2009 Eighth IEEE International Symposium on Network Computing and Applications

Maintaining Network QoS Across NIC Device Driver Failures Using Virtualization

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

Yoshio Turner
HP Laboratories
Palo Alto, CA
yoshio.turner@hp.com

***Abstract*—Device driver failures have been shown to be a major cause of system failures. Network services stress NIC device drivers, increasing the probability of NIC driver bugs being manifested as server failures. System virtualization is increasingly used for server consolidation and management. The isolated driver domain (IDD) architecture used by several virtual machine monitors, such as Xen, forms a natural foundation for making systems resilient to NIC driver failures. In order to realize this potential, recovery must be fast enough to maintain QoS for network services across NIC driver failures. We show that the standard Xen configuration, enhanced with simple detection and recovery mechanisms, cannot provide such QoS. However, with NIC drivers isolated in two virtual machines, in a primary/warm-spare configuration, the system can recover from an overwhelming majority of NIC driver failures in under 10ms.**

I. Introduction

A significant fraction of bugs in operating systems are found in device drivers [3]. Thus, errors in drivers are a major cause of system failures [18]. A faulty device driver can cause the entire system to crash, hang, or exhibit arbitrary incorrect behavior. In order to improve the reliability of systems, drivers must be isolated, limiting their ability to corrupt other parts of the system [18]. Furthermore, the system must be able to detect erroneous driver behavior and recover by restoring a working driver. In current systems, since a faulty driver can corrupt the entire system, recovery is likely to require a complete system reboot as well as recovery of the application state. For many applications, such as most network services, lengthy service interruption is unacceptable.

The resiliency of systems to driver failures can be improved by isolating drivers in light-weight domains [18], and by user-level drivers [9]. Isolating the device drivers from the kernel prevents buggy drivers from harming the kernel and crashing the system. Device driver recovery is done by restarting and re-attaching the device driver to the running kernel. These approaches require the kernel and device drivers be be modified or the use of non-standard device drivers.

System virtualization [16] is now widely used in data centers to provide workload isolation and flexible management of consolidated servers [2]. Several virtual machine monitors (VMMs) use an isolated driver domain (IDD) architecture to virtualize I/O devices [4, 12, 14]. With the IDD architecture, unmodified commodity device drivers (e.g., NIC drivers) run in a different virtual machine (VM) from applications. The IDD architecture does not eliminate the ability of a malicious device driver to prevent correct execution — for example, a NIC driver can drop all packets. However, the IDD architecture has the potential to prevent most non-malicious device driver failures from corrupting other VMs [4]. Since system virtualization is commonly used in data centers for other reasons, there is strong motivation to utilize the IDD architecture for resiliency to driver failures without resorting to special or modified drivers.

Unfortunately, without additional mechanisms, virtualization utilizing the IDD architecture is not sufficient to allow applications to continue uninterrupted across driver failures. On the contrary, with virtualization, the effects of failed drivers are worse since a single device driver failure can impact many VMs sharing the device. When drivers reside in a privileged VM, such as Dom0 in the Xen VMM, the entire virtualized system, including all the application VMs, must be restarted if the drivers crash the privileged VM. Even in a configuration where device drivers reside in separate non-privileged VMs [4], failure of a driver VM causes all VMs sharing the device exported by the driver VM to stop working.

Table I. Impact of fault injection in NIC device driver.

system configuration	# injections	% application failure
Linux	1987	66.0%
Xen-base	2574	66.1%
Xen-IDD	2809	63.4%

To illustrate the point above regarding the IDD architecture, Table I shows the results from fault injection into a NIC device driver (see Sections III and IV for details). The application is a simple user-level "ping" program between a separate physical system and the target system. Results are shown for a target system that is Linux without virtualization, Linux in an application VM on a standard Xen configuration (Xen-base) where the driver is in the privileged VM, and Linux in an application VM on a Xen configuration with a separate driver VM (Xen-IDD). In all three cases a similar fraction of injected faults caused the application to fail.

The focus of this paper is on achieving resiliency to NIC driver failures in virtualized systems using the IDD architecture. With the Xen VMM, we describe and evaluate several mechanisms that provide detection and recovery from NIC driver failure. Our evaluation is based on injecting faults in the driver code and measuring network service interruption. We present a driver failure detection mechanism that is capable of detecting hangs as well as crashes. Our results show that simply rebooting the VM with the NIC driver when driver failure is detected results in recovery delay of multiple

978-0-7695-3698-9/09 $25.00 © 2009 IEEE
DOI 10.1109/NCA.2009.42

seconds and thus cannot provide transparent recovery for network services.

We present a fast recovery mechanism based on maintaining two VMs with the NIC driver: the primary and a "warm spare." When driver failure is detected, recovery is performed by replacing the primary with the spare. While previous schemes reported recovery times on the order of a second [17] or hundreds of milliseconds [4], in the great majority of cases our scheme recovers in less than 10ms. Such fast recovery allows the system to meet QoS requirements for many network services. This is accomplished using unmodified application VMs, minimal modifications to the Xen VMM and driver VM kernel, and user-level scripts in the privileged VM. The scheme incurs no performance overhead and insignificant memory overhead during normal operation.

While our evaluation is based mainly on the Intel Pro 100 100Mb NIC, we have also validated our mechanism using another 100Mb NIC and a 1Gb NIC. With 100Mb NICs, *no* device driver modifications were required. With the 1Gb NIC, minor modifications (15 lines of code) were required.

The relevant aspects of virtualization technology are reviewed in Section II. The evaluation methodology is described in Section III. Section IV presents an evaluation of the impact of device driver failures on standard Linux and on Linux in a VM hosted by Xen. Section V describes and evaluates three system architectures that provide recovery from driver failures. Related work is presented in Section VI.

II. System Virtualization

Virtualization technology allows multiple VMs, each with its own OS, to run on a single physical machine [16]. A critical function of the VMM is to isolate VMs from each other so that activities in one VM cannot affect another VM [15].

In Xen [1], a privileged VM (PrivVM), often referred to as Dom0, is used to control and manage unprivileged VMs running applications (AppVMs). The PrivVM has direct access to the hardware devices on the system and typically houses all the device drivers. A failure of the PrivVM can cause the entire virtualized system to crash.

The split device driver architecture in Xen facilitates the sharing of devices among VMs [4]. With the split driver (Fig. 1), a *frontend* driver resides in each VM sharing the device. One *backend* driver together with the actual device driver reside in one VM. Requests from frontends are processed by the backend, using the actual device driver to perform the requested operations. Frontend/backend drivers are paravirtualized (PV) but can be used in fully virtualized (FV) VMs. FV VMs can use PV drivers in place of more complex device drivers to improve reliability and performance (by eliminating device emulation).

Communication between frontend and backend drivers is done using requests and responses on a ring data structure in

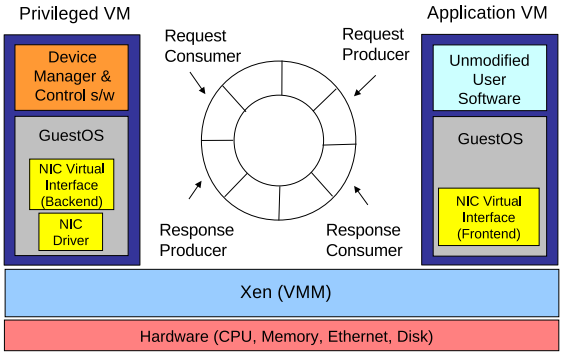

Figure 1: Xen virtualization architecture

memory shared between the respective VMs. The requests are often pointers to pages in memory containing the data, such as a packet that a VM wishes to send. These pointers, called grant references, allow one VM to give access to certain memory pages to other VMs. Once a request or response is placed on the shared ring, an event can be sent notifying the other side of the pending request/response. The use of the ring is coordinated using request/response producer/consumer indices. For network devices, there are two shared rings, one used for transmission (transmit ring) and one for reception (receive ring).

To initialize a frontend-backend connection, grant references (for setting up the shared ring) and event channel ports must be communicated between frontend and backend drivers residing on different VMs. To allow this, Xen provides a centralized store call the XenStore, implemented as a process running in the PrivVM. Since communication through the XenStore is asynchronous, involving multiple levels of indirection, operations are slow and are typically only used for frontend-backend connection setup and teardown.

In order to isolate the PrivVM from device drivers, Xen allows backends with their device drivers to be hosted on unprivileged VMs [4]. A VM hosting a device driver must be able to directly access the corresponding device controller. Xen allows VMs to have direct access to PCI devices by mapping the PCI I/O memory address space into the VMs virtual address space. VMs hosting drivers that directly access devices are referred to in this paper as driver VMs (DVMs). VMs which host only network devices are referred to as network driver VMs (NetDVMs). Multiple AppVMs can share a single NetDVM.

III. Evaluation Methodology

The results in this paper were obtained using several systems based on the Intel Core-2 processors. For most experiments, we used Intel Pro 100 NICs interconnected by a 100Mb switched ethernet network. Xen 3.3.0 was used as the VMM. The non-virtualized setup used the Linux kernel version 2.6.18.8. XenoLinux kernel version 2.6.18.8 was used for the PrivVM, AppVM, and NetDVMs.

Software-implemented fault injection was used to inject faults in the NIC driver. This was done using our *Gigan* fault injector that resides in the VMM and is capable of non-intrusively injecting into VMs [11]. Some of these injections simulate programming errors, such as invalid pointer access and incorrect loop termination conditions [18]. For *code* injection, faults were injected into every byte of the most frequently used functions[8] (six out of 75) in the E100 device driver, identified by the Xenoprof sampling profiler [13]. These six functions account for roughly 29% of the E100 code. The rest of the functions are used for device/driver initialization. Injection was triggered by setting hardware breakpoints on the virtual addresses targeted for injection. If the breakpoint fired, a random bit of the targeted byte was flipped. *Register* injection was similar, except that when the breakpoint fired a random bit of a random general-purpose register was flipped. In the reported results, an injection is counted only if it was actually activated — the breakpoint fired and caused a bit flip.

The outcome of each injection experiment is classified as either crash, hang, silent application failure, or non-manifested. A crash occurs when a kernel panics or the VM is killed by the VMM. A hang occurs when the VM stops responding with no explicit report of a crash. A silent application failure occurs when no hang or crash is detected but the application is unable to complete successfully (see below). Non-manifested means that no errors are observed.

Fault injection in a device driver often causes the kernel hosting the driver to crash or hang. To facilitate the experiments, for the Linux and Xen-base targets (see Section I), the targets were run inside a VM. This allowed the injection campaign software to resume the campaign after a crash or hang without manual intervention [11]. Thus, with the Xen-base system target, there were two levels of Xen: an inner-level Xen running inside the VM hosted by the outer-level Xen. In this case, injection was done from the *Gigan* injector located in the inner-level Xen. For the rest of the configurations, in which device drivers were isolated in their own VM, the target system ran directly on a physical machine.

A user-level *ping* was used to exercise the NIC drivers. This application, consisting of two processes running on separate physical hosts, sends a UDP packet every 1ms from the *sender host* to the target system. Upon delivery of this packet, an acknowledge UDP packet is sent back. When the target is a virtualized system, one of the application processes is run on an AppVM. The network interruption latency, which is a combination of failure detection and recovery latency, is the maximum time between reception of successive ping acknowledgements on the *sender host*.

The application process on the target system normally executes for 7 seconds during which it receives packets before reporting normal termination. If there is a network interruption, packets are not received so the application may execute for a longer period before accumulating 7s during which it receives packets. One injection run consists of booting the target system, randomly picking a time between 0 and 5s for when to set a breakpoint at the injection address, and running the application. If the application does not record normal termination within 22s (or in some experiments 14s), a timeout is triggered and that is interpreted to be a "silent application failure."

IV. Impact of Device Driver Failures

In a conventional system, device driver failure is likely to lead to overall system failure. In our experiments (Table II), 64.9% of injections into the E100 driver, running in a conventional Linux system, led to system crashes or hangs. As discussed earlier, recovery in this case requires a reboot of the entire system. This involves a long service interruption. Furthermore, without additional mechanisms, such as checkpointing/rollback, rebooting also leads to loss of application state.

Table II. Device driver code injection results

System Configuration	Linux	Xen-base	Xen-IDD
Injections	1987	2574	2809
System Crash	54.6%	0.0%	0.0%
System Hang	10.3%	0.0%	0.0%
PrivVM Crash	-	56.7%	0.0%
PrivVM Hang	-	7.7%	0.0%
NetDVM Crash	-	-	55.5%
NetDVM Hang	-	-	3.0%
Silent App Failure	1.1%	1.7%	4.9%

With the standard Xen configuration (Fig. 1), the device drivers are in the PrivVM, which is isolated from the AppVM. Table II shows that injection into the E100 driver in the PrivVM leads to crashes/hangs of the PrivVM (in 64.4% of injections) and not of the VMM or AppVM. However, the overall effect on the application running in the AppVM is the same (fails in 66.1% of injections). This is not surprising since crashes/hangs of the PrivVM effectively renders the entire virtualized system inoperable. Thus, it is desirable to remove the device drivers from the PrivVM and place them in a separate DVM. This allows the DVM to fail independently from the PrivVM and leave the virtualized system intact.

Fig. 2 shows a configuration where device drivers are isolated using DVMs. The NIC device driver is in the NetDVM while all the rest of the drivers are in one DVM. This configuration isolates all the device drivers from the PrivVM and AppVM and isolates the NIC driver from other drivers. As shown in Table II, injection into the NIC driver in this setup only crashes/hangs the NetDVM (in 58.5% of injections), leaving all other components of the system intact. While this configuration prevents driver failures from propagating to the AppVM and the PrivVM, the effect on the application running in the AppVM (fails in 63.4% of

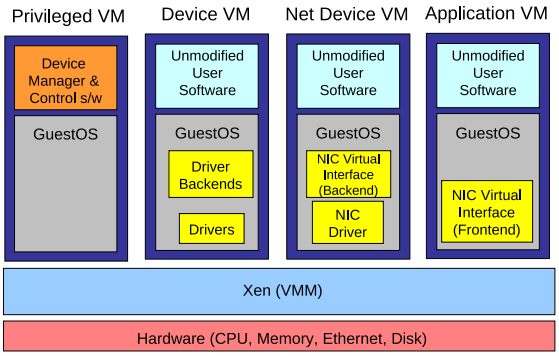

Figure 2: Xen-IDD – Xen with an Isolated Device Driver VM

injections) is the same as with the Xen-base and Linux configurations. This is because failures of the NetDVM disrupt the network connectivity of the AppVM. Without some mechanism to recover the faulty NetDVM, the application cannot continue to operate.

V. Device Driver Recovery

This section describes NIC driver recovery techniques that can be used in a virtualized system where the NIC device driver is isolated in a NetDVM. These recovery techniques are based on the assumption that the driver code is not completely dysfunctional. Rather, driver bugs may lead to failure in rare cases, under particular timing and ordering of asynchronous events in a system. Such Heisenbugs [6] are unlikely to recur once the driver is reset and the system is in a slightly different state. In addition, these techniques are effective for driver errors that are the result of transient hardware faults.

The first recovery scheme simply detects and reboots a failed NetDVM. While this works and enables the AppVM to continue, it is also slow. The next technique speeds up recovery by deploying a warm-spare NetDVM to take over once the primary NetDVM fails. Analysis of this mechanism reveals parts of the recovery process that can be further optimized, leading to the final recovery technique that can reduce network interruption time to less than 10ms.

A. Failure Detection Mechanisms

NetDVM recovery is initiated after the VMM detects a crash or hang. Crashes are detected when 1) the crash handler in the VM's kernel makes a hypercall to the VMM or 2) the VMM responds to illegal VM activity by killing the VM.

Hangs are diagnosed using two heuristics. The first heuristic maintains that a kernel with multiple runnable processes should be context switching among those processes. The VMM detects context switching by monitoring page table base register (PTBR) changes of the VM. If PTBR changes are not detected for a specified quanta of time (500ms), a hang is reported. To ensure that PTBR changes occur in a fault-free VM, the VM executes two simple processes that periodically (every 150ms) wake up and execute a few instructions.

The second heuristic applies to the transmit rings of NetDVMs. When an AppVM transmits a packet, it places it on the shared ring. An operational NetDVM should eventually remove the packet from the ring and place a response packet on the ring. Code in the VMM periodically (every 100ms) samples the shared ring indices. This code identifies a hang when it determines that there were packets on the transmit ring in one sample but none of these packets were processed and responded to by the NetDVM before the next sample. This mechanism cannot be applied to the receive ring since it is not known when packets will be received by the NetDVM.

B. Recovery Using Reboot

When a crash/hang is detected by the VMM, a message is sent to a user-level recovery agent running in the PrivVM. This message is sent using a shared ring and an event channel that the recovery agent sets up with the VMM when it is initialized. When the recovery agent is alerted to a crash/hang, it recovers the system by pausing the failed NetDVM, booting a new NetDVM, and establishing a new frontend-backend connection with the AppVM. As explained below, this requires: 1) modification of Xen's management tool, allowing it to remove access to the NIC device from a failed NetDVM; and 2) enhancement of the suspend/resume code in the AppVM kernel to facilitate frontend-backend reconnection. Fig. 2 shows the system setup for this recovery technique.

It might be expected that the first action upon detecting a crash/hang of a NetDVM would be to destroy it. However, despite the crash/hang of the NetDVM, the NIC may still be writing into the NetDVM's memory. If the failed NetDVM is destroyed, its memory might be reused by the VMM (e.g., for the new NetDVM), leading to memory corruption. Hence, the failed NetDVM is, initially, only paused, keeping all its memory, instead of being immediately destroyed.

A VMM that allows VMs to directly access PCI devices must prevent concurrent accesses to a single PCI device by multiple DVMs. Hence, Xen's management tools (in the PrivVM) prevent another VM from accessing the same PCI device as a paused VM. In order to allow a new NetDVM to boot and control the NIC, the new NetDVM must be given access to the PCI device. Therefore, a minor modification to the management tools was done to allow the paused NetDVM to be flagged as no longer accessing the NIC device. This change allows the new NetDVM to be granted access to the NIC device. The failed NetDVM is destroyed shortly after the new NetDVM is booted, without ever being unpaused.

A new frontend-backend connection must be established between the new NetDVM and the AppVM to complete recovery. This should be transparent to the application on the AppVM so that there will be no need to modify applications. Fortunately, this ability is already implemented in Xen for VM checkpointing/restoring. During checkpointing of a VM, all frontend drivers suspend activity and prepare for system-level

Table III. Device driver injection results with recovery.

system configuration	injections	NetDVM crash	NetDVM hang	silent app failure	non manifested	successful recovery
IDD Reboot Recovery	568 (code)	54.0%	3.2%	2.8%	40.0%	100.0%
IDD Spare Recovery	584 (code)	53.4%	3.4%	2.9%	40.3%	100.0%
IDD Fast Spare Recovery	2856 (code)	55.2%	3.1%	2.6%	39.1%	99.9%
IDD Fast Spare Recovery	1300 (register)	28.6%	3.0%	0.5%	67.8%	99.9%

suspension. During VM restoration, the frontend drivers resume activity and transparently reconnect with the backend drivers. Unfortunately, checkpointing and restoring a VM is slow. To avoid full checkpoint/restore, the suspend/resume code in the AppVM kernel was modified to allow individual frontend drivers to be selectively suspended/resumed. Using this mechanism, the recovery agent directs the AppVM to transparently reconnect only the network frontend driver to the backend driver on the new NetDVM.

This recovery technique was evaluated by performing fault injection into the code of the E100 device driver as described in section III. Table III shows the results. Recoveries were attempted on 325 detected hangs/crashes out of 568 injections and successfully restored the system 100% of the time. The mean time of recovery was 7.47s and ranged from 7.29s to 9.36s. The slowest component of this recovery mechanism was the time required to boot a new NetDVM (approximately 3s).

C. Recovery using a Warm-Spare NetDVM

The booting of a new NetDVM during recovery can be avoided by using a warm-spare NetDVM, paused, waiting to take over when a NetDVM fails. The system setup for this recovery technique is similar to the Xen-IDD architecture (Fig. 2), except for the addition of a spare NetDVM. To initialize the system, the PrivVM boots both the spare and primary NetDVMs, pausing the spare once its OS has booted. During recovery, the recovery agent: 1) pauses the failed NetDVM, 2) unpauses the spare NetDVM, 3) transfers NIC device control to the spare NetDVM, and 4) sets up a new frontend-backend connection between the spare NetDVM and the AppVM. Most of the required modifications are the same as for the reboot recovery technique. In addition, a simple mechanism is implemented to inform the PrivVM when to pause the spare NetDVM during initialization. Specifically, the PrivVM monitors an entry in the XenStore that the spare NetDVM writes once it boots.

Results of fault injection into the code of the E100 device driver (Table III) show that recoveries were attempted on 332 detected hangs/crashes out of 584 injections and successfully restored the system 100% of the time. The mean time of recovery was 4.52s and ranged from 4.33s to 7.69s. Most of this time was spent: (I) invoking the recovery agent inside the PrivVM, (II) transferring control of the NIC device to the spare NetDVM, and (III) establishing a new frontend-backend connection. As with the NetDVM reboot scheme, the recovery agent is invoked by a message sent from the VMM. Due to the asynchronous nature of this communication, it takes approximately 0.4s for the recovery agent to respond to a detected NetDVM failure. The following paragraphs detail the impact of operations (II) and (III) above on the recovery time.

In the reboot recovery mechanism, the time to boot a new NetDVM includes mapping the NIC to the NetDVM address space and then probing the NIC. Most of the boot time is saved by deploying a warm-spare NetDVM. However, the mapping and probing of the NIC device must still be delayed until recovery since the NIC device is exclusively accessed by the primary NetDVM up to that point.

The permission to map the PCI NIC device to the NetDVM's address space is controlled by a PCI-backend driver located in the PrivVM and a PCI-frontend driver located in the NetDVM. Before the spare NetDVM can access the NIC, the recovery agent issues a request for the spare NetDVM to be given the access. This process takes approximately 1s to complete due to the required multiple interactions, via the XenStore, among the recovery agent, Xen's user-management utilities, the PCI-backend driver, and the PCI-frontend driver.

To restore network connectivity to the AppVM, a frontend-backend connection must be established with the spare NetDVM. The same mechanism used in the NetDVM reboot technique is used here. Initiating and establishing this connection requires multiple handshakes, via the XenStore, between the PrivVM and AppVM and between the PrivVM and spare NetDVM. This takes about 1.8s to complete.

The memory overhead of keeping a spare NetDVM is 64MB, which is used to run the XenoLinux kernel and applications to setup and run the network backend. However, since the memory content of the spare NetDVM is nearly identical to the primary NetDVM, most of this overhead could be eliminated by using content-based memory sharing in the VMM [7]. Since the spare NetDVM is paused, it does not consume any CPU cycles.

D. Minimizing NetDVM Failover Latency

The second paragraph of the previous subsection lists the three operations responsible for most of the latency of recovery with the warm-spare NetDVM. Most of the latency of (I) can be eliminated by moving the main functionality of the recovery agent into the VMM. Most of the latency of (II) and (III) can be eliminated by allowing the spare NetDVM to

discover and set up the NIC device and form the network frontend-backend connection with the AppVM during system initialization, prior to recovery. With these optimizations, recovery entails pausing the failed primary NetDVM, unpausing the spare NetDVM, and performing minor NIC device and network backend re-initialization.

This recovery technique requires that the primary and spare NetDVMs have access to a single NIC device and use the same shared rings to perform network communication with the AppVM. The modifications required for this recovery technique are: (M1) VMM modification, allowing it to control the recovery of the NetDVM; (M2) VMM modification, allowing it to remap grant permissions on-the-fly; (M3) VMM modification, allowing it to redirect event notifications; (M4) VMM modification, allowing a VM to invoke a new hypercall to pause itself; (M5) modification to Xen's PCI backend driver, allowing PCI device co-ownership; (M6) a new kernel module in the spare NetDVM to perform NIC device and network backend re-initialization; and (M7) a new function in the NetDVM network backend device driver, allowing the existing shared ring to be attached to the backend during recovery. All of these modifications together required adding 246 lines of code to the Xen VMM, 200 lines of code to NetDVM kernel, and 51 lines of code to the PrivVM kernel.

The system configuration for this mechanism is similar to the previous subsection. However, system initialization differs in two ways: (1) the spare NetDVM pauses itself after setting up the frontend-backend connection with the AppVM and (2) when the primary NetDVM boots, it forms a frontend-backend connection with the AppVM by mapping in the shared rings used in the connection between the AppVM and the spare NetDVM. VMM code (M1) provides the functionality of the recovery agent. When an error is detected, this code pauses the failed NetDVM, initiates unmapping of some pages from the failed NetDVM (see below), and unpauses the spare NetDVM.

In this setup, the AppVM effectively has a single network frontend that is connected to two network backends (spare NetDVM and primary NetDVM), with only one network backend active at a time. It is transparent to the AppVM which NetDVM is handling its network requests. Hence, the AppVM is not involved in the recovery process, thus reducing the recovery latency. (M2) and (M3) implement this ability by remapping and redirecting grant references and event notifications to the active network backend. These modifications require two minor changes to the VMM. First, hooks are added to change grant references associated with the failed NetDVM to refer to the spare NetDVM. Second, one end of an event channel is allowed to be disconnected from one VM and reconnected to another VM without involving the VM at the other end.

During recovery, the VMM (M1) removes from the failed NetDVM the mapping to pages used for buffers by the shared rings between the AppVM and NetDVM. The spare NetDVM maps these pages into its address space on-the-fly (M2).

During recovery, the spare NetDVM re-initializes the NIC device and network backend driver (M6). This involves re-initializing the DMA regions. Furthermore, since the failed NetDVM may corrupt the NIC state, the re-initialization involves a reset of the NIC (see Subsection V.F). The private shared ring indices in the network backend driver are updated with current values from the shared rings (transmit and receive), so that the spare NetDVM can continue consuming requests and putting responses at the locations where the failed NetDVM left off. A new function is added in the network backend device driver in Linux (M7) to allow for this shared ring re-attachment to occur.

A kernel module is added to the spare NetDVM (M6). During system initialization, this module utilizes a new hypercall to the VMM that pauses the calling VM (M4). When the VMM unpauses the spare NetDVM, during recovery, the spare NetDVM begins to immediately execute code which initializes the NIC device and network backend.

Since setting up access to the NIC device for the spare NetDVM is slow, this step is moved to system initialization by allowing both the spare and primary NetDVMs to have active use of the NIC device (M5). This requires a modification to the PCI backend driver in the PrivVM to maintain multiple connections (one for each NetDVM) to a single PCI device. Concurrent access to the device is still prevented by ensuring only one NetDVM is unpaused at any time.

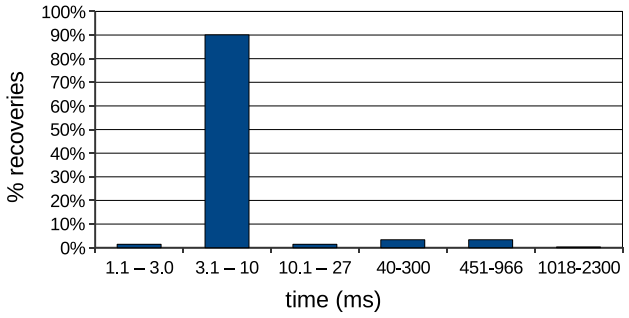

Figure 3: Network interruption latency using fast failover spare NetDVM

Evaluation of this approach consisted primarily of code and register injection into the E100 device driver. As shown in Table III, with code (register) injection, out of 2856 (1300) injections, 97.4% (99.5%) were not manifested or triggered recovery. When it was triggered, recovery was successful in 99.9% of the cases. In the two cases that failed, the AppVM no longer received packets after recovery. Fig. 3 shows the distribution of network interruption times (recovery times) with fault injection in the driver code. In 91% of recoveries, the network interruption time was less than 10ms. All of these cases were crashes that were immediately detected. Most

interruption durations over 10ms were due to detection latency of the hang detectors, ranging between 200ms for the transmit ring hang detector to 1s for the PTBR hang detector. There were also cases where crashes took as much as 500ms to detect since the NetDVM kernel was stuck recursively calling fault handlers before it finally panicked.

E. Silent Application Failures

A small portion of injections (less than 3%) caused the benchmark to fail silently. These were the results of errors that disrupted the NetDVMs ability to provide network connectivity but failed to be detected by the mechanisms discussed in Subsection V.A. Three examples of this class of errors, that resulted from code injection into the E100 NIC with the fast recovery scheme, are discussed below.

To allow interrupt number sharing in Linux, interrupt handlers for device drivers probe their devices to determine if they generated the interrupt. The first silent failure was caused by injecting into the code that performs this check, leading the interrupt handler to incorrectly determine that the NIC did not generate the interrupt. This caused the handler to exit without processing the interrupt, leaving the device unusable.

The second silent failure was a result of a fault that changed the condition check in a conditional jump. This caused a function in the driver to incorrectly return an error code, leading higher-level operations in the driver to fail. The result was that incomplete command sequences were sent to the NIC, preventing it from properly performing the desired function.

The third silent failure was caused by a fault in the code that allocates new buffers for received packets after old ones fill up. The fault prevents the buffers from being allocated, and thus, packets could no longer be received.

F. Recovering from Failures of other NIC Drivers

All the results reported so far were for the Intel E100 NIC driver. To determine the extent to which the the fast recovery scheme depends on a particular NIC, we evaluated the scheme with the RealTek 8139 100Mb NIC. The results with this NIC were similar to the results with the E100 NIC. Specifically, out of 1044 injected faults, 58% resulted in detectable NetDVM failures. Recovery was successful for all these detectable failures. In 87% of recoveries, the network interruption time was less than 16ms.

Our scheme is not able to achieve fast recovery for 1Gb NICs without small modifications to the device driver. A key to achieving fast recovery is the ability to quickly reset the NIC when failing over to the spare NetDVM. Resetting the NIC updates the transmit and receive descriptor rings in the device with memory locations from the spare NetDVM. Resetting the NIC also re-initializes the hardware state, which may have been corrupted by the failed NetDVM. The problem with 1Gb NICs is that a full reset can take up to 1.8s to complete. This was observed in three different 1Gb NICs: Broadcom Tigon 3, Attansic Atheros L1E, and Intel E1000. Most of this time is due to auto-negotiation and PHY training that must be performed on a reset [10].

In most cases, the physical link is not affected when a device driver fails. Hence, performing link negotiation is rarely necessary for recovery. Thus, as long as the NIC state is not corrupted, a full NIC reset can be avoided, thereby overcoming the above difficulty with 1Gb NICs. The only necessary operation is updating the descriptor rings in the NIC. While recovery may fail if the driver failure corrupts the NIC state, the experimental results below demonstrate that this is unlikely.

For the E1000 NIC we made small modifications to the driver to allow the descriptor rings to be updated without performing a hardware reset. The changes consisted of adding 15 lines of modified versions of existing functions.

As part of normal operation, the E1000 NIC caches receive descriptors [10]. This poses a problem during recovery since these cached descriptors contain memory locations belonging to the failed primary NetDVM. Incoming packets using these cached descriptors will be lost, thus increasing network interruption latency. To avoid this, these cached descriptors must be quickly purged immediately after recovery. Purging is performed by having the recovery module in the spare NetDVM send out ping requests immediately after recovery so that the returning ping replies flush out the invalid cached descriptors.

With the modifications above, out of 2289 injected faults, 37% resulted in detectable NetDVM failures. Avoiding the full NIC reset caused more unsuccessful recoveries: 18 out of 837 attempted recoveries, compared to 5 out of 2680 attempted recoveries for the 100Mb NICs. In 91% of detectable failures, the network interruption latency is less than 33ms. Most of the 33ms interruption latency is due to a 10ms sleep (actually measured between 10-19ms) that exists as part of the E1000 descriptor ring re-initialization code. When this sleep was removed, the network interruption latency was reduced to less than 10ms for 86% of recoveries. However, the number of unsuccessful recoveries increased to 54 out of 745 attempted recoveries.

VI. Related Work

Over the last few years there has been significant interest in techniques for enhancing system resiliency to driver failure [4, 12, 18, 17, 5, 9]. Excellent summaries can be found in the related work sections of two recent publications on the topic [5, 9]. The main idea in all of these mechanisms is to isolate drivers from the rest of the system so that a faulty driver is unlikely to corrupt or crash other parts of the system. Techniques that do not use virtualization typically require modifications to the device driver and the OS kernel. Schemes that use virtualization provide stronger isolation but incur overhead in performance and memory associated with

virtualization. However, since virtualization is widely used in data centers for other reasons [16, 2], there is little *additional* overhead for taking advantage of virtualization to provide resiliency to driver failures.

Most published mechanisms for enhancing resiliency to driver failure have been applied to NIC drivers. Published results for network interruptions associated with recovery from NIC driver failures range from hundreds of milliseconds to a few seconds [4, 9, 17].

Fraser et al. [4], describe the use of Xen's IDD architecture to provide recovery from NIC driver failure. Recovery is done by restarting the failed driver VM and reconnecting the guest VM to the new driver VM instance. The driver VM uses a customized kernel that boots from RAM disk, only comes up far enough to initialize the network device, and runs no user processes [19]. The evaluation is done by causing the driver to perform an illegal memory access, leading to guaranteed immediate detection and recorded network outages of around 275ms. Our mechanism uses primary and warm-spare driver VMs that recover from most driver failures in less than 10ms. Our evaluation is based on extensive fault injection and includes an evaluation of a practical detection mechanism in addition to the recovery procedure.

VII. Conclusions and Future Work

This paper shows that recovery from NIC driver failures can be accomplished while maintaining QoS. This is done by exploiting the capabilities of system virtualization, that is already in wide (and increasing) use for other reasons. Virtualization not only enforces strong isolation of the NIC driver from the rest of the system, but, as shown here, also provides mechanisms that enable the implementation of multiple recovery schemes with relatively little effort.

We have demonstrated that simple detection mechanisms, facilitated by virtualization, are able to detect the overwhelming majority (over 95%) of manifested faults in the NIC driver. Our driver recovery scheme, based on primary and warm-spare NetDVMs, is able to recover from 99.9% of detected driver failures. In most cases (91%), network interruptions for recovery were in the range of 1-10ms.

While most of our results were based on the E100 100Mb NIC, we also investigated the applicability of our scheme to other NICs. With the Realtek 8139 100Mb NIC we obtained similar results. For both of these 100Mb NICs no driver modifications were needed. We determined that for 1Gb NICs, recovery that involves a full NIC reset cannot be used since the reset itself is very slow. Using the E1000 1Gb NIC, we showed that fast recovery is achievable with small driver modifications that allow the reset to be avoided.

Future work will include the evaluation of the recovery scheme under multiple workload scenarios, using multiple AppVMs and NetDVMs, and with a greater variety of fault injection campaigns.

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

[2] J. P. Casazza, M. Greenfield, and K. Shi, "Redefining Server Performance Characterization for Virtualization Benchmarking," *Intel Technology Journal* **10**(3), pp. 243-251 (August 2006).

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating System Errors," *18th ACM Symposium on Operating Systems Principles*, Lake Louise, Alberta, pp. 73-88 (October 2001).

[4] 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)*, Boston, MA (October 2004).

[5] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The Design and Implementation of Microdrivers," *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, pp. 168-178 (March 2008).

[6] J. Gray, "Why Do Computers Stop and What Can Be Done About It?," *5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, pp. 3-12 (January 1986).

[7] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing Memory Redundancy in Virtual Machines," *8th Symposium on Operating Systems Design and Implementation*, San Diego, CA (December 2008).

[8] 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).

[9] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Failure Resilience for Device Drivers," *International Conference on Dependable Systems and Networks*, Edinburgh, UK, pp. 41-50 (June 2007).

[10] Intel, "PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual," *http://download.intel.com/design/network/manuals/8254x_GBe_SDM.pdf* (Accessed March 2009).

[11] 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).

[12] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," *6th Conference on Symposium on Opearting Systems Design*, San Francisco, CA (September 2004).

[13] 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* (June 2005).

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

[15] H. V. Ramasamy and M. Schunter, "Architecting Dependable Systems Using Virtualization," *Workshop on Architecting Dependable Systems*, Edinburgh, UK (June 2007).

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

[17] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering Device Drivers," *6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA (December 2004).

[18] 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).

[19] A. Warfield, *Private Communication* (July 2008).