot - A Technique for Cheap Recovery,’’ *6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, pp. 31-44 (December 2004).

[4] A. Depoutovitch and M. Stumm, ‘‘Otherworld - Giving Applications a Chance to Survive OS Kernel Crashes,’’ *5th ACM European Conference on Computer Systems*, Paris, France, pp. 181-194 (April 2010).

[5] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, ‘‘Safe Hardware Access with the Xen Virtual Machine Monitor,’’ *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS) (ASPLOS)* (October 2004).

[6] V. Goyal, E. Biederman, and H. Nellitheertha, ‘‘Kdump, A Kexec-based Kernel Crash Dumping Mechanism,’’ *lse.sourceforge.net/kdump/documentation/ols2oo5-kdump-paper.pdf* (2005).

[7] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, ‘‘Characterization of Linux Kernel Behavior Under Errors,’’ *International Conference on Dependable Systems and Networks*, San Francisco, CA, pp. 459-468 (June 2003).

[8] W. Gu, Z. Kalbarczyk, and R. K. Iyer, ‘‘Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors,’’ *International Conference on Dependable Systems and Networks*, Florence, Italy, pp. 887-896 (June 2004).

[9] I. Hsu, A. Gallagher, M. Le, and Y. Tamir, ‘‘Using Virtualization to Validate Fault-Tolerant Distributed Systems,’’ *International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, CA, pp. 210-217 (November 2010).

[10] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer’s Manual: Volume 3A*, 2010.

[11] 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* **59**(11), pp. 1466-1479 (November 2010).

[12] K. Kourai and S. Chiba, ‘‘A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines,’’ *International Conference on Dependable Systems and Networks*, Edinburgh, UK, pp. 245-255 (June 2007).

[13] K. Kourai and S. Chiba, ‘‘Fast Software Rejuvenation of Virtual Machine Monitors,’’ *IEEE Transactions on Dependable and Secure Computing* (May 2010).

[14] M. Le, A. Gallagher, and Y. Tamir, ‘‘Challenges and Opportunities with Fault Injection in Virtualized Systems,’’ *First International Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, Austin, TX (April 2008).

[15] M. Le, A. Gallagher, Y. Tamir, and Y. Turner, ‘‘Maintaining Network QoS Across NIC Device Driver Failures Using Virtualization,’’ *8th IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, pp. 195-202 (July 2009).

[16] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, ‘‘Diagnosing Performance Overheads in the Xen Virtual Machine Environment,’’ *First ACM/USENIX Conference on Virtual Execution Environments*, Chicago, IL, pp. 13-23 (June 2005).

[17] W. T. Ng and P. M. Chen, ‘‘The Systematic Improvement of Fault Tolerance in the Rio File Cache,’’ *29th Fault Tolerant Computing Symposium*, Madison, WI, USA, pp. 76-83 (June 1999).

[18] A. Pfiffer, ‘‘Reducing System Reboot Time With kexec,’’ *devresources.linuxfoundation.org/andyp/kexec/whitepaper/kexec.pdf* (April 2003).

[19] M. Rosenblum and T. Garfinkel, ‘‘Virtual Machine Monitors: Current Technology and Future Trends,’’ *IEEE Computer* **38**(5), pp. 39-47 (May 2005).

[20] M. Sullivan and R. Chillarege, ‘‘Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems,’’ *21st Fault-Tolerant Computing Symposium*, Montreal, Quebec, Canada, pp. 2-9 (June 1991).

[21] M. M. Swift, B. N. Bershad, and H. M. Levy, ‘‘Improving the Reliability of Commodity Operating Systems,’’ *ACM Transactions on Computer Systems* **23**(1), pp. 77-110 (February 2005).