UCLA Computer Science Department Technical Report #130027
University of California, Los Angeles, November 2013.

Resilience to Device Driver Failures Using Virtualization

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

Abstract— Faulty device drivers are a significant cause of system failures. Low overhead mechanisms that leverage virtualization can detect and recover from device driver failures without requiring modifications to the device driver, applications, or OS running in the VMs. These mechanisms can vary significantly in terms of coverage, recovery latency, and implementation complexity. This paper explores the design space of such mechanisms, provides a taxonomy for their characterization, and evaluates key points in the design space. Based on full implementations of a variety of mechanisms, design tradeoffs are described and key implementation challenges are identified. Schemes are evaluated on a variety of system configurations with multiple devices and multiple VMs running applications. Extensive fault injection campaigns are used to evaluate the effectiveness of the different mechanisms. It is shown that simple recovery schemes, transparent to the VMs running applications, can effectively recover from a very high percentage of faults. However, in order to minimize service interruption duration, it is necessary to use schemes that are slightly more complex, involving redundant device controllers.

Index Terms — Fault tolerance, recovery, virtual machine, VMM, hypervisor, network, storage

1. Introduction

Errors in device drivers are a major cause of system failures [4, 29]. A faulty device driver can cause the entire system to crash, hang, or exhibit arbitrary incorrect behavior. In order to tolerate device driver failures, drivers must be isolated so that they are not able to corrupt other parts of the system [29]. In addition, the system must detect erroneous driver behavior and restore, to applications, access to the corresponding I/O devices.

System virtualization [27] provides workload isolation and flexible resource management and is thus widely used in small servers as well as large data centers [3]. Some virtual machine monitors (VMMs) use an isolated driver virtual machine (IDVM) architecture to virtualize I/O devices [7, 20, 23]. With the IDVM architecture, commodity device drivers run in different virtual machines (VMs) from applications. We refer to a VM that hosts device drivers as a *driver VM* (DVM) and a VM that hosts applications as *application VM* (AppVM).

By isolating device drivers in their own VMs, the IDVM architecture has the potential to prevent most non-malicious device driver failures from directly corrupting other VMs [7]. However, such containment is not sufficient for providing device access to applications running in AppVMs across device driver failures. Specifically, the failure of a DVM or drivers within it will affect all the AppVMs that share devices through that DVM.

With the IDVM architecture, the *virtualization infrastructure* (VI) establishes connections from AppVMs to devices, with the connections passing through DVMs. When a device driver fails, possibly corrupting the DVM in which it runs, the connections to devices previously hosted by the potentially corrupted DVM must be reestablished. These alternate connections must not use the corrupted DVM. In addition to reestablishment of access to devices by the AppVMs, the ability of the system to recover from additional device driver failures must be restored. All this should be done with minimal overhead during normal operation and minimal interruptions to AppVM operation when device driver failure does occur.

There are many different mechanisms for providing recovery from device driver failure in virtualized systems based on the IDVM architecture. Mechanisms differ in their implementation complexity, memory overhead, performance overhead, coverage, and duration of service (device access) interruption when failure does occur. This paper is an exploration of the design space of such mechanisms. Different mechanisms are described, analyzed, and evaluated. We make the following key contributions: (1) provide a description of the design alternatives and a taxonomy of classifying them; (2) describe the tradeoffs among the different design alternatives; (3) identify critical implementation issues that must be addressed; (4) experimentally evaluate key design points based on measurements of actual implementations in a variety of system configurations.

There has been extensive previous work on improving the resiliency of systems to driver failures by isolating drivers in a VM [7, 20, 19, 17], user-level process [13, 14, 8], or light-weight domain [29, 28]. Unlike the work presented in this paper, these previous works only focused on a particular scheme and/or a particular device type (i.e., network or block device), and do not provide a systematic exploration or evaluation of the tradeoffs among different recovery approaches. Previous works also do not discuss nor evaluate mechanisms for restoring the system to its fault-tolerant configuration following recovery. Unlike previous works, we analyze the extent to which different types of faults affect detection and recovery latency, and thus total service interruption time, by conducting extensive fault injection campaigns. Furthermore, we analyze important system configuration parameters that can impact service interruption latency during DVM failure handling. These system configuration parameters include the number of AppVMs that share a DVM, the number of device controllers assigned to a DVM, and the amount of physical resources (CPUs) shared among VMs.

The relevant aspects of virtualization technology are reviewed in Section 2. The basic architectures and design tradeoffs of different mechanisms for providing resiliency to device driver failures are described in Section 3, which also presents a taxonomy of the mechanisms. Section 4 identifies key challenges in implementing the mechanisms and ensuring their reliable and fast operation. Section 5 describes our experimental setup and Section 6 presents our results. Related work is described in Section 7.

2. Split Device Drivers and Device Virtualization

System virtualization allows multiple VMs, each with its own OS, to run on a single physical machine [27]. A critical function of the VMM is to isolate VMs from each other so that activities in one VM cannot affect other VMs [26].

The IDVM architecture facilitates the sharing of devices among VMs using drivers that are split into *frontends* and *backends* [7, 20, 23]. A physical device is hosted by a DVM that includes a *backend* driver and the actual device driver, while a *frontend* driver resides in each AppVM sharing the device (Fig. 1). Requests from frontends are processed by the backend, using the actual device driver to perform the requested operations. There can be multiple DVMs in the system. A DVM can host multiple device drivers as well as multiple types of devices (e.g., both network and block devices). Multiple AppVMs can share a single DVM.

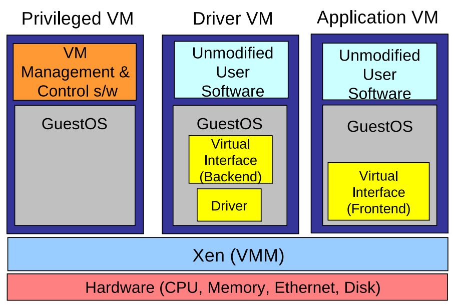
Privileged VM
Driver VM
Application VM
VM Management & Control s/w
Unmodified User Software
Unmodified User Software
GuestOS
GuestOS
GuestOS
Virtual Interface (Backend)
Driver
Virtual Interface (Frontend)
Xen (VMM)
Hardware (CPU, Memory, Ethernet, Disk)

Figure 1: Split device driver

A DVM hosting a device driver must be able to directly access the corresponding device controller. The VMM allows such access to PCI devices by mapping the PCI I/O memory address space into the DVM's virtual address space. With the Xen VMM [1], communication between the frontend and backend drivers is done using requests and responses on a ring data structure in memory shared between the respective 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.

A privileged VM (PrivVM) performs critical system management operations, such as creating/destroying VM, attaching/detaching devices to/from VMs, and pausing/unpausing VMs. With Xen [1], bootstrapping the frontend-backend connection involves exchanging information through a centralized store, implemented as a process running in the PrivVM, called the XenStore. The VMM, PrivVM, and DVMs together form the *virtualization infrastructure* (VI).

It should be noted that, in most Xen deployments, the PrivVM not only controls and manages other VMs, but also serves as a DVM. However, that configuration makes the PrivVM directly vulnerable to corruption by device driver failure and is thus a poor choice for resilience to driver failure.

The ubiquity and importance of virtualization has led to the development of self-virtualizing device controllers, such as the ones conforming to the SR-IOV standard [15]. With such device controllers, a single physical controller is partitioned into multiple virtual functions (VFs). Each VF is assigned to a different VM so that multiple VMs can safely and securely share a single device. In most cases, the VMs can directly access the VFs using drivers that are included in most OS distributions. As discussed below, there are disadvantages and limitations to self-virtualizing controllers so they do not remove the need for device access using the IDVM architecture. In addition, as a practical matter, many of the deployed servers are not configured with SR-IOV controllers and must use software solutions for device sharing.

While allowing AppVMs to have direct access to devices can improve performance, such setups can increase the chance of AppVM failures due to buggy device drivers. One of the main benefits of the IDVM architecture is that it isolates device drivers from the rest of the system, thus preventing device driver failures from, for example, crashing AppVMs. Since restoring a stateless DVM is much simpler than restoring a stateful AppVM, successful recovery is far more likely if a failed device driver can only crash a DVM. Thus, for dependable virtualized system, the use of the IDVM architecture is still highly desirable.

In addition to the issue of reliability, there is a need for the IDVM architecture due to limitations of current SR-IOV device controllers and limited support for SR-IOV controllers in many current VIs. An SR-IOV controller has a fixed number of VFs. Hence, without a software-based device sharing mechanism, the number of AppVMs that can share the physical device is limited. Furthermore, since special SR-IOV device drivers are needed to initially set up and manage the VFs, DVMs are needed to host these special drivers. Finally, live VM migration is more complex when the VM has direct access to a device controller. With many current VIs, without specialized add-ons, there is no support for live migration of VMs with such direct access. In other cases, the VI temporarily uses the IDVM software device sharing mechanism during migration [31].

3. Design Tradeoffs

When a device driver in a DVM fails, the connections to a device are severed. In order to tolerate device driver failure, the system can provide redundant connections to devices, such that the failure of one connection possibly degrades but does not eliminate the ability to access the device. Alternatively, the system must detect that a particular device driver in a DVM has failed and establish a new connection to the device using other new or existing DVMs.

The types of faults that affect device drivers and the resulting failure modes determine the recovery mechanisms required and the effectiveness of these mechanisms. We assume that device drivers are, for the most part, correct and functional. Hence, failures are caused by transient hardware errors or rare software bugs (Heisenbugs [9]) that occur only under particular timing and ordering of asynchronous events in the system. Thus, a recurrence of such failures after a device driver reset are unlikely. However, since faulty device drivers can potentially corrupt any part of the system (DVM) hosting them, when a driver failure is detected, the DVM is considered to be irreparable.

The following subsections discuss design options for mechanisms that provide resiliency to device driver failure. We first discuss design options for detecting driver/DVM failures. We then describe alternative system configurations that provide AppVMs with access to devices. These configurations guide the discussion of the recovery design options, which are grouped into three categories: (1) providing an alternative to the failed DVM for accessing the affected devices, (2) providing an alternative to the connection between the AppVM and the failed DVM, and (3) causing the AppVMs to begin using alternate connections to the device (failover). A taxonomy for describing these recovery design options is developed. In addition, for each category, we discuss how different recovery options affect the way fault tolerance is restored to the system after performing recovery. Furthermore, for each category, we qualitatively analyze the tradeoffs among the different recovery options based on implementation requirements, impact on service interruption latency, and resource overhead.

3.1. Failure Detection

Ideally, any incorrect behavior (outputs) from the driver/DVM should be detected. For example, this could be done using VM replication [16]. However, implementing this approach is challenging due to the need to eliminate all sources of non-determinism in the behavior of the DVM replicas dealing with timers and asynchronous interrupts. A simpler approach is to infer device driver failures by detecting only major deviation of the DVM from expected behavior, such as a crash or hang. This simple approach has the potential to result in many false negatives as well as long detection latencies that may allow significant system corruption to occur prior to detection or cause long service interruption delays. Fortunately, as shown in Section 6, the simple approach is highly effective.

Detection of drivers/DVMs misbehavior can be implemented in the VMM, AppVM, PrivVM, or the DVM itself. In some cases detection relies on knowledge of correct driver/DVM behavior and does not require any modifications to the DVM. For example, a DVM should respond to a request from an AppVM within a certain amount of time otherwise it is considered as failed [17, 19]. In other cases, the DVM is modified to change its behavior in a way that does not alter its basic functionality but leads to more accurate and/or faster failure detection. Such modifications can include direct flagging of errors, such as invoking a hypercall to the VMM on a DVM kernel panic. They can also include changes that expose behavior that is easily tracked so that deviations are simple to detect. An example of these latter modifications is running a process which wakes up periodically to force page table switches (coupled with monitoring that the switches actually occur) [19].

3.2. System Configurations

Some I/O functions that can be provided by a single physical device, may also be provided by multiple physical devices connected through multiple device controllers. For example, a system may access a network subnet using multiple NICs (physical device controllers) connected to the network switch through multiple cables (physical devices). Similarly, a storage ‘‘device’’ (RAID) may consist of multiple disks (physical devices), each connected through a separate device controller. We use the term *logical device* to encompass either a single physical device or a set of physical devices that, together, provide similar functionality to that of a physical device.

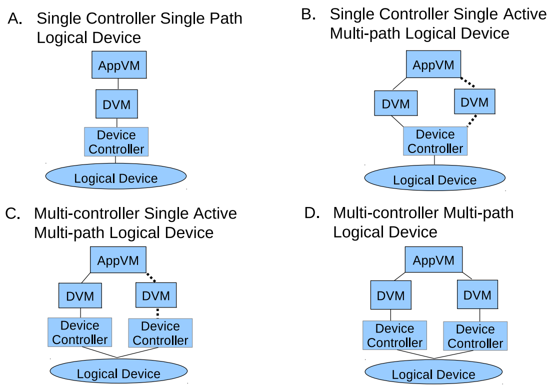
A. Single Controller Single Path Logical Device
AppVM
DVM
Device Controller
Logical Device
B. Single Controller Single Active Multi-path Logical Device
AppVM
DVM
DVM
Device Controller
Logical Device
C. Multi-controller Single Active Multi-path Logical Device
AppVM
DVM
DVM
Device Controller
Device Controller
Logical Device
D. Multi-controller Multi-path Logical Device
AppVM
DVM
DVM
Device Controller
Device Controller
Logical Device

Figure 2: System configurations for AppVM device access.

To provide an AppVM access to a logical device, there must be a path between them (Fig. 2). Components of this path include: the frontend-backend connection between the AppVM and DVM, the connection between the DVM and physical device controller, and the connection between the physical device controller and the device. Fig. 2.A shows a configuration where there is only a single path to the logical device. In Fig. 2.B, the configuration provides multiple disjoint paths (through different DVMs connected to the same physical device controller), where only one path is ever in use at a time. The configuration in Fig. 2.C is similar to the configuration in Fig. 2.B except the DVMs are connected to separate physical device controllers. An example of this configuration is access to a network subnet through only one of several NICs physically connected to the subnet. In Fig. 2.D, multiple disjoint paths are used simultaneously during normal operation, but degraded operation with fewer paths is possible. An example of this configuration is a software RAID.

3.3. DVM Replacement

Once driver/DVM failure has been detected, the system must provide an alternative to the failed DVM. This may involve the use of an existing DVM or the creation of a new DVM. The design alternatives for this, described below, are for a specific AppVM-logical device pair. A particular DVM may be part of the connections for multiple such pairs and different DVM replacement options may be used for different pairs in the system, even if they share a DVM.

- **N**o Replacement (D**N**): No DVM replacement is done upon failure (applies to the configuration in Fig. 2.D).
- **C**reate New (D**C**): A new DVM is created and booted upon failure (Fig. 2.A).
- **S**pare (D**S**): A spare DVM is booted as part of the initial system setup without access to the device controller. Upon failure, the device controller is attached to the spare DVM and initialized (Fig. 2.B).
- **A**ttached Spare (D**A**): A spare DVM is booted during system setup and a device controller is attached to it, enabling access to the controller by the spare. Until failure, the spare DVM is paused and the controller is not initialized by the spare DVM OS (Fig. 2.B).
- **P**rimed Spare (D**P**): Similar to DA, however, the device driver undergoes some amount of initialization as part of system setup [19]. Upon failure, the controller state may be inconsistent with the driver in the spare DVM, requiring the driver to perform some initialization operations in order to make the controller usable (Fig. 2.B).
- **R**eady Spare (D**R**): During system setup, a spare device controller is attached to the spare DVM and is fully initialized. The spare DVM *may* be paused until failure is detected. The device controller is only accessible by the spare and remains idle until failure (Fig. 2.C).

Fault-tolerance restoration: With all the alternatives except DC, once recovery from failure is complete, a new spare or redundant DVM must be booted to enable handling of future failures. With DN schemes, the redundant connection to the logical device must be restored. With DP schemes, in order to minimize recovery latency, the spare DVM OS partially initializes the device driver and, as a result, the device controller, in advance of recovery. Following recovery, when a new spare is booted, it must perform this driver initialization without interfering with the controller that is in use by the other DVM. Hence, the spare DVM must be prevented from accessing the actual controller by redirecting those accesses to an emulated controller, as done with full system virtualization.

Tradeoffs: *Implementation Requirements:* Of the six approaches, only DA and DP are likely to require modification to the *virtualization infrastructure* (VI). With both these options, a device controller is attached to the spare DVM and may also be attached to other (active) DVMs in the system. The VI typically allows controllers to be directly accessible to only one VM at a time and modification are needed to remove this restriction.

With DP, a device emulator must be provided by the VI to allow the spare DVM to initialize its device driver during fault-tolerance restoration. Device driver modifications may be necessary to perform a ‘‘minimal reset’’ of the controller upon recovery to restore the controller to a sane state [19]. As discussed in Subsection 3.5, the DN and DR options rely on support in the AppVM.

Service Interruption: The longest interruption latency is with DC, at it requires a new DVM to be booted. DS, DA, and DP use an existing spare, eliminating this boot time. DA and DP further shorten interruptions by performing some setup (attachment, initialization) for the controller prior to recovery. DR and DN minimize interruption as there is already a replacement or degraded connection ready for use.

Resource Overhead: Maintaining a spare DVM (DS, DA, DP, DR) requires additional memory. This overhead can be greatly reduced by using content-based memory sharing in the VMM [10].

3.4. AppVM-DVM Connections

When a DVM fails, AppVM connections to it are lost. The following are design options for providing replacements for those lost connections.

- **N**o Replacement (C**N**): Redundant connections between the AppVM and logical device (Fig. 2.D) degrade so there is no need to replace AppVM-DVM connections.
- **C**reate New (C**C**): As part of recovery, a new connection, either created from scratch or using existing data structures, is established. This can apply to all the alternatives in the previous subsection except for DN.
- **S**pare (C**S**): A connection is established but unused prior to recovery. This can be used with the DS, DA, DP and DR alternatives of the previous subsection.

Fault-tolerance restoration: With the CN and CS options, a new redundant or spare connection must be created following recovery.

Tradeoffs: *Implementation Requirements:* With CC, a new connection must be established and the old connection torn down without affecting applications running in the AppVM. VIs that support VM migration have the ability to perform such AppVM-OS-transparent connection migration, with support in the AppVM typically confined to the frontend drivers. Hence, no or very few VI modifications are needed. On the other hand, CS requires support for multiple connections to different DVMs for the same device. With existing VIs, this is likely to require modifications to the frontend drivers (see Subsections 3.5 and 4.4).

Service Interruption Latency: With CS and CN, a replacement or degraded connection is ready for use. CC results in longer interruption latency since a new connection is established during recovery. Recovery latency with CC can be reduced by changing the VI to reuse for the new connection data structures of the old connection [17, 19].

Resource Overhead: CS and CN require a small amount of memory for the spare or redundant connection. With Xen, a frontend-backend connection consists of a 4KB shared page and a small amount of bookkeeping information.

3.5. Failover at the AppVM

Once a DVM failure is detected, the AppVM must switch away from using the path to the logical device that involves the failed DVM. The relevant design alternatives are as follows:

- **T**ransparent (F**T**): The AppVMs are not involved in recovery. The VMM redirects the existing AppVM-DVM connection and any pending requests to an operational DVM [19].

- **F**rontend (F**F**): The AppVM is notified of DVM failure and pointed to an operational DVM. The frontend in the AppVM switches the device connection to the operational DVM.
- **B**uiltin Fault-Tolerance (F**B**): This requires an AppVM OS with a builtin mechanism for using multiple physical devices to provide a highly reliable logical device. The key examples of this are software RAID for storage and bonding [21] for network access. Upon DVM failure, the AppVM is notified of the loss of one of its redundant connection to the logical device (Fig. 2.D).

Fault-tolerance restoration: With the FT option, following recovery the VMM must be updated with information about a new operational spare DVM that can be used for the next failover. Similarly, with the FF and FB options, the VMM must maintain up-to-date information about AppVM/DVM pairings so that DVM failure notifications can be delivered to all relevant AppVMs.

Tradeoffs: *Implementation Requirements:* With both FT and FF schemes, AppVM requests waiting for responses from the failed DVM must be resubmitted to the new DVM or dropped. FT schemes require modifications to 1) the VMM, to forward AppVM requests destined for the failed DVM to the new DVM; and 2) the DVM backend driver, to seamlessly plug itself into the connection with the frontend driver in the AppVM. As discussed in Subsection 3.4, with VIs that support live migration, frontend drivers already have the requires support for CC schemes. Hence, many FF schemes do not require any additional changes to the frontend driver. However, existing frontend drivers typically do not support maintaining a spare connection, as used in CS schemes. Hence, for CS schemes, FF schemes do require changes to the frontend drivers. An example of such changes for the Linux network frontend driver is presented in Subsection 4.4.

Service Interruption Latency: With FT, there is no need to notify the AppVM of a failure, potentially leading to shorter interruptions than with FF or FB. However, in comparison to other operations required for recovery, the latency of failure notification is insignificant so all these options result in comparable interruption latencies.

4. Implementation Challenges

This section discusses key challenges and ways to address them in implementing DVM resiliency mechanisms and ensuring their reliable and fast operation. These challenges include preventing faulty device controllers from corrupting system memory or the devices themselves, preventing the failed driver/DVM from blocking interrupts, overcoming deficiencies in error detection and handling mechanisms with existing VIs, and supporting frontend network drivers with multiple network backends. The first two challenges are not specific to any particular VI or recovery approach. The remaining challenges are relevant to only a subset of mechanisms or to characteristics of only a subset of VIs. Our implementations of the various DVM resiliency mechanisms are based on the solutions presented in this section.

4.1. Mitigating Memory and Device Corruption

A faulty device driver can cause a device controller to corrupt system memory as well as corrupt device state by, for example, overwriting blocks on disks or transmitting packets that interfere with network operation. The damaging actions of the controller can continue even after the DVM hosting the driver crashes, until the controller is re-initialized by a working driver. While such actions cannot be completely blocked, mechanisms for recovery from driver failures should be designed to reduce the likely impact of this problem. Device corruption can also occur when a faulty driver drops (does not perform) certain operations. For example, with block devices, this can leave the file system in an inconsistent (corrupt) state. Mitigating this problem requires minimizing the probability of lost requests when a DVM fails.

Some systems include a hardware mechanism, called an IOMMU [2], that controls which parts of the host memory are accessible by each device controller. A recovery mechanism can use the IOMMU to block a potentially corrupted controller from overwriting any part of memory. For systems without an IOMMU, this problem can be partially mitigated by ‘‘quarantining’’ the memory locations likely in use by the corrupted device controller at the point of failure until the controller can be re-initialized. These locations include DMA regions belonging to the faulty DVM and/or AppVMs accessing the device. For DVM memory locations, the ‘‘quarantine’’ can be performed by delaying the destruction of the failed DVM until the controller is re-initialized, hence not freeing the DVM’s memory for other use [19]. Similarly, for memory locations in AppVMs, the frontend driver in the AppVM can delay the release of memory that is potentially in use by the device controller until it determines that the controller has been re-initialized. For example, such notification can be the establishment of a connection with a new DVM that has taken over the relevant device controllers.

An example of ‘‘device corruption’’ that can be mitigated deals with network operation. This problem occurs when using a recovery approach that fails over to a spare DVM controlling a spare NIC. Recovery may fail to restore network connectivity if the NIC controlled by the failed DVM continues to transmit packets on the network that cause the network switch to direct traffic destined for the AppVM to the failed DVM’s NIC. To quickly mitigate this problem, the AppVM frontend driver can periodically (e.g., every 10ms) broadcast ARP reply packets through the spare NIC from the time failure is detected until the NIC controlled by the failed DVM is re-initialized. To reduce reliance on this imperfect mechanism, immediately after failover, the failed DVM’s NIC can be mapped into the PrivVM’s memory address space and the PrivVM can then disable the transmit module of the NIC.

When a DVM fails, pending requests (operations) from/to AppVMs may be lost. This is generally not a problem with network devices, since the loss of packets is expected and dealt with by network protocols. However, as discussed above, with block devices, lost requests can result in device corruption. Hence, for block devices in configurations that involve a single controller and a single physical device (figures 2.A and 2.B), recovery should include resubmission of requests pending to the failed DVM to the replacement DVM. Xen's live VM migration mechanism performs this request resubmission for transparent connection migration. This capability is directly usable for DVM recovery.

4.2. Preventing Interrupt Blocking

Depending on how the system is configured, a faulty DVM can end up blocking the delivery of interrupts from device controllers to non-faulty DVMs. This problem can occur with two possible configurations: 1) multiple device controllers, assigned to different DVMs, share a single interrupt line (number); and 2) multiple DVMs are assigned to the same processor.

With some VI implementations (e.g., Xen), the VMM acknowledges an interrupt only after the DVM assigned to handle the interrupt informs the VMM that the interrupt has been handled. With the first configuration above, interrupts from devices that share an interrupt number are delivered to all the DVMs associated with those devices. In this case, the VMM waits for responses from *all* these DVMs before acknowledging the interrupt. A faulty DVM may not respond, preventing the VMM from acknowledging the interrupt. As a result, additional interrupts from other devices, controlled by non-faulty DVMs, are blocked until the faulty DVM is destroyed. This can take a long time since, as discussed in Subsection 4.1, there are reasons to delay the destruction of the faulty DVM.

The problem described in the previous paragraph cannot occur in system configurations that avoid sharing of interrupt numbers by multiple controllers. This can be achieved by using message signaled interrupts (MSIs) [5, 24], or, where possible, by simply changing the physical locations of one or more device controllers. Alternatively, the VMM can be modified to acknowledge interrupts on behalf of a DVM immediately upon detecting a DVM failure and prevent additional interrupts from being delivered to the failed DVM.

Typically, when an interrupt is being serviced, no interrupts of equal or lower priority can preempt it. Hence, with the second configuration above, a faulty DVM that does not acknowledge interrupts may end up blocking lower-priority interrupts from being delivered to the VMM, and thus, to the non-faulty DVMs. To prevent this problem, the DVMs can be pinned to different processors, or, alternatively, the VMM can be modified as described above.

4.3. Enhancing Failure Detection in AppVMs

Existing operating systems and their device drivers often include some mechanisms to detect and recover from device and/or driver failures. However, in many cases these mechanisms need to be supplemented in order to facilitate rapid recovery from DVM failures. With Linux, one example of this situation is with the bonding [21] driver. By default this driver uses polling with a 100ms interval to detect network disconnect, resulting in slow detection and thus long service interruptions. Another example is the Linux software RAID driver that relies on error codes propagated from the disk device driver. With the IDVM architecture, when a DVM fails, these error code are not received, preventing the RAID mechanism for functioning correctly.

The above problems can be solved using small changes to the VMM and frontend drivers. These changes allow the VMM to deliver virtual interrupts to frontend drivers in the AppVMs when a DVM failure is detected. Upon receipt of these interrupts, the frontend drivers can directly invoke the appropriate failure handling operations, such as switching to a different network device when Linux bonding is used. When RAID is used with block devices, the frontend drivers can initiate the cancellation of all pending I/O requests (return error codes) destined for the failed DVM, thus triggering the appropriate builtin failure handling in the AppVM. With a Linux AppVM and Xen, without the latter action, the RAID driver waits indefinitely for all pending requests to complete and fails to perform the failover.

4.4. Frontend Supporting Multiple Backends

Some OSs provide native support for managing multiple NICs to provide fault tolerance (e.g., Bonding [21] in Linux). With such AppVM OSs, configurations with multiple NICs (Figure 2.D) are natively supported. However, there are some OSs, such as pre-2012 versions of Windows, that require driver-level implementation to provide similar functionality. With such OSs, the network frontend driver can be modified to manage the failover between two backend drivers: an active and a spare (Figure 2.C). The frontend exports a *single* network interface to the AppVM kernel. When the DVM hosting the active backend fails, the frontend is alerted and switches to using the spare. Once the failed DVM is replaced, a new frontend-backend connection to the replacement DVM is formed and it becomes the active backend. All of this is done transparently to the rest of the AppVM kernel. We refer to this mechanism as *MultiNetIf*.

With MultiNetIf, when the network frontend driver switches from one connection to another, network traffic with the same endpoints flows through a different NIC with a different MAC. In order for recovery to be successful, immediately following the connection switch, an ARP reply packet must be broadcasted by the network frontend driver through the newly active connection to update the routing table at the network switch.

5. Evaluation Methodology

This section discusses the experimental setup used to evaluate the different mechanisms for resiliency to device driver failure. The following subsections present the recovery schemes (points in the design space) that we evaluated, details of the fault injection campaign, workload, latency measures, and failure detection mechanisms employed.

Experiments were run on systems based on the Intel Core-2 and Nehalem processors. The systems were interconnected by a 1Gb switched ethernet network using Intel E1000 NICs. The systems used SATA hard-drives connected through the JMicron 20360/20363 PATA/SATA controllers. We also used a USB flash drive connected through a NEC USB controller. The VMM used was Xen 3.3.0 with all the VMs running XenoLinux kernel version 2.6.18.8.

5.1. Recovery Schemes Evaluated

The taxonomy presented in Section 3, defines a 3-dimensional, 54-point implementation space. However, the number of implementable schemes is actually much smaller. DN, CN, and FB are only compatible with each other, in the DN-CN-FB configuration shown in Figure 2.D. DC and CS are incompatible as there cannot be a spare connection without a spare DVM.

Recovery mechanisms using DP and FT are presented in [19], and are not discussed further in this work. FT requires changes to the VI and does not have a significant advantage over simpler schemes (see Subsection 3.5). As discussed in Subsection 3.3 and based on our prior experience [19], DP is undesirable since it involves significant implementation complexity and operational problems.

Table 1. Practical, implementable, deployable recovery mechanisms. Names are given to recovery mechanisms we have implemented and evaluated.

Recovery Mechanism	Block Implementation Name	Network Implementation Name
DR-CS-FF	--	MultiNetIf
DR-CC-FF	--	--
DN-CN-FB	RAID	Bonding
DC-CC-FF	RebootBlk	RebootNet
DS-CC-FF	SpareBlk	SpareNet
DS-CS-FF	--	--
DA-CC-FF	AttSpareBlk	AttSpareNet
DA-CS-FF	--	--

Eight techniques that can be practically deployed are listed in Table 1. Five are selected to implement and evaluate in this work. In Table 1, names are assigned to specific scheme/device combinations for ease of reference. The selection of schemes is made in view of what has been evaluated in prior work and to ensure coverage of key design choices and system configurations. Since DR-CC-FF has been evaluated by Jo et al. [17] we do not evaluate that scheme here. Instead we implement and evaluate DR-CS-FF, which is very similar. DR-CC-FF and DR-CS-FF cannot be used with storage devices (disks) unless they can be connected to two device controllers simultaneously. We do not evaluate DS-CS-FF and DA-CS-FF as they are very similar to DS-CC-FF and DA-CC-FF, respectively, and do not provide any significant advantages. Specifically, the spare AppVM-DVM connection in DS-CS-FF and DA-CS-FF eliminates the service interruption time to form the connection as part of recovery. However, this time, which ranges from tens of milliseconds to a few hundred milliseconds, is eclipsed by the multiple seconds needed to reboot the DVM. While DC-CC-FF has been evaluated to some extent in other works [7, 20, 14], we include it in our evaluation as it serves as the foundation for other schemes.

5.2. Fault Injection Campaigns

Software-implemented fault injection is used to evaluate the effectiveness of the different DVM recovery mechanisms. Faults are injected in the network, block, and USB drivers in the DVMs. This is done using our *Gigan* fault injector that resides in the VMM and is capable of non-intrusively injecting into VMs [18].

A fault injection campaign consists of multiple runs. Each run takes 40s to 150s. In each run, a single fault is injected into a DVM at a random time 10s to 16s from the beginning of the run. An injection is triggered by a breakpoint in the target driver code. When the breakpoint fires, it causes the current instruction or a random register to be modified. Results are reported (counted) only for *activated* faults — the breakpoint fired and caused a fault to be inserted. If a fault is activated, the next run uses new (i.e., fault-free) instances of the DVM and the storage to which it had access. This prevents corruptions in one run from affecting future runs.

There are two types of campaign workloads: *static* and *dynamic*. The former are used to evaluate recovery latency and involve a fixed number of AppVMs booted at the beginning of the campaign. Results are collected with one, two, or three AppVMs. The effect of the number of AppVMs on the recovery latency can thus be evaluated.

Dynamic workload campaigns are used to stress the correctness and effectiveness of the recovery mechanisms. At the beginning of each run, an AppVM that is off (not booted), is booted and runs the benchmark with a probability of 0.7. An AppVM that is on (booted), is shut down with a probability of 0.3. These campaigns are designed to stress the ability to recover a DVM while it is being used by an application in an AppVM as well as during the AppVM-DVM connection establishment and tear-down phases.

After every recovery, the ability to handle future DVM failures is restored. For example, if a DVM failure occurs and the recovery mechanism uses a spare DVM, a new spare DVM is booted and prepared for the next DVM failure. Similar restoration of the fault-tolerance capability of the system is performed for other recovery mechanisms.

Table 2 shows the types of faults injected. As discussed in [25], some of these faults simulate typical programming errors. For the first seven fault types (*software faults*), we used the fault injection tool in [29] to generate a list of injections to perform by our Gigan fault injector. The random code faults further stress the device drivers. Register faults are included to emulate the effects of transient hardware faults.

Faults are injected into the E1000 NIC driver, the Linux AHCI SATA/SCSI block driver, and the Linux EHCI USB driver. We used the Xenoprof sampling profiler [22] to identify the functions in those drivers that are most frequently executed [11]. Only instructions in those functions are

Table 2. Types of faults injected

Fault Type	Description
NOP	Replace random instrs with NOP
Destination	Flip random bit in dst operand of instr
Source	Flip random bit in random src operand of instruction
Branch	Replace branch instrs with NOP
Loop	Reverse directions of loops
Pointer	Flip random bit in operand of memory access instructions
Interface	Use bad function arguments
Random code	Flip random bit in a random byte of an instruction
Register	Flip random bit in a random general purpose register, stack pointer, or program counter
Crash	Cause immediate crash of target VM

possible targets for fault injection. To ensure that register injections occur while executing driver code, the same breakpoints used for code injection are used to trigger the register injections.

The outcome of each injection run is classified as a DVM crash, a DVM hang, silent DVM failure, or non-manifested. A DVM crash occurs when the DVM's kernel panics or the DVM is destroyed by the VMM. A DVM hang occurs when the DVM stops responding with no explicit report of a crash. A silent DVM failure occurs when no hang or crash are detected but the application is unable to complete successfully (see Section 5.3). Non-manifested means that no errors are observed.

5.3. Benchmarks

Most of our evaluation is done using two micro-benchmarks, designed to facilitate accurate measurement of recovery effectiveness and latency: *netbench* and *filesysbench* that exercise the device drivers for the network and block devices, respectively. As discussed in Subsection 6.6, we also use a real-world application — the Apache web server.

Netbench is a user-level *ping* program that consists of two processes: one in an AppVM (*VM host*) and another on a separate physical machine (*PM host*). Every 1ms the PM host transmits a UDP packet to the VM host, which, upon receiving this packet, transmits a UDP packet back to the PM host. The inter-arrival time of successive packets on the PM host is used for measuring the latency of any network service interruption. If a DVM error is detected, a recovery failure is recorded if: (1) there is a service interruption of more than 30s at any time during a run, or (2) at any time after the recovery procedure completes, there is a 1s interval during which the rate of packet reception at the PM host drops by more than 10% compared to the rate during normal operation. A ‘‘silent failure’’ is recorded if no DVM errors are detected and, at any time during a run, there is a 1s interval during which the rate of packet reception at the PM host drops by more than 10% compared to the rate during normal operation.

Filesysbench runs in an AppVM and stresses the block device by creating and removing directories as well as creating, removing, and copying 1MB sized files on a disk hosted by a DVM. To ensure that the device driver in the DVM is exercised, caching of block and filesystem data by the AppVM's OS is prevented. This is accomplished by using the O_DIRECT flag to *open*, mounting the device with the synchronous flag, and dropping all filesystem caches after each filesystem operation. The time to perform each filesystem operation, e.g., read/write 1MB file or create/delete directory, is recorded in order to be used for measuring the block service interruption latency during recovery. If a DVM error is detected, a recovery failure is recorded if: (1) the application reports errors (failure of I/O operations), or (2) at the end of the run the files and directories created differ from a ‘‘gold’’ image. A ‘‘silent failure’’ is recorded if no DVM errors are detected and one or both of the above conditions for ‘‘recovery failure’’ are met.

5.4. Latency Measures

When a device driver in a DVM fails, AppVMs' access to the device controlled by that driver is lost. A key measure of mechanisms for resilience to device driver failure is the duration of this loss of access. We refer to this time as the *service interruption latency*. This latency consists of the failure detection latency followed by the recovery latency. This subsection precisely defines these latencies and discusses how they are measured in our experiments.

The key ‘‘events’’ in the fault's ‘‘life time’’ without any resiliency mechanism are: (1) the fault occurs (is injected), (2) the fault is activated — the faulty value is read, and (3) the fault is manifested — system deviates from correct behavior. With a resiliency mechanism, there are two additional key events: (A) the fault is detected, and (B) normal correct operation is resumed. In general, event (A) can occur at any time following event (1) and is followed by event (B) at any time thereafter. With the detection mechanisms considered in this work, event (A) can only occur after event (3).

For software and code (Subsection 5.2) faults, detection latency is measured between events (2) and (A). For register faults, it is measured between events (1) and (A) [12]. For crash ‘‘faults’’, since events (1) and (A) are the same, detection latency is zero. Recovery latency is measured between events (A) and (B). Service interruption latency is measured for a network device as an increase in the time between successive packets with *netbench* (Subsection 5.3). For block devices, it is an increase in the time of a single block read/write operation with *filesysbench*. These measured interruption latencies roughly, but not precisely, correspond to the time between events (3) and (B).

5.5. Failure Detection

Two types of fail-stop DVM failures are detected: crashes and hangs. Crashes are detected when 1) the crash handler in the VM's OS makes a hypercall to the VMM or 2) the VMM responds to illegal VM activity by killing the VM.

Hangs are detected using two mechanisms. The first mechanism is triggered when a VM with multiple runnable processes fails to perform context switches [19]. 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 (400ms), a hang is identified. 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 mechanism is triggered when there are requests placed on the shared rings (part of the frontend-backend connection) for a DVM, and none of the requests are removed for processing within some time frame. An additional trigger is if a response to the request is not generated within a short time after the request has been consumed by the DVM. The response generation time is dependent on the load of the DVM and the latency of the physical device. To minimize false positives, the time to pick up a request for network devices and block devices (disk and USB flash drive) is 50ms and 200ms, respectively. The time to generate a response for network devices, disks, and USB flash drive is 50ms, 200ms, and 1s, respectively. These mechanisms build on the mechanism used in [19] and are similar to the mechanism discussed in [17].

It should be noted that a hang detector that monitors timely response generation for requests from AppVMs is sufficient for correctly detecting a failed DVM. However, in many cases, the additional detectors provide faster detection time, and thus overall shorter interruption latency.

6. Experimental Results

The results of the experimental evaluation of the DVM recovery schemes described in Subsection 5.1 are presented and discussed in this section. The recovery success rates and recovery latencies are discussed in Subsections 6.1 and 6.2, respectively. The impact of the number of AppVMs on the recovery latency is analyzed in Subsection 6.3. Subsection 6.4 analyzes the impact of DVM failure detection latency on the overall service interruption time. The effect of sharing physical CPUs among multiple VMs on recovery latency is discussed in Subsection 6.5 and Subsection 6.6 demonstrates how well these recovery mechanisms work on a real-world application (Apache web server).

6.1. Recovery Effectiveness

The most important characteristic of a recovery mechanism is the success rate — the probability that applications complete successfully despite a component failure. Table 3 shows the impact of different types of faults (register, code injection, SW — simulated programming errors) on different device drivers, as well as the resulting recovery rates. There are several key observations: 1) if the failure is detected, there is a very high (>93%) probability of successful application completion; 2) for each device type, the rate of successful recovery from *detected* DVM failure is not dependent on the fault type or the recovery mechanism (the variations shown in the table are not statistically significant); 3) the main causes of application failures are undetected (silent) device driver failures; 4) the rate of silent failures is lowest for register faults and highest for SW faults.

Table 3. Injection outcome and recovery effectiveness of different DVM recovery schemes. DVMs host network and/or block device drivers. Campaign: *Dynamic AppVM* campaign with 3 AppVMs. Three fault types: software faults (SW) consist of injections selected from the first seven injection types in Table 2, random code (code), and register (reg).

Device Type	Recovery scheme (Injection type)	# Activated Injections	% Not manifested / Activated	Manifested			% Application completion / Detected (crash+hang)	% Application completion / Manifested
				% DVM crash	% DVM hang	% Silent failure		
Network	MultiNetIf (SW)	428	41.8	65.9	14.9	19.3	99.5	80.3
	Bonding (SW)	429	41.0	67.2	13.0	19.8	99.5	79.8
	RebootNet (SW)	425	40.9	65.7	14.3	19.9	100.0	80.1
	SpareNet (SW)	420	42.1	65.0	13.6	21.4	100.0	78.6
	AttSpareNet (SW)	461	42.3	66.2	14.7	19.2	100.0	80.8
	MultiNetIf (code)	647	33.1	85.0	4.6	10.4	99.7	89.4
	Bonding (code)	648	33.0	84.1	5.3	10.6	99.7	89.2
	RebootNet (code)	645	32.6	83.9	5.5	10.6	100.0	89.4
	SpareNet (code)	645	33.6	83.2	6.5	10.3	100.0	89.7
	AttSpareNet (code)	644	33.9	84.0	4.5	11.5	100.0	88.5
	MultiNetIf (reg)	600	72.3	88.0	9.0	3.0	100.0	97.0
	Bonding (reg)	598	71.9	89.9	7.7	2.4	100.0	97.6
	RebootNet (reg)	596	71.6	89.9	7.1	3.0	100.0	97.0
	SpareNet (reg)	600	71.3	88.4	8.7	2.9	100.0	97.1
	AttSpareNet (reg)	597	72.9	90.1	7.4	2.5	100.0	97.5
Block (disk)	RAID (SW)	491	40.3	62.5	21.5	16.0	96.7	81.2
	RebootBlk (SW)	491	35.4	58.7	23.7	17.7	95.8	78.9
	SpareBlk (SW)	493	35.9	58.9	23.7	17.4	94.3	77.8
	AttSpareBlk (SW)	491	35.6	58.2	24.7	17.1	93.1	77.2
	RAID (code)	469	36.5	79.5	13.8	6.7	97.8	91.3
	RebootBlk (code)	470	35.5	77.9	15.5	6.6	95.4	89.1
	SpareBlk (code)	471	34.6	77.6	16.2	6.2	94.1	88.3
	AttSpareBlk (code)	473	35.1	78.2	16.6	5.2	95.2	90.2
	RAID (reg)	663	67.0	89.0	9.1	1.8	97.7	95.9
	RebootBlk (reg)	652	65.5	87.6	8.9	3.6	99.1	95.6
	SpareBlk (reg)	657	64.4	83.8	12.4	3.8	99.1	95.3
	AttSpareBlk (reg)	682	64.2	86.5	9.8	3.7	98.7	95.1
Block (USB)	AttSpareBlk (SW)	519	43.2	52.2	41.7	6.1	96.8	90.8
	AttSpareBlk (code)	527	30.2	79.9	16.8	3.3	98.0	94.8
	AttSpareBlk (reg)	654	64.8	75.7	23.0	1.3	99.6	98.3

The recovery rate from *detected* DVM failures is higher for network devices compared to block devices. The main reason for this difference is that, with network devices, a fault can cause a few packets to be dropped without a significant impact on the overall packet reception rate, thus not leading to an ‘‘application failure’’ (see Section 5.3). However, a corruption of the block driver can cause the file system or a single file to become corrupted, leading to a wrong result

Table 4. Recovery latency of different recovery mechanisms for DVMs controlling network and block devices. During normal operations, with *netbench*, the median inter-arrival time of a *ping* message on the PM host is 1.03ms. With *filesysbench* accessing hard disks, the median time for reading/writing a 1MB file is about 8ms. When accessing a USB flash drive, the median time for reading and writing a 1MB file is 70ms and 600ms, respectively. Measurement is taken using one AppVM.

Recovery scheme	Min (ms)	1st Quartile (ms)	Median (ms)	3rd Quartile (ms)	Max (ms)
RebootNet	11007	11268	11409	11584	12641
SpareNet	3546	3650	3774	3908	5017
AttSpareNet	3209	3347	3461	3645	5769
MultiNetIf	0	0	0.14	0.4	0.68
Bonding	0	0	0.15	0.34	0.66
RebootBlk	10654	10710	10727	10744	10810
SpareBlk	4072	4115	4128	4143	4373
AttSpareBlk	3641	3687	3702	3721	3794
RAID	0	6.89	26.25	34.84	56.59
AttSpareBlk (USB)	7398	7461	7779	8216	10346
AttSpareNet	6376	7040	7137	7303	9193
AttSpareBlk	4028	4085	4134	4162	4243
MultiNetIf	0	0.18	0.63	1.23	3.0
RAID	0	0.87	25.76	35.48	53.81

which is identified as an ‘‘application failure.’’

The relatively low rate of silent failures due to register faults is a consequence of the fact that most of these faults result in DVM crashes, usually due to an illegal address dereference [18]. On the other hand, SW faults are more likely to change the functionality of the device driver, causing it, for example, to incorrectly skip the processing of I/O requests. One third of SW faults that resulted in silent failures were due to the replacement of an instruction with a NOP.

6.2. Recovery Latency

To measure the recovery latency (see Subsection 5.4) of different mechanisms, we use a configuration that consists of a single AppVM. In these experiments, a DVM crash is forced.

Table 4 shows the recovery latencies with different recovery schemes. With our setup, a DVM reboot takes approximately 10s. Recovery is, of course, slowest, when it requires such a reboot. Using a spare DVM eliminates the reboot time; but recovery latency still includes the time to re-initialize the device controller. Re-initializing the network or block controllers takes 2.5-3.5 seconds. The USB controller requires 6-7 seconds to probe and initialize the USB flash drive. Using a spare DVM with an attached device (AttSpare) removes the overhead of transferring the device controller to the spare DVM, thus reducing the recovery latency. Schemes using redundant devices (MultiNetIf, Bonding, RAID) have the shortest recovery latency since they do not require rebooting the failed DVM or re-initializing the device.

Table 5. Impact of No. AppVMs on recovery latency.

Recovery scheme: Network/ Block	# AppVMs	Recovery latency (ms) (with 95% confidence interval)	
		Network	Block
RebootNet / RebootBlk	1	11467 ± 30	10727 ± 5
	2	11518 ± 20	10876 ± 16
	3	11575 ± 18	11072 ± 9
MultiNetIf / RAID	1	0.14 ± 0.03	22.74 ± 1.8
	2	0.14 ± 0.02	40.45 ± 3.0
	3	0.18 ± 0.01	62.49 ± 3.8

The number of devices controlled by a DVM can affect its recovery latency. The bottom of Table 4 shows the recovery latency, with two different recovery schemes, for a DVM simultaneously controlling a network and a block device. We have run experiments with other combinations of recovery schemes but omit those results since the results in Table 4 are representative of the behaviors observed. With the AttSpare-Net/Blk scheme, which does not rely on redundant controllers, recovery latency is significantly longer than when the DVM controls a single controller. This is due to the DVM kernel serializing the re-initialization of the device controllers and creation of the virtual device backends. Specifically, the network controller is initialized only after the block controller. This example demonstrates that, for schemes without redundant controllers, recovery latency is minimized by assigning each controller to a separate DVM. The results with the MultiNetIf and RAID schemes, demonstrate that, for schemes with redundant controllers, the hosting of two controllers on the same DVM has very little impact on recovery latency. In this case, recovery does not involve device initialization or the creation of a backend.

6.3. Impact of No. AppVMs on Recovery Latency

In most deployments, a DVM is shared by multiple AppVMs. If DVM recovery involves specific operations with respect to each of the AppVM serviced by the DVM, recovery latency may be affected by the number of AppVMs sharing a DVM. To evaluate this effect, the experiments in Subsection 6.2 are extended to compare the recovery latency with one, two, and three AppVMs. The recovery latency is measured for each AppVM in a run. For each recovery mechanism and number of AppVMs, the campaign consists of approximately 200 runs and the mean recovery latency across all the AppVMs in all those runs is computed. Due to length limitations, we do not show the results from all the experiments performed. Table 5 shows a representative sample of those results.

For a DVM controlling a network device, there is little impact of the number of AppVMs on the recovery latency. With schemes involving rebooting the DVM or a spare DVM, recovery requires serially establishing the frontend/backend connections to each of the AppVMs. Hence, the time for establishing these connections does increase with the number of AppVMs. However, this time, of approximately 30ms per

AppVM, is hidden with respect to recovery latency. This is due to the fact that, in parallel with the frontend/backend connection establishment, the network device is initialized, taking approximately 3sec. With the RebootNet scheme, the small increase in the recovery latency with more AppVMs is due to checks performed by Xen when a VM is booted on each of the virtual disks in the system. With our setup, the number of these virtual disks increases with the number of AppVMs. For schemes that use spare devices (MultiNetIf and Bonding), recovery does not involve special operations by the DVM for each AppVM for the frontend/backend connections. Hence, there is essentially no impact of the number of AppVMs on recovery latency.

For a DVM controlling a block device (disk), there is a larger impact of the number of AppVMs on the recovery latency. With Reboot or Spare DVM schemes, unlike the operation with network devices, the frontend/backend connection establishment is performed only after the block device has been initialized. Hence, the connection establishment delay directly increases the recovery latency. With RAID, the large impact of the number of AppVMs is due to the operation of the RAID driver in the AppVM when the RAID array is reconfigured due to a failed disk. Specifically, the RAID driver flushes all pending write requests to disk. The latency of this flushing operation is dependent on the number of outstanding write requests, and thus dependent on the number of AppVMs.

6.4. Service Interruption Latency

As explained in Subsection 5.4, the service interruption latency is the sum of the detection and recovery latencies. This subsection is focused on experimental results showing the relationships among these three latencies for different fault types and different manifestations of these faults.

Figures 3 and 4 present a sample of our results, showing the detection, recovery, and interruption latencies per run. The code faults results are not shown since they are very similar to the results from software (SW) faults. For both network and block devices, recovery latency is independent of the manifestation of the fault (crash or hang). For both devices, when the fault causes a crash, recovery latency usually determines the interruption latency since detection is almost immediate. When the fault causes a hang, interruption latency is usually determined by the detection time since our hang detection mechanism is inherently slow. There are significant variations (orders of magnitude) among runs with respect to each of the three latencies measured. However, the overall conclusion is clear: the most important step towards reducing interruption latency would be the development of faster hang detection techniques.

6.5. Sharing CPUs and Service Interruption Latency

The results presented so far are for system configurations where each VM is assigned one virtual CPU (VCPU) that is pinned to a dedicated physical CPU (PCPU). However, for efficient resource utilization, in most deployments, multiple VCPUs often share a PCPU. For example, with the

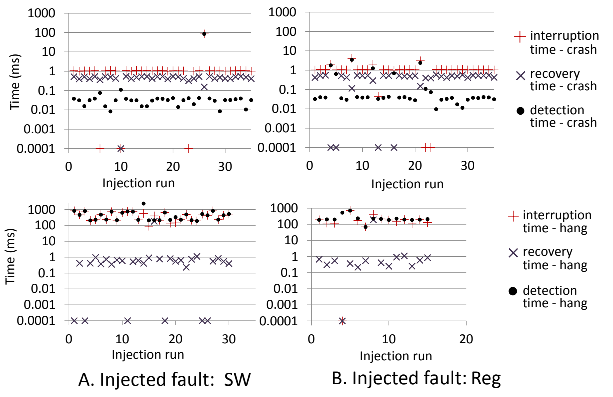

Figure 3: Service interruption with *netbench* using MultiNetIf recovery. SW and register faults. Top graphs: faults cause crashes. Bottom graphs: faults cause hangs.

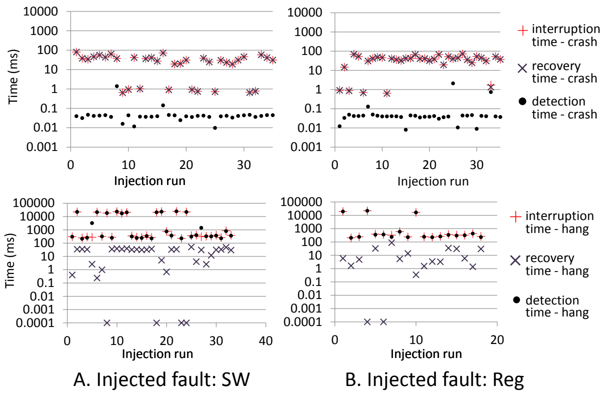

Figure 4: Service interruption with *filesysbench* using RAID recovery. SW and register faults. Top graphs: faults cause crashes. Bottom graphs: faults cause hangs.

configuration used in this work (Figure 1), the PrivVM is idle most of the time. Hence, it is wasteful to dedicate a PCPU to the PrivVM. This subsection focuses on the impact of such sharing on the duration of service interruption due to DVM failure and recovery.

While sharing PCPUs among VMs results in more efficient utilization, such sharing also affects how quickly DVM failures can be handled, and thus, the duration of the resulting service interruption. Specifically, if several VMs involved in the recovery process share a PCPU, recovery may be slower since all these VMs will have to execute sequentially. Furthermore, for recovery schemes employing redundant or spare DVMs, restoration of system fault tolerance requires booting a new DVM. If the new DVM is booted on a PCPU shared by an active AppVM or DVM, the latencies of ongoing I/O operations are likely to increase.

Since the number of combinations of recovery schemes and sharing configurations is very large, we do not attempt an exhaustive exploration of all possibilities. Instead, with both network and block devices, we evaluate selected schemes and configurations that illustrate the effects of PCPU sharing. The system setup includes two AppVMs, both running *netbench* or both running *filesysbench*, for evaluations with network and block devices, respectively. In each case, a DVM is crashed and the service interruption latency is measured. Our evaluation includes recovery schemes based on a spare DVM (SpareNet and SpareBlk), since with these schemes recovery and fault tolerance restoration involve the PrivVM, DVM, and AppVM. We also evaluate schemes that provide the fastest recovery using multiple devices: MultiNetIf and RAID. The PCPU sharing configurations evaluated are: (1) PrivVM shares with an AppVM or with a DVM, (2) DVM shares with an AppVM, (3) DVMs share with each other, and (4) all VMs share a single PCPU.

Due to space limitations, we do not show the raw experimental results but, instead, provide a brief summary of the main findings:

- With the spare DVM schemes, since the long recovery latency (multiple seconds) is dominated by the time to re-initialize the controller, PCPU sharing does not have a significant impact on interruption latency.
- With RAID, the only significant impact is when the PrivVM shares a PCPU with one or both DVMs. The median service interruption latency increases by approximately 25% (15ms) compared to the no sharing configuration. This increase occurs when the PrivVM and the DVM that does not fail share a PCPU. Immediately after the failure is detected, the PrivVM is active starting up a new DVM and cleaning up the failed DVM, thereby interfering with the remaining DVM with which it shares a PCPU.
- Sharing a PCPU between the PrivVM and a DVM or between PrivVM and AppVM have an impact on the service interruption latency with MultiNetIf for the same reason that PrivVM-DVM sharing affects RAID. For PrivVM-DVM sharing the median interruption latency increases from less than 0.1ms in the no sharing configuration to about 7.5ms. For PrivVM-AppVM sharing the median interruption latency increases to about 5.8ms.
- With MultiNetIf, if the two DVMs share a PCPU, the median interruption latency increases to about 23ms. This increase occurs since the failed DVM sometimes blocks interrupts destined for the working DVM until the failed DVM is destroyed (Subsection 4.2).

6.6. DVM Recovery with a Real Application

To assess DVM recovery with a real application, we use a host with two AppVMs, each running an Apache HTTP server. The two AppVMs access the network and disks through the same two DVMs. The clients are instances of the Apache benchmarking program *ab*. Each AppVM handles requests from one *ab* client. In a run, two *ab* clients, each running on a separate physical machine, sequentially issue 31 HTTP requests for a 1MB dynamically-generated file. The response to each request requires the server to execute the *filesysbench* program. The MultiNetIf and RAID schemes are used for recovery from DVM failure. Software faults (Subsection 5.2) are injected into the device drivers in the DVMs. The results measured are the percentage of request failures observed by the *ab* programs and the latency of request handling. A ‘‘silent failure’’ is recorded if no DVM errors are detected but a request results in a corrupted response or an *ab* client times out waiting for a response.

In this injection campaign, the 584 activated injections are approximately evenly split between the block and network drivers. Roughly 67% of the activated injections manifest as DVM failures: 83.2% DVM crashes, 6.6% DVM hangs, and 10.2% silent failures. Recovery is successful (no missing or corrupted replies) from all the detected DVM failures (crashes/hangs).

Table 6. Request latency without faults versus when there is DVM failure, detection, and recovery.

DVM Failure & Recovery?	Min (ms)	1st Quartile (ms)	Median (ms)	3rd Quartile (ms)	Max (ms)
No	1471	1587	1648	1738	2334
Yes	1479	1608	1660	1705	71380

Table 6 shows the request latencies in fault-free operation (based on 200 runs) and when there are detected faults followed by recovery. Due to the speed of recovery with MultinetIf and RAID, most DVM failures have negligible impact on the request latency. For the few faults that are manifested as hangs, request latencies increase by the latency of the hang detection mechanism. In Linux, the TCP SYN packet retransmission timeout is 3s. Hence, there are a few DVM crashes that result in request latencies of 3-4s, due to a lost TCP SYN packet during TCP connection establishment. Only 4.1% out of all the runs with detected DVM crashes or hangs resulted in a request latency greater than 2000ms.

7. Related Work

Most of the existing techniques for enhancing system resiliency to device driver failure rely on hardware protection mechanisms to isolate potentially faulty drivers in VMs [7, 20, 19, 17], user-level processes [13, 14, 8], or light-weight domains [29, 28]. There are also software techniques based on language extensions, static analysis, and runtime checks [32, 6]. Techniques that do not use virtualization typically require modifications to the device drivers and/or the OS kernel and incur performance overheads due to extra error checking in the driver code or resource tracking between the driver and other kernel subsystems. Schemes that leverage virtualization provide stronger isolation without requiring modifications to the driver or OS kernel.

Several works have investigated mechanisms to recover failed device drivers isolated in user-level processes [13, 14, 8]. A key goal is to isolate the drivers from critical OS kernel components. In [13], failed drivers are detected and restarted transparently to applications by a special ‘‘reincarnation server.’’ The technique is applied to both a network and a block device driver. An analysis and classification of the root causes of fault propagation in device drivers along with mechanisms to prevent such propagation are discussed in [14]. The proposed mechanisms are evaluated using extensive fault injection, injecting 3.2 million faults (software bug faults and random bit flips) into network device drivers, iteratively refining their design in response to injection results.

To facilitate the deployment of user space device drivers in commodity OSs, Ganapathy et al. [8] propose a technique to automatically partition existing device drivers in monolithic kernels into two components: a small, performance critical, core component that runs in the kernel, and a larger, possibly more buggy component, that runs in user space.

Swift et al. [29, 28] propose mechanisms to isolate device drivers in light-weight protection domains within the kernel space. Recovery is performed by unloading and then reloading the failed device driver. The scheme is evaluated using fault injection with several types of device drivers, including network and block. A drawback with the proposed mechanism is that it requires significant modifications to the existing kernel.

Isolation of device drivers in VMs was proposed in 2004 [20, 7]. LeVasseur et al. [20] discuss but do not implement or evaluate mechanisms for recovering from device driver failure. Fraser et al. [7] describe and evaluate the use of Xen's IDVM architecture to provide recovery from NIC driver failure. Recovery is done by restarting the failed DVM and reconnecting the AppVM to the new DVM instance. The DVM uses a customized kernel that boots from RAM disk, only comes up far enough to initialize the network device, and does not run any user processes [30]. The evaluation is done by causing the driver to perform an illegal memory access, leading to guaranteed immediate detection. The network outage duration as a result of a fault is around 275ms.

Xen's IDVM architecture has been used more recently to provide recovery from NIC driver failures (only), with a focus on decreasing the service interruption latency [19, 17]. Le et al. [19] evaluate recovery from NIC driver failures using DC-CC-FF, DS-CC-FF, and DP-CC-FT schemes. The lowest recovery latency is achieved with DP-CC-FT, using two DVMs and a single NIC. This scheme is implemented almost entirely in the VMM and is transparent to the AppVM. Data structures used for AppVM-DVM communication with the primary DVM are reused for the spare DVM and the VMM redirects AppVM I/O requests to the spare DVM. With a 100Mb NIC, recovery latency of less than 10ms is achieved. With a 1Gb NIC, such a recovery latency is achieved only with a partial instead of a full reset of the NIC during recovery, resulting in a significant increase in unsuccessful recoveries. Furthermore, due to the way the system must be initialized, it is difficult to restore the fault tolerance capability of the system after performing a recovery using this scheme.

Jo et al.[17] use a DR-CC-FF scheme, which they name DVH (Driver VM Handoff), based on a spare DVM connected to a spare NIC, to achieve recovery latency of 20-30ms. When the primary DVM fails, the AppVM connects with the spare DVM, reusing the existing AppVM-DVM communication data structures. This reuse allows packet loss during failover to be minimized. By using a spare NIC, recovery is not delayed by the latency of resetting the NIC, encountered in [19]. With the DVH mechanism, establishing the AppVM-DVM connection with the spare DVM during recovery requires interactions with the PrivVM (XenStore). With the DVH implementation in [17], due to these interactions, recovery involves a significant amount of time waiting for protocol state changes. Hence, DVH results in slower recovery compared to, for example, the MultiNetIf scheme evaluated in this work, that also uses a spare DVM and spare NIC, but achieves recovery latency of less than 1ms.

Fault injection is used in [17] to evaluate the effectiveness of the detection mechanism. Either random software faults are injected repeatedly until a failure is detected, or there is an injection to a specific location in the kernel/driver code of a software error that almost guarantees a DVM crash or hang. Hence, there is essentially no opportunity for faults to manifest as anything other than a crash or hang. Thus, this evaluation cannot provide critical information regarding the probability of system failure due to silent (undetected) failures (see Subsection 6.1).

In [17], with respect to recovery latency and packet loss, DVH is compared to two other recovery mechanism: (1) restarting the backend drivers and (2) failing over to a spare NIC using the Linux bonding driver [21]. However, there is no evaluation of the recovery effectiveness of these other mechanisms. The recovery latency reported for bonding, with a 1ms bonding driver detection interval, is greater than 70ms, compared to our measurements of latencies less than 1ms with bonding. We believe that this difference may be related to the way link failure detection is performed in [17].

8. Summary and Conclusions

The IDVM architecture, used in many virtualized systems, is particularly well suited for confining the effects of device driver failures and enabling resilience to such failures. We have analyzed design tradeoffs and developed a taxonomy for mechanisms that provide resiliency to device driver failure with the IDVM architecture. Key challenges and associate solutions in implementing these mechanisms and ensuring their reliable and fast operation were presented. Our evaluation was based on the implementation and measurement of five different recovery mechanisms, covering key points in the design space, with both network and block devices (hard disk and USB flash drive). The evaluation involved injecting thousands of software and hardware faults into device drivers with a variety of system configurations.

Our experiments show that the rate of successful recovery from detected DVM failures is high (>93%) across a wide range of faults. Recovery rates are essentially the same for all recovery mechanisms. These rates are higher for network devices than block devices since a few dropped packets do not significantly impact an application whereas a corrupted file or file system can cause an application failure. With the simple, low overhead DVM failure detection mechanisms we used, silent DVM failures, especially due to software faults, are the main cause of unsuccessful application completion. To reduce silent failures, more sophisticated detection mechanisms, such as using VM replication [16], will be needed.

We have shown that recovery latency is minimized by schemes that avoid the need to boot a new DVM or reset/initialize device controllers as part of the recovery process. The faster schemes use redundant device controllers and defer these operations to the fault tolerance restoration phase. With slower schemes, the number of controllers hosted by a DVM has a significant impact on recovery latency since device reset/initialization must typically be performed serially. The number of AppVMs sharing a DVM and the number of VMs sharing a physical CPU have relatively minor impact on recovery latency.

Our experiments show that the type of fault (register, code, software) has a major impact on the fraction of faults manifested as crashes versus hangs, and thus on the detection latency. Since the detection latency is the key factor in determining the service interruption latency, there is a major impact of the type of fault on that latency.

Most of our results are based on two micro benchmarks that stress the network and block devices and enable accurate measurements of the impact of faults and the recovery mechanisms. These results are validated using a real-world application (the Apache web server) with a combination of the fastest network and block recovery mechanisms (MultiNetIf and RAID). This validation demonstrates recovery from 100% of detected DVM failures without significant service interruptions.

This work focused on techniques that involve low performance/computation overhead during normal operation. The only exception is the RAID mechanism, for which there are other compelling reasons to deploy (disk failures). Our results show very high probabilities of failure detection and recovery using these low overhead techniques. However, the coverage of these techniques is not 100%. Future work will involve investigating ways to increase this coverage at the cost of moderate increases in performance overhead.

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] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. V. Doorn, A. Mallick, J. Nakajim, and E. Wahlig, ‘‘Utilizing IOMMUs for Virtualization in Linux and Xen,’’ *Proceedings of the Linux Symposium* (July 2006).

[3] 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).

[4] 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).

[5] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang, ‘‘Towards high-quality I/O virtualization,’’ *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009).

[6] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, ‘‘XFI: Software Guards for System Address Spaces,’’ *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, Washington, pp. 75-88 (2006).

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

[8] 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).

[9] 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).

[10] 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).

[11] 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).

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

[13] 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).

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

[15] Intel Corporation, *PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology*, January 2011.

[16] C. M. Jeffery and R. J. Figueiredo, ‘‘A Flexible Approach to Improving System Reliability with Virtual Lockstep,’’ *IEEE Transactions on Dependable and Secure Computing* **9**(1), pp. 2-15 (2012).

[17] H. Jo, H. Kim, J.-W. Jang, J. Lee, and S. Maeng, ‘‘Transparent Fault Tolerance of Device Drivers for Virtual Machines,’’ *IEEE Transactions on Computers* **59**(11), pp. 1466-1479 (November 2010).

[18] 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).

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

[20] 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).

[21] Linux Foundation, *Bonding*, http://www.linuxfoundation.org/collaborate/workgroups/networking/bonding, 2009.

[22] 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).

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

[24] T. L. Nguyen, M. Silbermann, and M. Wilcox, ‘‘The MSI Driver Guide HOWTO,’’ *https://www.kernel.org/doc/Documentation/PCI/MSI-HOWTO.txt* (Accessed September, 2013).

[25] W. T. Ng and P. M. Chen, ‘‘The Design and Verification of the Rio File Cache,’’ *IEEE Transactions on Computers* **50**(4), pp. 322-337 (April 2001).

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

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

[28] 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).

[29] 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).

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

[31] E. Zhai, G. D. Cummings, and Y. Dong, ‘‘Live Migration with Pass-through Device for Linux VM,’’ *Ottawa Linux Symposium*, Ottawa, Canada, pp. 261-267 (July 2008).

[32] F. Zhou, J. Condit, Z. Anderson, and I. Bagrak, ‘‘SafeDrive: Safe and Recoverable Extensions Using Language-Based Techinques,’’ *7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, pp. 45-60 (November 2006).