IEEE International Conference on Software Security and Reliability
Washington, D.C., June 2012.

Applying Microreboot to System Software

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

***Abstract*—Availability is increased with recovery based on component microreboot instead of whole system reboot. There are unique challenges that must be overcome in order to apply microreboot to low-level system software. These challenges arise from the need to interact with immutable hardware components on one hand and, on the other hand, with a wide variety of higher level workloads whose characteristics may be unknown. As an example, we describe our experience with applying microreboot to system-level virtualization software. Specifically, implementing microreboot for all the components of the widely-used Xen virtualization infrastructure. We identify the unique difficulties with applying microreboot for such low-level software and present our solutions. We present measures of the complexity of different classes of solutions and experimental results, based on extensive fault injection, showing the effectiveness of the solutions.**

Keywords– Recovery; Virtualization; Fault injection

1. Introduction

Microreboot enables fast restoration of a valid state of a failed process or system by rebooting the specific failed component instead of the entire system [5]. *Low-level system software* (LLSS) manages and controls hardware resources and mediates access to hardware resources by higher software layers. While in previous work microreboot has been used mostly for user-level processes, the technique is also applicable to system software. However, as explained below, due to key characteristics of system software, using microreboot to recover from failures in such software is particularly challenging.

In general, it is difficult to apply microreboot to software with tightly-coupled components and non-modular organization. Unfortunately, LLSS tends to possess these characteristics. LLSS has additional properties that makes it less amenable than application software to the microreboot technique. One such property is that LLSS interacts directly with hardware resources. The hardware cannot be modified and the interfaces to the hardware may not provide well-defined semantics in the face of an LLSS component reboot. Hardware resources may be shared by multiple LLSS components, potentially coupling these components to each other, making it difficult to microreboot a single component.

A second property that complicates the use of microreboot for LLSS is that the software that runs on top of the LLSS (application software) is typically not under the control of the LLSS developer. This makes it difficult to microreboot LLSS components while allowing the application software, that interacts with the LLSS, to eventually continue operating normally. Finally, microreboot is simpler if a component with greater privilege can manage the microreboot process. With LLSS there may not be any software component in the system with greater privilege. Hence, the LLSS must somehow microreboot itself while in a potentially corrupted state.

Key contributions of this paper include identifying unique challenges to applying microreboot to low-level system software and presenting general approaches to addressing these challenges. This is based on our experience using these approaches with complex low-level software — software that provides system-level virtualization. Specifically, we have applied microreboot to all the components of the Xen [1] virtualization software. As explained in Section 4, this software consists of three components: the privileged virtual machine (PrivVM), the device driver virtual machine (DVM), and the virtual machine monitor (VMM). We denote all three of these components as the *virtualization infrastructure* (VI).

We describe how we have employed the approaches to using microreboot with LLSS with all three components of the VI. In each case, we explain the specific difficulties and present experimental results that show the effectiveness of the techniques. While the use of microreboot to recover from failures of two of the VI components has been presented in prior work [13, 10, 14, 15], recovery from failures of the PrivVM is presented here for the first time. Hence, the description and evaluation of the specific technique used for PrivVM recovery, based on microreboot, is another key contribution of this work.

The next section is an overview of microreboot. Section 3 describes the challenges with the use of microreboot for LLSS and general approaches to addressing these challenges. The Xen VI is described in Section 4. Section 5 explains how we have employed microreboot for recovery from failures of each one of the Xen VI components. The experimental setup used to evaluate the recovery mechanism is described in Section 6. The results of the evaluation are in Section 7. Section 8 provides measures of the implementation complexity of our recovery schemes and related work is presented in Section 9.

2. Microreboot Overview

When system components fail, a simple way to recover is to reboot the entire system. This is slow and reduces system availability. Microreboot is a recovery technique that

reduces the down time caused by failure of some components in the system by recovering (rebooting) only the failed components in the system while allowing other functioning components to continue operating normally.

As explained in [5], there are three main design goals for microrebootable software: (1) fast and correct component recovery, (2) localized recovery to minimize impact on other parts of the system, and (3) fast and correct reintegration of recovered components. To achieve these goals, software components need to be loosely coupled, have states that are kept in dedicated state store separate from program logic, and be able to hide component failures by providing mechanisms for retrying failed requests. To decrease recovery latency, the software components should be fine-grained to minimize the restart and reinitialization time. Lastly, to facilitate cleaning up after microreboots and prevent resource leaks, system resources should be leased and requests should carry time-to-live values.

3. Microrebooting Low-level System Software

There are unique challenges to employing microrebooting with low-level system software. In the rest of this section, these challenges are described and general approaches to addressing these challenges are presented.

Immutable shared hardware: LLSS interacts directly with hardware that may not provide interfaces with the properties required for ‘‘clean’’ microreboot and may cause undesirable coupling among different LLSS components that interact with the hardware. The coupling among LLSS components may force multiple components to be microrebooted together, thus violating the key goal of the microreboot technique and increasing the recovery latency of the system.

An LLSS component failure can lead to corruption of hardware state by providing the hardware with faulty inputs or failing to respond properly to inputs from the hardware. The only way to restore the hardware component (e.g., a device controller, an interrupt controller, or a bus) to a sane state may be to reset it. The hardware component that needs to be reset may be coupled to other hardware components and/or shared by multiple LLSS components. Hence, the reset of the hardware component may initiate a ‘‘domino effect’’ that forces the reset and reboot of multiple hardware and LLSS components. Furthermore, the reset of some hardware components may be inherently slow [13].

When an LLSS component, such as an OS kernel, is rebooted, it often goes through a process of probing the hardware in order to identify its characteristics. For some hardware components, such probing can change the state of the device, which can lead to corruption and inconsistencies if the device is being used by other components in the system.

The hardware cannot be changed in order to accommodate LLSS microreboot. Hence, microreboot for LLSS components may require implementing work-arounds to deal with specific problematic properties of specific hardware components. This may involve identifying ways to perform partial resets of the hardware, even if that may not always be sufficient [13]. It may lead to special operations performed during microreboot which, while not an actual hardware reset, can eliminate specific known ‘‘problem states’’ in the hardware. An example of this is to acknowledge any pending interrupts, waiting for a response from the rebooted LLSS component, that would otherwise block future interrupts. Finally, in some cases, the considerations above may force a recovery technique where system functionality is restored before the microreboot of the failed LLSS component and/or the reset of some particular hardware component. This can be done by fail-over mechanisms that use redundant software and hardware resources while the failed components are restored.

Workload transparency: Typically, many different applications and user-level subsystems (workloads) run on top of the LLSS and interact with LLSS components. The software that runs on top of the LLSS may not be known at the time the LLSS is developed and/or may not be under the control of the LLSS developer. Hence, ideally, the microreboot of LLSS components should be transparent to the layers above it.

A key challenge to meeting the transparency goal described above is the possibility of requests from the workload that are in progress in the LLSS when an LLSS component fails. Since the workload is typically not designed to interact with components that may be rebooted, it cannot be expected to retry such in-progress requests once the microreboot is performed. Hence, an LLSS recovery mechanism that employs microreboot must have some way to log, in a safe location, information that allows in progress requests to be retried and completed following microreboot. Ideally, this is done strictly on the LLSS side and must be considered part of implementing the microreboot capability.

“Last” software layer: It is typically simplest for microreboot of a component to be performed by a component that has higher privilege and thus has access to all the resources required for the rebooted component. In many cases, LLSS components interact directly with the hardware and have the highest privilege. In such cases, the challenge is that, once the LLSS component fails, there is no higher privilege component that can manage the microreboot process. Furthermore, since the failed component has the highest privilege, no part of the system can be considered safe from corruption by the failed component.

The only viable approach to dealing with the above challenge is to have an equal privilege LLSS component, a failure handler, that is invoked when an error is detected.

Since the failure handler is of equal privilege, the handler itself may be corrupted by the failed component and/or recovery may rely on the reuse of potentially corrupted state. Redundant data structures can be used to minimize the probability of corrupted state reuse. However, this possibility cannot be eliminated.

System time management: System time is maintained by the system software. Microreboot of an LLSS component may lead to erratic changes in system time. For example, system time may cease to advance for some duration and/or may advance suddenly by a significant amount when an LLSS component microreboot completes. Applications running on the system may rely on time that is monotonically increasing at a constant rate. For example, this may be the case for applications that use timer events to trigger some actions. In addition, the system may be interacting with the outside world that is expecting time on the system to remain approximately synchronized with real time.

With respect to the workload running on top of the LLSS, the problem above can be partially mitigated by ensuring that the workload is not allowed to execute (is scheduled out) if time is not advancing while an LLSS component is being rebooted. The minimum disruption to the workload can then be achieved by restoring time to the value just before the reboot (just before the workload is paused) when the workload is finally allowed to resume execution. To minimize problems related to interactions with the outside world, time must then be slowly accelerated until it catches up with real time.

Imperfect recovery: None of the approaches described above for dealing with the challenges of employing microreboot for LLSS guarantee that recovery will be successful. LLSS with high privilege may corrupt all software layers above it and may corrupt the state of the hardware to the point of requiring power cycling the entire system. The probability of such undesirable behavior can be reduced by performing extra work during normal operation to provide self-checking capabilities and log redundant data that can be used during recovery. The microreboot process itself can be made safer by performing extra checks on state that is reused as well as on hardware components. The challenge is to determine the costs and benefits of such overhead and know when the mechanism is sufficient to meet system requirements.

An approach to meeting the above challenge is to rely on incremental refinement of the recovery mechanism based on experimental evaluation [19]. Specifically, the first step is to implement a minimal recovery mechanism, that can deal with "well-behaved" fail-stop failures. Fault injection campaigns are then used to evaluate the recovery success rate under realistic conditions. The results are used to determine the most important cause of recovery failures. This cause is a deficiency in the recovery mechanism that is then eliminated or mitigated. The process is then repeated until the desired success rate is achieved.

4. System-level Microreboot Example: Xen

As will be discussed in Section 5, we have applied the techniques discussed in Section 3 to employ microreboot for recovery from failures in the Xen [1] *virtualization infrastructure* (VI). To facilitate understanding of this work, this section is a brief review of key features of Xen.

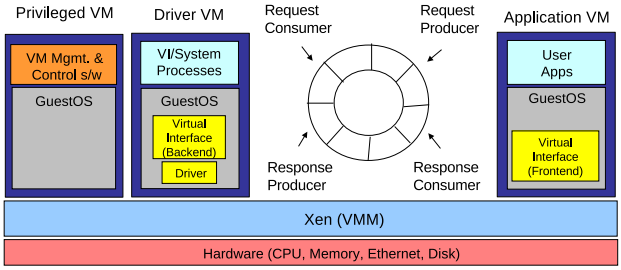

Figure 1: Virtualization infrastructure and the split device driver architecture.

System-level virtualization allows multiple VMs, each with its own OS, to run on a single physical computer [20]. The *virtualization infrastructure* (VI) consists of all the software components involved in multiplexing hardware resources among VMs. The Xen VI is composed of three components: virtual machine monitor (VMM), driver VM (DVM), and privileged VM (PrivVM). We refer to VMs that are not part of the VI as *application VMs* (AppVMs).

A common VI organization for allowing multiple VMs to share I/O devices is called the split device driver architecture [7, 16, 18]. With this organization, a *frontend driver* resides in each VM sharing a device. As shown in Figure 1, the actual device driver together with a *backend driver* reside in a VM that is referred to as the driver VM (DVM). In each AppVM, I/O requests are forwarded by the frontend driver to the backend driver, which invokes the actual device driver. In Xen [1], the frontend and backend drivers communicate through a ring data structure in an area of memory shared between the AppVM and DVM.

The privileged VM (PrivVM) is used to perform system management operations such as creating, destroying, and checkpointing VMs. The VMM does not permit these operations to be invoked by any other VMs.

The functionality of the PrivVM is provided by a combination of kernel modules and user-level processes running in the PrivVM. One user-level process, XenStored, provides access to a dynamic database of system configuration information, called XenStore. XenStored also provides mechanisms for VMs to be informed of changes to certain configuration states by allowing VMs to register *watches* on those states in the XenStore. A VM communicates with the XenStore through XenStored using

a shared ring data structure, similar to the communication mechanism between a DVM and AppVM.

5. Employing Microreboot for Xen VI Recovery

As originally implemented, failures in any of the Xen VI components results in the failure of the entire virtualized system, requiring full system reboot that involves restarting all the AppVMs. We have employed microreboot to implement mechanisms that allow the virtualized system to recover from most failures of any of the VI components, without requiring the AppVMs to be restarted. Following recovery, the fault tolerance capability of the system is restored so that it maintains its ability to recover from future VI component failures. The rest of this section describes the configuration of the virtualized system that makes it amenable to fault tolerance. The mechanisms used for detecting failures of the VI components are explained. A definition of what is meant by ''successful recovery'' is then provided. Three successive subsections present how microreboot is used to recover from failures of each of the VI components. A final subsection relates the specific difficulties with implementing microreboot for the Xen VI to the challenges discussed in Section 3.

The PrivVM manages the virtualized system [1] and, as described later, also plays important roles in most of the fault tolerance mechanisms that we have developed. Hence, it is desirable to minimize the probability of PrivVM failure and minimize the complexity of recovery when the PrivVM does fail. In a typical Xen system, the PrivVM is used to access all hardware devices in the system [1]. However, such a configuration exposes the PrivVM to possibly buggy device drivers [21] or faulty device controllers, thus increasing the probability of PrivVM failure.

Due to the considerations explained above, we use a configuration that minimizes the interactions of the PrivVM with device controllers. Specifically, as explained in Section 4, access to most devices is through a separate driver VM (DVM). The PrivVM's root file system is in memory so that the PrivVM does not have access to any storage devices. The PrivVM has access to a network interface card (NIC) in order to enable remote management of the virtualized system. In addition, the PrivVM controls the PCI bus configuration space and access to the PCI configuration space from any VM is performed through the PrivVM. Furthermore, the PrivVM hosts the virtual serial console device.

A prerequisite for recovery is the detection of component failures. Crashes of the DVM or VMM are easily detected since they cause the invocation of *panic* handlers [13, 14]. DVM and VMM hangs are detected by mechanisms implemented in the VMM that identify when these components fail to perform expected operations over a given period of time [13, 14]. Similar mechanisms are used to detect crashes or hangs of the PrivVM's kernel. As discussed in Section 4, the PrivVM hosts user-level processes that are essential to the correct operations of the VI. To detect the failure of these processes, we deploy a user-level monitoring process, called *hostmon*, that periodically checks for the existence of these processes. When hostmon detects the disappearance of one of these processes, it invokes the VMM to crash the PrivVM and trigger full PrivVM recovery.

We use the rate of *successful recovery* from VI component failures to quantify the effectiveness of our resiliency mechanisms. Virtualization is commonly used to consolidate the workloads of multiple physical systems on a single physical host. With multiple physical systems, a single fault may cause the failure of one of the systems. Due to physical isolation, other systems are not directly affected. Since a virtualized system does not have the benefits such physical isolation, an aggressive reliability goal for VI resiliency mechanisms is to do no worse than a cluster of physical systems. Hence, we define recovery from a VI component failure to be *successful* as long as it does not lead to the failure of more than one AppVM and the recovered VI is able to continue hosting the remaining VMs as well as create and host new VMs [14, 15].

We have implemented recovery mechanisms, based on microreboot, for all the VI components of Xen version 3.3.0. As discussed in Section 3, we have used the incremental refinement approach to optimize the recovery scheme for each VI component. Mechanisms for microrebooting the VMM [14] and DVM [13, 15] are only briefly reviewed since they have already been presented elsewhere. The mechanism for recovering a failed PrivVM has not been described elsewhere and is thus presented in more detail.

5.1. Microrebooting the VMM

ReHype [14] microreboots the VMM while preserving the states of all running VMs in memory. Failure detectors in the VMM initiate the VMM microreboot. The failure handler in the VMM controls the microreboot process: stopping all but one CPU from running, preserving critical VMM state, and refreshing parts of the VMM memory with a pristine VMM image stored elsewhere in memory. The CPUs are instructed to halt by the handler of a special non-maskable interrupt sent from the CPU handling the failure. Before refreshing VMM memory, VMM state in the static data segments is preserved since it contains data that is critical for resuming execution of the existing VMs. ReHype reserves a space in the uninitialized (bss) static data segment to which this state is copied and from which it is later restored. The overwritten (refreshed) VMM memory includes VMM code, the initialized static data segment, and the non-reserved area of the bss.

To prevent the loss of VM states across a VMM microreboot, the memory of the VMs, which is allocated on

the VMM's heap, must be preserved. The memory states of VMs can be very large. Hence, to minimize recovery time, the VMM's heap is preserved in place. The VMM's boot code has been modified to restore critical VM management data structures, saved in the reserved space in the bss and the preserved heap, and to avoid marking as free pages that were allocated by the old VMM instance.

Since the VMM can fail at any time, inconsistencies can occur between the new VMM instance and the rest of the preserved system. The basic microreboot scheme discussed above is not capable of resolving most of these inconsistencies, resulting in a low successful recovery rate (5.6% of detected VMM failures). We incrementally improved the basic recovery scheme and were able to obtain a successful recovery rate of over 94% [14] of detected VMM failures. Some of these improvements involved resolving inconsistencies that can arise from partially executed hypercalls, acquired locks in the preserved structures, and unacknowledged hardware interrupts.

5.2. Microrebooting the DVM

When a DVM fails, applications accessing I/O through that DVM are blocked. If the DVM is microrebooted and hardware devices are reset, the duration of the interruption may be on the order of seconds [13]. Such long interruptions can result in the failure of the workload running in the AppVMs. Therefore, unlike other VI components, we do not rely on microreboot to recover from DVM failure. Instead, recovery from a DVM failure involves failing over to a redundant DVM with access to separate hardware devices [15]. However, microreboot must still be used to replace the failed DVM so that the fault tolerance capabilities of the system are restored.

The PrivVM controls the process of microrebooting the DVM, which includes: pausing the failed DVM, booting a new DVM instance, destroying the failed DVM, and integrating the new DVM with existing VMs on the system. The destruction of the failed DVM and subsequent releasing of all its memory to the VMM must be done after all the devices that the failed DVM owns are re-initialized by the newly booted DVM. This is to prevent ongoing DMA operations initiated by the failed DVM from corrupting memory that has been released to the VMM. The new DVM instance is re-integrated with existing VMs by reforming the respective frontend-backend connections. This is done transparently to the applications in the AppVM by extending existing mechanisms in the frontend drivers responsible for resuming and suspending devices [13].

5.3. Microrebooting the PrivVM

As described above, microreboot of a DVM is controlled by the PrivVM. The PrivVM is, obviously, not operational during PrivVM microreboot. Without a functional PrivVM, only the VMM has the privileges required to replace a failed PrivVM. Hence, the VMM is responsible for releasing all the resources of the failed PrivVM and booting the new PrivVM instance. Since the PrivVM kernel and root file system may be corrupted during PrivVM failure, pristine PrivVM kernel and filesystem images must be used for the new PrivVM. The required pristine images are stored, compressed, in the VMM address space, consuming approximately 128MB.

A key requirement for microrebooting the PrivVM is to restore state in the PrivVM needed for managing and controlling the system. This state includes the XenStore, stored as a file in the PrivVM, and *watches* in the XenStored process. Since all the PrivVM state, including the file system, is in memory, failure of the PrivVM results in the complete loss of its state. Hence, to preserve the critical PrivVM state, the XenStore and XenStored states are replicated. To survive PrivVM failure, the replicated states must be stored in a different VI component. While there are several alternatives, the simplest choice is to maintain the replicated state in one of the DVMs. This DVM is referred to as DVM_XS.

The backup copy of the critical PrivVM state is maintained in the DVM_XS by a user-level process — the XenStore Backup Agent (XBA). The XBA on the DVM_XS communicates with XenStored on the PrivVM over shared rings, similar to the connection between AppVMs and a DVM. XenStore write requests, watch registrations, and requests to start or end XenStore transactions are forwarded by XenStored to the XBA before performing the operations in the PrivVM. The XBA performs all operations on a local copy of the XenStore located on the filesystem of the DVM_XS. After a microreboot, the new PrivVM acquires up-to-date XenStore and XenStored states from the XBA. If the DVM_XS fails, the XenStore and XenStored states are transmitted from the PrivVM to the XBA on the newly recovered DVM_XS.

During PrivVM microreboot, the frontend-backend connection with the XBA must be established before the PrivVM can obtain the XenStore and XenStored states from the XBA. Establishing this connection requires the PrivVM to have information such as the frame number of the shared page being used by the connection (Section 4). Hence, the VMM has been modified (new hypercalls) to allow the PrivVM to store this information in VMM memory when the PrivVM is first booted and retrieve the information when a new instance is booted during PrivVM microreboot.

During PrivVM microreboot, once the XenStore and XenStored states have been restored, connections involving the PrivVM are re-established, connecting frontends in the various VMs to the XenStored, virtual serial console, and PCI backends in the PrivVM. Re-establishment of PCI frontend-backend connections required a small extension to the PCI frontend driver in all the VMs. This extension

incorporates the abilities to disconnect/connect from/to the PCI backend driver using the existing suspend/resume functionalities of the split device driver mechanism.

Table 1. Fault injection into the PrivVM. Percentage of successful recoveries out of detected PrivVM failures.

Mechanism	PrivVM State	Successful Recovery Rate
Basic	Idle	92.16%
Basic	Active	35.71%
+ T_VMCreate	Active	86.42%
+ T_Requests	Active	96.27%

The mechanisms presented up to this point provide basic capabilities for PrivVM microreboot. Table 1 presents the results from fault injection into registers while the PrivVM is executing (Section 6). As shown in the top row, when the PrivVM is idle, the rate of successful recoveries out of all detected faults is greater than 92%.

If faults occur while the PrivVM is active, recovery is more difficult. As shown in the second row of Table 1, if faults are injected while the PrivVM is in the process of creating a new VM, recovery rate plummets to below 36%. Most failures (> 65%) occur because, during recovery, the new XenStored process fails while trying to acquire information from the restored XenStore about the existing VMs on the system. Since the PrivVM fails in the middle of creating an AppVM, only a subset of the expected entries for the new AppVM are present in the XenStore backup. Hence, XenStored in the new PrivVM instance fails while attempting to process incomplete (invalid) XenStore state.

T_VMCreate: To avoid the above problem, VM management operations, such as VM creation and destruction, including associated changes to XenStore, have to be atomic. Transactionalizing these operations requires maintaining a log that tracks the individual steps of each operation. This allows the recovery mechanism to determine how far along the operation progressed before failure and, if necessary, how to undo partially completed operations. With this information, even across PrivVM failures, VM management operations are either executed to completion or aborted, leaving the VI in a consistent state. In the case of a VM create operation, either the VM is created successfully or the VM is destroyed and any information written to the XenStore is removed.

To demonstrate the feasibility and effectiveness of the above approach, we have implemented transactional versions of the VM create and destroy operations. This involved modifying *Xend* — a user-level process in the PrivVM that receives requests for management operations and carries out these requests by interacting with the VMM and XenStored. Xend acknowledges requests *after* performing the requested operation. Our modified Xend uses the XBA and replicated XenStore to create a log that is maintained across PrivVM recovery and is available to the new Xend process.

Creating a VM requires sending two requests to Xend: (1) create the VM and (2) unpause the VM. Our modified Xend creates a log entry before executing the first request and removes the log entry before acknowledging this request. After recovery, if Xend detects a log entry for creating a VM, that indicates that the PrivVM had failed before acknowledging the ‘‘create VM’’ command and thus may have failed before completing all the steps involved in creating a VM. Hence, Xend destroys the VM and cleans up any XenStore entries associated with that VM. With respect to the PrivVM, the processing of the ‘‘unpause VM’’ request is inherently atomic (one hypercall) and thus no modifications are needed to the processing of that request. The processing of the ‘‘VM destroy’’ request is made atomic using the same approach as for the ‘‘VM create’’ request.

With transactional VM creation, the successful recovery rate is above 86% (Table 1). More than 63% of remaining failures are cases in which the AppVM hangs waiting for a response to a XenStore request. Such requests are sent by the AppVM during its boot-up process to set up access to its disks. The AppVM hangs if the request is lost due to PrivVM failure while processing the request.

T_Requests: To overcome the problem of AppVM hangs, when the PrivVM is microrebooted, the recovery mechanism must ensure that pending XenStore requests from other VMs are performed and proper responses are sent to the requesters. Similarly to T_VMCreate, this is done using logging to detect partially processed requests — requests for which responses have not been sent to the requesters.

XenStore requests/responses from/to another VM are placed in circular buffers shared between the PrivVM and the other VM. Shared producer and consumer indices are used to coordinate the use of the shared buffers. Since the shared buffer page is owned by the other VM, the requests/responses and the circular buffer indices remain intact across a PrivVM microreboot (although there is the possibility that they will be corrupted by the failed PrivVM).

As originally implemented, XenStored updates the request consumer index once it reads a request, before processing it. XenStored updates the response producer index after it places a response in the response ring. To help detect pending requests, our modified XenStored updates the request consumer index only after it updates the response producer index. Our modified XenStored also logs in the replicated XenStore the request consumer index before it begins to process a request and logs the response producer index after placing the response in the shared buffer but before updating the shared index. Thus, while XenStored is processing a request, the logged request consumer index equals the shared value. If the response to

an in-progress request has already been made available to the requesting VM, the logged response producer index is *not* equal to the shared value. During PrivVM recovery, the new XenStored uses the values of the logged indices and the indices in the shared buffer to determine where in the request buffer to resume request processing.

With PrivVM recovery, the mechanism described above may result in the re-execution of requests that have been completely executed but for which the response has not been made available to the requesting VM. This is not a problem with idempotent requests, such as XenStore reads and writes. However, error responses may be sent to the requester if there is an attempt to re-execute non-idempotent requests, such as removing entries in the XenStore, starting/ending XenStore transactions, or setting watches. For such non-idempotent requests, *before* request processing begins, XenStored logs to the replicated XenStore relevant information regarding the state of XenStore and XenStored. During PrivVM recovery, the new XenStored uses this logged information to avoid re-executing non-idempotent requests that have already been executed, yet provide proper responses to the requester. With this enhancement, the successful recovery rate for detected PrivVM failures is above 96%.

5.4. System-level Implementation Challenges

This subsection highlights how the general approaches for overcoming challenges of system-level microreboot (Section 3) apply to specific difficulties with microrebooting the Xen VI components. Some of the problems and solutions are not exclusively system-level problems but are discussed in order to provide a complete picture of the issues involved in microrebooting system-level software.

Immutable shared hardware: Microrebooting Xen VI components requires resetting hardware devices with which these components interact. It is not possible to change the fact that some devices cannot be individually reset and some reset operations may be slow, possibly causing applications waiting in the AppVMs to fail (Section 3). This is a problem that affects all three components of the VI. Getting around these problems requires modifications to the software that initializes hardware components.

An example where a modified re-initialization operation is needed is during the microreboot of the PrivVM. Since the PrivVM has access to the PCI subsystem (PCI buses and PCI configuration space), PrivVM boot-up usually includes the probing of PCI devices. This probing involves the reading of the PCI configuration, which requires writing into a control register. The control register may be in use for different purposes during normal system operations. Hence, reading the PCI configuration during PrivVM recovery may have unintended side effects. Avoiding this problem required modifying the PCI driver in the PrivVM kernel. The PCI information gathered during initial PrivVM boot-up is saved in the VMM. A modified PCI probing routine is used during reboot to retrieve the PCI information from the VMM.

When microrebooting the DVM, special re-initialization operations can be used to decrease the device reset time. Specifically, for reducing the time it takes to reset network interface devices, we have experimented with a special network device reset routine that bypasses the link layer negotiation phase [13]. This phase can be time consuming (orders of seconds). The danger of bypassing the link layer negotiation is that the device may be left in a corrupted state. Our solution for dealing with this problem is described in Section 5.2 [15].

Workload transparency: To make the microreboot of VI components transparent to other system components, the handling of in-progress requests from other components must continue across recovery. This requires maintaining additional state in the VI components in order to detect and resume partially-executed operations. For instance, microrebooting the PrivVM requires a mechanism, based on logging, for ensuring completion of in-progress XenStore requests (Section 5.3). Without this mechanism, the AppVMs would have to be modified to retry these XenStore operations themselves.

Partially executed hypercalls are a key problem with recovery from VMM failures [14]. The hypercall mechanism allows code in VMs to send requests to the VMM. If the VMM fails while executing a hypercall, the operation may never complete and the invoking VM may be blocked forever waiting for a response. Hence, VMM microreboot requires a facility to retry in-progress hypercalls. Hypercall retry is 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 (trap) before allowing the VM to run. This mechanism is already used in the Xen [1] VMM to allow the preemption of long running hypercalls transparently to the VMs [14].

Ideally, there should be no need to modify workload software to support microrebooting of system components. However, in some cases, limited workload modifications are justified since they simplify the implementation of microreboot. With our implementation of microreboot for the Xen VI components, all workload modifications are in the AppVM kernels. The applications running in the AppVMs are not modified. For example, small modifications to drivers in the kernels of the AppVMs support reforming frontend-backend connections upon DVM or PrivVM recovery [13]. Without these modifications, the VMM would need to maintain additional state and have mechanisms to reroute requests from AppVMs to the new DVM or PrivVM instance, further

complicating the VMM [13].

"Last" software layer: The VMM is the most privileged layer in the software stack. Thus when the VMM fails, it must microreboot itself. Recovery, however, can fail if the VMM's failure handler uses corrupted data.

The VMM's failure handler must access the stack to prepare the system for a VMM microreboot. To prevent the VMM from accessing a corrupted stack, the VMM failure handler sets the stack pointer to a valid stack location obtained from a fixed location in memory. In addition, to prevent CPUs from blocking interprocessor interrupts (IPI), the CPU detecting the VMM failure uses NMI-based IPIs to signal to other CPUs to initiate failure handling [14].

System time management: When the VMM is microrebooted, system time and all VMs running on the system are momentarily stopped. We have applied the approach for restoring system time after recovery discussed in Section 3. To prevent applications in the VMs from failing, system time is restored to the value right before the reboot. This allows timer events in the VMs to fire in the correct order and mask the microreboot latency. External entities may be exposed to the incorrect time in the recovered system immediately after a VMM microreboot. However, time can be slowly adjusted to the correct value using a time synchronization service (*ntp*).

6. Experimental Setup

This section discusses the experimental setup used to evaluate our microreboot-based recovery mechanisms for the Xen VI. It discusses details of the fault injection campaigns and the workloads stressing the VI components.

We use the UCLA *Gigan* fault injector [12, 9] to inject single bit flip faults into CPU registers (general purpose and the program counter) while the CPUs are executing VI code. This type of injection is used since it causes arbitrary corruptions in the VI components.

Our evaluation uses two workloads: synthetic and LVS. For each workload, three fault injection campaigns are performed, differing in when faults were injected: 1) while the CPU executes VMM code, 2) while the CPU executes DVM code, and 3) while the CPU executes PrivVM code. The time of injection is random, and, for the DVM and PrivVM campaigns, includes both user and kernel level code.

A fault injection campaign consists of many fault injection runs. For the synthetic workload, a ''run'' begins by booting the VMM, PrivVM, two DVMs, and two AppVMs. One AppVM runs a network intensive application while the other runs an OS intensive application. After the benchmarks begin executing in the AppVMs, a single fault is injected into a VI component. To ensure the VI is still operational, a third AppVM is booted after a possible VI component recovery and runs a disk intensive application.

The LVS workload is a deployment of the Linux Virtual Server (LVS [22, 17]) on a clusters of VMs (virtual cluster). LVS is an open-source load-balancing solution for building highly-scalable and highly-available servers using clusters of servers. For the LVS workload, a run begins by booting the VMM, two DVMs, and five AppVMs. Three AppVMs run the Apache web server and two AppVMs act as primary/backup load balancers (directors). The virtual cluster is stressed by running five instances of the Apache *ab* benchmark on a remote client. Each instance performs a series of HTTPS requests for statically and dynamically generated web pages. After the remote client begins to generate requests, a single fault is injected into one of the VI components. One AppVM is randomly selected to be rebooted about 50 seconds after a fault is injected to ensure the VI is still operating correctly.

To simplify the setup for software-implemented fault injection, the entire target system runs inside a fully-virtualized (FV) VM [12]. Among other benefits, this simplifies the restoration of pristine system state before each run, thus isolating the run from previous runs.

Table 2. Injection outcomes.

Outcome	Description
Detected VI component failure	Crash: (1) VMM/DVM/PrivVM kernel panics (2) System process in PrivVM dies Hang: VMM/DVM/PrivVM makes no observable progress
Silent failure	Undetected failure: No detected VI component failures but the workload fails to execute correctly
Non-manifested	No errors observed

Table 2 summarizes the three possible consequences of an injection. Only detected failures lead to recovery attempts. As discussed in Section 5, recovery is considered successful if no more than one AppVM fails and the recovered VI maintains its ability to host the existing AppVMs as well as create and host new AppVMs. All silent failures and failed recoveries of the VI are considered *system failures*.

7. Fault Isolation and Recovery Success Rate

This section presents the results from the fault injection campaigns. Some of the injected faults are manifested as errors — a component of the system deviates from correct operation (component failures). Recovery using microreboot is effective only if single faults do not manifest as errors in multiple components, i.e., there is strong *fault isolation* among system components. Figures 2-4 present the distribution of component failures caused by faults injected during VMM/DVM/PrivVM execution with the synthetic and LVS workloads. The results show that the vast majority of component failures are confined to the component into which faults are injected. This indicates

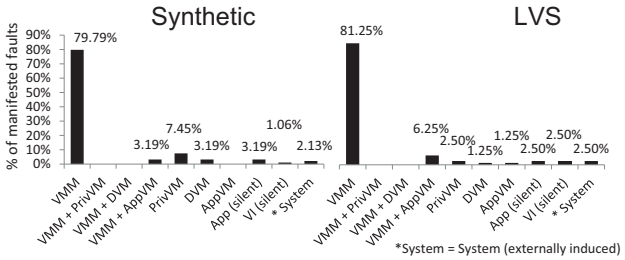

Figure 2: Distribution of component failures caused by injecting faults into CPU registers during VMM execution.

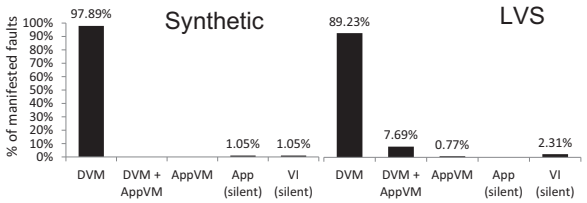

Figure 3: Distribution of component failures caused by injecting faults into CPU registers during DVM execution.

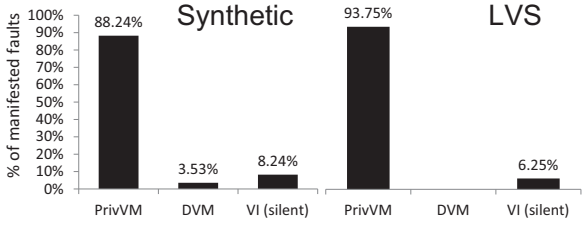

Figure 4: Distribution of component failures caused by injecting faults into CPU registers during PrivVM execution.

that the Xen VI provides a high degree of fault isolation and is thus an appropriate platform for deploying microreboot.

The fault isolation among Xen VI components is not perfect. A single fault in a VI components can cause other components to fail either together with the faulty component or independently. For example, when faults occur during VMM execution (Figure 2), with the synthetic and LVS workloads, about 3% and 6%, respectively, of manifested faults cause an AppVM to fail together with the VMM. Not all failure of multiple components are due to poor fault isolation. Some of these failures can be a result of an incomplete recovery leaving the system in an inconsistent state, leading to the failure of other components.

Some faults in VI components are not detected by our detection mechanisms but cause the workloads to fail or prevent the VI from correctly hosting or creating AppVMs. These faults are manifested as silent application failures (“App (silent)”), violating fault isolation (no faults were injected in AppVMs), or silent VI failures (“VI (silent)”).

A small fraction of faults result in the outer VMM terminating the entire target system. The impact of these faults is categorized as “System (externally induced)”. These failures occur if there is critical state corruption in the target system, preventing the outer VMM from performing some operation on behalf of the target system, or if the target system causes a triple fault exception to occur [15].

Table 3. Recovery success rates, out of all manifested faults, for faults in the different VI components.

Workload	VI Component	Successful Recovery Rate
Synthetic	VMM	86.2%
	DVM	94.7%
	PrivVM	88.2%
LVS	VMM	87.5%
	DVM	96.2%
	PrivVM	92.2%

Table 3 shows the effectiveness of our microreboot-based recovery mechanisms with respect to faults in each of the three Xen VI components. Despite imperfect fault isolation, a large fraction of component failures do not result in system failures — the system maintains correct operation.

For faults injected during VMM execution, approximately half of system failures are caused by two main problems: (1) the inability of the VMM to correctly recover itself or the PrivVM due to state corruption in the VMM, and (2) the simultaneous failure of two VI components, overwhelming our recovery mechanisms. The remaining causes of system failures are due to undetected VI failures, about half of which are externally induced.

For faults injected during DVMs or PrivVM execution, the majority of system failures are caused by silent application failures or silent VI failures. Silent application failures can occur when faults in the DVM cause data corruption when reading from or writing to I/O devices. Faults in either PrivVM or DVM can cause silent VI failures as both components are used to provide device access to AppVMs and the PrivVM is used to create AppVMs.

Table 4. Lines of code (LOC) needed to implement the different microreboot mechanisms. The Final mechanism category includes the LOC for all improvements made in *addition* to the Basic mechanism.

Component	Mechanism	User- level	Kernel-level	VMM-level
VMM	Basic	0	0	830
	Final	+0	+0	+50
DVM	Basic	20	285	0
PrivVM	Basic	1730	1770	350
	Final	+575	+0	+15

8. Implementation Complexity

To provide insight regarding the engineering effort required to implement microreboot-based recovery for the Xen VI components, Table 4 shows the breakdown of the implementation complexity, in terms of lines of code (LOC), for the different microreboot mechanisms. The basic PrivVM microreboot mechanism has the highest LOC count. Most of this code is related to backing up the XenStore and XenStored state. On the other hand, microrebooting the DVM requires the least amount of code. The DVM has no internal state that needs to be maintained

and the information needed to boot and reconnect a new DVM instance to existing AppVMs is kept in the PrivVM.

Similarly to the PrivVM, the VMM has state that must be maintained across a microreboot. However, unlike the PrivVM, the VMM preserve this state in place, in memory. This reduces the amount of code needed for saving and restoring state.

9. Related Work

The original work on the use of microreboot as an inexpensive recovery technique was presented in [5]. That work, along with others in [3, 2, 4], discussed the main design principles of microrebootable software and presented examples of applying microreboot to application-level software systems. The work in this paper leverages ideas from this previous work and extends them by examining how to address the unique challenges associated with employing microreboot for system-level software.

There has been prior work on applying microreboot to system software components. The focus in prior work has been, almost exclusively, on investigating a specific mechanism for a specific component or a few mechanisms for a specific component. Much of this work focused on improving the resiliency to device driver failures [21, 7, 16, 8, 13, 10]. Microreboot has been applied to recovery from failures of the Linux kernel [6] and the Xen VMM [14]. Microreboot has also been used for proactive rejuvenating of the Xen VMM and PrivVM [11]. None of the prior works presented the general challenges to implementing microreboot for low-level system software, based on experience with multiple components and multiple mechanisms. Furthermore, no prior work has presented a mechanism for recovery from PrivVM failures.

10. Summary and Conclusions

We have identified unique challenges to applying microreboot with low-level system software (LLSS) and presented general approaches to addressing these challenges. To demonstrate the utility of these approaches, we have applied microreboot to all three components of the Xen VI: the VMM, DVM, and PrivVM. We have presented some of the difficulties of applying microreboot to each VI component in the context of the earlier discussion of generic challenges and solutions with system software. Using fault injection, we have shown that microreboot can be the key building block of low-overhead techniques that successfully recover from failures in LLSS, restoring the system to full operation for a great majority of manifested faults.

Acknowledgements

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

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *19th ACM Symp. on Operating Systems Principles*, Bolton Landing, NY, pp. 164-177 (October 2003).

[2] G. Candea and A. Fox, "Crash-Only Software," *9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii (May 2003).

[3] G. Candea, A. B. Brown, A. Fox, and D. Patterson, "Recovery-Oriented Computing: Building Multitier Dependability," *IEEE Computer* **37**(11), pp. 60-67 (November 2004).

[4] G. Candea and J. Cutler, "Improving Availability with Recursive Microreboots: A Soft-State System Case Study," *Performance Evaluation Journal* **56**(1-4), pp. 213-248 (March 2004).

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

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

[7] 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).

[8] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Fault Isolation for Device Drivers," *Int. Conf. on Dependable Systems and Networks*, Estoril, Lisbon, Portugal, pp. 33-42 (June 2009).

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

[10] 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 (Nov 2010).

[11] K. Kourai and S. Chiba, "A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines," *Int. Conf. on Dependable Systems and Networks*, Edinburgh, UK, pp. 245-255 (June 2007).

[12] M. Le, A. Gallagher, and Y. Tamir, "Challenges and Opportunities with Fault Injection in Virtualized Systems," *1st Int. Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, Austin, TX (April 2008).

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

[14] M. Le and Y. Tamir, "ReHype: Enabling VM Survival Across Hypervisor Failures," *7th ACM Int. Conf. on Virtual Execution Environments*, Newport Beach, CA, pp. 63-74 (March 2011).

[15] M. Le, I. Hsu, and Y. Tamir, "Resilient Virtual Clusters," *17th IEEE Pacific Rim International Symposium on Dependable Computing*, Pasadena, CA, pp. 214-223 (December 2011).

[16] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," *6th Symp. on Operating Systems Design & Implementation*, San Francisco, CA, pp. 17-30 (December 2004).

[17] Linux Virtual Server, *http://linuxvirtualserver.org*.

[18] Microsoft, *Hyper-V Architecture*, http://msdn.microsoft.com/en-us/library/cc768520.aspx.

[19] W. T. Ng and P. M. Chen, "The Systematic Improvement of Fault Tolerance in the Rio File Cache," *29th Fault Tolerant Computing Symp.*, Madison, WI, USA, pp. 76-83 (June 1999).

[20] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *IEEE Computer* **38**(5), pp. 39-47 (May 2005).

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

[22] W. Zhang and W. Zhang, "Linux Virtual Server Clusters," *Linux Magazine* **5**(11) (November 2003).