Transparent Fault Tolerance of Device Drivers for Virtual Machines

Heeseung Jo[*] KAIST

Hwanju Kim[†] KAIST

Jae-Wan Jang[‡] KAIST

Joonwon Lee[§]
Sungkyunkwan University

Seungryoul Maeng[**]
KAIST

CS/TR-2009-312

June 29, 2009

K A I S T
Department of Computer Science

[*] heesn@camars.kaist.ac.kr
[†] hjukim@camars.kaist.ac.kr
[‡] jwjang@camars.kaist.ac.kr
[§] joonwon@skku.edu
[**] maeng@camars.kaist.ac.kr

Transparent Fault Tolerance of Device Drivers for Virtual Machines

Heeseung Jo, Hwanju Kim, Jae-Wan Jang, Joonwon Lee, and Seungryoul Maeng

Abstract — In a consolidated server system using virtualization, physical device accesses from guest virtual machines (VMs) need to be coordinated. In this environment, a separate driver VM is usually assigned to this task to enhance reliability and to reuse existing device drivers. This driver VM needs to be highly reliable, since it handles all the I/O requests. This paper describes a mechanism to detect and recover the driver VM from faults to enhance the reliability of the whole system. The proposed mechanism is transparent in that guest VMs cannot recognize the fault and the driver VM can recover and continue its I/O operations. Our mechanism provides a progress monitoring-based fault detection that is isolated from fault contamination with low monitoring overhead. When a fault occurs, the system recovers by switching the faulted driver VM to another one. The recovery is performed without service disconnection or data loss and with negligible delay by fully exploiting the I/O structure of the virtualized system.

Index Terms—D.4.5 Reliability, D.4.5.c Disconnected operation, D.4.5.d Fault-tolerance, D.4.5.e High availability

1 Introduction

VIRTUALIZATION enables multiple operating systems to run on a single physical machine, and the consolidated server systems using virtualization significantly expand their territory, especially in large-scale computing or cluster systems. This trend is based on the effort to lower the management cost, which is one of the primary factors for server hosting centers or the server market. With server consolidation, fewer physical machines are needed to run the same number of servers and hence lessen power and space. These factors are directly related to the total management cost, and it is known that 50-70% of reduction is possible [1]. Moreover, virtualized systems are also advantageous in their availability and manageability of servers.

Most work in virtualization is focused on improving performance, since the sharing hardware among guest virtual machines (VMs) incurs significant overhead. The para-virtualization approach, which makes an operating system aware of the virtualization layer with a slight modification of a guest kernel, has accomplished a performance breakthrough [28]. As the para-virtualization becomes prominent, it is partially adopted to full-virtualization software such as VMware [30] and KVM [29] for performance improvement.

Despite the advantages of a virtualized system, it also includes the risk of centralization. Since many guest VMs are consolidated in a single physical machine, a fault in device drivers or virtual machine monitor (VMM) can affect all other VMs. The code of VMM is relatively reliable in that it is developed and published by a closed group, and many kinds of tests are performed during the development process. On the other hand, the device driver codes, which are used by a VMM or privileged VMs, are unreliable, since most device drivers are developed independently and cooperate with other components in a kernel address space. It is known that 85% of Windows XP crashes are caused by device drivers [2], and the device driver bugs occur seven times more often than any other kernel bugs [3]. Thus, the reliability of a virtualized system, especially related with device drivers, becomes one of the most important factors for high availability.

This paper describes a fault detection and recovery mechanism for device drivers in a virtualized system. Most virtualized systems adopt the isolated I/O architecture that enables a driver VM, rather than a guest VM, to perform I/O device accesses. This isolated I/O architecture is advantageous for software reusability and reliability [5], [42]. With the isolated I/O architecture, our mechanism detects the faults of a driver VM including kernel crash and malfunction, and then recovers from them by switching from the faulted driver VM to another one. Thus, it hides the fault of a driver VM from guest VMs and enables guest VMs to continue their I/O operations transparently without data loss.

For transparency, which is one of the primary goals of our mechanism, the implementation of the proposed mechanism is confined to the virtualization layer: VMM and virtual device drivers. Based on this feature, it requires no kernel or application level modification of VMs, and thus the full-virtualization-based VMM can also adopt our scheme. Another design goal is lightweightness. Our mechanisms are well aware of the unified interfaces and the I/O path of the virtualized system, which are exploited to lower overhead. Thus, CPU and memory overhead is negligible and acceptable with respect to scalability. Our

- *Heeseung Jo, Hwanju Kim, Jae-Wan Jang, and Seungryoul Maeng are with the Computer Science Department, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea. E-mail: {heesn, hjukim, jwjang, maeng}@camars.kaist.ac.kr.*
- *Joonwon Lee (corresponding) is with the School of Information and Communication Engineering, SungKyunKwan University, Suwon, Korea. E-mail: joonwon@skku.edu.*

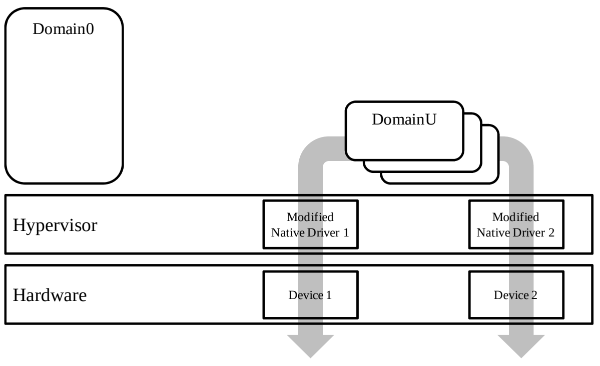


Figure 1: The I/O architecture of the initial Xen. Hypervisor includes modified device drivers and guest domains request I/O operations to the hypervisor.

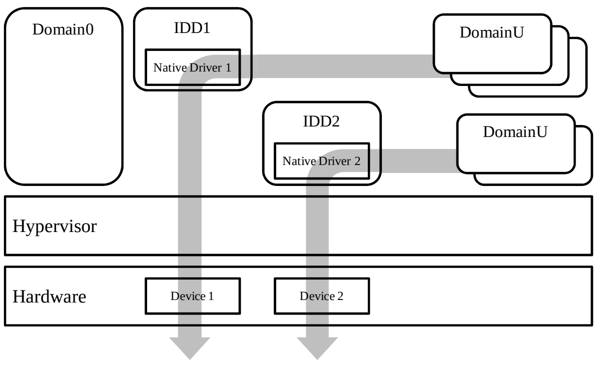


Figure 2: The I/O architecture of the reliability enhanced Xen. The newly introduced IDD includes unmodified native device drivers and processes the I/O requests of guest domains instead of the hypervisor.

fault detection module is isolated from device drivers and the address space of the guest kernel. This architecture protects it from the fault contamination, which was a challenging problem in the previous work. For the recovery of faults, our mechanism inspects the I/O status and redeems pending requests. This recovery is performed without I/O service disconnection and data loss.

The rest of this paper is organized as follows. The next section summarizes the Xen [21] overview and its I/O architecture that is our implementation target. Section 3 describes our fault model to detect and recover. Sections 4 and 5 illustrate the design and detailed mechanism of the fault detection and recovery, respectively. Section 6 presents and analyzes the evaluation results, and Section 7 describes the related work. Finally, we conclude in Section 8.

2 Background

This section presents an overview of Xen focusing on the isolated I/O architecture and its subsystems.

2.1 Xen Virtual Machine Monitor

Xen is an open source para-virtualization software [21] that exceeds in performance compared with the full-virtualization approach. Since the para-virtualization exports virtualization-aware interfaces to VMs, each guest kernel can be optimized to minimize the virtualization overhead. Currently Linux and BSD kernel are successfully ported for Xen and are named XenoLinux and XenoBSD respectively. The portion of the kernel modification is small while the performance of Xen is much better than that of full-virtualization implementation [4]. Xen also supports full-virtualization by using hardware-assisted virtualization such as Intel-VT [35] and AMD-V [36].

In Xen, a guest VM is called a *domain*, and each guest domain runs on a VMM called a *hypervisor*, which resides between the hardware and guest domains. Hypervisor virtualizes hardware to show it to guest domains and schedules guest domains. There is a privileged guest domain called *domain0*, and it has an interface to control the hypervisor and other guest domains.

2.2 Isolated Driver Domain

Xen 3.0 architecture introduced an isolated driver model for the I/O device virtualization. Figures 1 and 2 show the difference between the I/O architecture of the initial Xen and that of the reliability-enhanced Xen. An isolated driver domain (IDD) is a special purpose guest domain for directly handling of I/O device accesses. In the initial design of Xen, the hypervisor embeds modified device drivers, and the guest domains interact directly with the hypervisor for I/O operations. In the reliability enhanced Xen, however, the hypervisor excludes device drivers, and the newly introduced IDD includes them instead. All the requests of guest domains are forwarded to their corresponding IDDs. For example, it is possible that a request for device1 is forwarded to IDD1, and that for device2 is forwarded to IDD2. Each IDD processes these requests using its native device driver and replies results back to each requesting guest domain.

The IDD I/O architecture improved the reliability and reusability of Xen. In the IDD I/O architecture, unmodified native device drivers can be used as-is in an IDD because the IDD is based on a commodity OS such as Linux. Furthermore, the failure or malfunction of a device driver does not spread to the entire system, and its damage is isolated within the IDD. Such failures, however, affect guest domains that rely on the failed IDD.

2.3 I/O Descriptor Ring

In the IDD I/O architecture, there are two virtual device drivers: (1) a frontend driver in the guest domain is responsible for requesting, and (2) a backend driver in an IDD forwards the request from the frontend driver to the native device driver. An *I/O descriptor ring*, a circular queue data structure containing descriptor information, is used to communicate between the frontend and backend drivers. For a network IDD, there are two I/O rings for transmission (TX) and receiving (RX), and an I/O ring has two pairs of producer/consumer pointers: one for request and the other for the response to the request. These pointers indicate the location of the request and the response that each virtual driver should process next.

Figure 3 shows the I/O path between a guest domain

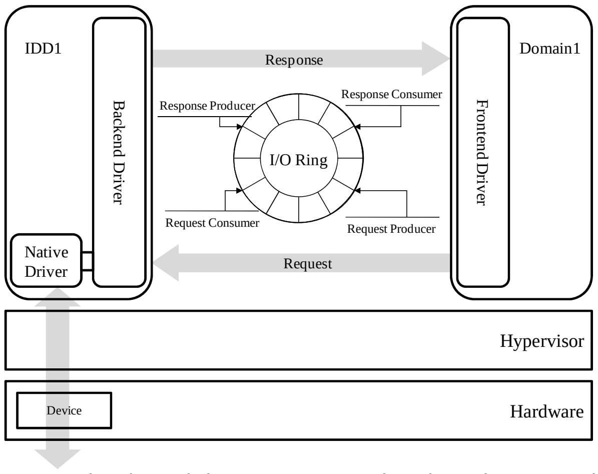


Figure 3: The I/O path between a guest domain and an IDD. The request and response are managed via the I/O ring, which is a structure partially shared between an IDD and a guest domain.

and an IDD. An I/O request of domain1 is forwarded to the backend driver in IDD1 via its frontend driver. The frontend driver inserts a request into its I/O ring, and the backend driver consumes it. After processing the request, the backend driver puts a response into the I/O ring. The response means that the request is properly processed by the backend driver and the native device driver. Then the frontend driver consumes the response. Since the address spaces of an IDD and a guest domain are isolated, shared memory is used for the I/O ring. The access permissions, however, are separated for isolation.

2.4 Grant Table Mechanism

The IDD I/O architecture uses a *grant table mechanism* to transfer actual I/O data. This mechanism was originally devised to share the memory pages between guest domains. Fundamentally, Xen does not allow a guest domain to access the memory page of another domain for isolation. Exceptionally for special purposes, a privileged domain can be allowed to access the memory page of other guest domains using the grant table mechanism. It is strictly limited to the memory page and guest domain permitted by the owner. In the IDD I/O architecture, the descriptive information of an I/O operation is transferred via an I/O ring, and the actual data is transferred by the grant table mechanism.

2.5 Event Channel

The purpose of *event channel* in Xen is for asynchronous notification between guest domains or between a guest domain and the hypervisor. It is similar to the signal mechanism of commodity OSes. The inter-domain event channel, a bidirectional channel between the frontend and backend drivers, is used to notify requests and responses in the IDD I/O architecture.

2.6 Xenbus & Xenstore

Interactions between frontend and backend drivers use *Xenbus*, which is a bus abstraction that communicates between guest domains. When a frontend device is loaded, it probes the corresponding backend driver based on the information in *Xenstore*, a database for configuration and status shared among all domains. A guest domain is able to acquire the information in XenStore by requesting a transaction on Xenbus.

3 Environment and Fault Model

A consolidated server system takes advantage of operating system virtualization in terms of load balancing and management. Live VM migration [9], [10], [11], [12] and the online maintenance features [33], [34] of virtualized systems are attractive to a system administrator. In addition, high resource utilization is achieved by relocating VMs to an underutilized physical machine.

Our mechanism is designed for a server machine equipped with multiple I/O devices in a highly consolidated system. Although it is applicable to all types of I/O interfaces or devices due to the unified virtual driver interface of the isolated I/O architecture, our prototype implementation is focused on the network driver VM, since network traffic takes most of the I/O operations in a large server consolidation environment. Even with respect to disk I/O, a large number of nodes typically share separated data storage through a network in a highly consolidated system. To extend our mechanism to block I/O paths such as a disk and other devices is an implementation issue, and we will address this in future work.

In this paper, we define IDD fault when an IDD no longer provides its I/O service for the connected guest VMs. Since our work focuses on the continuous working of guest VMs, the unavailability of an IDD is fatal and is used as a criterion to decide whether an IDD is in a fault state or not. Though it is practically impossible to discern if an IDD is in a fault state or not when it does not respond for a long period of time, it is better to consider it as crashed to be on the safe side. Note that our fault detection mechanism does not try to identify fault reasons because it aims to detect faults and recover from them, not to debug faults.

An IDD fault can be largely divided into two cases: kernel crash and malfunction. The hypervisor can explicitly recognize the crash of a guest domain. The design principles of the hypervisor make the detailed activities of a domain irrelevant. Hypervisor, however, cares about the abnormal behavior of a guest domain such as crash due to domain scheduling. If a domain crash is recognized, the hypervisor cleans up the data structures of the crashed domain and discards them from the domain-scheduling queue.

It is probable that the IDD kernel is alive but cannot serve the I/O requests of guest domains for some reason. The malfunction of the kernel or device drivers is more difficult to detect than a domain crash. Due to the semantic gap between the hypervisor and a guest domain, the hypervisor does not have any insight about the context of a guest domain. Therefore, our detection mechanism focuses on tracking the malfunction of an IDD at the virtualization layer rather than a crash.

There is another case in which a device driver returns an error. For example, a driver might return an I/O error when the network link is temporarily off-line or the disk fails. Our fault detect mechanism ignores these error cases and regards them as part of the normal operations, because it is an error handling issue.

4 Fault Detection

Since an IDD is a kind of guest domain, it may fault unexpectedly. Considering that most faults are caused by device drivers, an IDD is more vulnerable to faults, since it contains native device drivers. The IDD I/O architecture [5] is introduced for the reliability of the overall virtualized system, and it achieves this aim by separating device drivers from the VMM. The risk, however, is transferred over to the IDDs. The reliability of an IDD is still critical, since an IDD fault disables all the I/O operations of connected guest domains.

To detect the fault of an IDD, we introduce a detection module, named *Driver VM Monitor (DVM),* which periodically decides the fault state of IDDs. Its first design goal is transparency that requires no guest kernel or application level modification. The DVM resides in the hypervisor and requires minimal support from the virtual device drivers. This transparent design enhances flexibility and enables other full-virtualization-based VMMs to adopt our fault detection mechanism.

Another advantage of the DVM is its isolation from other components. In the previous work to detect the faults of the device drivers or kernel in a native machine, protecting the detection module itself is a challenging problem, even with considerable overheads [31], [18]. Due to the hypervisor level monitoring, the faults of an IDD do not affect the hypervisor or DVM. This feature protects our detection module from being contaminated by the faults of other modules.

For a lightweight mechanism and scalability, the DVM is designed to use a small amount of memory per guest domain, and the memory usage is linearly proportional to the number of guest domains. Furthermore, it consumes trivial CPU overhead, since it executes only a few lines of codes in the normal state.

4.1 Basic Detect Mechanism

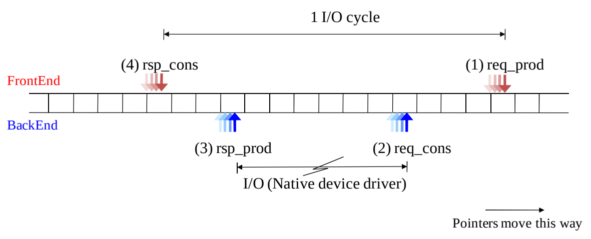
The DVM is a core component that is activated periodically for detecting IDD faults implemented in the hypervisor. If invoked, the DVM logs the information about the I/O ring and physical IRQ (PIRQ). DVM mainly refers the I/O ring pointer information for fault decision, and the PIRQ is used for a more detailed inspection. The DVM keeps the logged information in the hypervisor stack. Since it is isolated from the IDD kernel address space, the hypervisor stack is safe from fault contamination. After gathering the information, the DVM inspects whether the targeted IDD is in a normal state or in a fault state.

Before starting the monitoring, the DVM expects to receive a *hypercall* that is an interface for privileged operations at the VMM level. Once the virtual frontend driver is initialized, it calls a newly added *ringnoti* hypercall to let the DVM know the hardware memory location of the I/O ring structure. Since the I/O ring is a data structure of the frontend driver and belongs to the address space of a guest domain kernel, the DVM cannot refer to the information of the I/O ring without the aid of the ringnoti hypercall. Once alerted, the DVM starts monitoring the information of the I/O ring.

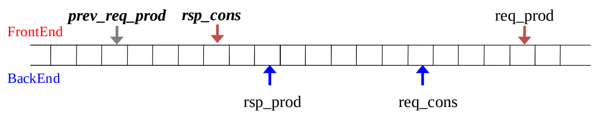
The DVM is invoked at a periodical IDD scheduling time. We modified the VM schedule function of Xen to count how many times the targeted IDD is scheduled. When the counter reaches a pre-configured value named the *DVM frequency*, the schedule function invokes the DVM. Thus, the timing is not regular. This periodical IDD scheduling time is advantageous, since it leverages the load of an IDD. For example, a default VM scheduler (*credit scheduler*) of Xen adopts a *boosting mechanism*, which preferentially schedules an IDD having pended requests to enhance the responsiveness of the I/O access. With an increased number of requests, the IDD is scheduled more times, and the DVM checks the status of the IDD more frequently.

To determine the IDD fault, the DVM compares the current I/O ring state with the logged information. The I/O ring has four pointers: request producer/consumer (req_prod, req_cons) and response producer/consumer (res_prod, res_cons), as described in Section 2. For example, Figure 4(a) shows a linear operation sequence of the I/O ring pointers for processing network transmission. To send a packet, a guest domain generates a request and transfers it to its frontend driver. The frontend driver puts the request into the I/O ring, and sets the req_prod (1). Then the backend driver of the IDD consumes the request (2). The backend driver processes the request with native device drivers and sets the rsp_prod (3). During this process, the DMA operations, grant table operations, and interrupts handlers are performed. An I/O operation cycle ends when the frontend driver consumes the response (4).

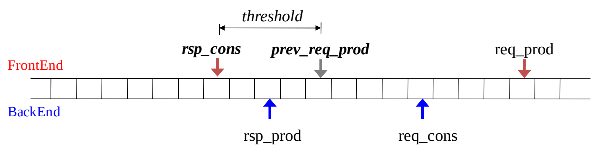
When invoked, the DVM is faced with either Figure 4(b) or (c). Note that prev_req_prod denotes the location of req_prod at the previous DVM check-time. In Figure 4(b), the prev_req_prod is smaller than the rsp_cons. This means that the previous request is already completed. In this case, we can assume that the IDD is in a normal state. On the other hand, in Figure 4(c), the prev_req_prod is larger than the rsp_cons, meaning that the previous request is not yet finished. In this case, there are two possibilities; the IDD might be in a fault state or might be too busy to handle the request. To assess the case more precisely, the DVM uses a *threshold* value. If the difference between the prev_req_prod and the rsp_cons is greater than the threshold, the DVM decides that the IDD has failed. Otherwise, the DVM assumes that the IDD is too busy to handle the previous requests and gives a chance until the next check turn. As a result, the threshold value is an important factor for our fault detection mechanism. A small threshold value would cause a false-positive by which a busy IDD can be sentenced as having a fault. With a large threshold value, the fault detection could be deferred, while the false-positive ratio decreases. The false-negative case does not occur, since the difference between the prev_req_prod and rsp_cons always increas-



(a) One I/O cycle and I/O ring pointers



(b) Normal operation case



(c) A fault possible case and threshold

req_prod: request producer	req_cons: request consumer
rsp_prod: response producer	rsp_cons: response consumer
prev_req_prod: previous req_prod	

Figure 4: A linear operation sequence for network transmission. (a) shows a operational sequence in terms of pointers. (b) and (c) are normal operations and a possible fault case for each.

es, and the DVM detects it in the end.

4.2 Deciding Threshold

The threshold value of the DVM should be carefully selected for reactive detection with a low false-positive ratio. The number of guest domains and their workloads change continuously, and so does the load of an IDD. As a result of the aforementioned reasons, the value of the threshold needs to be dynamic to reflect the dynamic nature of the IDD load.

The DVM uses the number of PIRQs as an indicator of the IDD load. In the IDD I/O architecture, the PIRQ intensity of an I/O device represents for the IDD load. In order to maintain a transparent design principle, we exclude approaches to monitor the load within an IDD, and the DVM cannot monitor much information of the upper layer. The DVM works at the hypervisor level and only gathers low level I/O information. Considering this, the PIRQ is a feasible representation of the load of an IDD, since an IDD is a lightweight domain designed to conduct only I/O operations with its physical devices. Using the PIRQ is also advantageous in that it is not dependent on the number of guest domains.

The formula to get a *dynamic threshold (DThres)* is defined as:

$$\mathrm{PIRQ}_{\mathrm{diff}} = \mathrm{PIRQ}_{\mathrm{curr}} - \mathrm{PIRQ}_{\mathrm{prev}} \qquad (1)$$

$$\mathrm{DThres} = \mathrm{PIRQ}_{\mathrm{diff}} \times \mathrm{T}_{\mathrm{ratio}} \qquad (2)$$

The DVM logs the counter of PIRQ and calculates the $\mathrm{PIRQ}_{\mathrm{diff}}$, which is the difference between the previous and the current values of the PIRQ (Eq. 1). The $\mathrm{PIRQ}_{\mathrm{diff}}$ denotes how many PIRQs are raised during a DVM check period and represents the load of an IDD. The DThres is calculated by multiplying $\mathrm{PIRQ}_{\mathrm{diff}}$ by the $\mathrm{T}_{\mathrm{ratio}}$ (Eq. 2), which is a compensation value for leveraging the strictness of the DVM. With the smaller $\mathrm{T}_{\mathrm{ratio}}$, the DThres decreases, and tighter detection is possible.

$$\mathrm{TotalReq}_{\mathrm{diff}} = \sum_{n=1}^{\#\ of\ VMs} |\,\mathrm{rsp\_cons} - \mathrm{prev\_req\_prod}\,| \qquad (3)$$

To check the global progress of the I/O operations, the DVM gets the total difference of all requests ($\mathrm{TotalReq}_{\mathrm{diff}}$) by summing up the each difference between rsp_cons and prev_req_prod for each guest domain connected to an IDD (Eq. 3) and compares it with the DThres. In summary, the DVM decides the fault state of an IDD by comparing the $\mathrm{TotalReq}_{\mathrm{diff}}$ with DThres.

4.3 Asynchronous I/O Operation

Most types of I/O requests, such as sending packets or disk block accesses are explicitly triggered by a guest domain. The asynchronous I/O operations, such as receiving a packet, however, are triggered from the outside of the physical machine. Using only the I/O ring information, the DVM cannot decide whether an IDD is in a fault state or has no incoming packet. To address this problem, we exploit the PIRQ for receiving ($\mathrm{PIRQ}_{\mathrm{RX}}$) since most of device accesses involve interrupts. If the req_prod of the RX I/O ring and $\mathrm{PIRQ}_{\mathrm{RX}}$ do not increase, the DVM decides that there are no incoming packets. This case is regarded as a normal operation. If the $\mathrm{PIRQ}_{\mathrm{RX}}$ increases without changes in the req_prod, it means that the native device driver or the backend driver of the IDD is not properly processing the receiving PIRQ. Thus, the DVM decides that the IDD is in a fault state.

Unfortunately, the objective of the PIRQ is not classified at hypervisor level. Although the hypervisor could detect the fact that a PIRQ is raised, it is hard to define whether the PIRQ is for TX or RX. To classify the objective of a PIRQ, the hypervisor should inspect the native device driver codes in an IDD. This mechanism is too complex and would ruin the flexibility of the DVM, since it would have to inspect the data structures of each device. Hence, the DVM estimates the number of $\mathrm{PIRQ}_{\mathrm{RX}}$ using the following formula:

$$\mathrm{PIRQ}_{\mathrm{TX}} = \mathrm{req\_prod}_{\mathrm{TX}} - \mathrm{prev\_req\_prod}_{\mathrm{TX}} \qquad (4)$$

$$\mathrm{PIRQ}_{\mathrm{RX}} = \mathrm{PIRQ}_{\mathrm{total}} - \mathrm{PIRQ}_{\mathrm{TX}} \qquad (5)$$

DVM gets $\mathrm{PIRQ}_{\mathrm{RX}}$ by subtracting PIRQ for transmission ($\mathrm{PIRQ}_{\mathrm{TX}}$) from the total number of PIRQ ($\mathrm{PIRQ}_{\mathrm{total}}$)

(Eq. 5). $PIRQ_{TX}$ can be inferred from the rsp_prod logs (Eq. 4).

The control packet types such as ARP or ICMP also pass through the I/O ring and are counted by the DVM. Although the control packets are not delivered to the user level of a guest domain, they should be handled by the guest domain kernel.

4.4 Pointer Integrity

DVM uses only the I/O ring pointers of a frontend driver to avoid contamination by faults. Since a backend driver works in the kernel address space of an IDD, a fault in the IDD might taint the location of the backend driver pointers. Although the pointers of an I/O ring are shared by the frontend and backend drivers, they have different write permissions. For example, the backend driver has no write permission to the req_prod and the rsp_cons in a TX I/O ring. To guarantee the integrity of pointers, contrary to the described example in Figure 4, the DVM refers to req_cons, prev_req_cons, and rsp_prod for network receiving. Due to the different write permission, the pointers of the frontend driver guarantee their correctness even if an IDD is tainted by faults.

We assume that a guest domain and its frontend driver run faultlessly and limit our need to detect IDD faults. For all our assumptions, the DVM checks the order and the address range of the frontend pointers to ensure correctness. The pointers always increase and the values of pointers should be within the address range of their I/O ring. Although this ensuring process requires a little overhead, we included it in the current DVM implementation for higher reliability.

4.5 Limitation

Since our detection mechanism is based on the I/O ring information, if a fault contaminates only the data page of a packet, the current DVM implementation cannot detect the fault. Although it is hard for a fault to taint only the packet data page without being caught by our scheme, it is theoretically possible. Addressing this problem accompanies much overhead to check the integrity of every packet datum, and therefore, we excluded this from our work.

5 Fault Recovery

To recover from IDD faults, we introduced the *Driver VM Hand-off (DVH)* mechanism, which transparently redirects I/O requests and responses of the fault IDD to another IDD.

Besides transparency, the DVH has two other design principles, one of which is the I/O seamlessness. From the view point of a guest domain, the previous requests, responses, and connection should not be lost during switching IDDs. Our DVH design aims to assure seamless service even if an IDD kernel crashes. The DVH supports seamless switching, and therefore guest domains are not aware of the IDD switching and can continue their services.

The other design principle is delay minimization. To shorten the switching delay of the DVM, the I/O ring is reused. The DVH modifies the pointers of the I/O ring and the grant table entries to get rid of the reinitialization overhead and to reduce delay. Since the I/O ring is created and owned by the frontend driver, the I/O ring itself and its data survive even when an IDD kernel or the backend driver crashes.

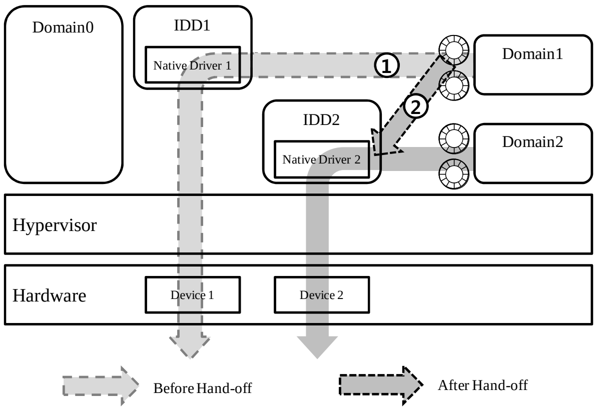


Figure 5: A DVH scenario. Domain1 that is served by IDD1 is handed off to IDD2. Then, IDD2 serves Domain1 and 2 simultaneously.

Although it is similar to the active-backup or duplication backup scheme [7], [8], [13], the DVH is based on the circumspect inspection of the I/O mechanism of a virtualized system. Therefore, the DVH has the advantages of transparency, seamlessness, and low overhead.

5.1 Hand-off

The DVH design assumes that multiple IDDs are in charge of the I/O devices instead of the domain0. This configuration takes advantage of the high reliability, since a fault in domain0 seriously affects the entire system. As a machine is equipped with multiple I/O devices, it is safe to distribute I/O processing to several IDDs. In the case of an IDD fault, the damage is enclosed within only related guest domains that are served by the IDD. Especially in a highly consolidated server, this configuration is recommended.

Figure 5 shows a DVH scenario. Domains 1 and 2 are served by IDDs 1 and 2, respectively. Both devices 1 and 2 are network devices. When the IDD1 goes into a fault state, the connection with domain1 is broken, but the I/O rings of domain1 are sustained. The DVH cleans up the connection ① and remaps the I/O rings of domain1 to the backup IDD, IDD2 in Figure 5 ②. Note that a backup IDD denotes a destination IDD of the DVH, and a backup backend driver denotes the backend driver in the backup IDD.

In this hand-off process, the pointers of the I/O ring are relocated to guarantee seamlessness. The DVH should check and resubmit the unfinished I/O requests that are consumed by the backend driver, but not completed at the fault time. Note that the response producer is set once the backend driver confirms the I/O completion. Therefore, we should consider an I/O operation incomplete if its response producer has not been set until fault. The

DVH moves the request consumer of the I/O ring to the location of response producer, which is the last successfully committed operation. Due to this pointer relocation, the DVH can guarantee the I/O seamlessness without the IDD fault recognition of the guest domains.

In a limited server system, reserving a backup IDD might be overhead. A fast VM instantiation technique, however, can be adopted for a backup IDD. Michael et al. proposed a mechanism to instantly clone a VM [37]. In this work, flash cloning shows about a 300 ms delay in initiating a guest domain. Otherwise, domain0 can assume the role of the backup IDD, since it always runs as a privileged domain for managing the whole virtualized system with its hypervisor control interface.

5.2 Interface

Our implementation provides two interfaces to initiate DVH. One is *VMM interface* using a virtual interrupt to trigger DVH from the outside of a guest domain such as the hypervisor or domain0. This interface is useful to trigger DVH when the hypervisor decides to switch an IDD. If the DVM detects IDD faults, it generates a virtual interrupt to launch the DVH process.

The other interface is a *guest domain interface* using a virtual file system to interact with the kernel. This interface provides DVH triggering from the inside of a guest domain. This interface is devised for a guest domain to initiate DVH, if needed. Through this interface, a third party fault-monitoring tool can trigger the DVH process. For example, if a user level heart-beat monitoring daemon of a guest domain detects an I/O problem, it can execute the DVH with the guest domain interface.

5.3 DVH Steps

The DVH has to interact with the frontend driver of a guest domain and the backend driver of an IDD. To allow for this communication, we add two protocol states as shown in Figure 6. One is the *BackupWait*, which means that a backup backend driver is ready for the DVH. When the DVH is triggered, the frontend driver reprobes the backend driver whose state is BackupWait, through Xenbus. The other protocol state is the *Reconfigured* denoting that the frontend driver finishes the reconfiguration for the backup backend driver during DVH. When the backend driver recognizes this state, it tries to connect to the reconfigured frontend driver. It does not reinitialize the producer/consumer pointers, but rather, updates them to the next successfully committed points.

The DVH process is largely divided into the following three steps: preparation, reconfiguration, and establishment. Before execution of the DVH, some preparation is necessary. The information of the frontend driver in the Xenstore should be modified to adapt to a backup backend driver. The current backend information of the frontend driver is replaced with the information from the backup backend driver. This modification does not affect the operation of the frontend driver since the information is not referred to after the initial probing. In the preparation step, the information of the backup backend driver is also configured. The backup backend information, but not

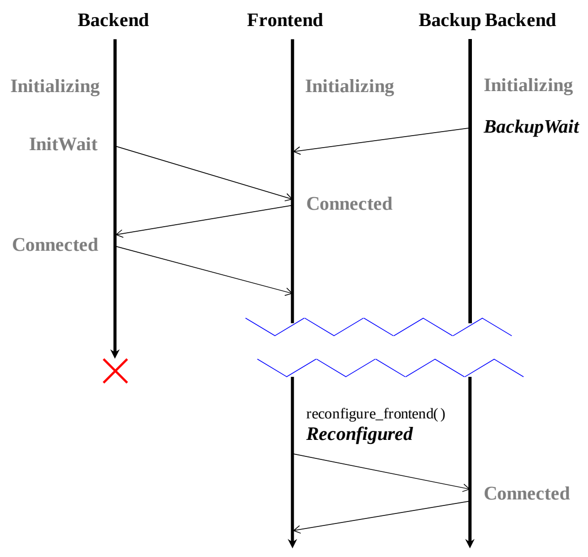


Figure 6: The Xenbus protocol among the virtual network device drivers. For the DVH, the BackupWait and Reconfigured state are added.

state, is copied from the current backend driver; the initial state of the backup backend driver is set to BackupWait. After finishing the preparation, the backup backend driver waits reconfiguration of the frontend driver in the BackupWait state.

When the DVM detects an IDD fault, the DVH starts the reconfiguration step, which revises the connection of the frontend drivers. All the connection information is modified to be redirected to the backup backend driver. Most of modification is for I/O ring-related subsystem such as the grant table, event channel, and mapped memory pages. This reconfiguration is fast because the original I/O ring is reused and the outstanding requests are not discarded. After completing this reconfiguration process, the state of frontend driver is changed to Reconfigured, and the backup backend driver is notified.

For the establishment step, the backup backend driver tries to connect to the frontend driver as soon as the Reconfigured state is detected. Since the I/O ring maintains the last committed request pointers, the backup backend driver knows the next entry to be requested. Thus, the backup backend driver just updates its pointers to the next entries to be requested. Finally, the DVH is finished if all the above procedures are successfully conducted, and the I/O requests are served by the backup IDD.

5.4 Limitation

The DVH mechanism has a limitation derived by hardware device switching. If an incoming network packet arrives at the network device buffer from the outside of the machine and is not delivered to its backend driver, the packet will be lost when the IDD fails. In this case, there is no way to handle it with software. Conversely, in a network packet transmission and disk I/O request, losses do not occur, since the I/O ring has the information about the pending requests, and DVH reinstates them.

Another issue is the state of a device. The current version of the DVH does not provide a mechanism to keep and recover the device states; this has been left for future work. Most block I/O devices such as a network, however, are almost stateless or can be easily reinitialized [5].

6 Evaluation

This section presents the evaluation results of our fault detection and recovery mechanism. In particular, we are interested in four aspects: the overhead and the accuracy of the DVM, and the delay and the data loss during DVH.

6.1 Experimental Environment

Our mechanism is implemented on a Xen-3.0.4-1 with a para-virtualized Linux 2.6.16.33 kernel for the x86 architecture. All machines in the evaluation were connected to a dedicated local 1 Gbps Ethernet. The machine for Xen had an Intel Core2 Duo 2.33 GHz CPU with 2 GB of RAM. The memory size of the IDDs and guest domains was set to 256 MB, and the network devices were 1 Gbit (Intel 82541PI). We used several native servers communicating with domains, and they had an Intel Core2 Quad 2.4 GHz CPU with 4 GB of RAM. These machines are expected to run VMs at a reasonable performance level.

6.2 Fault Detection Frequency

To decide on a DVM frequency value, we ran a pre-evaluation test using the *httperf* benchmark [26], [38]. Httperf was modified to log timestamps for each web request. We ran the apache2 web server on a native machine and performed httperf with a guest domain and an IDD. Httperf is configured to fully saturate the network bandwidth of the guest domain. The x-axis of Figure 7 shows how frequently the domain scheduler calls DVM. The number 50 indicates that the DVM is called when the IDD is scheduled every 50 times. Note that Baseline is the vanilla version of Xen. The y-axis shows the average response time of each request, and the error line of each bar shows the maximum and the minimum values.

In the case of 50 and 100, the response time and variation were relatively large. Beyond 150, the response time was significantly reduced and stable. The standard deviation was also stabilized. We assume that 150 is a reasonable value for the DVM frequency, and thus it is used as a default value in the following evaluations. Since this evaluation supposes saturated network throughput, a DVM frequency of 150 is sufficient for general workloads. Below 50, the response time and the standard deviation were extremely high compared with others, and thus were excluded from the results. The DVM frequency is not really affected by the number of guest domains, since the DVM works at periodical scheduling time. An IDD will be scheduled more frequently for more guest domains and more I/O requests.

With respect to the DVM frequency, we also evaluated the delay to detect a fault state with the same configuration. We injected a fault and measured how long the DVM takes to detect it. This evaluation is designed to show how the DVM frequency affects the detection delay. In Figure 8, as the DVM frequency increases, the detection delay becomes larger. At the DVM frequency of 150, it detected faults within 1.35 second. We can conclude that the detection delay is closely related to the DVM frequency, since it decides the monitoring period.

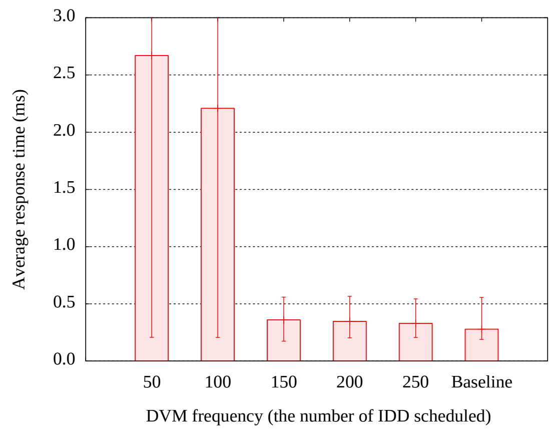


Figure 7: The Httperf average response time for DVM frequencies.

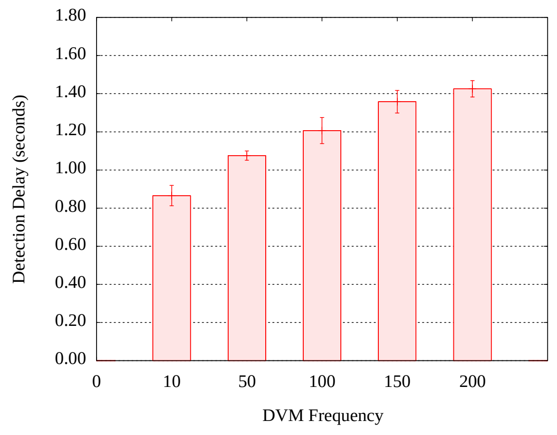


Figure 8: The detection delay for DVM frequencies.

6.3 Fault Detection

For the evaluation of fault detection, we used an automatic fault injection tool, which was also used for Nooks [18, 19], [20] and RIO file cache [39]. The tool injects 27 types of faults into a device driver module. The faults include tainting operation codes, memory access violations, synchronization errors, and so forth. The httperf benchmark is executed from a separate physical machine to a guest domain via an IDD. Most of the injection trials did not produce IDD faults on the first trial, since the tool was designed to inject fault codes at random instruction, according to the assigned type. Therefore, the injections are iterated with different seeds for each fault type until the device driver fails to process the I/O requests. We performed these iterative injections for each of 27 fault types, and repeated them ten times for a fault type. In the automatic fault injection test, the fault time was unpredictable, and the results were nondeterministic, even for the same fault type. There were 263 cases of kernel crash, and only

TABLE 1
THE FAULTS TYPES USED FOR MANUAL INJECTION

Fault types	Activity
NULL access	Access NULL pointer
PANIC()	Call PANIC() macro
Divide by zero	Divide by zero
Infinite loop	while (1) loop
IRQ dead lock	Make dead lock with irq_save
BUG()	Call BUG() macro
kmalloc() loop	Call kmalloc() infinitely
Return 0	Return without excuting function codes
I/O addr. modification	Modify I/O buffer address
Queue entry	Suffle queue entry
Return IRQ_HANDLED	Return without excuting IRQ handler
Spin lock	Hold spin lock

TABLE 2
THE DETECTION RESULTS OF MANUAL FAULT INJECTION

Kernel fault	Result	Detection
NULL-access	N	N
Divide-by-zero	N	N
PANIC()	C	✓
BUG()	C	✓
IRQ dead lock	M	✓
Infinite loop	M	✓
kmalloc() loop	M	✓

Driver fault		Result	Detection
Send	Return 0	M	✓
	I/O addr. modification	M/C	✓
	Queue entry	M	✓
	Return IRQ_HANDLED	M	✓
	Spin lock	M	✓
Recv	Return 0	M	✓
	I/O addr. modification	M/C	✓
	Queue entry	M	✓
	Return IRQ_HANDLED	M	✓
	Spin lock	M	✓

N: No fault, C: Crash, M: Malfunction, M/C: Crash after Malfunction

seven cases of malfunction. In spite of the nondeterministic characteristics, our mechanism was successful at detecting all faults.

In addition to the automatic fault injection tool, we also manually injected faults to see the deterministic behavior and more malfunction cases. We modified the IDD kernel and the device driver to explicitly cause faults. The fault types and activities for the manual injection are described in Table 1, and the detection results are presented in Table 2. For general kernel bugs, null-pointer-access and divide-by-zero did not generate an IDD fault. These types of faults are handled by the kernel fault handler, and the I/O services are not affected by them. Thus, DVM assumes that the IDD is in a normal state. The explicit fault macros such as PANIC() and BUG() cause IDD crashes, and the IRQ dead lock and Infinite loop types caused malfunction. For device driver bugs, the five types of faults were injected to both the sending and receiving codes of the network device driver. Most of faults showed malfunction, and one of them caused a kernel crash after malfunction. In the crash after malfunction case, the DVM detected the malfunction successfully, and the connection had already been switched when the IDD crashed.

Although the detection delay might be prolonged, false negative cases do not occur due to the mechanism of the DVM. Our approach, however, can lead to false-positives. The Dthres value used to decide a fault state is dynamically calculated by the PIRQ and the T_{ratio} value. To evaluate the false-positive ratio, we ran the *iperf* benchmark with the same VM configuration. Figure 9 shows the false-positive ratio as the T_{ratio} value increases. When T_{ratio} is more than 0.16, false-positives did not occur. Based on the results of this evaluation, we determined that the possibility of a false-positive is very low, even with a small T_{ratio} value. This result is due to the boosting mechanism of the Xen domain scheduler, which preemptively schedules an IDD to process pending requests so that the number of pending requests is small, and the responsiveness of guest VMs is increased. In the mean time, the detection delay is not different to the T_{ratio}. We can conclude that the detection delay is more dependent on the DVM frequency rather than the T_{ratio}.

6.4 Recovery Delay and Packet Loss

In this section, we evaluate how fast the DVH recovers from an IDD fault, and how much data might be missed. For the evaluation of recovery delay and packet loss, we made a simple streaming benchmark saturating the network bandwidth using TCP and UDP. With the streaming server on a native machine, we used an IDD and a

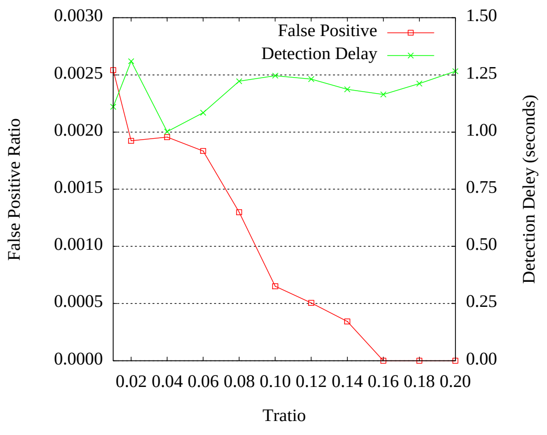
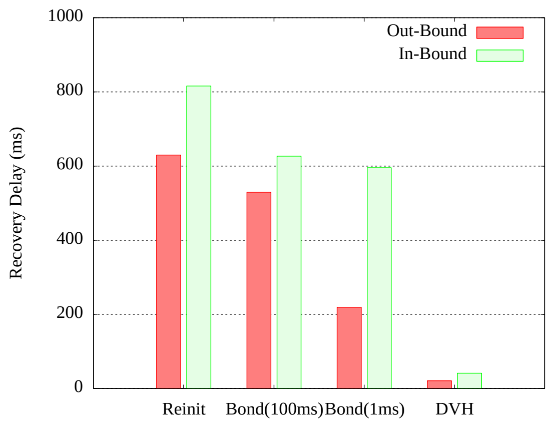


Figure 9: The false-positive ratio and the detection delay for the T_{ratio} value.

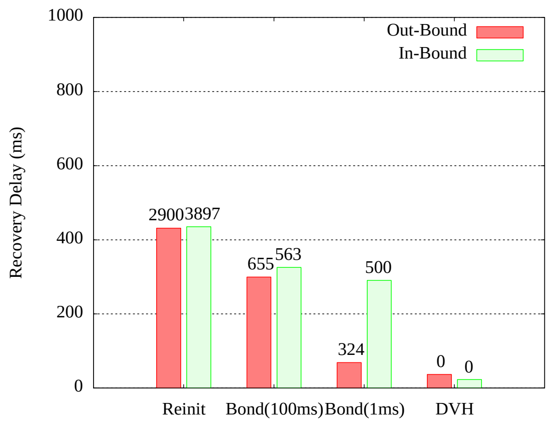
guest domain to run the client for in-bound evaluation, and vice versa for out-bound. For the comparison with DVH, the evaluation was also performed on the *reinitialization* and *Linux bonding driver*. The reinitialization means that the backend driver is reloaded instead of restarting an IDD [5]. It takes additional overheads to reconstruct the I/O rings and the event channel with driver initialization, compared to DVH. The Linux bonding driver [22] shows several network devices as a single interface to the upper layer, and therefore the upper layer cannot see multiple network devices. This increases the bandwidth and enhances the availability of the network devices [23]. Even if a network device incurs faults, the Linux bonding driver lets other network devices back it up.

Figure 10 shows the recovery delay of the reinitialization, the Linux bonding driver, and the DVH. The evaluation was performed on TCP and UDP for each recovery method. The x-axis shows each recovery method, and the y- axis shows the inter-arrival time of the packets when each recovery method is performed. In the case of UDP, the label on the top of each bar reveals the number of lost packets. The average inter-arrival delay of normal operation was 0.25 ms, and the benchmark streams were about 12000 packets per second for TCP and 21000 packets per second for UDP. To minimize the time to process a packet, the data size was configured to 8 bytes.

The reinitialization, denoted as Reinit, showed the largest recovery delay among the three methods. This is mainly due to link detection, I/O ring setup, and event channel configuration. The Linux bonding driver shows a lower recovery delay than the reinitialization. The configuration of the Linux bonding driver, including the monitoring period or delay time, can leverage the switching delay. We evaluated the Linux Bonding driver with two different configurations for the monitoring period and delay time. One is 100 ms, a default value, denoted as Bond100, and the other is 1 ms, a minimum value, denoted as Bond1. Both Bond100 and Bond1 show better recovery delay than Reinit. Further, Bond1, the smaller value, has a smaller recovery delay than Bond100. In contrast, the DVH only took about 3-8% of the switching de-



(a) TCP



(b) UDP

Figure 10: The recovery delays of Reinitialization, Linux bonding driver, and DVH.

lay compared with reinitialization. Most benefits result from the reusing mechanism of the DVH.

The delays of TCP and UDP in the DVH case were almost similar, while the delay of TCP was definitely larger than that of UDP in the case of reinitialization and Linux bonding driver. These results are caused by the reliable data transmission of the TCP. The DVH eliminates this delay by reusing the original I/O structure at the virtualization layer.

During this evaluation, we also monitored how many packets were lost. Owing to the retransmission functionality, the evaluation of TCP does not cause packet loss. On the other hand, all other experiments showed packet loss in UDP, except DVH, and it was proportional to the recovery delay. The packet loss from the reinitialization was more than that of the Linux bonding driver. In the case of DVH, no packet was lost, as expected.

The recovery method for compared targets is different from DVH in that the reinitialization does not require a backup IDD, and the Linux bonding driver simulates two network devices as one in an IDD. This evaluation, how-

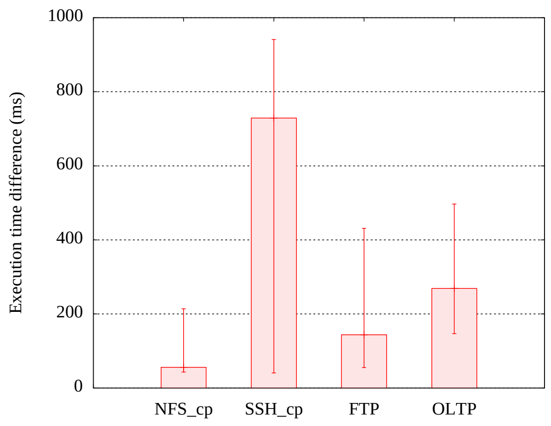


Figure 11: The execution time difference for realistic workloads.

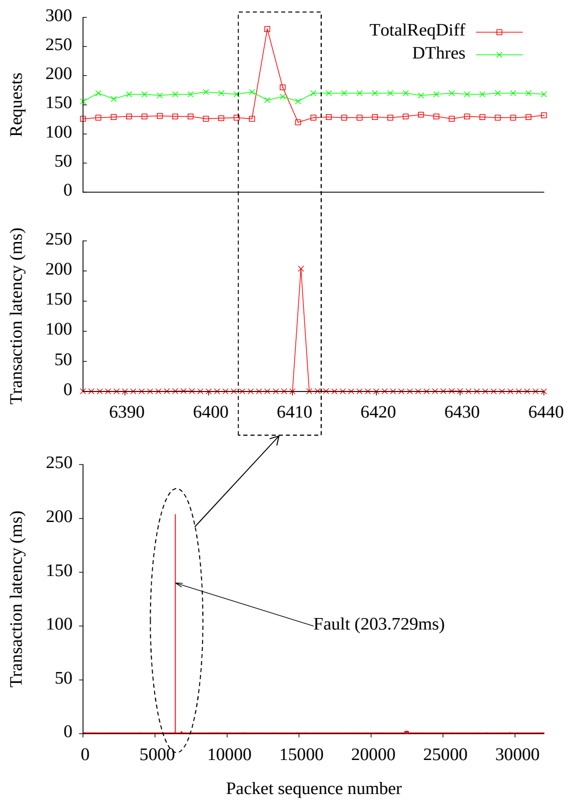


Figure 12: The detailed progress of transactions, TotalReq$_{diff}$, and DThres in the OLTP evaluation.

ever, is to compare only the recovery delay among these methods. From these results, we could verify that reusing the I/O ring with DVH is efficient. In the reinitialization, although we reinitialized only the backend driver, its recovery delay would be much longer if the faulted IDD need to be rebooted. Although the Linux bonding driver shows better results than the reinitialization, it has no way to recover when the IDD crashes or malfunctions. The Linux bonding driver is not an appropriate approach for achieving a fault tolerance of the device driver, since it

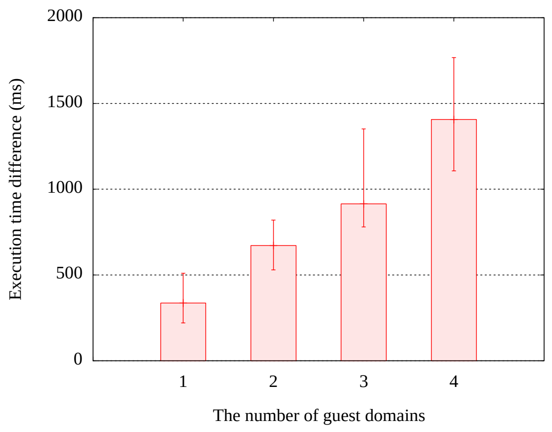


Figure 13: The execution time difference of the OLTP evaluation for multiple guest domains.

was originally designed for high availability, and not for fault recovery.

6.5 Real Workload

For the realistic evaluation of DVM and DVH, we ran several real workloads as shown in Figure 11. In this evaluation, we performed these workloads in a guest domain that was served by an IDD, and the guest domain communicated with a native server machine. The NFS_cp and the SSH_cp denote the copy operation of the Xen kernel source code using NFS and SSH, respectively. FTP represents the download of the Xen kernel source using FTP. OLTP denotes the emulating of online transaction processing (OLTP) with Mysql and sysbench [53], which requests about 130 database transactions per second. Each bar shows the difference between the execution time of the vanilla version of Xen and that of a fault and recovery using our mechanism. We used the difference as an evaluation metric, since the running time of each workload is different. The error bars show the maximum and the minimum difference. We confirmed the integrity of each execution result by using the *diff* command. The variation is large, since the difference relies on when a fault is injected and how many I/O operations are conducted at the injection time. Especially, the case of SSH_cp shows the largest variation, since the computing takes a large portion of the workload. All of the workloads, however, were recovered within a second, and data loss did not occur.

The detailed working of the DVM and DVH of the OLTP workload is presented in Figure 12, which shows the transaction latency, TotalReq$_{diff}$, and DThres for each transaction. When a fault occurs, the DThres decreases slightly, since the number of PIRQs decreases. On the other hand, the TotalReq$_{diff}$ increases due to the pending requests by the fault. Therefore, the DVM determines that this state is a fault. Following this decision, the DVH is initiated, and the transaction latency drops to normal operation.

We performed the same evaluation of the OLTP for multiple guest domains as shown in Figure 13. Each bar shows the averaged execution time difference for a domain. The difference was proportional to the number of guest domains, since the total amount of work to recover from a fault is proportional to the number of guest domains connected an IDD. This result, however, might not be a problem in the real world, since the difference is almost negligible compared to the total execution time of workload.

6.6 Overhead

We evaluated the overhead of our mechanism with respect to performance and resource consumption. Figure 14 shows the normalized throughput for several workloads. This is an evaluation without faults to check fault monitoring overhead. For I/O workload, we ran the httperf benchmark and grep command over the *NFS* file system [40]. We also ran the *stream* benchmark [27, 41] for heavy CPU workload. All workloads were executed in a guest domain via an IDD. The results show that the DVM has negligible overhead for logging monitoring information and for making a fault decision. The DVM shows 2% of performance overhead at most in the stream benchmark due to its heavy CPU-bound workload.

The computational overhead of the DVM and DVH was measured using the *time stamp counter* as presented in Table 4. One execution of DVM consumes 1,755 clocks on average, which are less than 1 us on our 2.66 GHz CPU machine. Actually, the DVM executes only a few lines of codes in the normal state. The DVH codes consume about 2 million clocks, which is 784 us. Besides the code execution, since DVH co-works with the Xen subsystems for event channel operations, grant mappings, and changing protocol, the actual completion with DVH takes more time. Most of the time for DVH completion is consumed by waiting protocol state changes.

To evaluate memory overhead, we summed the memory usage of the DVM and DVH. Most of memory consumption is for the information logged by the DVM. It takes 120 bytes per VM and is linearly proportional to the number of VMs. It is not a significant space overhead.

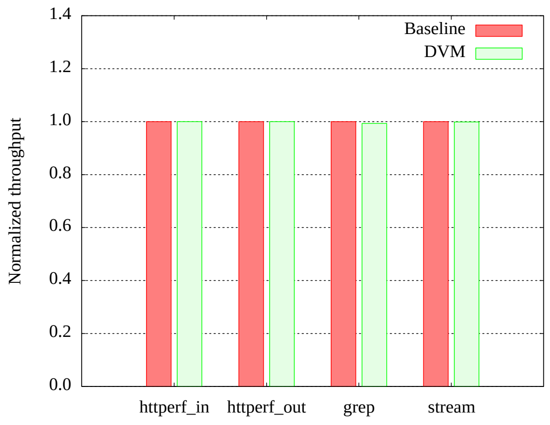


Figure 14: The overhead of DVM.

TABLE 3
THE CPU CONSUMPTION

Component	Clock	Time (us)
DVM	1,755	less than 1
DVH	1,828,029	784

7 Related Work

There has been a lot of previous work on fault detection and recovery mechanisms. The proposed mechanism is an approach to transparently detect and recover from a fault in a device driver in a virtualized system. Thus, it differs in many ways from the fault detection or recovery of native machines.

Douceur et al. proposed MS Manner, which is a progress-based regulation mechanism to prevent resource contention [49]. It employs statistical mechanisms to deal with stochastic progress of processes. They showed a good progress detection mechanism in a generic way. Although the DVM monitors the states of the IDDs similarly, it only uses the limited information in the VMM layer for isolation and transparency.

The fault tolerance of device drivers in a native machine has been widely researched over the past thirty years. Swift et al. proposed Nooks [18], [19] and the shadow driver [17] for the fault tolerance of a device driver. They hook communication between the kernel and device drivers, and then simulate the role of a device driver to hide faults when the device driver failure occurs. Alsberg et al. suggested a single primary-multiple backup scheme in the distributed computing environment [14], and Guerraoui et al. suggested a software-based replication for fault tolerance [16].

Microkernels [43], [44] and their variations [45], [46], [47] proposed another approach for reliability. These systems isolate extensions into separate address spaces and interact with the OSes using kernel communication services, such as messages or remote procedure call [48]. Thus, the overall system is isolated from the faults of an extension within an address space. The Minix 3 is a highly reliable and self-repairing OS based on a microkernel [50]. A failing component is replaced transparently in most cases, and Herder et al. proposed a fault isolation mechanism for device drivers and prototyped their ideas in the Minix 3 [51], [52].

Fraser et al. proposed a safe hardware interface in Xen and the reinitialization of the backend driver for the fault tolerance of virtual machines [5], [6]. Their safe hardware interface becomes the basis of the current I/O architecture of Xen 3.0. With the IDD I/O architecture, the native device drivers are reused without modification, and the failure of a device driver does not lead to a crash of the entire system. This work, however, uses the reinitialization method of the backend driver to recover the device driver, and therefore loses the requests and responses in the I/O rings. LeVasseur et al. [42] proposed a device driver reusing model for the L4 microkernel [43]. This approach enables unmodified device drivers to run with their original operating system and improves the dependability of the system by isolating defective or malicious drivers. With the fault of a device driver, they restart the fault device driver. During the reinitialization, their interface intercepts and delays accesses to the device. Although this approach is successful at reusing device drivers and shows little overhead, it lacks the ability to provide fault detection mechanisms.

8 Conclusion

Server consolidation using virtualization is a viable approach, since a virtualized system allows higher efficiency and lower management cost. As its popularity has increased, virtualization has evolved in its performance for the past decades. Most work on virtualization has focused on performance issues, since the sharing hardware among guest VMs entails significant overhead. On the other hand, the reliability of a virtualized system becomes an important factor in a highly consolidated environment. In particular, considering that most failure is caused by the device driver, the reliability of the driver VMs becomes crucial.

In this paper, we propose a transparent fault detection and recovery mechanism of device drivers for virtualized server systems. Our mechanism is transparent and secure in that it requires no kernel or application level modification, and the detection module is isolated from the address spaces of the guest domains. To achieve these features, all implementation is confined to the virtualization layer. If the DVM detects faults, the hypervisor switches the faulted driver VM to another one, and this process is performed without service disconnection or data loss by fully exploiting the I/O structure of the virtualized system. From the evaluation results, the DVM shows negligible overhead with a low false-positive ratio, and the DVH only takes dozens of milliseconds to recover from a fault.

Acknowledgment

The authors wish to thank A, B, C. This work was supported in part by a grant from XYZ.

References

[1] http://www.vmware.com

[2] R. Short, Vice President of Windows Core Technology, *Microsoft Corp. private communication*, 2003.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. *In Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *In Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164-177, 2003.

[5] K. Fraser, S Hand, R Neugebauer, I Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS)*, 2004.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. *Tech. Report, UCAM-CL-TR-596, 2004.*

[7] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. *In Proceedings of the 2nd International Conference on Software Engineering*, pages 562-570, 1976.

[8] K. Birman, T. Joseph, T. Raeuchle, and A. Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502-508, 1985.

[9] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273-286, 2005.

[10] M. Nelson, B. Lim, and G. Hutchins. Fast transparent migration for virtual machines. *In Proceedings of the USENIX Annual Technical Conference*, pages 25-25, 2005.

[11] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schioberg. Live wide-area migration of virtual machines including local persistent state. *In Proceedings of the 3rd international conference on Virtual execution environments*, pages 169-179, 2007.

[12] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. Oudenaarde, S. Raghunath, and P. Wang. Seamless live migration of virtual machines over the man/wan. *Future Generation Computer Systems*, 22(8):901-907, 2006.

[13] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. *In Proceedings of the 3rd IFIP Conference on Dependable Computing for Critical Applications*, pages 187-198, 1992.

[14] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. *In Proceedings of the 2nd international conference on Software engineering,* pages 562-570, 1976.

[15] S. Chetan, A. Ranganathan, and R. Campbell. Towards fault tolerance pervasive computing. *Technology and Society Magazine, IEEE*, pages 38-44, 2005.

[16] R. Guerraoui and A. Schiper. Towards fault tolerant pervasive computing. *Computer,* 30(4):68-74, 1997.

[17] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering Device Drivers. *In Proceedings of 6th Operating Systems Design & Implementation,* pages 1-16, 2004.

[18] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems*, 22(4):77-110, 2004.

[19] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. *In Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[20] M. Swift, S. Martin, H. Levy, and S. Eggers. Nooks: an architecture for reliable device drivers. *In Proceedings of the Tenth ACM SIGOPS European Workshop*, 2002.

[21] http://www.xensource.com

[22] Linux Ethernet Bonding Driver HOWTO. *Documentation/networking/bonding.txt*

[23] SysKonnect. Link aggregation according to ieee 802.3ad. *Technical report*, SysKonnect GmbH, 2002.

[24] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. *In Proceedings of the ACM Symposium on Operating Systems Principles*, 2001.

[25] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault-Tolerance. *In Proceedings of Symposium on Operating Systems Principles*, pages 1-11, 1995.

[26] http://www.hpl.hp.com/research/linux/httperf/

[27] http://www.cs.virginia.edu/stream/

[28] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. *In Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 195–209, 2002.

[29] I. Molnar. KVM paravirtualization for Linux. *linux kernel mailing list, http://lkml.org/lkml/2007/1/5/205*.

[30] Performance of VMware VMI. *White paper of VMware.*

[31] T. Borden, J. Hennessy, and J.Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104-123, 1989.

[32] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. *In Proceedings of the USENIX Annual Technical Conference*, pages 1-14, 2001.

[33] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *In Proceedings of the 11th international conference on Architectural support for programming languages and operating systems,* pages 211-223, 2004.

[34] H. Chen, R. Chen, F. Zhang, B. Zang, and P. C. Yew. Live updating operating systems using virtualization. *In Proceedings of the 2nd international conference on Virtual execution environments,* pages 35-44, 2006.

[35] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48-56, 2005.

[36] AMD. AMD64 virtualization codenamed pacifica technology: Secure virtual machine architecture reference manual. May 2005.

[37] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. *In Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[38] D. Mosberger, T. Jin. Httperf: A Tool for Measuring Web Server Performance. *In Proceedings of Workshop on Internet Server Performance*, 1998.

[39] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. *In Proceedings of the Seventh ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[40] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. *In Proc. USENIX Summer Conference*, pages 137-152, 1994.

[41] McCalpin, John D. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.

[42] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. *In Proceedings of the 6th Symposium on Operating Systems Design and Implementation,* 2004.

[43] J. Liedtke. On µ-kernel construction. *In Proceedings of the 15th ACM Symposium on Operating Systems Principles,* pages 237-250, 1995.

[44] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian. Mach: A new kernel foundation for UNIX development. *In Proceedings of the Summer USENIX Conference,* pages 93-113, 1986.

[45] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an operating system architecture for application-level resource management. *In Proceedings of the 15th ACM Symposium on Operating Systems Principles,* pages 251-266, 1995.

[46] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: a substrate for OS language and research. *In Proceedings of the 16th ACM Symposium on Operating Systems Principles,* pages 38-51, 1997.

[47] S. M. Hand. Self-paging in the Nemesis operating system. *In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation,* pages 73-86, 1999.

[48] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems,* 8(1):37-55, 1990.

[49] J. R. Douceur and W. J. Bolosky, Progress-based regulation of low-importance processes. *In Proceedings of the 17th ACM Symposium on Operating Systems Principles,* pages 247-260, 1999

[50] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, Minix 3: a highly reliable, self-repairing operating system. *In Proceedings of the 38th International Conference on Dependable Systems and Networks (DSN)*, 2009.

[51] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, Fault Isolation for Device Drivers. *In Proceedings of the 38th International Conference on Dependable Systems and Networks (DSN)*, 2009.

[52] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, Failure Resilience for Device Drivers. *In Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*, pages 41-50, 2007.

[53] http://sysbench.sourceforge.net/