UCLA Computer Science Department Technical Report #140007
University of California, Los Angeles, March 2014.

UNIVERSITY OF CALIFORNIA

Los Angeles

Resilient Virtualized Systems

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

Michael Vu Le

2014

© Copyright by

Michael Vu Le

2014

ABSTRACT OF THE DISSERTATION

Resilient Virtualized Systems

by

Michael Vu Le

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2014

Professor Yuval Tamir, Chair

System virtualization allows for the consolidation of many physical servers on a single physical host by running the workload of each physical server inside a Virtual Machine (VM). This is facilitated by a set of software components, that we call the Virtualization Infrastructure (VI), responsible for managing and multiplexing physical resources among VMs. While server consolidation using system virtualization can greatly improve the utilization of resources, reliability becomes a major concern as failure of the VI due to hardware or software faults can result in the failure of all VMs running on the system.

The focus of this dissertation is on the design and implementation of mechanisms that enhance the resiliency of the virtualized system by way of enhancing the resiliency of the VI to transient hardware and software faults. Given that the use of hardware redundancy can be costly, one of the main goals of this work is to achieve high reliability using purely software-based techniques.

The main approach for providing resiliency to VI failures used in this work is to partition

the VI into subcomponents and provide mechanisms to detect and recover each failed VI component transparently to the running VMs. These resiliency mechanisms are developed incrementally using results from fault injection to identify dangerous state corruptions and inconsistencies between the recovered and existing components in the system. A prototype containing mechanisms proposed in this dissertation is implemented on top of the widely-used Xen virtualized system. In this prototype, three different recovery mechanisms are developed for each Xen VI component: the virtual machine monitor (VMM), driver VM (DVM), and privileged VM (PrivVM).

With the proposed resiliency mechanisms, applications can continue to correctly provide services over 86% of detected VMM failures and over 96% of detected DVM and PrivVM failures. The proposed mechanisms require no modifications to applications running in the VMs and minimal amount of modifications to the VI. These mechanisms are light-weight and can operate with minimal CPU and memory overhead during normal system operations. The mechanisms in this work do not rely on redundant hardware but can make use of redundant resources to achieve, in many instances, sub-millisecond recovery latency.

The dissertation of Michael Le is approved.

Songwu Lu

William Kaiser

David A. Rennels

Yuval Tamir, Committee Chair

University of California, Los Angeles

2014

To my parents.

Table of Contents

Abstract ii

Acknowledgments xv

Vita and Publications xvi

Chapter One - Introduction 1

1.1. Motivation 2

1.2. Thesis Contributions 4

1.3. Thesis Organization 6

Chapter Two - Background and Related Work 8

2.1. System Virtualization 8

2.2. Microreboot 12

2.3. Fault Model 13

2.4. Reducing Failures in Virtualized Systems 14

2.4.1. Reducing Complexity 14

2.4.2. Rejuvenation 16

2.5. Enhancing the Resiliency of Virtualized Systems 18

2.5.1. Failure Detection 18

2.5.2. Enhancing the Resiliency of Virtual Machines 20

2.5.2.1. VM Replication 21

2.5.2.2. VM Checkpointing 22

2.5.3. Enhancing the Resiliency of the Virtualization Infrastructure 24

2.5.3.1. Tolerating Device Driver Failures 25

2.5.3.2. Tolerating OS Kernel Failures 27

2.6. Software-implemented Fault Injection 28

Chapter Three - Applying Microreboot to System Software 31
Chapter Four - Resilience to Device Driver Failures in Driver VMs 37
4.1. Impact of Device Virtualization 39
4.2. Design Tradeoffs 40
4.2.1. Failure Detection 41
4.2.2. System Configurations 42
4.2.3. DVM Replacement 44
4.2.4. AppVM-DVM Connections 46
4.2.5. Failover at the AppVM 47
4.3. Implementation Challenges 49
4.3.1. Mitigating Memory and Device Corruption 49
4.3.2. Preventing Interrupt Blocking 51
4.3.3. Enhancing Failure Detection in AppVMs 52
4.3.4. Frontend Supporting Multiple Backends 53
4.4. Evaluation Methodology 54
4.4.1. Recovery Schemes Evaluated 55
4.4.2. Fault Injection Campaigns 56
4.4.3. Benchmarks 59
4.4.4. Latency Measures 60
4.4.5. Failure Detection 61
4.5. Experimental Results 62
4.5.1. Recovery Effectiveness 64
4.5.2. Recovery Latency 65
4.5.3. Impact of No. AppVMs on Recovery Latency 67
4.5.4. Service Interruption Latency 69
4.5.5. Sharing CPUs and Service Interruption Latency 70

4.5.6. DVM Recovery with a Real Application 72
4.6. Summary 74
Chapter Five - Fast Device Driver Recovery 76
5.1. Experimental Setup 77
5.2. Optimizing Network Device Driver Recovery 79
5.3. Recovering from Failures of other NIC Drivers 88
5.4. Summary 90
Chapter Six - Resilience to Privileged VM Failures 91
6.1. PrivVM Configuration 92
6.2. Failure Detection 92
6.3. Basic Recovery 93
6.4. Experimental Setup 95
6.5. Improving Recovery 98
6.6. Effectiveness of Detection 102
6.7. Summary 102
Chapter Seven - Resilience to VMM Failures 103
7.1. Tolerating VMM Failure 105
7.1.1. Detection 106
7.1.2. Repairing VMM Corruption 107
7.1.3. Resolving Inconsistencies 110
7.2. Transparent VMM Microreboot 115
7.2.1. Preserving VMM and VM States 116
7.2.2. Re-integrating VMM and VM States 119
7.3. Improving Recovery 120
7.4. Experimental Setup 129
7.4.1. System Configurations 130

7.4.2. Workloads 133
7.4.3. Fault Injection Campaigns and Fault Types 134
7.4.4. Numbers of Injections 136
7.4.5. Fault Injection Outcome 137
7.4.6. Failure Detection 137
7.5. Impact of VMM Modifications on ReHype 138
7.6. Enhancements to ReHype 141
7.7. Analysis 144
7.7.1. Recovery Effectiveness 144
7.7.2. Silent Failures 149
7.7.3. Impact of Injection Distribution 151
7.8. Validating ReHype on Bare Hardware 153
7.9. Validating ReHype with FV VMs 156
7.10. Recovery Latency 160
7.11. Summary 162
Chapter Eight - Resilience to Virtualization Infrastructure Failures 164
8.1. FTVI Framework 164
8.2. Experimental Setup 165
8.3. Effectiveness Results 169
8.4. Implementation Complexity 173
8.5. Summary 174
Chapter Nine - Gigan Fault Injector 175
9.1. Software-Implemented System Fault Injection 177
9.2. Virtualization and Fault Injection 178
9.2.1. Virtualization Facilitating SWIFI 178
9.2.2. Fault Injection Performance 181

9.2.3. Implementation Challenges 184
9.2.4. Evaluating Virtualized Systems 187
9.3. Fault Injector Design and Implementation 189
9.3.1. Basic Capabilities 189
9.3.2. Page Table Injection Fidelity 190
9.3.2.1. Paravirtualized Page Tables 191
9.3.2.2. Fully-virtualized Page Tables 194
9.3.3. Logging in a Virtualized System 195
9.3.4. Measuring Latency in VMs 195
9.4. Experimental Setup 199
9.5. Evaluating the Fault Injector 201
9.5.1. Fault Isolation Enabling Fault Injection 202
9.5.2. VM vs. Bare Hardware 203
9.5.3. Impact of Injector in Target System 208
9.5.4. Impact of Logging Mechanism 212
9.5.5. Fault Injection Performance 213
9.6. Gigan Deployments 214
9.6.1. Evaluating Non-virtualized Systems 215
9.6.2. Evaluating Virtualized Systems 217
9.7. Summary 218
Chapter Ten - Summary and Conclusions 220
Bibliography 223

List of Figures

2.1 Components in a virtualization infrastructure 9

2.2 Types of VMM 10

2.3 Isolated driver VM architecture 10

4.1 System configurations for AppVM device access 43

4.2 Service interruption with *netbench* using MultiNetIf recovery 69

4.3 Service interruption with *blkbench* using RAID recovery 70

5.1 Basic target system configuration showing a PrivVM, NetDVM, and AppVM 78

5.2 Network interruption latency using fast failover spare NetDVM 86

6.1 System configuration for evalauting PrivVM. 96

7.1 Layout of system memory for ReHype. 117

7.2 System configuration for 1AppVM. 131

7.3 System configuration for 3AppVM. 131

7.4 System configuration for 3AppVM deployed inside a FV VM 131

7.6 Fault injection results of ReHype: FTVI_HD vs. FTVI_HDP 140

8.1 Overview of the FTVI showing the main components providing resiliency 164

8.2 System configuration for evalauting FTVI. 166

8.3 Structure of Linux Virtual Server 167

8.4 Fault injection results into VMM hosting LVS 170

8.5 Fault injection results into DVM hosting LVS 171

8.6 Fault injection results into PrivVM hosting LVS 171

9.1 Fault injection in a virtualized system 179

9.2 Page table injection scenarios violating fault isolation. 192

9.3 VMM injection outcome distributions: FV VM vs. bare hardware. 204

9.4 VMM crash cause distributions: FV VM vs. bare hardware. 204
9.5 Kernel injection outcome and crash cause distributions. 207
9.6 Kernel crash latency distribution. 207
9.7 Kernel injection outcome distributions: inside vs. outside 209
9.8 Kernel crash cause distributions: inside vs. outside 209
9.9 VMM injection outcome distributions: inside vs. outside 210
9.10 VMM crash cause distributions: inside vs. outside 210

List of Tables

4.1 DVM recovery mechanisms implemented and evaluated 55

4.2 Types of faults injected 58

4.3 Injection outcome and effectiveness of different DVM recovery schemes 63

4.4 Recovery latency of different recovery mechanisms 65

4.5 Impact of No. AppVMs on recovery latency 66

4.6 Request latency Apache benchmark 73

5.1 Device driver injection results with recovery 81

6.1 Fault injection results into the PrivVM. 98

7.1 Recovery improvement description 121

7.1 Improvements over the basic ReHype recovery. 121

7.2 Injection outcomes. 122

7.3 Fault injection results for 1AppVM system setup. 123

7.4 Fault injection results using 3AppVM system setup. 128

7.5 Comparing success rates of ReHype in FTVI_HDP and FTVI_HD 139

7.7 Successful recovery rate when using the enhanced *Ack interrupts* mechanism. 143

7.8 Recovery success rates of ReHype across different fault types 145

7.9 Percentages of manifested faults that result in silent failures 149

7.10 Fault injection results based on weighted VMM execution time 152

7.11 Recovery success rates across different fault types: bare hardware 154

7.12 Percentages of manifested faults that result in silent failures: bare hardware 154

7.13 Recovery success rates of ReHype recovering VMM hosting FV AppVMs. 158

7.14 Percentages of silent failures: VMM hosting FV AppVMs 159

7.15 Breakdown of recovery latency using ReHype. 160

8.1 Recovery success rates for different VI components: VMM hosting LVS 172
8.2 Lines of code needed to implement different microreboot mechanisms 173
9.1 Distributions of crash latency measurements: FV VM vs. bare hardware. 206
9.2 Distributions of crash latency measurements (Linux): inside vs. outside. 211
9.3 Distributions of crash latency measurements (VMM): inside vs. outside. 211
9.4 Time breakdown, in seconds, of operations in a single injection run. 214

ACKNOWLEDGMENTS

I would like to acknowledge the following people for keeping the light on at the end of the tunnel. First and foremost, my advisor Yuval Tamir, for his *almost* never ending patience and guidance. Needless to say, without him, this work would not have been possible. My family for their unwavering support and trust that spending the golden years of my youth in pursuit of a doctorate is worthwhile. Andrew Gallagher for his friendship and technical contributions to parts of this work. My current and former CSL labmates for their antics, friendships, and above all, their shoulders to lean on through the slog that is grad school. For those who should be on this list but are not, my sincerest apologies. Consider yourselves formally acknowledged.

VITA

2004	B.S., Information and Computer Science School of Information and Computer Science University of California, Irvine Irvine, California
2008	M.S., Computer Science Department of Computer Science University of California, Los Angeles Los Angeles, California
2005-2014	Teaching Assistant/Graduate Student Researcher Department of Computer Science University of California, Los Angeles Los Angeles, California

PUBLICATIONS

M. Le and Y. Tamir, ‘‘Applying Microreboot to System Software,’’ *IEEE Int. Conf. on Software Security and Reliability*, pp.11-20 (June 2012).

M. Le, I. Hsu, and Y. Tamir, ‘‘Resilient Virtual Clusters,’’ *17th IEEE Pacific Rim International Symposium on Dependable Computing*, pp.214-223 (December 2011).

M. Le and Y. Tamir, ‘‘ReHype: Enabling VM Survival Across Hypervisor Failures,’’ *7th ACM Int. Conf. on Virtual Execution Environments*, pp.63-74 (March 2011).

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

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, pp.195-202 (July 2009).

K. Lee, M. Le, J. Harri, and M. Gerla, ‘‘LOUVRE: Landmark Overlays for Urban Vehicular Routing Environments,’’ *2nd IEEE International Symposium on Wireless Vehicular Communications* (September 2008).

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* (April 2008).

M. Le, J.-S. Park, and M. Gerla, ‘‘UAV Assisted Disruption Tolerant Routing,’’ *IEEE Military Communications Conference* (October 2006).

Chapter One

Introduction

System virtualization allows multiple Virtual Machines (VMs), each with its own workload and OS, to run on a single physical host [Rose05]. This is facilitated by a set of software components, that we call the Virtualization Infrastructure (VI), responsible for managing and multiplexing physical resources among VMs. A common use of system virtualization is to consolidate many physical servers onto a single physical host by running the workload of each physical server inside a VM. The use of system virtualization for server consolidation improves resource utilization and eases system management operations. However, reliability becomes a major concern since failure of the VI due to faults can result in the failure of all VMs running on the system.

The focus of this dissertation is on the design and implementation of software-based mechanisms that enhance the resiliency of virtualized systems through enhancing the resiliency of the VI to transient hardware and software faults. The main approach for providing resiliency to VI failure is to partition the VI into subcomponents and provide mechanisms to detect and recover each failed VI component transparently to the running VMs.

Since a faulty VI component can potentially cause arbitrary corruption and inconsistency in a system, one of the main challenges of this work is determining which corruption and inconsistency are most likely to result in recovery failure and devise mechanisms to fix them. To address this challenge, the development of the VI resiliency mechanisms is done

incrementally using results from fault injection as a guide in determining the causes of recovery failures.

To aid in the development of the resiliency mechanisms and ultimately to evaluate the effectiveness of these mechanisms, a software-implemented fault injector is developed. This injector leverages facilities provided by system virtualization to allow transient hardware and software faults to be injected into virtualized as well as non-virtualized systems with minimum intrusion and campaign run-time overhead.

A prototype containing mechanisms proposed in this dissertation is implemented on top of the wide-used Xen virtualized system [Barh03]. In this prototype, three different recovery mechanisms are developed for each Xen VI component: the virtual machine monitor (VMM), driver VM (DVM), and privileged VM (PrivVM). All of these resiliency mechanisms together form the core of our fault tolerant VI (FTVI). With our FTVI, VMs running applications performing intensive network and disk I/O operations can continue to correctly provide services over 86% of detected VMM failures and over 96% of detected DVM and PrivVM failures.

The motivation for the research in this dissertation is provided in the following subsection. Research contributions are then presented followed by a description of how the rest of this dissertation is organized.

1.1. Motivation

In the past, with commodity hardware, clusters of interconnected servers had to be used to meet the high performance requirements of some applications. To meet the performance

demands, each server runs a single application or provides services to a single client. As performance of individual servers improve over the years, the servers in the clusters are becoming under-utilized a larger fraction of time. With the availability of system virtualization and more powerful multi-core systems, workloads from many physical servers can be efficiently consolidated on a single physical host. Besides improved resource utilization, system virtualization also provides better facilities for managing the cluster such as support for VM migration and dynamic resource allocation.

Unfortunately, server consolidation using system virtualization has drawbacks. In particular, a cluster of VMs (virtual cluster) is less reliable than a cluster of physical servers (physical cluster). In a physical cluster where servers are physically separated, a single fault can at most cause one server to fail. The services running on the physical cluster can be made to tolerate such failures by employing well known fault tolerance mechanisms such as replication and failover. These same mechanisms can also be used on a virtual cluster to increase reliability. However, on a virtual cluster, a single fault that affects the physical host or the VI can result in the failure of multiple or all VMs in the system. This is likely to overwhelm the typical fault tolerance mechanisms designed for clusters. Hence, the existence of these single points of failure is a major barrier to the use of system virtualization for server consolidation.

Current mechanisms allowing a virtual cluster to tolerate failure of the physical host or VI require the use of hardware redundancy [Tech14, VMwa10, Bres96, Cull08]. However, for transient faults, which have been found to occur more often than permanent faults [Gray86], the use of hardware redundancy to tolerate failure is costly and unnecessary. Given that services provided by the VMs can be made more reliable by employing existing cluster-based

fault tolerance mechanisms, the resiliency of the virtualized system is, therefore, hinged on the ability of the VI to tolerate transient hardware and software faults, which is the main topic of this dissertation.

1.2. Thesis Contributions

Following the discussion from the previous subsection, an aggressive reliability goal for a virtualized system is to do no worse than a cluster of physical servers. The subject of this dissertation is on the development of key mechanisms used to enhance the resiliency of the VI in order to meet this goal. Hence, with the proposed mechanisms, if a transient hardware or software fault in any component causes only one VM running applications to fail, the goal is met. To evaluate the effectiveness of the proposed resiliency mechanisms, a fault injector designed to inject faults into virtualized systems is developed. The main contributions of this dissertation can be summarized as follows:

- An analysis of the unique challenges to performing recovery by rebooting subcomponents in system software and general approaches to addressing these challenges are discussed.
- A set of resiliency mechanisms to detect and recover a VI component failure transparently to applications running in the VMs. A prototype of the proposed mechanisms is implemented for the Xen virtualized system, providing resilience with respect to faults in all three Xen VI components: the virtual machine monitor (VMM), driver VM (DVM), and privileged VM (PrivVM).
- An extensive evaluation and analysis of the effectiveness of the proposed resiliency

mechanisms using fault injection. The fault injection campaigns stress the resiliency mechanisms of the VI using transient hardware and software faults while the VI is hosting VMs running a variety of workloads.

- A systematic exploration of different approaches for providing resiliency to device driver failures in the isolated driver VM architecture, a typical architecture for sharing devices in commodity virtualized systems. The design tradeoffs and a taxonomy for classifying these design alternatives are also presented along with an evaluation of key design points based on actual implementations in a variety of system configurations.
- The development of a software implemented fault injector, called *Gigan*, that can be used to evaluate resiliency mechanisms in virtualized and non-virtualized systems. Gigan allows a variety of faults to be injected into different components of a virtualized system and leverages system virtualization to provide low-intrusive log collection and efficient campaign automation capabilities.

The proposed set of VI resiliency mechanisms do not require any modifications to the applications running in the VMs and require only a small amount of modifications to the VI and kernels of VMs running applications. In addition, these mechanisms are light-weight and do not intrude on the normal operations of the system. Furthermore, these mechanisms minimize service interruption time since only the failed components must be recovered.

The work in this dissertation is not the first to have recognized the importance of allowing services provided by the VMs to survive failure of the physical host or components in the VI. Several works have proposed mechanisms that leveraged hardware redundancy to allow uninterrupted VM execution across complete physical host failure [VMwa10, Tech14,

Bres96, Cull08]. Other works aim to reduce the chances of failure or the effects of failure by simplifying and isolating components in the VI [Fras04, LeVa04, Murr08] or by proactively removing faults from the VI through periodic rejuvenation [Kour10]. Still other works try to specifically address the issue of recovering a failed VI component, in particular the DVM [Fras04, LeVa04, Jo10, Herd07]. However, no work exists that provides a comprehensive set of mechanisms to tolerate failure of the entire VI without requiring the use of hardware redundancy.

The development of resiliency mechanisms typically requires the use of fault injection to evaluate and validate those mechanisms. Extensive work in software-implemented fault injectors have existed [Hsue97, Carr99], including works that have used virtualized systems for fault injection [Buch01, Sieh02, Buch03, Poty07] or have used fault injection to evaluate mechanisms or properties of virtualized systems [Pham11, Jeff12]. Unfortunately, these injectors were either not built for commodity virtualized systems or lack features needed to evaluate the resiliency mechanisms of a VI running directly on bare hardware.

1.3. Thesis Organization

The rest of this dissertation is organized as follows:

In Chapter 2, background materials on system virtualization, microreboot, and fault model are presented. In addition, previous works related to enhancing the resiliency of virtualized systems are presented which include mechanisms to increase reliability of VMs and of the VI. Also included is a discussion of works related to performing fault injection in virtualized systems.

Chapter 3 discusses challenges of applying microreboot to system software and general approaches to resolving those challenges. These general approaches are used throughout this dissertation to guide the implementation of mechanisms to recover Xen VI components.

Chapter 4 presents a systematic analysis of different techniques for detecting and recovering failed device drivers in a virtualized system. A taxonomy of the different approaches is presented along with an evaluation of key design options. Chapter 5 discusses mechanisms for reducing device driver recovery latency on systems without redundant devices, in particular network devices.

Chapter 6 discusses mechanisms to recover the PrivVM and Chapter 7 discusses mechanisms to recover the VMM. An evaluation and analysis of the effectiveness of those mechanisms using fault injection is presented in each of the respective chapters.

Chapter 8 presents the overall framework of our fault tolerant VI (FTVI) and explains how all the resiliency mechanisms fit together. Fault injection results demonstrating the effectiveness of our combined resiliency mechanisms at allowing applications running in the VMs to survive failure of any VI component is also presented.

Chapter 9 discusses the *Gigan* fault injector built to evaluate and validate the proposed resiliency mechanisms. An analysis of the interactions between fault injection and system virtualization is presented along with an evaluation of the use of fault injection in virtualized systems.

Chapter 10 gives a summary of the work presented in this dissertation and briefly points out directions for future work.

Chapter Two

Background and Related Work

The focus of this dissertation is on improving the reliability of virtualized systems through improving the resiliency of the virtualization infrastructure (VI). One of the main goals of this work is to develop mechanisms that can provide VI resiliency without relying on redundant hardware.

To facilitate the discussion of the topics presented in this dissertation, Section 2.1 presents background materials on system virtualization. Section 2.2 discusses the microreboot approach to recovery and Section 2.3 presents our fault model. Related work on building reliable virtualized systems by reducing the probability of failures in those systems is presented in Section 2.4. Section 2.5 discusses prior work on enhancing the resiliency of the virtualized system. Related work on software-implemented fault injection is presented in Section 2.6.

2.1. System Virtualization

System-level virtualization allows multiple VMs, each with its own OS, to run on a single physical host [Rose05]. This is facilitated by a set of software components, that we call the *Virtualization Infrastructure* (VI), responsible for managing and multiplexing physical resources among VMs. A typical VI, shown in Figure 2.1, is composed of three components: virtual machine monitor (VMM), driver VM (DVM), and privileged VM (PrivVM/Dom0). We refer to VMs that are not part of the VI as application VMs (AppVMs). The VMM

isolates VMs from each other so that activities in one VM cannot affect another VM [Rama07]. Isolation is maintained by controlling accesses to shared hardware resources such as CPU, memory, and I/O devices. Special dedicated DVMs are used to directly control devices and provide device access to other VMs on the system [Fras04]. The PrivVM is used to perform system management operations such as creating, destroying, and checkpointing VMs. The VMM does not permit these operations to be invoked by any other VMs. The following paragraphs describe these VI components in more detail, focusing on the particulars of the Xen [Barh03] VI which is used as a platform for implementing the resiliency mechanisms proposed in this dissertation.

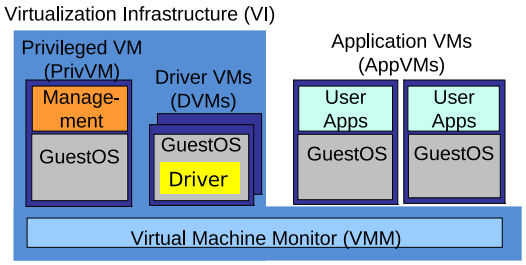


Figure 2.1: Components in a virtualization infrastructure.

A VMM can be run directly on bare hardware or on top of an existing operating system. As shown in Figure 2.2, the former type of VMM is referred to as type 1 VMM while the latter is known as type 2 VMM. The work in this dissertation focuses on type 1 VMM as this type of VMM is typically deployed in platforms used for server consolidation. Type 1 VMM also has the advantage of having a smaller code base, thus reducing the amount of work necessary to harden the system software against faults. The VMM in the Xen VI is a type 1 VMM.

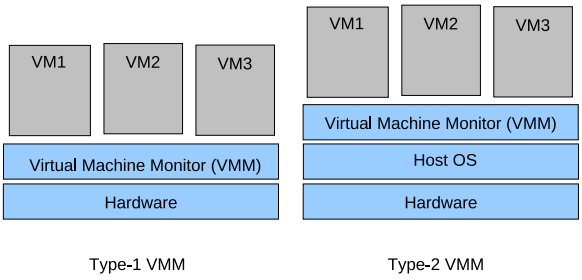


Figure 2.2: VMM types

A common VI organization for allowing multiple VMs to share I/O devices is called the isolated driver VM architecture (IDVM) [Fras04, LeVa04, Micr]. The IDVM architecture, shown in Figure 2.3, facilitates the sharing of devices among VMs using drivers that are split into *frontends* and *backends* [Fras04, LeVa04, Micr]. 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. 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.

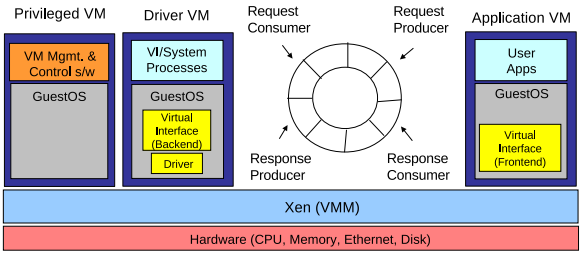


Figure 2.3: Isolated driver VM architecture

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 VI, 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. The requests are often pointers to pages in memory containing the data, such as a packet that a VM wishes to send. These pointers, called grant references, allow one VM to give access to certain memory pages to other VMs. Once a request or response is placed on the shared ring, an event can be sent notifying the other side of the pending request/response. The use of the ring is coordinated using request/response producer/consumer indices, as shown in Figure 2.3.

The functionality of the PrivVM is provided by a combination of kernel modules and user-level processes running in the PrivVM. One user-level process, *XenStored*, provides access to a dynamic database of system configuration information, called XenStore. XenStored also provides mechanisms for VMs to be informed of changes to certain configuration states by allowing VMs to register *watches* on those states in the XenStore. A VM communicates with the XenStore through XenStored using a shared ring data structure, similar to the communication mechanism between a DVM and AppVM. VM management operations are controlled and performed through a user-level process called *Xend*. Xend receives requests for management operations and carries out these requests by interacting with the VMM and XenStored.

To initialize a frontend-backend connection, grant references (for setting up the shared ring) and event channel ports must be communicated between frontend and backend drivers residing on different VMs. This is done using the XenStore. Since communication through the XenStore is asynchronous, involving multiple levels of indirection, operations are slow

and are typically only used for frontend-backend connection setup and teardown.

It should be noted that, in typical 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 system configuration for building resilient virtualized systems (see Chapter 6 for more details).

There are two types of virtual machines: paravirtualized (PV) [Rose05] and fully-virtualized (FV) [Dong06]. PV VMs are aware that they are running within a virtual system while FV VMs are not. PV VMs are modified (paravirtualized) to incorporate this knowledge and the modification must be carried over to newer generations of the kernel. Dedicated hardware mechanisms are required to support FV VMs [Uhli05]. While hardware support for virtualization is becoming common [Uhli05], PV VMs are still needed to serve as the PrivVM and DVMs.

Transitions from the VM to the VMM are called VM exits and subsequent resumption of the VM by the VMM is called a VM entry. VM's CPU state is saved and restored across VM exits and entries, respectively.

2.2. Microreboot

The main approach for providing resiliency to VI failures used in this work is to partition the VI into subcomponents and provide mechanisms to detect and recover (reboot) each failed VI component transparently to the running VMs. This technique of rebooting subcomponents in a system for recovery is known as *microreboot* [Cand04]. A brief overview of this technique is presented in this subsection.

When system components fail, a simple way to recover is to reboot the entire system. This is slow and reduces system availability. Microreboot is a recovery technique that reduces the down time caused by failure of some components in the system by recovering (rebooting) only the failed components in the system while allowing other functioning components to continue operating normally [Cand04].

As explained in [Cand04], there are three main design goals for microrebootable software: (1) fast and correct component recovery, (2) localized recovery to minimize impact on other parts of the system, and (3) fast and correct reintegration of recovered components. To achieve these goals, software components need to be loosely coupled, have states that are kept in dedicated state store separate from program logic, and be able to hide component failures by providing mechanisms for retrying failed requests. To decrease recovery latency, the software components should be fine-grained to minimize the restart and reinitialization time. Lastly, to facilitate cleaning up after microreboots and prevent resource leaks, system resources should be leased and requests should carry time-to-live values.

While the Xen VI does not satisfy all requirements of a ‘‘perfect’’ microrebootable system, it has a modular design that when combined with some modest amount of engineering effort discussed in this dissertation can be made to work well with the microreboot technique.

2.3. Fault Model

System failures can be caused by permanent/transient faults in hardware [Mukh05] or software. Software faults are bugs in the design or implementation. The types of faults that affect a system determine the recovery mechanisms that must be employed.

In this dissertation, we assume that VI software is for the most part correct and functional. Hence, failures can only be caused by faults in hardware or software. We focus on transient hardware and software faults as these types of faults have been found to occur more often in deployed systems [Gray86]. With respect to software faults (bugs), we assume they are rare and occur only under particular timing and ordering of asynchronous events in the system (Heisenbugs) [Gray86]. Thus, a recurrence of failures caused by these types of faults after a VI component has been recovered are unlikely.

2.4. Reducing Failures in Virtualized Systems

This section discusses previous works that have investigated ways to enhance the reliability of applications running in virtualized systems by reducing the probability of failures in these systems. The following subsections discuss works that focus on reducing the complexity of the VI and works that make use of software rejuvenation to counteract problems associated with software aging in both VMs and the VI.

2.4.1. Reducing Complexity

One way to reduce the probability of system failure is to simplify the system by separating and isolating components that are complex or prone to errors. This general approach has been applied to virtualized systems. In particular, as discussed below, previous works have tried to decouple components in the PrivVM and VMM in order to help improve reliability and enhance security.

Fraser et al. [Fras04] separated device drivers from the PrivVM for the Xen [Barh03] virtualized system. These drivers are placed in their own VMs, called driver VMs (DVMs),

isolated from the rest of the system. Failure of a device driver is therefore contained within the DVM and does not cause failure of other system components. Work from LeVasseur et al. [LeVa04] also investigated the benefits of isolating device drivers using VMs for the L4 microkernel.

Murray et al. [Murr08] investigated ways to decouple the VM creation ability from the PrivVM for the purpose of improving system security. In this work, the VM creation functionality in the PrivVM is moved to a different VM. This new VM's sole purpose is to receive and execute VM creation commands from the PrivVM. In addition, the ability for user-space applications in the PrivVM to make privileged hypercalls (e.g. mapping foreign VM pages) have been removed to prevent rogue processes in the PrivVM from harming the virtualized system.

Despite the relatively small code size of VMMs compared to commodity OSs like the Linux kernel, there are works that try to further simplify the VMM in order enhance security or improve reliability. In particular, Steinberg et al. [Stei10] propose a VMM architecture that splits the VMM into a critical privileged component and unprivileged components. In their work, the VMM is only 9K LOC compared to about 100K for Xen. The micro-VMM only contains the scheduler, MMU, communication mechanisms between the micro-VMM and unprivileged VMM components, and a smaller set of hypercall handlers. The unprivileged VMM components run in the root unprivileged space while the VMs run in non-root space. The unprivileged VMM components handle instruction emulation, device and BIOS emulation, and maintains multiprocessor contexts for the VMs. Device drivers are run at the same privilege level as the unprivileged VMM components. The unprivileged VMM components interact with these drivers to access I/O.

One extreme approach of reducing the size of a VMM is to remove the VMM altogether from the virtualized system. Keller et al. proposes such a system [Kell10]. In this work, VMM functionality is provided through the hardware and resources in virtualized system cannot be over subscribed. Each CPU core can run only one VM, physical memory is statically partitioned and assigned to VMs, and VMs are given direct access to devices. While the work claim that there is no VMM, there is still small amount of code that must be run with VMM privilege in order to manage the VMs (start/stop/migrate). This manager code only needs to run when managing VMs and is not run during normal system operations. In this configuration, the VMs never perform any exits to the VMM during normal operations.

2.4.2. Rejuvenation

To reduce the probability that a fault will manifest as a failure, proactive fault tolerance can be employed. Proactive fault tolerance requires the ability to accurately predict when failures are about to occur and gracefully take actions to prevent the pending failure from causing system outage. This graceful action takes the form of rejuvenation, a technique to get rid of faults in a system by cleanly rebooting the entire system or subcomponents of a system. This general technique has been applied to both VMs running applications and the VI. The following paragraphs discuss the main works in this area in more detail.

In terms of proactive fault tolerance in VMs, the graceful action entails cleanly rebooting the entire VM. Of course, while the VM is being booted, the service provided by that VM is not available. Works in this area try different ways to make use of the virtualized environment to reduce the downtime of services provided by these VMs across a proactive reboot.

Silva et al. [Silv07] propose the use of virtualization to perform VM rejuvenation. The

VM-FIT project by Reiser et al. [Reis07a, Reis07b] aims to increase system availability by booting a new instance of the VM on the same host while the original VM is still running to minimize downtime associated with rejuvenation. To achieve the same level of availability in a non-virtualized environment would require spare physical hosts. To improve availability further, Distler et al. [Dist08] aims to reduce the downtime associated with transferring application state to the rejuvenated VM by leveraging the VI. Application state must first be gotten from all the applications and converted to a non-application specific format. Application states include states in memory and on disks. These states can be transferred through shared memory or by a virtualized disk. Works from Nagarajan et al. [Naga07] and Vallee et al. [Vall08] use live VM migration to proactively move a VM from an unhealthy node detected using hardware sensors on the node.

The RootHammer project by Kourai and Chiba [Kour10] aims to rejuvenate critical components of the VI while preserving the VMs running on the system. The focus of this work is to increase availability by decreasing rejuvenation time of a virtualized system. In particular, the Xen virtualized system is proactively rejuvenated by quickly rebooting the VMM and Domain 0 while preserving the state of the running VMs. During rejuvenation, Domain 0 is properly shutdown and the VMM signals the VMs to suspend themselves by cleanly disabling device access and halting all running VCPUS. VM memory is preserved across the reboot along with a reserved memory area that contains saved CPU state and VM configuration. The mechanism to quickly reboot a new instance of the Xen VMM and Domain 0 without going through the BIOS based on the kexec mechanism [Pfif03]. When the Xen VMM and Domain 0 has been booted, to re-integrate the preserved VMs and to restore the state of Domain 0, Domain 0 must re-instantiate and resume the VMs. This re-

instantiation and resumption requires modifications to tools in Domain 0 to work with VM configuration and state already resident in memory.

The mechanisms employed in RootHammer is directly applicable to tolerating VMM failure as recovery of the VMM should also preserve the state of the running VMs on the system. While much of the mechanism of RootHammer can be used to tolerate a VMM failure, there must be additional mechanisms to handle recovery of a failed VMM whose state may be corrupted and have inconsistencies within the VMM state and between the VMM and VMs and VMM and hardware.

2.5. Enhancing the Resiliency of Virtualized Systems

This section discusses previous works that have investigated ways to enhance the reliability of applications running in virtualized systems by mean of enhancing the resiliency of the VMs and the VI to faults. These works are grouped into the following categories: works that devise mechanisms to detect errors in VMs and/or VI components, works that tolerate failures of the VMs by leveraging facilities in virtualized systems, and works that focus on mechanisms that enhance the resiliency of the VI. The following subsections provide more details about works in these different categories.

2.5.1. Failure Detection

Before recovery can be performed, failures must first be detected. The quality of failure detection mechanisms greatly affect the overall reliability of a system.

In general, faults can manifest as system crashes, hangs, or silent data corruption.

Failures caused by transient hardware or software faults almost always are the result of state corruption. If access to the corrupted state lead to a fatal exception, then a crash has occurred. Crashes can be detected by determining if the fatal exception handler in the kernel or VMM has executed. State corruptions can also lead to system hangs. A system is hung if it fails to make observable progress. Detecting hangs require the use of heuristics and knowledge of the application domain. Typically, hang detectors rely on the execution of periodic events such as timer events [Depo10] and keep alive signals from some component in the system [Herd07]. Hangs can also be detected by relying on knowledge of how the system should behave. For instance, a response to an I/O request needs to be generated within a short period of time after the I/O request has been made [Jo10]. When these events are not executed on time, a hang is declared.

When corruption occurs but no crashes or hangs are detected, then the corruption is silent. Detecting silent corruptions is much more challenging and requires the use of redundancy. For example, failures can be detected by using a voter component to compare outputs from replicated VMs as described in [Reis06]. Self-check mechanisms such as checksums and assertions can be added into the code to detect for corruption [Gu03].

An approach proposed by Tan et. al., called TinyChecker [Tan12], makes use of domain knowledge and hardware memory protection mechanisms to detect memory corruption in the VMM and prevents wild writes by the VMM into a VM's memory space and CPU registers. TinyChecker relies on nested virtualization [Ben-10] to run a small checker VMM beneath the main VMM. Critical data structures used for managing the system in the main VMM and most of a VM's memory are write protected by the CPU's MMU unit. Writes to these protected memory area traps to the checker VMM. Based on the context of the write,

TinyChecker determines what memory areas can be safely modified. For potentially unsafe modifications, TinyChecker uses on-demand checkpointing to save a copy of the memory location before allowing the update. When VMM failure occurs, the checkpoints can be used to obtain a valid memory state. In addition, the VM's CPU registers and state in the VM's control structure [Uhli05] are checked before every VM entry to ensure there are no illegal modifications by the VMM.

There are limitations to what corruption TinyChecker can check for. In particular, TinyChecker has no way to ensure that updates to valid memory locations in the VMM (VMM data structures or VM's CPU state) are correct. Incorrect values even in valid memory locations can result in VMM failure. In addition, failures that are not caused by memory corruption such as faults in CPU registers cannot be detected by TinyChecker. As with any failure detectors, faults can affect the checker VMM itself, causing incorrect detection of errors in the system.

2.5.2. Enhancing the Resiliency of Virtual Machines

Since faults can occur anywhere in the system, including when VMs running applications (AppVMs) are executing, one way to improve the reliability of applications running in a virtualized system is to directly tolerate failures of the AppVMs. Numerous techniques to achieve this can be found in the literature. Many of these techniques are from research done for reliable clusters and non-virtualized systems. These techniques aim to increase the resiliency of the AppVMs through replicating the AppVMs on separate physical hosts or checkpointing the VMs. As the VI contains components that are VMs (PrivVM and DVM), techniques in these works can potentially be leveraged to help improve the reliability

of the VI.

2.5.2.1. VM Replication

System virtualization vendors have products that can tolerate hardware failures by restarting VMs on a different physical host if the current host fails or on the same host if only the VM crashes [VMwa10]. However, this technique can cause VMs to lose states that have not been saved to permanent storage before the failure. A software product called everRun [Tech14], can replicate a VM across two separate physical hosts and allow failure of one of the physical hosts to be tolerated. everRun can also tolerate component failures by redirecting I/O requests from the host with failed component to working components on the other physical host. In both cases, these techniques require redundant hardware to seamlessly tolerate failures.

Academic research on the use of VM replication to increase reliability have existed for decades. Bressoud et al. [Bres96], proposed mechanisms that protects systems from hardware failure by running the system as replicated VMs. The primary and backup VMs execute in lock-step on separate physical hosts. Similar to work by Bressoud et al., the Remus [Cull08] project allows a VM to survive physical host failure by using rapid checkpointing where the primary VM sends to the backup VM residing on a separate physical host snapshots of its state. Unlike Bressoud et al., Remus does not require lock-step execution between the primary and backup. The VM running in the primary host executes slightly ahead of the VM in the backup host and sends its state to the backup host asynchronously. Externally visible outputs (network and disk) are buffered until the checkpoint state has been committed in the backup host. Tamura et al. [Tamu08] extends Remus by eliminating the need to buffer I/O

and instead forces synchronous state update in the backup when an I/O event occurs, such as reading/writing to a disk. Cui et al. [Cui09] proposed mechanisms to protect VMs in the KVM [Kivi07] VMM from failures by employing a similar replication technique as in Remus but uses a network attached storage (NAS) for disks to avoid replicating disk requests.

While the works from above require the use of two physical hosts as they aim to tolerate hardware failure, VM replication can be done on one physical host to protect against transient hardware faults. Reiser et al. [Reis06] proposed an infrastructure to allow network-based applications to run redundantly in two separate VMs on a single physical machine. A separate voter VM is used to vote on the network outputs from the replicas to detect and tolerate application or VM failure. Reiser et al. extended this work in [Reis07b] to allow redundant execution of the network-based services to go across physical hosts. A technique to perform lock-step VM replication on the same physical host by leveraging the VI to synchronize events is done in [Jeff12]. Unfortunately, lock-step execution cannot be used to tolerate software faults. Instead techniques such as N-version programming [Aviz85] must be employed. For instance, each VM replica must run a different OS. However, to fully tolerate VM failures using a single physical host requires that the VI also be resilient to failures by using mechanisms discussed in this dissertation.

2.5.2.2. VM Checkpointing

Besides using replication to tolerate VM failures, checkpointing of the VMs can also be used for fault tolerance. Primary-backup VM replication techniques discussed in the previous subsection can be considered a subset of the checkpoint/restart mechanism. With checkpoint/restart, the VM state is periodically stored in persistent storage and is used to

restart a VM when failure occurs. Most virtualized systems already have built-in capabilities to checkpoint and restart VMs. A checkpoint consists of both VM memory and CPU states as well as VM persistent storage such as disks. Depending on the size of VM memory or storage, checkpointing a VM can be time consuming. Works that focus on VM checkpoint/restart in virtualized systems try to decrease the time needed to perform a checkpoint and thus decrease the time the VM is suspended and is unavailable.

Works by Vallee et al. [Vall06] and Goiri et al. [Goir10] propose the use of stackable file systems such as *aufs* [Okaj] to reduce the amount of VM storage that needs to be saved per checkpoint. Wang et al. [Wang10] proposes mechanisms for rapid checkpointing of VMs and keeping the checkpoints in memory for fast recovery. This work uses mechanisms similar to the ones propose by Sun et al. in [Sun10] to decrease VM checkpointing time by relying on Copy-on-Write to quickly take a snapshot of VM memory. Wang et al. use dirty page prediction to reduce performance penalty when using the CoW method of snapshotting VM memory.

Taking checkpoints of many VMs comprising a virtual networked environment is done by Kangarlou et al. [Kang09]. In this work, the focus is to reduce the downtime of the networked environment by leveraging live VM migration techniques.

The obvious drawback of using checkpoint/restart is that the most up-to-date VM state at the point of failure is not saved and restored across a failure. Hence, some amount of work in the VMs will be lost. In addition, performing periodic checkpoints of VM state during normal operations can hurt overall system performance.

2.5.3. Enhancing the Resiliency of the Virtualization Infrastructure

The idea of enhancing the resiliency of the VI itself as a way to increase the chance of VM survival is not new. Works that have been discussed in the previous subsections have investigated ways to make the VI more reliable by reducing the probability a fault can lead to system failure through simplifying and isolating components that are critical to the normal operations of the system. Simplification reduces the chance of failures while isolation can be used to contain the failure and prevent faults from propagating and affecting other components of the virtualized system.

Despite attempts at reducing the chance of failures in a virtualized system, failures can still occur. In particular, failures can occur in any components of the VI (PrivVM/DVM/VMM). Mechanisms must exist to tolerate these failures by detecting and recovering the failed component.

Of the three VI components, only the DVM has received attention in terms of research into recovering from failure. The main cause of DVM failures is errors in device drivers, hence, techniques to tolerate device driver failures in both virtualized and non-virtualized systems will be discussed in this subsection. While mechanisms to recover the PrivVM and VMM have not been proposed, many ideas and techniques used for recovering from OS failures and rejuvenating the VMM and PrivVM can be leveraged to implement such mechanisms.

The following subsections discuss previous works that propose mechanisms to tolerate device driver failures and failure of an OS kernel.

2.5.3.1. Tolerating Device Driver Failures

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 [Fras04, LeVa04, Jo10], user-level processes [Herd07, Herd09, Gana08], or light-weight domains [Swif05, Swif04]. There are also software techniques based on language extensions, static analysis, and runtime checks [Zhou06, Erli06]. 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 [Herd07, Herd09, Gana08]. A key goal is to isolate the drivers from critical OS kernel components. In [Herd07], 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 [Herd09]. 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. [Gana08] 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. [Swif05, Swif04] 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 [LeVa04, Fras04]. LeVasseur et al. [LeVa04] discuss but do not implement or evaluate mechanisms for recovering from device driver failure. Fraser et al. [Fras04] 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 [Warf08]. 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 [Jo10]. Jo et al.[Jo10] introduced a scheme named 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. 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 [Jo10], 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. In [Jo10], 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 [Linu09]. However, there is no evaluation of the recovery effectiveness of these other mechanisms.

2.5.3.2. Tolerating OS Kernel Failures

This subsection discusses a prior work that presented mechanisms to recover the Linux kernel from failure by rebooting kernel transparently to processes in the system. The ideas from this work can be used to tolerate failure of the kernel in the PrivVM and/or DVM and serve as the basis for developing mechanisms to tolerate failure of the VMM.

The Otherworld project by Depoutovitch and Stumm [Depo10] allow a Linux kernel to be recovered while preserving the state of the running processes. The recovery mechanism involves the use of KDump [Goya05] to load a kernel used for crash handling into a reserved area of memory and booting that new kernel. The new kernel only boots within the reserved memory space and therefore preserves the memory content of the failed system. Once booted, a selected number of processes are resurrected by recreating the process descriptors and copying the process memory from the old memory region. Data integrity checking must be used to ensure recovery does not end up using corrupted data.

Besides resurrecting the process descriptors, Otherworld must individually resurrect kernel resources that are used by the processes such opened files, signal handlers, and shared

memory IPC. Not all IPCs can yet be resurrected such as the network stack or pipes. In addition, Otherworld requires applications that require the use of such kernel resources to have custom crash procedures registered with the kernel so that they can be invoked to perform application specific recovery tasks such as re-opening sockets or restart the applications after saving necessary state. After process resurrection, the old memory region is released and is integrated into the running system. Due to the complexity of resurrecting individual kernel resources for a process, Otherworld requires an application to register a crash handler to be able to fully recover from a kernel failure.

2.6. Software-implemented Fault Injection

Developing reliable systems require the development of techniques and tools to evaluate and validate the proposed resiliency mechanisms. The simplest and most flexible technique for such task is software-implemented fault injection (SWIFI). SWIFI, without virtualization, has been used for decades to evaluate systems [Sega88, Hsue97, Carr99, Stot00]. This has included injection into OS kernels [Carr98, Gu03, Gu04]. As discussed further in the rest of this section, more recently, fault injection in conjunction with system virtualization has been used to evaluate non-virtualized systems [Buch01, Sieh02, Buch03, Poty07, Hsu10, Subk07, Swif05, Oyam11] as well as virtualized systems [Jeff12, Pham11]. However, there has been only very limited prior work focused on analyzing and evaluating the *interactions* between virtualization and fault injection, and of the benefits and challenges of using virtualization for fault injection. To the best of our knowledge, there is no prior published work analyzing the tradeoffs and comparing the results from injections with or without virtualization, or of injections from inside or outside the system under test. Furthermore, prior publications have

not discussed the challenges of implementing fault injection in virtualized systems, such as accurately emulating errors in page tables or measuring crash latency.

Starting in 2001, a sequence of related papers proposed and expanded on the use of VMs to facilitate low-cost fault injection for evaluating non-virtualized systems [Buch01, Sieh02, Buch03, Poty07]. These papers discussed the design, implementation, and use of a hardware emulator to run a VM into which faults targeting devices, CPU registers, and memory can be injected. Benefits of using virtualization for fault injection, such as the ease of automation, and reduced intrusion and interference, are also presented.

The benefits of using system virtualization to automate fault injection campaigns have been recognized by previous works. Subkraut et al. [Subk07] discuss the benefits of using fast snapshot and rollback mechanisms in VMs to decrease the time to perform a single fault injection run when evaluating the robustness of software APIs. Swift et al. [Swif05] use VMs to facilitate fault injection automation and to speedup fault injection experiments for evaluating OS reliability.

Use of fault injection to evaluate *virtualized systems* has emerged only over the last few years. Jeffery and Figueiredo [Jeff12] used a VMM-level injector to inject faults into registers of a VM to evaluate their VM replication mechanism. Pham et al. [Pham11] developed a fault injection framework that uses a kernel-level injector for evaluating and comparing the reliability of different virtualization infrastructures. With this framework, they are able to inject faults into the KVM hypervisor, which is implemented as a kernel module for Linux. Oyama and Hoshi [Oyam11] combine fault injection and system virtualization to non-intrusively evaluate properties of security systems. Faults in specific processes and data structures of the target VM are injected from another VM. Only the work in this dissertation

has included injection into a separate hypervisor (VMM), as used with the Xen VI.

Chapter Three

Applying Microreboot to System Software

Microreboot enables fast restoration of a valid state of a failed process or system by rebooting the specific failed component instead of the entire system [Cand04]. *Low-level system software* (LLSS) manages and controls hardware resources and mediates access to hardware resources by higher software layers. The virtualization infrastructure (VI) is a type of LLSS. While in previous work microreboot has been used mostly for user-level processes, the technique is also applicable to system software [Le12]. However, as explained below, due to key characteristics of system software, using microreboot to recover from failures in such software is particularly challenging. In the rest of this chapter, unique challenges of microrebooting LLSS are described and general approaches to addressing these challenges are presented. Materials from this chapter are used to directly guide the development of mechanisms to microreboot individual components of the Xen VI, which are described in detail in Chapters 4-8 of this dissertation.

In general, it is difficult to apply microreboot to software with tightly-coupled components and non-modular organization. Fortunately, unlike LLSS such as conventional monolithic OS kernels, the Xen VI is well-suited for microreboot as it is organized in a modular fashion consisting of three separate components that communicate through narrow and well defined interfaces. However, even with a modular design, the Xen VI, and in general, LLSS has properties that makes it less amenable than application software to the microreboot technique. One such property is that LLSS interacts directly with hardware

resources. The hardware cannot be modified and the interfaces to the hardware may not provide well-defined semantics in the face of an LLSS component reboot. Hardware resources may be shared by multiple LLSS components, potentially coupling these components to each other, making it difficult to microreboot a single component.

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

In the following paragraphs, five specific challenges with applying microreboot to LLSS are described along with general approaches to addressing these challenges.

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

An LLSS component failure can lead to corruption of hardware state by providing the hardware with faulty inputs or failing to respond properly to inputs from the hardware. The

only way to restore the hardware component (e.g., a device controller, an interrupt controller, or a bus) to a sane state may be to reset it. The hardware component that needs to be reset may be coupled to other hardware components and/or shared by multiple LLSS components. Hence, the reset of the hardware component may initiate a ‘‘domino effect’’ that forces the reset and reboot of multiple hardware and LLSS components. Furthermore, the reset of some hardware components may be inherently slow. (see Section 4).

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

The hardware cannot be changed in order to accommodate LLSS microreboot. Hence, microreboot for LLSS components may require implementing work-arounds to deal with specific problematic properties of specific hardware components. This may involve identifying ways to perform partial resets of the hardware, even if that may not always be sufficient (see Chapter 5). It may lead to special operations performed during microreboot which, while not an actual hardware reset, can eliminate specific known ‘‘problem states’’ in the hardware. An example of this is to acknowledge any pending interrupts, waiting for a response from the rebooted LLSS component, that would otherwise block future interrupts. Finally, in some cases, the considerations above may force a recovery technique where system functionality is restored before the microreboot of the failed LLSS component and/or the reset of some particular hardware component. This can be done by fail-over mechanisms that use redundant software and hardware resources while the failed components are restored.

Workload transparency: Typically, many different applications and user-level

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

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

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

The only viable approach to dealing with the above challenge is to have an equal privilege LLSS component, a failure handler, that is invoked when an error is detected. Since the failure handler is of equal privilege, the handler itself may be corrupted by the failed component and/or recovery may rely on the reuse of potentially corrupted state. Redundant

data structures can be used to minimize the probability of corrupted state reuse. However, this possibility cannot be eliminated.

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

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

Imperfect recovery: None of the approaches described above for dealing with the challenges of employing microreboot for LLSS guarantee that recovery will be successful. LLSS with high privilege may corrupt all software layers above it and may corrupt the state of the hardware to the point of requiring power cycling the entire system. The probability of such undesirable behavior can be reduced by performing extra work during normal operation to provide self-checking capabilities and log redundant data that can be used during recovery.

The microreboot process itself can be made safer by performing extra checks on state that is reused as well as on hardware components. The challenge is to determine the costs and benefits of such overhead and know when the mechanism is sufficient to meet system requirements.

An approach to meeting the above challenge is to rely on incremental refinement of the recovery mechanism based on experimental evaluation [Ng99]. Specifically, the first step is to implement a minimal recovery mechanism, that can deal with ‘‘well-behaved’’ fail-stop failures. Fault injection campaigns are then used to evaluate the recovery success rate under realistic conditions. The results are used to determine the most important cause of recovery failures. This cause is a deficiency in the recovery mechanism that is then eliminated or mitigated. The process is then repeated until the desired success rate is achieved.

Chapter Four

Resilience to Device Driver Failures in Driver VMs

As described in Subsection 2.1, with the isolated driver VM architecture (IDVM), commodity device drivers run in driver VMs (DVMs) separate from VMs running applications (AppVMs). Failure of the DVM is typically caused by errors in device drivers which have been found to be a major cause of system failures [Chou01, Swif05]. 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 [Fras04]. 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 chapter 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 [Fras04, LeVa04], user-level process [Herd07, Herd09, Gana08], or light-weight domain [Swif05, Swif04]. Unlike the work presented in this chapter, 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 remainder of this chapter is organized in the following way. A discussion on the current state of device virtualization and its impact on the IDVM architecture is presented in the next section. The basic architectures and design tradeoffs of different mechanisms for providing resiliency to device driver failures are described in Section 4.2, which also presents a taxonomy of the mechanisms. Section 4.3 identifies key challenges in implementing the mechanisms and ensuring their reliable and fast operation. Section 4.4 describes our experimental setup and Section 4.5 presents our results.

4.1. Impact of Device Virtualization

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 [Inte11]. 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 [Zhai08].

4.2. 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. As discussed

in Section 2.3, we assume that device driver failures are caused by transient hardware and/or software faults. 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.

4.2.1. Failure Detection

Ideally, any incorrect behavior (outputs) from the driver/DVM should be detected. For example, this could be done using VM replication [Jeff12]. 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 4.5, 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 [Jo10] (see Section 4.4.5). 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) (see Section 4.4.5).

4.2.2. 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.

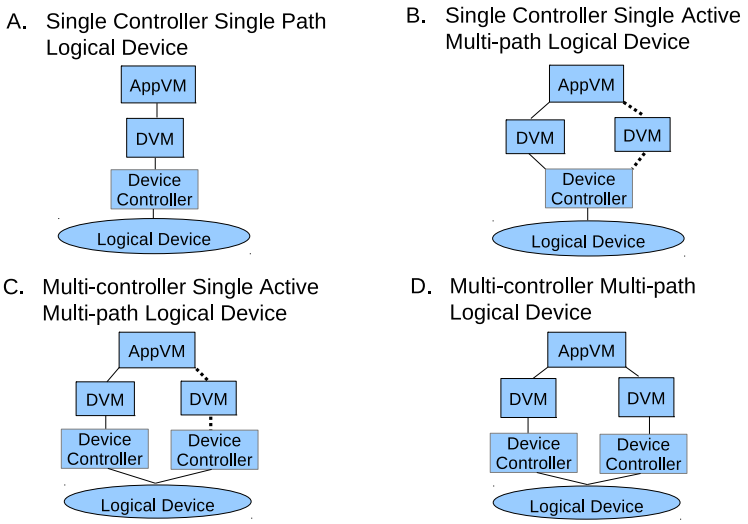


Figure 4.1: System configurations for AppVM device access.

To provide an AppVM access to a logical device, there must be a path between them (Fig. 4.1). 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. 4.1.A shows a configuration where there is only a single path to the logical device. In Fig. 4.1.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. 4.1.C is similar to the configuration in Fig. 4.1.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. 4.1.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.

4.2.3. 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. 4.1.D).
- **C**reate New (D**C**): A new DVM is created and booted upon failure (Fig. 4.1.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. 4.1.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. 4.1.B).
- **P**rimed Spare (D**P**): Similar to DA, however, the device driver undergoes some amount of initialization as part of system setup (see Chapter 5). 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. 4.1.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. 4.1.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 (see Chapter 5). 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 (see Chapter 5). As discussed in Subsection 4.2.5, 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 [Gupt08].

4.2.4. 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. 4.1.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 (see Chapter 5). 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 4.2.5 and 4.3.4).

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 [Jo10] (see Chapter 5).

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.

4.2.5. 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 (see Chapter 5).
- **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 [Linu09] for network access. Upon DVM failure, the AppVM is notified of the loss of one of its redundant connection to the logical device (Fig. 4.1.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 (see Chapter 5). As discussed in Subsection 4.2.4, 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 4.3.4.

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.

4.3. 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.

4.3.1. 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 [Ben-06], 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 (see Chapter 5). 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 4.1.A and 4.1.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.

4.3.2. 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 4.3.1, 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) [Dong09, Nguy13], 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.

4.3.3. 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 [Linu09] 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.

4.3.4. Frontend Supporting Multiple Backends

Some OSs provide native support for managing multiple NICs to provide fault tolerance (e.g., Bonding [Linu09] in Linux). With such AppVM OSs, configurations with multiple NICs (Figure 4.1.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 4.1.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.

4.4. 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.

4.4.1. Recovery Schemes Evaluated

The taxonomy presented in Section 4.2, 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 4.1.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 Chapter 5, and are not discussed further in this chapter. FT requires changes to the VI and does not have a significant advantage over simpler schemes (see Subsection 4.2.5). As discussed in Subsection 4.2.3 and based on our work discussed in Chapter 5, DP is undesirable since it involves significant implementation complexity and operational problems.

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	--	--

Table 4.1. Practical, implementable, deployable recovery mechanisms. Names are given to recovery mechanisms we have implemented and evaluated.

Eight techniques that can be practically deployed are listed in Table 4.1. Five are selected to implement and evaluate in this work. In Table 4.1, 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. [Jo10] 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 [Fras04, LeVa04, Herd09], we include it in our evaluation as it serves as the foundation for other schemes.

4.4.2. 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 (see Chapter 9).

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 4.2 shows the types of faults injected. As discussed in [Ng01], some of these faults simulate typical programming errors. For the first seven fault types (*software faults*), we used the fault injection tool in [Swif05] to generate a list of injections to perform by our Gigan fault injector. The random code faults further stress the device drivers. Register faults

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

Table 4.2. Types of faults injected

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 [Meno05] to identify the functions in those drivers that are most frequently executed [Gu03]. Only instructions in those functions are 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 4.4.3). Non-manifested means that no errors are observed.

4.4.3. Benchmarks

Most of our evaluation is done using two micro-benchmarks, designed to facilitate accurate measurement of recovery effectiveness and latency: *netbench* and *blkbench* that exercise the device drivers for the network and block devices, respectively. As discussed in Subsection 4.5.6, 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.

Blkbench 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.

4.4.4. 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 4.4.2) faults, detection latency is measured between events (2) and (A). For register faults, it is measured between events (1) and (A) [Gu04]. 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 4.4.3). For block devices, it is an increase in the time of a single block read/write operation with *blkbench*. These measured interruption latencies roughly, but not precisely, correspond to the time between events (3) and (B).

4.4.5. 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. 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 are similar to the mechanism discussed in [Jo10].

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.

4.5. Experimental Results

The results of the experimental evaluation of the DVM recovery schemes described in Subsection 4.4.1 are presented and discussed in this section. The recovery success rates and recovery latencies are discussed in Subsections 4.5.1 and 4.5.2, respectively. The impact of the number of AppVMs on the recovery latency is analyzed in Subsection 4.5.3. Subsection 4.5.4 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 4.5.5 and Subsection 4.5.6 demonstrates how well these recovery mechanisms work on a real-world application (Apache web server).

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

Table 4.3. 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 4.2, random code (code), and register (reg).

4.5.1. 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 4.3 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.

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 4.4.3). However, a corruption of the block driver can cause the file system or a single file to become corrupted, leading to a wrong result 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 (see Chapter 9). 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.

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

Table 4.4. 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 *blkbench* 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.

4.5.2. Recovery Latency

To measure the recovery latency (see Subsection 4.4.4) of different mechanisms, we use a configuration that consists of a single AppVM. In these experiments, a DVM crash is forced.

Table 4.4 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.

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
SpareNet / SpareBlk	1	3795 ± 21	4132 ± 4
	2	3812 ± 16	4231 ± 16
	3	3828 ± 13	4375 ± 19
AttSpareNet/ AttSpareBlk	1	3623 ± 44	3705 ± 4
	2	3607 ± 31	3817 ± 6
	3	3633 ± 33	3960 ± 10
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
Bonding	1	0.13 ± 0.04	--
	2	0.15 ± 0.02	--
	3	0.13 ± 0.02	--

Table 4.5. Impact of No. AppVMs on recovery latency.

The number of devices controlled by a DVM can affect its recovery latency. The bottom of Table 4.4 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 4.4 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.

4.5.3. 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 4.5.2 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. Table 4.5 shows results from those campaigns.

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.

4.5.4. Service Interruption Latency

As explained in Subsection 4.4.4, 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.

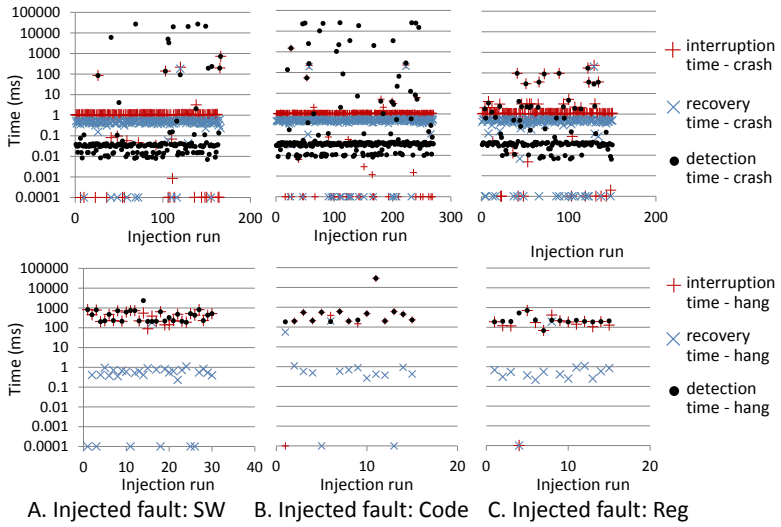


Figure 4.2: Service interruption with *netbench* using MultiNetIf recovery. SW and register faults. Top graphs: faults cause crashes. Bottom graphs: faults cause hangs.

Figures 4.2 and 4.3 show the detection, recovery, and interruption latencies per run of our MultiNetIf and RAID fault injection campaigns. 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

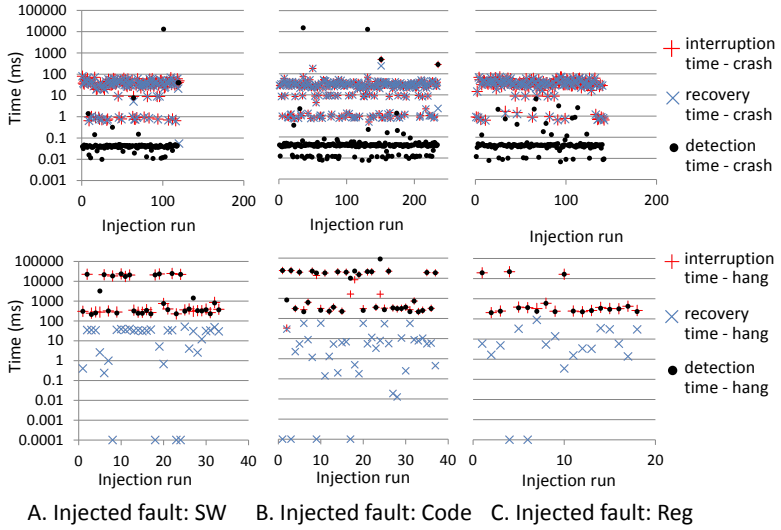


Figure 4.3: Service interruption with *blkbench* using RAID recovery. SW and register faults. Top graphs: faults cause crashes. Bottom graphs: faults cause hangs.

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.

4.5.5. 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 configuration used in this work, 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 *blkbench*, 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 4.3.2).

4.5.6. 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 *blkbench* program. The MultiNetIf and RAID schemes are used for recovery from DVM failure. Software faults (Subsection 4.4.2) 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).

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

Table 4.6 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.

4.6. Summary

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 [Jeff12], 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.

Chapter Five

Fast Device Driver Recovery

The discussion and evaluation of different mechanisms to tolerate device driver failures in an IDVM architecture have been presented in Chapter 4. As explained in that Chapter, due to the long device reset time, to achieve low recovery latency, redundant devices had to be used. However, redundant devices are not always available in every system. Some of these systems may host VMs running network services that have stringent quality of service (QoS) requirements and where lengthy service interruption is unacceptable. The focus of this chapter is on achieving fast recovery of device drivers residing in DVMs without relying on device redundancy. Specifically, the device of interest in this chapter are NICs. NICs are an important class of devices in virtualized system as the network is often the primary way to communicate with the outside world including being used for accessing storage devices.

The fast recovery scheme discussed in this chapter employs two DVMs: the primary and a spare. When driver failure is detected, recovery is performed by quickly replacing the primary with the spare, completely transparent to the AppVMs [Le09]. This scheme trades off a small amount of correctness of the recovered device driver for a large reduction in recovery latency. While previous schemes reported recovery times on the order of a second [Swif04] or hundreds of milliseconds [Fras04], in the great majority of cases our scheme recovers in less than 10ms. Such fast recovery allows the system to meet stringent QoS requirements for many network services. This is accomplished using unmodified application VMs, minimal modifications to the Xen VMM and driver VM kernel, and user-

level scripts in the privileged VM. The scheme incurs no performance overhead and insignificant memory overhead during normal operation.

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

We start with the implementation details and evaluation of the simple but slow reboot recovery scheme presented in the previous chapter (RebootNet). The SpareNet mechanism (also presented in the previous Chapter) is then briefly described as a way to improve the RebootNet scheme. Analysis of the SpareNet scheme reveals parts of the recovery process that can be further optimized, leading to the development of the fast recovery scheme. Using the taxonomy from the previous chapter, this new scheme would be classified as DP-CS-FT.

The experimental setup used to improve and evaluate the recovery mechanisms discussed in this chapter are presented in Section 5.1. Section 5.2 presents the development and evaluation of the fast recovery scheme and Section 5.3 discusses and evaluates the application of the fast recovery scheme to different types of NIC drivers.

5.1. Experimental Setup

The results in this chapter were obtained using several systems based on the Intel Core-2 processors. For most experiments, we used Intel Pro 100 NICs interconnected by a 100Mb switched ethernet network. Xen 3.3.0 was used as the VMM. Linux kernel version 2.6.18.8 was used for the PrivVM, AppVM, and DVMs. We refer to a DVM providing access to only

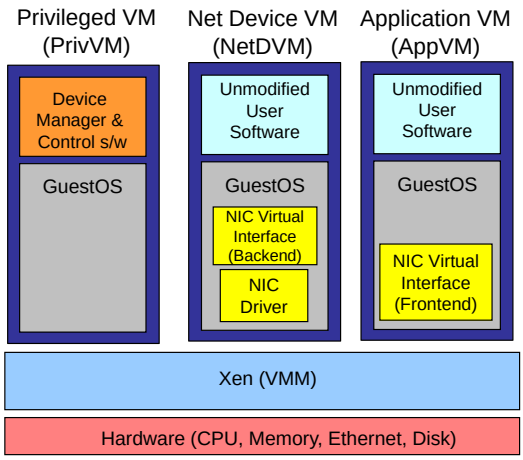


Figure 5.1: Basic target system configuration showing a PrivVM, NetDVM, and AppVM.

a network device as a NetDVM.

We use our *Gigan* fault injector (see Chapter 9) to inject faults into the NIC driver. Some of these injections simulate programming errors, such as invalid pointer access and incorrect loop termination conditions [Swif05]. For *code* injection, faults were injected into every byte of the most frequently used functions[Gu03] (six out of 75) in the E100 device driver, identified by the Xenoprof sampling profiler [Meno05]. These six functions account for roughly 29% of the E100 code. The rest of the functions are used for device/driver initialization. Injection was triggered by setting hardware breakpoints on the virtual addresses targeted for injection. If the breakpoint fired, a random bit of the targeted byte was flipped. *Register* injection was similar, except that when the breakpoint fired a random bit of a random general-purpose register was flipped. In the reported results, an injection is counted only if it was actually activated — the breakpoint fired and caused a bit flip.

The outcome of each injection experiment is the same as what was described in Section 4.4.2. Mechanisms to detect device driver failure have been described in Section 4.4.5. To

stress the driver code in the DVM, the *netbench* program (see Section 4.4.3) is used. Similar to what was described in the previous chapter, the network interruption latency, which is a combination of failure detection and recovery latency, is the maximum time between reception of successive ping acknowledgments on the *sender host*.

The basic system configuration for the experiments in this section is shown in Figure 5.1. It includes the PrivVM, a NetDVM, and a single AppVM running the netbench program. The application process (netbench) running in the AppVM normally executes for 7 seconds during which it receives packets before reporting normal termination. If there is a network interruption, packets are not received so the application may execute for a longer period before accumulating 7s during which it receives packets. One injection run consists of booting the target system, randomly picking a time between 0 and 5s for when to set a breakpoint at the injection address, and running the application. If the application does not record normal termination within 14s, a timeout is triggered and that is interpreted to be a ‘‘silent application failure.’’

5.2. Optimizing Network Device Driver Recovery

This section starts be describing basic NIC driver recovery techniques that can be used in a virtualized system where the NIC device driver is isolated in a NetDVM. The first recovery scheme simply detects and reboots a failed NetDVM. While this works and enables the AppVM to continue, it is also slow. The next technique speeds up recovery by deploying a spare NetDVM to take over once the primary NetDVM fails. Analysis of this mechanism reveals parts of the recovery process that can be further optimized, leading to the final recovery technique that can reduce network interruption time to less than 10ms.

5.2.1. Recovery Using Reboot

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

It might be expected that the first action upon detecting a crash/hang of a NetDVM would be to destroy it. However, despite the crash/hang of the NetDVM, as explained in Subsection 4.3.1, the NIC may still be writing into the NetDVM's memory. If the failed NetDVM is destroyed, its memory might be reused by the VMM (e.g., for the new NetDVM), leading to memory corruption. Hence, the failed NetDVM is, initially, only paused, keeping all its memory, instead of being immediately destroyed.

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

Recovery Mechanism	injections	NetDVM crash	NetDVM hang	silent app failure	non manifested	successful recovery
RebootNet	568 (code)	54.0%	3.2%	2.8%	40.0%	100.0%
SpareNet	584 (code)	53.4%	3.4%	2.9%	40.3%	100.0%
SpareNet (fast)	2856 (code)	55.2%	3.1%	2.6%	39.1%	99.9%
SpareNet (fast)	1300 (register)	28.6%	3.0%	0.5%	67.8%	99.9%

Table 5.1. Device driver injection results with recovery.

A new frontend-backend connection must be established between the new NetDVM and the AppVM to complete recovery. This should be transparent to the application on the AppVM so that there will be no need to modify applications. Fortunately, this ability is already implemented in Xen for VM checkpointing/restoring. During checkpointing of a VM, all frontend drivers suspend activity and prepare for system-level suspension. During VM restoration, the frontend drivers resume activity and transparently reconnect with the backend drivers. Unfortunately, checkpointing and restoring a VM is slow. To avoid full checkpoint/restore, the suspend/resume code in the AppVM kernel was modified to allow individual frontend drivers to be selectively suspended/resumed. Using this mechanism, the recovery agent directs the AppVM to transparently reconnect only the network frontend driver to the backend driver on the new NetDVM.

This recovery technique was evaluated by performing fault injection into the code of the E100 device driver as described in section 5.1. Table 5.1 shows the results. Recoveries were attempted on 325 detected hangs/crashes out of 568 injections and successfully restored the system 100% of the time. The mean time of recovery was 7.47s and ranged from 7.29s to 9.36s. The slowest component of this recovery mechanism was the time required to boot a new NetDVM (approximately 3s). Due to some differences in the kernel of NetDVM, this

boot time is faster than the DVM boot time described in Chapter 4 by about 4s.

5.2.2. Recovery using Spare NetDVM

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

Results of fault injection into the code of the E100 device driver (Table 5.1) show that recoveries were attempted on 332 detected hangs/crashes out of 584 injections and successfully restored the system 100% of the time. The mean time of recovery was 4.52s and ranged from 4.33s to 7.69s. Most of this time was spent: (I) invoking the recovery agent inside the PrivVM, (II) transferring control of the NIC device to the spare NetDVM, and (III) establishing a new frontend-backend connection. As with the NetDVM reboot scheme, the recovery agent is invoked by a message sent from the VMM. Due to the asynchronous nature of this communication, it takes approximately 0.4s for the recovery agent to respond to a detected NetDVM failure. The following paragraphs detail the impact of operations (II) and (III) above on the recovery time.

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

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

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

The memory overhead of keeping a spare NetDVM is 64MB, which is used to run the Linux kernel and applications to setup and run the network backend. However, since the memory content of the spare NetDVM is nearly identical to the primary NetDVM, most of this overhead could be eliminated by using content-based memory sharing in the VMM [Gupt08]. Since the spare NetDVM is paused, it does not consume any CPU cycles.

5.2.3. Minimizing NetDVM Failover Latency

The second paragraph of the previous subsection lists the three operations responsible for most of the latency of recovery with the warm-spare NetDVM. Most of the latency of (I) can be eliminated by moving the main functionality of the recovery agent into the VMM. Most of the latency of (II) and (III) can be eliminated by allowing the spare NetDVM to discover and set up the NIC device and form the network frontend-backend connection with the AppVM during system initialization, prior to recovery. With these optimizations, recovery entails pausing the failed primary NetDVM, unpausing the spare NetDVM, and performing minor NIC device and network backend re-initialization.

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

The system configuration for this mechanism is similar to the previous subsection. However, system initialization differs in two ways: (1) the spare NetDVM pauses itself after

setting up the frontend-backend connection with the AppVM and (2) when the primary NetDVM boots, it forms a frontend-backend connection with the AppVM by mapping in the shared rings used in the connection between the AppVM and the spare NetDVM. VMM code (M1) provides the functionality of the recovery agent. When an error is detected, this code pauses the failed NetDVM, initiates unmapping of some pages from the failed NetDVM (see below), and unpauses the spare NetDVM.

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

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

During recovery, the spare NetDVM re-initializes the NIC device and network backend driver (M6). This involves re-initializing the DMA regions. Furthermore, since the failed NetDVM may corrupt the NIC state, the re-initialization involves a reset of the NIC (see Subsection 5.3). The private shared ring indices in the network backend driver are updated

with current values from the shared rings (transmit and receive), so that the spare NetDVM can continue consuming requests and putting responses at the locations where the failed NetDVM left off. A new function is added in the network backend device driver in Linux (M7) to allow for this shared ring re-attachment to occur.

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

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

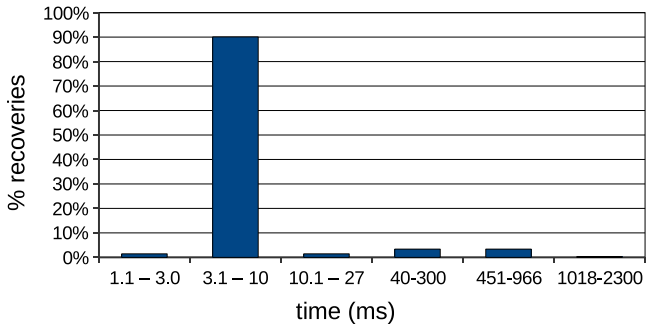


Figure 5.2: Network interruption latency using fast failover spare NetDVM

Evaluation of this approach consisted primarily of code and register injection into the

E100 device driver. As shown in Table 5.1, with code (register) injection, out of 2856 (1300) injections, 97.4% (99.5%) were not manifested or triggered recovery. When it was triggered, recovery was successful in 99.9% of the cases. In the two cases that failed, the AppVM no longer received packets after recovery. Fig. 5.2 shows the distribution of network interruption times (recovery times) with fault injection in the driver code. In 91% of recoveries, the network interruption time was less than 10ms. All of these cases were crashes that were immediately detected. Most interruption durations over 10ms were due to detection latency of the hang detectors, ranging between 200ms for the transmit ring hang detector to 1s for the PTBR hang detector. There were also cases where crashes took as much as 500ms to detect since the NetDVM kernel was stuck recursively calling fault handlers before it finally panicked.

5.2.4. Silent Application Failures

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

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

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

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

5.3. Recovering from Failures of other NIC Drivers

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

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

that must be performed on a reset [Inte09].

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

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

As part of normal operation, the E1000 NIC caches receive descriptors [Inte09]. This poses a problem during recovery since these cached descriptors contain memory locations belonging to the failed primary NetDVM. Incoming packets using these cached descriptors will be lost, thus increasing network interruption latency. To avoid this, these cached descriptors must be quickly purged immediately after recovery. Purging is performed by having the recovery module in the spare NetDVM send out ping requests immediately after recovery so that the returning ping replies flush out the invalid cached descriptors.

With the modifications above, out of 2289 injected faults, 37% resulted in detectable NetDVM failures. Avoiding the full NIC reset caused more unsuccessful recoveries: 18 out of 837 attempted recoveries, compared to 5 out of 2680 attempted recoveries for the 100Mb NICs. In 91% of detectable failures, the network interruption latency is less than 33ms. Most of the 33ms interruption latency is due to a 10ms sleep (actually measured between 10-19ms)

that exists as part of the E1000 descriptor ring re-initialization code. When this sleep was removed, the network interruption latency was reduced to less than 10ms for 86% of recoveries. However, the number of unsuccessful recoveries increased to 54 out of 745 attempted recoveries.

5.4. Summary

This chapter shows that fast recovery (<10ms) from NIC driver failures is possible and does not require access to redundant devices. This is done by exploiting the capabilities of the IDVM architecture. However, such fast recovery comes at the cost of requiring the modification of the VMM and for some NICs, NIC device drivers.

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

Chapter Six

Resilience to Privileged VM Failures

The failure of the Privileged VM (PrivVM) prevents the ability to manage and control the virtualized system. As a consequence, no VMs can be created, destroyed, migrated, reconfigured, etc. Typically, the AppVMs and other VI components do not need to access any services provided by the PrivVM during normal system operations. Hence, applications executing in existing AppVMs may continue to correctly run as the VMM and DVM are still functional. However, to regain control of the virtualized system, the PrivVM must be recovered.

This chapter presents the mechanisms to tolerate failure of the PrivVM [Le12]. Following the incremental refinement methodology in this dissertation, the development of the PrivVM recovery mechanisms is iterative using results from fault injection as a guide.

The rest of this chapter is organized in the following way. Section 6.1 discusses how to configure the PrivVM in order to make it less susceptible to failure and more amenable to recovery. Section 6.2 describes the mechanisms to detect failure of the PrivVM. The basic mechanism to recover the PrivVM is presented in Section 6.3. Section 6.4 describes the experimental setup used to improve and evaluate the PrivVM recovery mechanism. Improvements to the basic recovery scheme using results from fault injection as a guide are discussed in Section 6.5. Section 6.6 analyzes the effectiveness of the failure detection mechanisms.

6.1. PrivVM Configuration

Besides managing the virtualized system, the PrivVM also plays important roles in the resiliency mechanisms that we have developed (see Chapters 4 and 5). Hence, it is desirable to minimize the probability of PrivVM failure and minimize the complexity of recovery when the PrivVM does fail. In a typical Xen system, the PrivVM is used to access all hardware devices in the system as well as play the role of the DVM [Barh03]. However, such a configuration exposes the PrivVM to possibly buggy device drivers [Swif05] or faulty device controllers, thus increasing the probability of PrivVM failure.

Due to the considerations explained above, we use a configuration that minimizes the interactions of the PrivVM with device controllers. Specifically, as explained in Chapter 2.1, access to most devices is through a separate DVM. The PrivVM's root file system is in memory so that the PrivVM does not have access to any storage devices. The PrivVM has access to a network interface card (NIC) in order to enable remote management of the virtualized system. In addition, the PrivVM controls the PCI bus configuration space and access to the PCI configuration space from any VM is performed through the PrivVM using the PCI frontend/backend device drivers. Furthermore, the PrivVM hosts the virtual serial console device.

6.2. Failure Detection

The mechanisms for detecting crashes and hangs of the PrivVM kernel are similar to the mechanisms used for detecting crash/hang of the DVM kernel described in Subsection 4.4.5. Besides the kernel, the correct operations of the PrivVM, and ultimately the VI, depends on special user-level processes (*XenStored* and *Xend*) that are run in the PrivVM (see Subsection

2.1). To detect the failure of these processes, we deploy a user-level monitoring process, called *hostmon*, that periodically checks for the existence of these processes. When hostmon detects the disappearance of one of these processes, it invokes the VMM to crash the PrivVM and trigger full PrivVM recovery. Of course, hostmon itself can fail, leaving the PrivVM without any mechanism to detect failure of the critical user-level processes. To overcome this problem, we deploy two hostmon processes so that they can detect each other's presence. If one of the hostmon processes fail, then full PrivVM recovery is triggered.

6.3. Basic Recovery

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

Since the PrivVM has access to the PCI subsystem (PCI buses and PCI configuration space), PrivVM boot-up usually includes the probing of PCI devices. One of the main challenges of microrebooting the PrivVM is to avoid such probing from affecting other components on the system. PCI probing involves the reading of the PCI configuration, which requires writing into a control register. The control register may be in use for different purposes during normal system operations, such as holding the base memory address used for

accessing a particular device. Hence, reading the PCI configuration during PrivVM recovery may have unintended side effects. Avoiding this problem required modifying the PCI driver in the PrivVM kernel. The PCI information gathered during initial PrivVM boot-up is saved in the VMM. A modified PCI probing routine is used during reboot to retrieve the PCI information from the VMM.

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

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

During PrivVM microreboot, the frontend-backend connection with the XBA must be established before the PrivVM can obtain the XenStore and XenStored states from the XBA. Establishing this connection requires the PrivVM to have information such as the frame number of the shared page being used by the connection (Section 2.1). Hence, the VMM has been modified (new hypercalls) to allow the PrivVM to store this information in VMM memory when the PrivVM is first booted and retrieve the information when a new instance is booted during PrivVM microreboot.

During PrivVM microreboot, once the XenStore and XenStored states have been restored, connections involving the PrivVM are re-established, connecting frontends in the various VMs to the XenStored, virtual serial console, and PCI backends in the PrivVM. Re-establishment of PCI frontend-backend connections required a small extension to the PCI frontend driver in all the VMs. This extension incorporates the abilities to disconnect/connect from/to the PCI backend driver using the existing suspend/resume functionalities of the split device driver mechanism (see Subsection 5.2.1).

6.4. Experimental Setup

This section describes the system setup and fault injection campaigns used to improve and evaluate the effectiveness of the PrivVM recovery mechanism. During normal system operations, the PrivVM is mostly idle unless it must perform some management operations such as create a VM. The experiments in this section are thus designed to evaluate the effectiveness of the PrivVM recovery mechanism while the PrivVM is in one of two states: idle and active.

In all experiments, we use Xen 3.3.0 with all the VMs running XenoLinux version

2.6.18.8. Faults are injected into the CPU's general purpose and instruction pointer registers while the CPU is executing code in the PrivVM (both kernel and user-level). Faults in registers are a good choice as they can cause arbitrary corruption allowing the PrivVM recovery mechanism to be stressed.

To evaluate the recovery mechanism when the PrivVM is idle, we make use of the Linux Virtual Server system configuration described in Section 8.2. A fault is injected into the PrivVM once the entire target system has been started and the workload has been run. Similar to what is discussed in Section 8.2, to ensure the PrivVM is functional after recovery, one AppVM is selected to be rebooted immediately after recovery of the PrivVM.

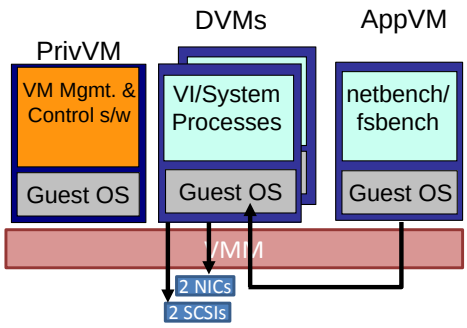


Figure 6.1: System configuration for evaluating the PrivVM recovery mechanism. The AppVM accesses I/O devices through the DVMs. The AppVM has access to two disks, hosted on separate DVMs, which the AppVM uses to form a single RAID level-1 block device. The AppVM also has access to NIC through one of the DVMs.

To evaluate the recovery mechanism when the PrivVM is active, we use the system configuration shown in Figure 6.1. One of the main tasks that the PrivVM must perform when creating an AppVM is to initiate and facilitate virtual device setup between the AppVM and one or more DVMs. Hence, in order to force the PrivVM to perform more work during the AppVM's creation process, the AppVM in this setup is configured to make use of both

DVMs to access devices. Specifically, the AppVM's root disk is configured as a RAID level-1 device with each disk of the RAID hosted on a separate DVM. The AppVM accesses the network through one of the DVMs. The AppVM's workload consists of running sequentially the *netbench* and *fsbench* programs (discussed in Subsection 4.4.3).

Faults are injected into the PrivVM during the creation of the AppVM. This is accomplished by setting an injection trigger based on time before starting the AppVM. The trigger is set to fire at a random time between 100ms and 2.5s. A fault is only injected after this timer trigger has fired and a some number of PrivVM instructions, randomly selected from 0 to 20,000, has been executed. This time interval gives a fault a chance to be inserted into the PrivVM before, during, and slightly after the creation of the AppVM, but before the AppVM has been fully booted.

We have not evaluated the recovery mechanism when the PrivVM is actively performing other management activities such as migrating a VM. However, as discussed above, the PrivVM is mostly idle during normal operations and the techniques discussed in this chapter can already provide significant resiliency to failure as shown in the following section.

A PrivVM recovery is considered successful if after recovery the virtualized system is able to continue hosting existing VMs and the functionality of the PrivVM is restored, e.g., create/destroy VMs.

As with other VI components, the development of the PrivVM recovery mechanism is iterative. Hence, the number of injections at each improvement step is selected to be enough to allow the main cause of recovery failures to be identified. Thus, at each step, we performed about 500 injections resulting in more than 85 manifested faults. For the final recovery

scheme, to provide results that are more statistically significant, the number of injections is increased to 835 which results in about 140 manifested faults. To gauge the stability of the results for the final scheme (T_Requests - see next Section), we split the total number of injections into two equal halves and calculated the recovery success rates of the two halves. The outcome of this analysis yield less than 1% difference in the recovery success rates of the two halves, indicating that the results are stable.

6.5. Improving Recovery

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

Table 6.1. Fault injection into the PrivVM. Percentage of successful recoveries out of detected PrivVM failures.

Section 6.3 presented mechanisms that provide basic capabilities for PrivVM microreboot. Table 6.1 shows the results from our fault injection experiments evaluating this basic scheme. As shown in the top row, when the PrivVM is not interacting with VMs or performing any VM management operations (idle), the rate of successful recoveries out of all detected faults is greater than 92%.

If faults occur while the PrivVM is active, recovery is more difficult. As shown in the second row of Table 6.1, if faults are injected while the PrivVM is in the process of creating a new VM, recovery rate plummets to below 36%. Most failures (> 65%) occur because, during recovery, the new XenStored process fails while trying to acquire information from the

restored XenStore about the existing VMs on the system. Since the PrivVM fails in the middle of creating an AppVM, only a subset of the expected entries for the new AppVM are present in the XenStore backup. Hence, XenStored in the new PrivVM instance fails while attempting to process incomplete (invalid) XenStore state.

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

To demonstrate the feasibility and effectiveness of the above approach, we have implemented transactional versions of the VM create and destroy operations. This involved modifying *Xend* (see Subsection 2.1). Xend acknowledges requests *after* performing the requested operation. Our modified Xend uses the XBA and replicated XenStore to create a log that is maintained across PrivVM recovery and is available to the new Xend process.

Creating a VM requires sending two requests to Xend: (1) create the VM and (2) unpause the VM. Our modified Xend creates a log entry before executing the first request and removes the log entry before acknowledging this request. After recovery, if Xend detects a log entry for creating a VM, that indicates that the PrivVM had failed before acknowledging the ‘‘create VM’’ command and thus may have failed before completing all the steps involved

in creating a VM. Hence, Xend destroys the VM and cleans up any XenStore entries associated with that VM. With respect to the PrivVM, the processing of the ‘‘unpause VM’’ request is inherently atomic (one hypercall) and thus no modifications are needed to the processing of that request. The processing of the ‘‘VM destroy’’ request is made atomic using the same approach as for the ‘‘VM create’’ request.

With transactional VM creation, the successful recovery rate is above 86% (Table 6.1). More than 63% of remaining failures are cases in which the AppVM hangs waiting for a response to a XenStore request. Such requests are sent by the AppVM during its boot-up process to set up access to its disks. The AppVM hangs if the request is lost due to PrivVM failure while processing the request.

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

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

As originally implemented, XenStored updates the request consumer index once it reads

a request, before processing it. XenStored updates the response producer index after it places a response in the response ring. To help detect pending requests, our modified XenStored updates the request consumer index only after it updates the response producer index. Our modified XenStored also logs in the replicated XenStore the request consumer index before it begins to process a request and logs the response producer index after placing the response in the shared buffer but before updating the shared index. Thus, while XenStored is processing a request, the logged request consumer index equals the shared value. If the response to an in-progress request has already been made available to the requesting VM, the logged response producer index is *not* equal to the shared value. During PrivVM recovery, the new XenStored uses the values of the logged indices and the indices in the shared buffer to determine where in the request buffer to resume request processing.

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

6.6. Effectiveness of Detection

Before recovery can be performed, a failure must first be detected. How well the failure detection mechanisms work greatly influences the overall effectiveness of the PrivVM resiliency mechanism. Fortunately, the detection mechanisms described in Section 6.2 work quite well (detect over 96% of manifested faults), despite its simplistic nature.

Out of a total of about 835 injections we performed for the final PrivVM recovery scheme using the active PrivVM setup, roughly 140 manifested as failures. Of these manifested failures, 85.6% manifest as crashes, 10.8% manifest as hangs, and 3.6% manifest as silent failures (no PrivVM crash/hang detected but AppVM creation failed). Roughly 25% of the PrivVM crashes were detected by the user-level detection mechanism (*hostmon*).

6.7. Summary

This chapter presents the mechanisms to recover a PrivVM from failure. Unlike recovering the DVM, recovering the PrivVM requires maintaining state that must be restored after recovery. These states include the XenStore and some states maintained by the XenStored process. With just the basic recovery mechanism and with the PrivVM being idle, recovery of the PrivVM is successful over 92% of detected PrivVM failures. However, a PrivVM in the middle of performing management operations cause recovery success rate to drop to 36%. Enhancements to the basic recovery mechanism mainly include transactionalizing operations in the PrivVM in order to eliminate inconsistencies in the recovered PrivVM state. Based on transactionalizing only two critical VM management operations and adding mechanisms to ensure responses are generated for every pending requests, recovery success rate increases to over 96% of detected PrivVM failures.

Chapter Seven

Resilience to VMM Failures

The hypervisor† is a key component of the VI and is responsible for managing VMs' access to physical resources and is critical to the operation of the entire system. This chapter introduces and evaluates a mechanism for recovery from hypervisor failures called *ReHype* [Le11b]. ReHype is a mechanism that allows VMs to survive hypervisor failures without any loss of work in progress and without any performance overhead during normal operation. Upon hypervisor failure, ReHype boots a new hypervisor instance while preserving the state of running VMs. VMs are stalled for a short duration during the hypervisor reboot. After a new hypervisor is booted, ReHype integrates the preserved VM states with the new hypervisor to allow the VMs to continue normal execution.

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

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

A key contribution of our work is to identify the specific sources of state corruptions and inconsistencies, determine which of those are most likely to prevent successful recovery, and devise mechanisms to overcome these problems. We use the results of fault injection to incrementally enhance [Ng99] an initial basic version of ReHype. These incremental steps improve the rate of successful recovery from an initial 5.6% of detected faults to over 88% of detected faults. Our evaluation of the final scheme points to ways in which the success rate can be further improved.

ReHype builds upon the Otherworld [Depo10] mechanism for microrebooting the Linux kernel while preserving process states and the RootHammer [Kour07] mechanism for rejuvenation of the Xen hypervisor through a reboot while maintaining VM state without the overhead of saving VM memory images to persistent storage. Compared to Otherworld, the memory overhead for ReHype is significantly lower. Furthermore, while ReHype does not involve any changes to the VMs or the applications in the VMs, Otherworld requires modifications for system call retries and, in many cases, application-level ‘‘crash procedures’’ that are invoked upon recovery. Finally, we design and evaluate enhancements to the basic recovery mechanism in the context of a hypervisor, while Otherworld is focused on microrebooting of an OS kernel. RootHammer does deal with the Xen hypervisor and provides proactive rejuvenation. However, proactive rejuvenation is simpler since it does not address recovery from failure and thus does not deal with possible arbitrary corruptions and inconsistencies throughout the system.

The following section discusses the requirements and approaches to recover from a failed hypervisor. Section 7.2, describes the implementation of a version of ReHype that provides basic transparent hypervisor recovery but does not deal with many problems caused

by state corruptions and inconsistencies. Incremental improvements to ReHype, based on fault injection results, are described in Section 7.3. The details of the experimental setup are presented in Section 7.4. Section 7.5 discusses the impact of incorporating the PrivVM recovery mechanism into the hypervisor on ReHype. Additional enhancements to ReHype are discussed in Section 7.6. An evaluation of the effectiveness of the final version of ReHype with respect to different fault types is presented in Section 7.7. Sections 7.8 and 7.9 present the validation of ReHype's effectiveness on bare hardware and its ability to recover a hypervisor hosting FV VMs, respectively. The recovery latency of ReHype is discussed in Section 7.10.

7.1. Tolerating VMM Failure

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

Recovery from VMM failure that avoids losing work in progress in the VMs necessarily

relies on utilizing the VM states at the time of failure detection. One or more of those VM states may be corrupted, resulting in the failure of those VMs even if the rest of the system is restored to correct operation. Based on the reliability goal of a virtualized system discussed in Subsection 1.2, we define recovery from VMM failure to be successful if no more than one of the existing VMs running applications fails *and* the recovered VMM maintains its ability to host the other existing VMs as well as create and host new VMs.

Successfully ‘‘tolerating’’ VMM failure requires detection of such failures and successfully recovering from them, as defined above. To accomplish this goal, mechanisms must exist to: (1) detect VMM failure, (2) repair VMM corruption, and (3) resolve inconsistencies within the VMM, between the VMM and VMs, and between the VMM and the hardware. Detecting VMM failure boils down to being able to detect a VMM crash, hang, or silent data corruption, as described in Subsection 7.1.1. Subsection 7.1.2 discusses different approaches to repairing VMM corruption and the tradeoffs among them in terms of implementation complexity and expected success rates. Inconsistencies among the states of different components following recovery may be resolved entirely in the VMM or may require VM modifications. Details of the sources of inconsistencies and techniques for resolving inconsistencies are described in Subsection 7.1.3.

7.1.1. Detection

VMM failures can be manifested as VMM crashes, hangs, or silent corruption (arbitrary incorrect actions). Crashes can be detected using existing VMM panic and exception handlers. If the VMM panics, then a crash has occurred. Detecting VMM hangs requires external hardware. A typical hang detector, such as the one implemented in the Xen VMM,

uses a watchdog timer that sends periodic interrupts to the VMM. The interrupt handler checks whether the VMM has performed certain actions since the last time the handler was invoked. If it has not, the handler signals a hang.

Silent VMM corruption is more difficult to detect. Detection mechanisms involve redundant (e.g., replicated) data structures and redundant computations (e.g., performing sanity checks). TinyChecker [Tan12] can potentially be used to prevent some VMM memory corruption but requires the use of nested virtualization. Fortunately, our fault injection results (Section 7.7) indicate that the majority of VMM failures (65%-80%) are caused by crashes and hangs and are thus detectable using the simple mechanisms discussed above.

7.1.2. Repairing VMM Corruption

Repair is initiated when the detection mechanism invokes a failure handler. Corrupted VMM state can then be repaired by either identifying and fixing the specific parts of the VMM state that are corrupted or simply booting a new VMM instance. A major difficulty with the first alternative is the requirement to identify which parts of the state are erroneous. This is likely to require significant overhead for maintaining redundant data structures. Furthermore, complex repair operations performed in the context of a failed VMM can increase the chance of failed recoveries [Sull91] The approach of using nested virtualization with on-demand checkpointing proposed by [Tan12] can potentially be used to repair VMM state corruption but incurs a high runtime overhead. Hence, we focus on the latter alternative.

Normally, a system reboot would cause the loss of all VM states. The simplest approach to preserve all VM states across a reboot is to checkpoint the VMs to stable storage in the failure handler. Once a fresh system boots up, the VMs can be restored. However,

checkpointing VM state in the context of a failed VMM can increase the chance of failed recoveries since the VMM must perform I/O and access possibly corrupted structures that hold VM state. In addition, the space and time overhead of checkpointing can be large as the state of all VMs must be copied to stable storage.

An alternative approach to a system-wide reboot, is to microreboot [Cand04] the VMM. In this approach, VM states are preserved in memory across the reboot. This avoids the space and time overhead of checkpointing VM states to stable storage. Once the new VMM has been booted, it must be re-integrated with the preserved VMs. This re-integration can be done by either recreating the VMM structures used to manage the VMs or reusing VMM structures preserved from the old VMM. Either way, some amount of VMM data needs to be preserved across a VMM reboot for the re-integration process.

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

The choices of whether to reboot the VMM in place or in a reserved memory area affects the operations that must be done in the failure handler. This, in turn, can affect the chances of a successful recovery. When the new VMM is booted in place, the failure handler must perform two operations: 1) preserve VMM state (data structures) needed for re-integration by copying it to a reserved memory location, and 2) overwrite the existing VMM image in memory with a new image. These two operations do not have to be performed if the new VMM is booted into a reserved memory area. There is no VMM state to copy since the new

VMM is confined to the reserved memory area on boot so no old VMM memory state is lost. In addition, the new VMM image can be preloaded into the reserved memory area without affecting the operation of the current VMM. Performing fewer operations in the failure handler, as is the case when rebooting into a reserved area, can increase the chance of a successful recovery. However, as discussed below, the choice of booting the new VMM instance in a reserved memory area requires complex operations for the re-integration of the new VMM instance with the preserved VMs.

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

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

values in the new VMM's data structures. However, if the preserved structures are reused, there is a greater risk of also introducing into the new VMM instance corrupted pointers, which may lead to further corruption.

There is no perfect solution to the problem of ending up with corrupted values in the new VMM's data structures. As discussed above, some structures can be initialized with static or ''safe'' values. In other cases, data structures may be populated based on other data structures from the failed VMM instance, at least ensuring consistency (see Section 7.1.3). In general, the recovery process needs to ensure that all values that will be used by the new VMM instance are safe — will not lead to subsequent VMM failure. This may involve performing integrity checks, possibly requiring maintaining redundant information, such as logs or checkpoints [Tan12] during normal operation.

Given the tradeoffs presented in this subsection, ReHype uses the microreboot approach and opts for a simple implementation that does not require major modifications to the VMM. Hence, ReHype preserves and reuses almost all of the VMM's dynamic memory but updates a few key data structures with ''safe'' values, as described in Sections 7.2 and 7.3. The benefit of reusing most of the VMM's data structures is that it allowed ReHype to be easily integrated into a VMM (Xen in our case) with minor (900 LOC added/modified) modifications. Despite the reuse of old VMM data in ReHype, fault injection results show a high rate of successful recoveries (Section 7.7).

7.1.3. Resolving Inconsistencies

By design, the VMM is assumed to be reliable by the VMs and hardware. Typically, the VMM itself is, for the most part, implemented with the assumption that there are no hardware

faults and no software faults in the VMM code. These assumptions are violated when the VMM fails. Thus, after recovery from a failed VMM, even if none of the states of the system components are corrupted, these states may be *inconsistent*, preventing the system from operating correctly.

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

Sources of VMM/VMM inconsistencies include partially updated structures, unreleased locks, and memory leaks. The options for resolving these inconsistencies are, essentially, special cases of the options for dealing with state corruption, discussed in the previous subsection. Resolving inconsistencies caused by partially updated structures requires either reconstructing the data structures using information from the failed VMM or using redundant information logged prior to failure to fix the new VMM's state. A scheduler's run queue is an example of a data structure for which the former technique can be used. Inconsistency can occur if a VCPU becomes runnable but the VMM fails before inserting it into the run queue.

Resolving this inconsistency requires re-initializing the run queue to empty upon bootup and re-inserting all runnable VCPUs (obtained from the failed VMM) into the run queue. For other data structures, such as the ones that track memory page usage information, reconstruction is more difficult so the latter technique may be preferable. For instance, an inconsistency can occur if a failure happens right when a page use counter has been updated but before that page has been added to a page table. Resolving this inconsistency by traversing all page table entries to count the actual mappings to that page can be done but is complex and slow. Instead, the entire mapping operation can be made atomic with respect to failure using *write-ahead logging*, involving a small overhead during normal operation and simple, fast correction of any inconsistencies upon recovery.

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

A memory leak can occur if a failure happens between allocation and freeing of a memory region in the VMM. Such a memory leak is benign if the leaked region is small. After VMM recovery, the system can be scheduled to be rebooted to reclaim leaked memory. Alternatively, leveraging work by Kourai and Chiba[Kour10], after recovery, the virtualized system can be quickly rejuvenated to get rid of any memory leaks.

Sources of VMM/VM inconsistency include erratic and/or non-monotonic changes in system time, partially executed hypercalls, and undelivered virtual interrupts. The correct operation of many VMs depends on a system time that is monotonically increasing at a constant rate. In a virtualized system, the VMs' source of time is the VMM. When a VMM

is rebooted, its time keeping structures are reset, potentially resulting in a time source for VMs that is erratic (e.g., ceases to advance or advances suddenly by a significant amount) or that is not monotonically increasing. In addition, such a reset can result in timer events set using time relative to the VMM's system time prior to recovery to be delayed. One technique for resolving this inconsistency is to simply save the VMM time structures upon failure and restore those structures after the VMM reboot and before the VMs are scheduled to run. This allows time to continue moving forward with no interruption visible by the VMs. For external entities that interact with the VMs and expect time to remain approximately synchronized with real time, additional mechanisms such as NTP will be required to slowly accelerate time in the virtualized system to catch up with real time.

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

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

Hypercall retry can be implemented without modifying the VMs. To force re-execution of a hypercall after recovery, the VMM adjusts the VM's instruction pointer to point back to the hypercall instruction (usually a trapping instruction). When the VM is scheduled to run,

the very next instruction it executes will be the hypercall. This mechanism is already used in the Xen [Barh03] VMM to allow the preemption of long running hypercalls transparently to the VMs.

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

A source of VMM/hardware inconsistency is unacknowledged interrupts. The processor blocks delivery of pending interrupts that is of lower or equal priority than the current interrupt being serviced by the VMM. These blocked interrupts can be delivered once the in-service interrupt has been acknowledged by the VMM. In addition, for level-triggered interrupts, the interrupt controllers will block an interrupt source until the previous interrupt from that source has been acknowledged. Since VMM failure can occur at any time, the interrupt being serviced at the point of failure may never get acknowledged, thus blocking interrupts of lower or equal priority indefinitely. If VMM recovery is done without

performing a hardware reset, a mechanism is needed to either reset both the pending interrupt state in the processor and the I/O controller or acknowledge all pending interrupts during recovery. In the case of acknowledging pending interrupts, the interrupt source must be blocked at the interrupt controller before the interrupt is acknowledged to prevent another interrupt from slipping by before the VMM is ready to handle the interrupt.

7.2. Transparent VMM Microreboot

This section describes the implementation of a version of ReHype that provides the basic capability to microreboot the VMM while preserving the running VMs and allowing them to resume normal execution following the microreboot. Improvements to the basic scheme that enhance recovery success rates are discussed in Section 7.3.

To microreboot the VMM, ReHype uses the existing Xen port of the Kdump [Goya05] tool. Kdump is a kernel debugging tool that provides facilities to allow a crashed system to load a pristine kernel image, in this case the VMM image, on top of the existing image and directly transfer control to it. The Kdump tool by itself, however, does not provide any facilities to preserve parts of memory, such as those holding VM states. The burden of memory preservation is on the kernel or VMM being booted.

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

On boot, the stock Xen VMM initializes the entire system memory and allocates memory for its own use. The modifications for ReHype must ensure that, upon recovery, the new VMM instance preserves the memory used by VMs and memory that was used by the previous VMM instance to hold state needed for managing the VMs. Hence, as described in Subsection 7.2.1, the ReHype version of the Xen VMM allocates ‘‘around’’ the preserved memory regions during a VMM microreboot. When the new VMM instance is booted and initialized, it does not contain information about the running VMs, and thus has no way to run and manage them. Subsection 7.2.2 describes how the VMM and VM states preserved during VMM recovery are re-integrated with the new VMM instance.

7.2.1. Preserving VMM and VM States

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

VMM microreboot involves overwriting the existing VMM image (code, initialized data, and bss) with a pristine image. The VMM’s static data segments (initialized data and bss) contain critical information needed for the operation of the system following bootup of the new VMM instance. For example, this includes interrupt descriptors and pointers to structures on the Xen heap, such as Domain 0’s VM† structure and the list of domains. Hence, some of the information in the old static data segments must be preserved. While it is

† The terms Domain 0 and Privileged VM (PrivVM) are used interchangeably.

only necessary to preserve a subset of the static data segments, since they are relatively small, we reduce the implementation complexity by preserving the segments in their entirety.

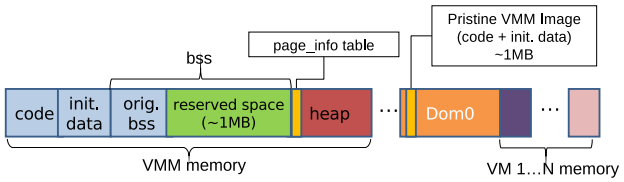


Figure 7.1: Layout of system memory for ReHype.

Figure 7.1 shows the memory layout for Xen, as modified for ReHype. In particular, the bss segment is extended by approximately 1MB — sufficient space to hold complete copies of the original bss and initialized data segments. This is done by changing the linker script used for creation of the Xen image. Since the area reserved is in the bss segment, no extra disk space is taken up for the new VMM image. The bss segment only takes up space when it is loaded into memory. Upon failure detection, before initiating a VMM microreboot, the failure handler copies the initialized data and (unextended) bss segments to this new reserved memory.

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

access the old heap, the page table from the old VMM must be restored. This requires copying the old page directory from the old bss segment, preserved as discussed above, to the new bss segment.

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

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

Preserving the heap and static data segments of a failed VMM is unsafe — it can result in recovery failure if those preserved values are corrupted by the failed VMM. Section 7.3 will discuss mechanisms that dramatically improve the chances of successful recoveries despite re-using the preserved heap and static data segments.

7.2.2. Re-integrating VMM and VM States

Following a VMM microreboot, the VMM does not have the information required to resume execution and manage the VMs that were running at the time of failure. The missing system state includes the list of VMs it was managing, the system time that was provided to the VMs, information for interrupt routing (to processors) and forwarding (to VMs), and timer events that were scheduled by the VMs. To allow the virtualized system to continue running, these components of the system state must be restored. As discussed in the previous subsection, all the required state is preserved across a VMM microreboot. Hence, all that is needed is to re-integrate the preserved information with the new VMM instance. This re-integration is accomplished by copying a few key values from the old static data segments to the new static data segments. The restoration is done before the VMs can be scheduled to run. The following structures must be restored:

- Pointer to xmalloc free list: prevent memory leaks. Pointers to the domain list and hash table: allow Xen to access the state of the running VMs.
- Pointer to the Domain 0 descriptor: since Domain 0 is not rebooted as part of recovery, the pointer to it must be restored to allow Xen access to the Domain 0 structure.
- Pointers to timer event objects: restore pending timer events on the old timer heap to the new timer heap.
- Pointer to the machine-to-physical (m2p) mapping table: make available mapping of machine frame numbers to physical frame numbers.
- System time variables: maintain monotonically increasing time. The time-stamp counter (TSC) must not be reset.

- IRQ descriptor table and IO-APIC entries, including correct IRQ routing affinity and mask: allow VMs to continue to receive interrupts from their devices.
- Structures for tracking the mappings of pages shared between VMs and the VMM: prevents overwriting mappings that are still in use.

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

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

7.3. Improving Recovery

The scheme presented in the previous section provides basic capabilities for VMM microreboot. However, as explained below, with this basic mechanism the probability of successful recovery is very low. The section starts with the basic scheme and incrementally improves on that scheme to achieve higher recovery success rates. Table 7.1 shows the mechanisms used to improve the basic recovery scheme. As in [Ng99], the choice of improvements is guided by results from fault injection experiments. The last two mechanisms

Mechanism	Description
NMI IPI	Use NMI IPIs in failure handler. Avoid IPI blocking by failed VMM.
Acknowledge interrupts	Acknowledges all in-service interrupts in all processors to avoid blocked interrupts after recovery.
Hypercall retry	All partially executed hypercalls are retried transparently to the VMs.
FixSP	Stack pointer set to ‘‘safe’’ value in failure handler.
NMI ‘‘ack’’	Execute iret to ‘‘ack’’ NMI when hang detected on non-CPU0.
Reinitialize locks	Dynamically allocated spin locks and non-spin locks are unlocked.
Reset page counter	Reset page use counter based on page validation bit.
Acknowledge interrupts (enhanced)	Acknowledges all in-service and pending interrupts in all processors to avoid blocked interrupts after recovery.
Clear ‘‘running’’ VCPU flag	Clear the VCPU flag that indicates the VCPU is currently running before rescheduling the VCPU.

Table 7.1. Improvements over the basic ReHype recovery.

are discussed in more detail in Section 7.6.

We used software-implemented fault injection to introduce errors into CPU registers when the VMM is executing. The goal of the injection was to cause arbitrary failures in the VMM and evaluate the effectiveness of different recovery mechanisms. Two system setups were used: 1AppVM and 3AppVM. The 1AppVM setup, shown in Figure 7.2, was comprised of two VMs: Domain 0, the *Privileged VM* (PrivVM), and an Application VM (AppVM) running a disk I/O (block) benchmark (AppVM_Blk). This setup was used to quickly identify major short comings with the recovery mechanisms. The more complex 3AppVM system setup, shown in Figure 7.3, was used to further stress the virtualized system, once the majority of sources of failed recoveries had been fixed. In this setup, AppVM_Blk is designed to boot after VMM recovery. This is a check of whether the recovered virtualized

system maintains the ability to create new VMs.

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

Table 7.2. Injection outcomes.

Table 7.2 summarizes the possible outcomes from an injected fault (injection experiment). Only detected VMM failures lead to VMM recoveries. With the 1AppVM setup, a recovery is considered successful if the benchmark in the AppVM_Blk completes correctly. With the 3AppVM setup, following the explanation in Section 7.1, a recovery is considered successful if either AppVM_Net or AppVM_Unix completes its benchmark correctly and AppVM_Blk is able to boot and run its benchmark to completion without errors. Silent failures, discussed in Section 7.7, do not trigger VMM recovery and are thus excluded from further discussion in this section.

The number of injections performed in different fault injection campaigns in this section progressively increases from 300 to over 2800 as the main causes of recovery failures become harder to identify due to the increased effectiveness of ReHype. Details of the experimental setup, fault injection campaign, the number of injections performed, and failure detection mechanisms are in Section 7.4.

We start the incremental refinement process of ReHype with the VMM hosting paravirtualized (PV) VMs. PV VMs are an important class of VMs as they include special purpose VMs such as the PrivVM and DVMs [Fras04]. For the purpose of stressing ReHype,

PV VMs are a good choice to use as AppVMs since the VMM is highly involved in supporting many typical VM operations such as page table updates, process scheduling, and timer operations. The ability of ReHype to recover a VMM hosting fully-virtualized (FV) VMs is presented in Section 7.9.

In the rest of this section, each version of the recovery scheme is described, and fault injection results are presented. This is followed by an analysis of the main cause of failed recoveries, motivating the next version of the recovery scheme. At each step, only the problem that leads to the plurality of failed recoveries is analyzed and fixed. Table 7.3 summarizes the rate of successful recoveries with the basic ReHype scheme and the various incremental improvements that are made.

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

Table 7.3. Fault injection results for 1AppVM system setup. Percentage of successful recoveries out of detected VMM failures (VMM crash/hang).

Basic: As shown in Table 7.3, with the Basic recovery scheme (Section 7.2), the successful recovery rate is only 5.6%. A large fraction of recovery failures (44%) occur because the failure handler is unable to initiate the VMM microreboot. Normally, the failure handler relies on interprocessor interrupts (IPIs) to force all processors to save VM CPU state and halt execution before microrebooting the VMM. Microrebooting the VMM cannot

proceed until all processors execute the IPI handler. Therefore, the failure handler is stuck if a processor is unable to execute the IPI handler due to a blocked IPI or memory corruption.

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

Table 7.3 shows an increase in recovery success rate to 17.6% when these fixes are used. Only 8.2% of the failures are now caused by an inability to initiate the VMM microreboot. The plurality of the remaining failures (45%) are due to interrupts from the block device not getting delivered to the PrivVM. This causes the block device driver in the PrivVM to time out, thus leading to the failure of block requests from the AppVM.

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

Acknowledge interrupts: To prevent level-triggered interrupts from being blocked, the failure handler must acknowledge all pending interrupts on all processors. Pending interrupts can be classified as interrupts waiting to be serviced by the VMM or interrupts currently being serviced by the VMM (in-service). Acknowledging in-service interrupts can be easily accomplished by executing the *EOI* instruction for every interrupt that has been delivered to the VMM. It is necessary to perform this operation in the failure handler since information about pending interrupts in the CPU are cleared after a CPU reset during a VMM reboot. A more complete, but more involved, approach at acknowledging *all* pending interrupts

(including interrupts waiting to be serviced) is discussed in Section 7.6.

Table 7.3 shows that when this mechanism is added, the successful recovery rate jumps to 48.6%. Of the remaining unsuccessful recoveries, 52.8% are caused by a crashed AppVM or PrivVM after recovery. The crashes are caused by bad return values from hypercalls. Since VMM failures can occur in the middle of a hypercall, it is necessary to be able to transparently continue the hypercall after recovery. Without mechanisms to do this, after recovery, the VM starts executing right after the hypercall, using whatever is currently in the EAX register as the return value.

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

Table 7.3 shows that, with hypercall retry, the successful recovery rate is 62.6%. Out of the remaining unsuccessful recoveries, 41% are caused by the same symptom encountered and partially solved with the Basic scheme — the inability of the failure handler to initiate the VMM microreboot. With the improved recovery rate, the causes of this symptom not previously resolved are now responsible for the plurality of failed recoveries.

The experimental results show two causes for the symptom above: 1) NMI IPIs sent to the wrong destination CPU due to stack pointer corruption and 2) NMIs are blocked due to

NMI-based watchdog hang detection. Problem (1) occurs because a corrupted stack pointer is used to obtain the CPUID of the currently running processor. The obtained CPUID is incorrect and is, in turn, used to create a CPU destination mask for the NMI IPI. This mask can end up containing the sending processor as one of the destination CPUs. The result of this is that an IPI is incorrectly sent to the sending processor. This IPI is dropped and the sender waits forever for the completion of the IPI handling.

Problem (2) is due to the fact that NMI delivery is blocked if a CPU is in the middle of handling a previous NMI — an *iret* instruction matching a previous NMI has not been executed [Corp10] The Xen hang detector is based on periodic NMIs from a watchdog timer. If a hang is detected on a processor, that processor immediately executes the panic handler and never executes an *iret* instruction. This prevents the processor from getting an NMI IPI from the boot processor to initiate recovery.

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

registers after recovery, the VMM may try to restore a corrupted value as the VCPU code segment register. This may cause the VMM to continue executing with VMM privilege using corrupted (incorrect) register values.

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

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

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

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

Reinitialize locks: Re-initializing dynamically-allocated spin locks requires tracking the allocation and de-allocation of these locks. All locks that are still allocated upon recovery are initialized to unlocked state. This tracking of spin locks is the only extra work that ReHype

must perform during normal operation. The associated performance overhead is negligible since the allocation and de-allocation of spin locks is normally done only as part of VM creation and destruction. Furthermore, there are only about 20 spin locks that are tracked per VM.

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

As shown in Table 7.3, re-initializing locks increases successful recovery rate to 95.8%. For the remaining recovery failures there is not one dominant cause.

While the 1AppVM system setup is useful for uncovering the main problems with the Basic ReHype recovery, it is very simple, thus potentially hiding important additional problems. To better stress the virtualized system, the rest of the experiments in this section use the 3AppVM setup. The results with this setup are summarized in Table 7.4.

As shown in Table 7.4, with the 3AppVM setup, the reinitialize locks mechanism results in a recovery success rate of 88.6%. Hence, there is a small decrease in the success rate compared to the 1AppVM setup.

Mechanisms	Successful Recovery Rate
Reinitialize locks	88.6%
+ Reset page counter	92.2%

Table 7.4. Fault injection results using 3AppVM system setup. Percentage of successful recoveries out of detected VMM failures (VMM crash/hang).

Out of the remaining recovery failures, about 35% are due to the VMM hanging

immediately after recovery. This problem is caused by a data inconsistency resulting from a VMM failure while in the middle of handling a page table update hypercall. This hypercall promotes an unused VM page frame into a page table type by incrementing a page type use counter and performing validity check on the page frame. After the validity check, a validity bit is set to indicate that the page can be used as a page table for the VM. Inconsistency arises when a VMM failure occurs before the validity check is completed but after the page type use counter has been incremented. When the hypercall is retried after recovery, since the page use counter is not zero and the validity bit is not set, it assumes validation is in progress and waits by spinning. Of course, there is no validation taking place, and the CPU is declared hung by the hang detector.

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

With the page counter fix employed, recovery success rate improves to 92.2%. The remaining causes of failed recoveries vary widely and are discussed in more detail in Section 7.7.

7.4. Experimental Setup

This section presents the experimental setups used to evaluate and validate the recovery effectiveness of ReHype. Specifically, this section discusses details of the different system configurations used for stressing ReHype, the different workloads, the fault injection campaigns and fault types, the number of faults to inject, the fault injection outcomes, and the failure detection mechanisms.

7.4.1. System Configurations

The physical machines used for running experiments are equipped with dual quad-core Intel Nehalem and Core 2 Duo processors and 8GB of memory. In general, the virtualized system under evaluation is comprised of the Xen VMM, augmented with ReHype, hosting the PrivVM, possibly a DVM (depending on the experiment), and one or more AppVMs. The number and type of AppVMs (PV or FV) and the workload that is run is determined by the specific experiment.

All experiments in this work make use of two basic system configurations: 1AppVM and 3AppVM. As mentioned in Section 7.3, the 1AppVM and 3AppVM configurations are used in experiments designed to identify and help improve recovery problems of ReHype. With the exception of one set of experiments, the AppVMs are all PV VMs. The one exception is for experiments validating ReHype's ability to tolerate VMM failure when hosting *FV* VMs. We describe these two basic system configurations below.

The 1AppVM configuration is shown in Figure 7.2. In this configuration, the system hosts two VMs: a PrivVM (Domain 0) and a single AppVM (AppVM_Blk). AppVM_Blk runs the *blkbench* benchmark, described below, which continuously performs disk I/O (block) operations. The PrivVM hosts the block backend for the AppVM_Blk. Each of the VMs consists of one virtual CPU (VCPU) that is pinned to its own physical CPU (PCPU).

The 3AppVM configuration is shown in Figure 7.3. In this configuration, the PrivVM's root filesystem is in memory and the PrivVM does not access any devices. A separate DVM [Fras04] has direct access to the network and SCSI controllers and serves as the network backend for AppVM_Net and block backend for AppVM_Blk. AppVM_Unix has

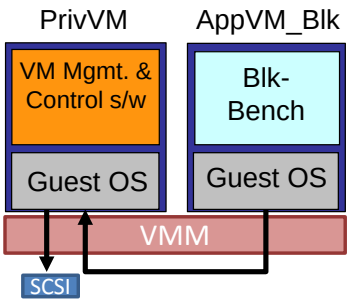


Figure 7.2: System configuration for 1AppVM. AppVM_Blk accesses block device (disk) through the PrivVM.

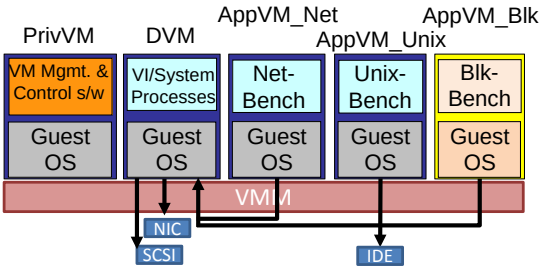


Figure 7.3: System configuration for 3AppVM. AppVM_Blk created after recovery. AppVM_Net and AppVM_Blk accesses I/O devices through the DVM. AppVM_Unix accesses block device directly.

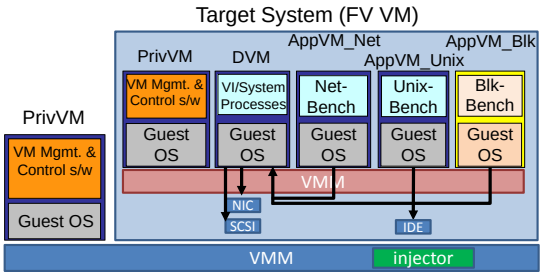


Figure 7.4: Target system is 3AppVM deployed inside a FV VM. Injection is performed from the ‘‘outer’’ VMM.

direct access to the IDE controller, and thus does not rely on the DVM for any device access. Each VM consists of one VCPU which is pinned to its own PCPU. To check whether the recovered system maintains its ability to create new VMs, AppVM_Blk is designed to boot up after a possible VMM recovery and run the *blkbench* benchmark.

In order to simplify the setup for software-implemented fault injection and speedup the campaigns, for experiments using the 1AppVM and 3AppVM configurations, the target system was run inside a fully-virtualized (FV) VM. Figure 7.4 shows this setup when the target system is the 3AppVM configuration. This made it easy to restart the target system and refresh its disk images after each injection run to isolate the effects of faults injected in different runs.

One potential problem with performing the evaluation of ReHype in a virtualized environment is that the results may not be representative of results obtained if ReHype had been evaluated on bare hardware. Differences in fault injection outcome may be due to differences in hardware configurations between a virtualized and bare machine and differences in behavior of emulated versus real devices. To ensure our results are meaningful, we validate the results obtained when ReHype is deployed in a virtualized environment using the 3AppVM configuration with results obtained from deploying the same configuration on bare hardware. The main configuration differences between the bare hardware and virtual machine setup are system memory size (8GB vs. 2GB) and the type of block (disk) controller accessed by AppVM_Unix (SCSI instead of IDE).

As mentioned above, the 3AppVM configuration is also used to validate ReHype's ability to tolerate VMM failure when hosting *FV* VMs. In these experiments, the target

system is run on bare hardware and all the AppVMs are FV VMs. In this setup, AppVM_Unix accesses its block device through the DVM rather than directly as in the original 3AppVM configuration.

7.4.2. Workloads

We use a synthetic workload to evaluate ReHype. The synthetic workload consists of three micro-benchmarks: *netbench*, *blkbench*, and *UnixBench*. Details of *netbench* and *blkbench* are described in Subsection 4.4.3. For *netbench*, similar to the description in Subsection 4.4.3, if a VMM error is detected, an unsuccessful application completion is recorded if: (1) there is a interruption of more than 10s 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. However, given a fault, if no VMM errors are detected, an unsuccessful application completion is recorded if 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. While these failure criteria cover more complicated failure modes, based on our experiences, the effects of a failed VMM on netbench is much more simplistic — in the vast majority of cases, the PM host simply stops receiving packets from the VM host.

For *Blkbench*, given a fault, regardless if there is a VMM error detected or not, an unsuccessful application completion 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 reference image.

The third benchmark (*UnixBench*) is a subset of the set of programs in UnixBench [Unix] with minor modifications to improve logging and failure detection. The selected programs were chosen for their abilities to stress the VMM's handling of hypercalls such as virtual memory management and process scheduling. Given a fault, regardless if there is a VMM error detected or not, an unsuccessful application completion is recorded if: (1) one or more programs in UnixBench terminate prematurely due to failed system calls, or (2) the resulting program output differs from a reference output.

7.4.3. Fault Injection Campaigns and Fault Types

We used the *Gigan* fault injector, described in Chapter 9, to inject faults into the VMM. A fault injection campaign consists of many fault injection runs. A single fault injection run that uses the 1AppVM system configuration consists of first booting the VMM along with the PrivVM and AppVM_Blk. AppVM_Blk begins running the blkbench benchmark and a fault is injected into the VMM. The injection campaign infrastructure allows the target system sufficient time for the VMM to recover and for the benchmark to complete. If the benchmark does not complete, a timeout mechanism identifies system failure. At the end of each run, fault injection logs and benchmark output are retrieved and stored for analysis.

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

Three types of faults are used to evaluate ReHype: random single bit-flips in CPU registers during execution of VMM code, random single bit-flips in VMM code segment (Code), and software faults in the VMM (SW). Table 4.2 in Chapter 4 shows the different fault types in more detail. For register faults, the experiments in this chapter also inject into the system flags register (EFL). However, when the target system is run inside a FV VM, we do not inject into the reserved bits and the VM-8086 bit of the EFL register. This is due to the limitation of performing fault injection into a VM as some hardware faults cannot be accurately emulated (see Chapter 9 for more details). These three fault types were chosen as they can cause a wide range of possible corruptions in the entire system, hence, satisfying the main goal of the evaluation, which is to ‘‘stress’’ the recovery mechanisms in order to identify problem areas.

An injection is triggered after a random time period between 500ms to 6.5s after the AppVMs begin running their benchmarks. For software faults and code injection, a breakpoint is used to trigger an injection. We used the fault injection tool in [Swif05] to generate a list of injection targets. The breakpoint is set on the target VMM instruction after the designated time has elapsed. To increase the activation rate of software faults and code injection, we used the Xenoprof [Meno05] sampling profiler to identify the most frequently executed functions in the VMM, considering only the instructions in those functions as possible targets for fault injection. In each run, a breakpoint is set on a randomly selected CPU of the target system. For register injection, to ensure that the injection occurs only when the VMM is executing, a fault is only injected after the designated time has elapsed and 0 to 20,000 VMM instructions, chosen at random, have been executed.

7.4.4. Numbers of Injections

The number of faults injected in a campaign is determined by a tradeoff between execution time and statistical significance. The number of faults injected should be minimized since it is directly proportional to the total time required to run the campaign. However, the accuracy (statistical significance) of the results increases as the number of injections is increased. Hence, the number of faults injected should be maximized. As a practical matter, the number of faults to inject is dependent upon the goal of the injection campaign and the manifestation rate of the fault. Specifically, when the goal is to identify the main recovery problems of ReHype (Sections 7.3 and 7.6), the number of injections can be less than when the goal is to analyze and validate the overall effectiveness of the final version of ReHype (Sections 7.7-7.9). In addition, faults into CPU registers have lower manifestation rates than software faults or code corruption whose target instructions are selected from the most executed functions identified through profiling. Hence, the number of injections needed to yield statistically-significant results for campaigns targeting CPU registers will be higher than campaigns injecting software faults or code corruption.

In accordance with the discussion above, in Section 7.3, we performed roughly 300 to 600 injections into CPU registers for each campaign (each improvement step) using the 1AppVM setup. As recovery failures decrease and the 3AppVM setup is used, the number of injections per campaign is increased to roughly 2800. This same number of injections is used for each experiment in Sections 7.5 and 7.6.

For campaigns used for analyzing the overall effectiveness of ReHype (see Section 7.7), we injected roughly 2800 CPU register faults, 830 SW faults, and 650 Code faults. For campaigns used for validating ReHype (see Sections 7.8 and 7.9), a fault injection run takes

much longer to complete as the system is deployed on bare hardware (see Subsection 9.5.5). Hence, we injected fewer faults per campaign with the numbers being approximately 900 injections into CPU registers, 400 injections using SW faults, and 340 injections using Code faults. With these numbers of injections, the outcomes are sufficiently stable to allow meaningful analysis of the results (details of significance analysis can be found in the respective sections).

7.4.5. Fault Injection Outcome

As summarized in Table 7.2, the outcome of each injection run is classified as: *detected* (VMM crash/hang), *silent* (undetected failure), or *non-manifested*. A crash occurs when the VMM panics due to unrecoverable exceptions. Hangs occur when the VMM does not perform its expected operation in a timely manner. A silent failure occurs when no VMM hang or crash is detected but the applications (workloads) in one or more AppVMs fail to complete successfully. What constitutes unsuccessful completion is application specific and is discussed above. Non-manifested means that no errors are observed.

7.4.6. Failure Detection

We rely on simple, but effective, techniques to detect when the VMM has crashed or hang. In particular, a crash is detected when the VMM invokes the panic handler due to unrecoverable exceptions. VMM hangs are detected using a watchdog mechanism built into Xen. Specifically, Xen maintains a watchdog counter that is supposed to be incremented by a normal timer event every 100ms. A watchdog NMI is generated every 100ms of unhalted CPU cycles. If the watchdog NMI handler detects that the watchdog counter has not been

incremented for 300ms, the system is declared hung.

7.5. Impact of VMM Modifications on ReHype

As part of the work to increase the resiliency of virtualized systems, aside from ReHype, we have developed mechanisms to tolerate failures of the DVMs (Chapter 4) and the PrivVM (Chapter 6). All of these resiliency mechanisms together form the core of our fault tolerant VI (FTVI), discussed in more detail in Chapter 8.

The evaluation of ReHype up to this point has been conducted using a system that included only the DVM recovery mechanisms. However, the DVM recovery mechanisms were disabled as the focus was on the evaluation of ReHype. We refer to the FTVI version with only ReHype and DVM recovery mechanisms as $FTVI_{HD}$ and with all mechanisms including the PrivVM recovery mechanisms as $FTVI_{HDP}$. The focus of this section is to assess the extent to which VMM modifications in $FTVI_{HDP}$ (discussed below) impact the behavior of ReHype.

Compared to $FTVI_{HD}$, the VMM in $FTVI_{HDP}$ has gone through significant changes in order to support recovery of the PrivVM. In addition, the mechanism to log VMM debug output used for post-mortem analysis when a VMM failure is detected has also been simplified in the $FTVI_{HDP}$ version. This latter modification is made based on analysis that suggests some failed recoveries may be caused by logging operations accessing corrupted state in the failed VMM.

With respect to PrivVM recovery, modifications to the VMM include adding about 365 lines of code and the use of about 128MB of VMM memory to store the PrivVM's kernel and

filesystem images (see Chapter 6 for details). With respect to logging simplifications, all outputs by the VMM to the serial console are disabled to prevent the potential recovery failure associated with using the serial console during VMM failure handling. In addition, separate 8KB memory buffers are created for each CPU (64KB total) to store outputs from the VMM's crash handler. This departs from a smaller 16KB buffer shared by all CPUs in the $FTVI_{HD}$ version.

Modifications to the VMM can alter the alignment and location of critical data structures in the VMM which ultimately can affect the outcome of an injected fault. For example, an unrecoverable VMM failure due to a corruption of a critical VMM data structure located near the VMM's stack caused by a fault in the stack pointer may *not* occur if that same data structure is allocated instead in a completely different area of memory. Hence, it is important to ensure that the VMM modifications outlined above do not qualitatively change the behavior and effectiveness of ReHype as reported in the previous sections.

Configuration	Mechanism	Successful Recovery Rate
1AppVM	Reinitialize locks	93.3% (95.8%)
3AppVM	Reinitialize locks	83.0% (88.6%)
3AppVM	+ Reset page counter	88.1% (92.2%)

Table 7.5. Comparing success rates of ReHype in $FTVI_{HDP}$ and $FTVI_{HD}$ with respect to different system configurations and recovery mechanisms. Results for ReHype in $FTVI_{HD}$ reshown in parenthesis.

In the following paragraphs, we compare fault injection results presented in the previous section using $FTVI_{HD}$ to results using $FTVI_{HDP}$. Similar to the DVM recovery mechanism, the PrivVM recovery mechanism is disabled during the evaluation of ReHype.

FTVI Version	Manifested / Injected	Manifested		Successful Recovery Rate / Detected
		Detected Failure	Silent Failure	
$FTVI_{HD}$	31.2%	81.5%	18.5%	92.2%
$FTVI_{HDP}$	32.9%	81.4%	18.6%	88.1%

Table 7.6. Comparing fault injection results of ReHype with all the enhancements up to and including *Reset page counter* using the 3AppVM setup between $FTVI_{HD}$ and $FTVI_{HDP}$.

The main outcomes comparing the success rates of ReHype in $FTVI_{HDP}$ and $FTVI_{HD}$ with respect to different system configurations and recovery mechanisms discussed in Section 7.3 are shown in Table 7.5. From these results, we find the behavior of ReHype under the two FTVIs to be very similar to each other. In particular, in both versions, there is a decrease in the success rates when going from the simple 1AppVM configuration to the more complex 3AppVM configuration and a small improvement in success rates when the final recovery enhancement (Reset page counter) is applied.

While the relative improvement trends may be similar, the success rates using the final *Reset page counter* enhancement for $FTVI_{HDP}$ is significantly less than $FTVI_{HD}$, 88.1% vs. 92.2%. This discrepancy most likely has to do with the differences in exactly where critical data structures such as descriptor tables and page directories/tables are located in memory between the two FTVI versions. Corruption of these data structures will often lead to failure during a VMM microreboot. In fact, close to 19% of failed recoveries using the $FTVI_{HDP}$ version were caused by a combination of fatal page faults during the reboot of the new VMM and corruption of segment descriptor tables leading to the crash of the entire target system (FV VM) by the host VMM conducting the experiment (see Subsection 7.4.1). These particular types of failures, while possible, did not occur in the experiments using $FTVI_{HD}$.

Table 7.6 shows in more detail the results from fault injection experiments using the latest enhancements between the two FTVI versions. Despite the small decrease in recovery success rates, the overall behavior of ReHype with respect to faults between the two FTVI versions are very similar. These results suggest that the enhancements in $FTVI_{HDP}$ do not qualitatively alter the behavior of ReHype under faults. The $FTVI_{HDP}$ setup will be used in all further evaluation in this chapter.

7.6. Enhancements to ReHype

Over the course of several months of experimenting with ReHype, we identified two simple enhancements to ReHype that can increase the overall effectiveness of the mechanism. The first enhancement reduces single AppVM failures in the case of successful recoveries and the second enhancement prevents the VMM from getting hung when the VM pause hypercall is retried after recovery.

As explained in Section 7.1, VMM recovery that leads to the failure of only a single AppVM is considered successful. However, it is clearly preferable for none of the AppVMs to fail. With all the improvements to ReHype up to this point, the rate of successful recovery is 88.1% of detected VMM failures. Out of these successful recoveries, approximately 28% result in a single AppVM failure. The majority of such cases are due to the failure of the benchmark in AppVM_Net which is primarily caused by blocked network interrupts at the I/O controller after recovery of the VMM. This prevents the benchmark from receiving additional packets from the sender host.

Unfortunately, the *acknowledge interrupts* mechanism discussed in Section 7.3 has shortcomings that prevent it from completely solving the above problem. In general, to

acknowledge an interrupt, the CPU must be currently servicing that interrupt. For level-triggered interrupts, the acknowledgment from the CPU gets propagated back to the I/O controller and only then can additional interrupts from the device be sent to the CPU. The *acknowledge interrupts* mechanism proposed in Section 7.3 only acknowledges interrupts that are currently being serviced. However, a VMM failure can occur after an I/O controller delivers an interrupt to the CPU but before the CPU begins servicing the interrupt. Upon recovery, the CPU state containing in-service and pending interrupts are reset, thus the CPU cannot acknowledge any interrupts that came in prior to recovery. This problem leaves level-triggered interrupts blocked at the I/O controller.

To resolve the above problem, the *acknowledge interrupts* mechanism needs to be enhanced to allow all pending interrupts, including ones that are not being serviced by the VMM at the point failure, to be acknowledged. This is accomplished by first installing a new interrupt descriptor table that contains dummy interrupt handlers for each CPU. These handlers simply acknowledge the interrupt and return. Interrupts are then blocked at the I/O controller so that no new interrupts will be delivered to the CPU and then interrupts are enabled on each CPU. Recovery can resume when there are no more pending interrupts (checked by reading the Interrupt Request Register on the CPU). With this enhancement, the majority of recoveries resulting in netbench failures caused by blocked interrupts are greatly reduced. As shown in Table 7.7, the rate of successful recovery out of detected VMM failures that result in no AppVM failures increases from 63.7% to 73.0%. The remaining single AppVM failures are caused by kernel panics from bad hypercall return values in AppVM_Unix.

The above enhancement can also help with improving the overall successful recovery

rate as it can potentially reduce the number of recovery failures associated with two AppVM failures where one of the AppVM failure is due to a blocked interrupt. Unfortunately, the number of cases where this scenario occurs in our experiment is small (˜2% of detected failures), hence, the overall successful recovery rate remains about the same at 89.1%.

Mechanisms	Successful Recovery Rate	Successful Recovery Rate (w/ no AppVM Failure)
Reset page counter	88.1%	63.7%
+ Ack interrupts (enhanced)	89.1%	73.0%

Table 7.7. Successful recovery rate with no AppVM failure when using the enhanced *Ack interrupts* mechanism. System configuration: 3AppVM with $FTVI_{HDP}$. Percentage of successful recoveries out of detected VMM failures (VMM crash/hang).

The need to implement the second enhancement mentioned above came about when evaluating the resiliency of the complete FTVI (see Chapter 8). The problem is that a VMM failure in the middle of handling a DVM failure can cause the recovered VMM to fail immediately after recovery when DVM failure handling is resumed. As explained below, this problem is due to inconsistency within parts of the VM state that is maintained by the VMM.

Our approach of recovering a failed DVM requires the PrivVM to first pause the failed DVM (see Chapter 4). In Xen, when the VM pause hypercall is invoked, the VMM first increments a pause counter kept for each VM before sending an IPI to stop all running VCPUs of the VM. The VMM then waits until all VCPUs have been descheduled by checking to see if the VCPU's ‘‘running’’ flag has been cleared. Normally, this flag is set when the VCPU is scheduled to run and cleared when the VCPU is descheduled. In addition, a VM that is paused (non-zero pause counter) cannot be scheduled to run. If a VMM failure occurs during the VM pause hypercall and the VM's VCPU has not been scheduled out (the

‘‘running’’ flag is not cleared), after recovery, the retry of the hypercall may cause the VMM to wait forever for this ‘‘running’’ flag to be cleared. This can happen if the VM’s pause counter has been incremented before VMM failure is detected but before the VM’s VCPU has been descheduled. The inconsistency is between the pause counter and the VCPU’s ‘‘running’’ flag. Hence, a straight forward approach to resolve this problem is to clear the running flag in all VCPUs after recovery before rescheduling the VCPUs. This enhancement allows for the resumption of DVM recovery after having recovered the VMM.

7.7. Analysis

This section analyzes fault injection results for the final version of ReHype, with all the recovery improvements and enhancements from Sections 7.3, 7.5, and 7.6. The following subsections discuss: I) the recovery success rates and causes of VMM recovery failures with respect to different fault types, II) the causes of silent VMM failures, and III) the impact of the distribution of injections across CPUs.

7.7.1. Recovery Effectiveness

Up to this point, we have been injecting into registers of CPUs while the CPU is executing VMM code in order to stress ReHype. These injections were meant to cause arbitrary corruption and trigger recovery allowing deficiencies in ReHype to be identified and fixed. In this subsection, we evaluate and analyze the effectiveness of ReHype at recovering from not only faults in CPU registers but also software faults and random single-bit VMM code corruption (see Section 7.4). We also classify the recovery failures and identify the main causes of such failures.

Fault Type	Successful Recovery Rate	Successful Recovery Rate (w/ no AppVM Failure)
Reg	89.1%	73.0%
SW	88.1%	67.7%
Code	88.6%	72.0%

Table 7.8. Recovery success rates of ReHype across different fault types: register faults (Reg), software faults (SW), and VMM code single-bit corruption (Code). Percentages of successful recoveries out of detected VMM failures (VMM crash/hang). Target system: 3AppVM running on FV VM.

We injected roughly 2800 register faults, 830 SW faults, and 650 Code faults, of which 950, 530, and 450 faults manifested as failures, respectively. Table 7.8 shows the recovery success rates of ReHype when deployed using the 3AppVM configuration (see Subsection 7.4.1) with respect to different types of faults. To evaluate the statistical significance of these results, for each fault type, we divide the total number of injections into three equal groups and calculated the mean of each measure and margin of error at 95% confidence interval. The results of the confidence test for the rate of successful recovery and successful recovery with no AppVM failure are the following: Reg: 89% ± 4, 73% ± 4; SW: 88% ± 6, 68% ± 8; and Code: 89% ± 4, 72% ± 8. These results indicate that the effectiveness of ReHype is similar across the different fault types and the variations shown in Table 7.8 are not significant.

Overall, ReHype failed to recover the VMM in about 12% of detected failures. Based on the discussion in Section 7.1, failed VMM recoveries can be roughly classified into four categories: (i) unsuccessful VMM reboot, (ii) successful VMM reboot but all AppVMs fail to complete their benchmarks successfully, (iii) successful VMM reboot and benchmark in AppVM_Blk completes successfully but benchmarks in both AppVM_Net and AppVM_Unix complete unsuccessfully, and (iv) successful VMM reboot and successful completion of

benchmarks in AppVM_Net and/or AppVM_Unix but benchmark in AppVM_Blk completes unsuccessfully. Considering faults in CPU registers, about 65% of failed recoveries belong in category (i) and the remaining failed recoveries are about evenly split among categories (ii)-(iv). For SW, about 34% belong in category (i), 29% belong in category (ii), 24% belong in category (iii), and 12% belong in category (iv). For Code, about 55% belong in category (i) and the remaining failed recoveries are about evenly split among categories (ii)-(iv).

The majority of recovery failures in category (i) across three fault types are caused by triple fault exceptions generated during the execution of the VMM failure handler triggering a hardware system reset. A triple fault exception is generated if an exception is triggered while trying to invoke the double fault handler. A double fault exception is generated if an exception is triggered while trying to invoke an exception handler. The inability to invoke an exception handler is due to the corruption of the interrupt descriptor table or memory address mapping data structures such as page tables or the global descriptor table caused by the injected fault. In our experiments, such corruption occur most frequently when the injected fault affects the stack pointer register. In fact, nearly half of failures in category (i) for register injection is due to stack pointer corruption leading to triple fault exceptions. The higher probability of corrupting critical system state when injecting into registers such as the stack pointer may explain why register injection have a significantly higher chance of causing VMM reboot failure (category (i)) than Code and particularly SW injection.

A small fraction of recovery failures belonging to category (i) may be an artifact of our fault injection setup. Specifically, with register injection, in less than 5% of failures, the host VMM crashes the FV VM running the target system. This scenario occurred once for SW injection but none for code injection. This type of failure can happen if the host VMM

attempts to access invalid state in the FV VM while performing some operations on its behalf. For example, as part of emulating a hardware task switch for a FV VM using PAE and with the host system using hardware-assisted paging, the host VMM will map in the page pointed to by the FV VM's CR3 register. If the mapping fails (no valid page) due to a corrupted CR3, the host VMM will crash the FV VM. With ReHype running directly on hardware, such a scenario would likely result in a page fault and a detected VMM failure, allowing recovery to be attempted. Other recovery failures in category (i) are caused by various VMM corruptions and inconsistencies including: (1) corruption of VM's VCPU registers causing the new VMM to crash after recovery when restoring the VCPU, (2) corruption of the timer heap which leads to a page fault in the VMM when the old timer heap is walked to restore timer events, and (3) page table corruption that causes the new VMM to page fault early in the boot code.

Mechanisms to prevent memory corruption [Tan12] or to check and ensure consistency such as data structure replication as discussed in Subsection 7.1.2 may be helpful in mitigating the above problems.

Recovery failures in categories (ii)-(iv) are caused by several independent problems that manifest from a single injected fault. These problems include: (a) VM kernel panic due to error return values from hypercalls and/or VM state corruption, (b) VM I/O requests/replies and timer events not handled due to the lost of virtual interrupts sent by the VMM, and (c) VMM access corrupted state after recovery and repeatedly fails. In some cases, a single problem affects multiple VMs and in other cases different problems affect different VMs. In all cases, more than one AppVM fails to execute its benchmark successfully and/or AppVM_Blk cannot execute its benchmark successfully which in most cases is because the VMM can no longer create AppVM_Blk. An example of how a hypercall can fail after

recovery is provided below as this type of problem causes the majority of failures in categories (ii)-(iv).

While ReHype has the ability to retry hypercalls that were interrupted due to the VMM failure upon recovery (see Section 7.3), some hypercalls are not idempotent and cannot be re-executed. Retries of these hypercalls will result in further state corruption and early hypercall termination with error values returned to the VM. For example, retries of the hypercall to unmap a grant reference can fail depending on how much of the hypercall has been executed before VMM failure was detected. This hypercall is used by a VM to unmap a shared page with another VM and update the grant table structure in the VMM indicating the grant reference is no longer being used (page mapped in) by that VM. If the VMM fails after the hypercall has removed the mapping from the caller VM's page table, the retry of the hypercall will fail as the hypercall will expect the page to still be mapped. Such failure prevents the grant table from being properly updated, hence, the VM that owns the page will fail (panic) when it tries to reuse the page but discovers that the page is still being ''used'' by the other VM.

Preventing the problems that cause recovery failures in categories (ii)-(iv) may require maintaining redundant information during normal operations to transactionalize certain hypercalls and re-deliver pending virtual interrupts or preventing critical memory areas from being written. Often time, the VM that fails due to these problems is the PrivVM and/or the DVM which affects the correct operation of multiple AppVMs. The impact of a failed PrivVM or DVM can be partly resolved by providing mechanisms to recover a DVM or PrivVM from failure (see Chapters 4 and 6, respectively).

7.7.2. Silent Failures

Fault Type	Silent 1 AppVM Failure	Silent System Failure
Reg	8.1%	10.9%
SW	0.6%	35.0%
Code	0.9%	22.5%

Table 7.9. Percentages of manifested faults that result in silent failures of a single AppVM and silent failures of the entire system. Target system: 3AppVM running on FV VM.

VMM corruptions can lead to failures that are not detectable by simple crash and hang detectors. These silent failures can manifest as failures of one or more VMs and/or failure of the VMM to host additional VMs. As discussed in Section 7.1, the reliability goal of ReHype is met if no more than one AppVM fails due to a fault and the VMM can still host existing VMs and create new VMs. In the case of a silent single AppVM failure, the reliability goal is met. However, silent system failures, which are silent VMM failures that result in more than one VM failure and/or the failure of the VMM to host additional VMs, reduce the reliability of the virtualized system. Hence, the rest of this subsection discusses the causes of these failures.

Table 7.9 shows the percentages of silent VMM failures out of manifested faults for the different fault types. The results of the confidence test for silent single AppVM failure and silent system failure are the following: Reg: 8% ± 6, 11% ± 2; SW: 0.4% ± 2, 35% ± 10; Code: 1% ± 2, 23% ± 4. In our experiments, silent VMM failures range roughly between 20% and 35% of all manifested faults with register injection causing the least amount of silent failures and SW injection causing the most. Approximately 1%-8% cause a benchmark

(netbench or unixbench) in a single AppVM to complete incorrectly. This is mostly caused by failed hypercalls causing VM kernel panics or a blocked/lost virtual interrupt as discussed above. The remaining silent VMM failures cause system failures.

Similar to failed recoveries, silent system failures can be grouped into four main categories: (i) failure of the entire system (i.e., hardware reset). (ii) all AppVMs fail to complete their benchmarks successfully, (iii) benchmark in AppVM_Blk completes successfully but benchmarks in both AppVM_Net and AppVM_Unix complete unsuccessfully, and (iv) benchmarks in AppVM_Net and/or AppVM_Unix complete successfully but benchmark in AppVM_Blk completes unsuccessfully. Considering faults in CPU registers, roughly 63% of silent system failures are in category (i), 7% of silent system failures are in category (ii), 0% of silent system failures are in category (iii), and 30% of silent system failures are in category (iv). For SW and code injections, roughly 10% and 25% are in category (i), 41% and 43% are in category (ii), 2% and 2% are in category (iii), 47% and 30% are in category (iv), respectively.

Compared to register injection, the rate of silent system failures that belong to categories (ii)-(iv) is much higher for SW and Code injection. This means that the vast majority of silent system failures when injecting SW or Code faults are caused by the inability to create a new AppVM or host the two existing AppVMs. The main reason for this phenomenon is in the nature of how register, SW, and Code injections are performed. Unlike faults in registers, SW faults and Code faults are permanent (remains until the system is rebooted). Since VMM code is shared among all CPUs, a fault in the VMM code that causes one VM to fail will most likely cause another VM executing on the same CPU or another CPU to fail when accessing the same faulty VMM code. For instance, a fault in a routine use for updating page table

entries of a VM will likely cause any VMs making use of hypercalls that invokes the same faulty routine to fail. In contrast, with register injection, in most cases, the fault is confined to the target CPU and the fault is masked when the target register is overwritten. Hence, as each of our VM is pinned to a single CPU, register faults in the VMM that cause VM failures tend to only affect the lone VM running on that CPU. This has the effect of increasing the chance of causing a single AppVM failure, as shown in Table 7.9.

The main causes of silent system failures for each category are similar to what have already been discussed for recovery failures above. Specifically, for category (i), the main cause of failures are triple faults that are generated before VMM failures are ever detected. For category (ii)-(iv), there are two main causes of silent system failures: (1) error return values from hypercall causing the kernel in the PrivVM, DVM, and/or AppVMs to panic and (2) blocked/lost virtual interrupts sent from the VMM or another VM via the VMM. Reducing silent failures will require more sophisticated detection mechanisms that make use of redundant data or computation than the simple crash and hang detectors we currently employ.

7.7.3. Impact of Injection Distribution

With respect to hardware faults, it is expected that faults in a CPU that rarely executes VMM code are less likely to lead to VMM failures than faults in a CPU that spends a larger fraction of time executing VMM code. Similarly, assuming software faults are roughly distributed evenly across code of the VMM, software faults in the VMM are more likely to manifest and cause VMM failures on a CPU that executes VMM code a larger fraction of

Fault Type	Manifested	
	Detected	Silent Failure
	Successful Recovery Rate	
Reg	89.1% (94.3%)	19.0% (25.0%)
SW	88.1% (91.1%)	35.6% (35.0%)
Code	88.6% (87.7%)	23.4% (22.8%)

Table 7.10. Fault injection results with injections distributed across CPUs uniformly or weighted (in parenthesis) by VMM execution time. Percentage of successful recoveries are out of detected VMM failures. Percentage of silent failures are out of manifested faults. Target system: 3AppVM running on FV VM.

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

We use the Xenoprof profiler [Meno05] to measure the distribution of Xen execution time across CPUs when running the 3AppVM setup. As explained in Section 7.4, each VM is pinned to a CPU. The results are as follows: 4.9% PrivVM CPU, 2.1% DVM CPU, 2.1% AppVM_Net (NetBench) CPU, and 90.8% AppVM_Unix (UnixBench) CPU. The AppVM_Blk CPU is not included in the profile because it is created after the injection.

Table 7.10 shows the injection results using the uniform distribution across CPUs and when injection distribution is weighted VMM execution time. The results are very similar. This is mainly due to the fact that under weighted distribution of injections, most of the injections are applied to a CPU whose behavior with respect to injection closely matches the behavior of the overall system under uniform distribution of injections. For example, with

code injection under uniform distribution of injections, 11.4% of detected VMM failures result in unsuccessful recoveries. Considering only injections into AppVM_Unix, the corresponding number is very close — 12.4%. Since fault injection results are not very sensitive to the two injection distributions, the rest of this chapter will continue to use the uniform distribution for selecting the target CPUs.

7.8. Validating ReHype on Bare Hardware

Thus far, the system under evaluation (target system) has been run in a virtualized environment (FV VM). As discussed in Section 7.4, using a FV VM to host the target system simplifies and speeds up the fault injection campaigns. Since the deployment of a production virtualized system will be on bare hardware, having only results showing the effectiveness of ReHype running in a FV VM may not be representative. In this section, to better gauge the extent to which our evaluation of ReHype presented in the previous section is valid, we compare those results to results obtained when ReHype is deployed on a bare machine.

For the most part, deploying ReHype on bare hardware required almost no additional modifications to the VMM. We had to make only two minor changes in the VMM to prevent the VMM from hanging during recovery. Specifically, on a microreboot with our bare hardware setup, the VMM will hang if it tries to perform low-level BIOS accesses to gather information about the hard disks using the Enhanced Disk Drive Services and information about the display through accessing the Extended Display Identification Data. These problems did not occur when the VMM was deployed in a FV VM. Normally, the VMM accesses this information using the BIOS upon system bootup. As part of the support for para-virtualization, the VMM would store and later provide this information to any VM that

may need it during the VM's own bootup process. While we do not know the exact cause of these problems, we suspect possible BIOS bugs that prevent a machine that has not been ''normally'' (fully) rebooted to access such functionality. Part of our modifications to the VMM is to skip such BIOS operations during a reboot. This is not a major problem as such information can be restored after the VMM has been rebooted. The functionality to skip the BIOS probing of these devices is already provided as a command line argument to the Xen VMM and even to the Linux kernel in order to overcome buggy BIOS implementations.

Fault Type	Successful Recovery Rate	Successful Recovery Rate (w/ no AppVM Failure)
Reg	90.9% (89.1%)	69.2% (73.0%)
SW	87.6% (88.1%)	72.1% (67.7%)
Code	91.0% (88.6%)	73.0% (72.0%)

Table 7.11. Recovery success rates of ReHype across different fault types. Percentages of successful recoveries out of detected VMM failures (VMM crash/hang). Values in parenthesis are results from running target system inside a FV VM (copied from Table 7.8). Target system: 3AppVM running on bare hardware.

Fault Type	Silent 1 AppVM Failure	Silent System Failure
Reg	5.8% (8.1%)	12.6% (10.9%)
SW	1.2% (0.6%)	36.0% (35.0%)
Code	1.3% (0.9%)	22.3% (22.5%)

Table 7.12. Percentages of manifested faults that result in silent failures. Values in parenthesis are results from running target system inside a FV VM (copied from Table 7.9). Target system: 3AppVM running on bare hardware.

With the target system (3AppVM configuration) deployed on bare hardware, we performed the same fault injection experiments that were used for evaluating ReHype in a

virtualized environment. We injected roughly 990 register faults, 410 SW faults, and 320 Code faults, of which 300, 250, and 230 faults manifested as failures, respectively.

Successful recovery rates from our bare machine experiments are shown in Table 7.11 and the percentages of silent VMM failures are shown in Table 7.12. Results from previous experiments where the target system is run in a FV VM are re-shown in parenthesis. The results of the confidence test (discussed in Subsection 7.7.1) for the rate of successful recovery and successful recovery with no AppVM failure are the following: Reg: 91% ± 7, 69% ± 14; SW: 88% ± 2, 72% ± 7; Code: 91% ± 3, 73% ± 15. The results of the confidence test for silent single AppVM failure and silent system failure are the following: Reg: 6% ± 9, 13% ± 8; SW: 1% ± 3, 36% ± 2; Code: 1% ± 3, 23% ± 7. These results suggest that the behavior of the two setups are very similarly to each other. In addition, the distributions of failed recoveries and silent system failures for the two setups have very similar characteristics although with large variations between the distributions as the number of recovery and silent failures are not large enough to warrant a meaningful comparison. For example, for the bare hardware setup when considering faults in CPU registers, the vast majority of failed recoveries (87%) belong to category (i) (described in Subsection 7.7.1). This approximates the results from the FV VM setup which has about 65% of recovery failures belonging to the same category.

Due to the limitation of logging failure outputs on a bare machine, determining the similarities between the two setups with regards to the causes of failed recoveries or silent system failures is difficult. In particular, on bare hardware, unlike on a virtualized system, it is not possible to directly determine if the recovery failure or silent system failure is caused by a triple fault. Given the more limited abilities to log failure information on the bare hardware

setup, from what we can observe, there is strong indication that the causes of failed recoveries and silent system failures for each fault type is very similar to what was described in Section 7.7. The results in this section, for the most part, validate our results presented in the previous section with ReHype deployed in a virtualized environment.

7.9. Validating ReHype with FV VMs

Unlike running in PV VMs, guest software running in FV VMs do not have to be modified. This makes FV VMs more attractive for hosting applications (AppVMs). However, up until now, the evaluation of ReHype has been done on systems running only PV VMs. As the VMM must perform different tasks when hosting FV VMs than when hosting PV VMs, a thorough evaluation of ReHype must include the configuration in which the VMM is hosting FV VMs. In this section, we show that ReHype, with the mechanisms discussed thus far and without any major enhancements, can be as effective at recovering the VMM hosting FV VMs as when the VMM is hosting PV VMs. As will be further discussed below, the main reason for this is that while there are different tasks that the VMM must perform for FV VMs, many of the tasks that the VMM must perform on behalf of FV VMs are very similar to what must be done for PV VMs, especially when the system supports hardware assisted paging [AMD08] and the FV VMs are configured to use para-virtualized devices.

To host FV VMs, the VMM must set up the processor to make use of hardware virtualization capabilities and provide handlers to handle world switches between VM and the VMM. In addition, the VMM must typically provide mechanisms to emulate I/O operations, emulate page table operations (e.g., shadow page table when there is no hardware-assisted paging support), and emulate privileged instructions used for querying or manipulating system

state. All of these operations require state to be maintained in the VMM for the VMs which must be saved/restored and possibly checked for consistency across a VMM recovery. Reducing the involvement of the VMM during normal operations of the FV VM can reduce the probability of inconsistency after recovery and the amount of additional VMM state that needs to be saved/restored.

Fortunately, the mechanisms for saving and restoring a FV VM CPU state is mostly handled by the hardware virtualization mechanism in the processor. Hence, to save/restore the CPU state of FV VMs across a VMM failure does not require any additional mechanisms beyond what is already employed for PV VMs. In addition, by utilizing para-virtualized drivers and hardware assisted paging, the level of involvement of the VMM on normal FV VM operation can be greatly reduced. While utilizing para-virtualized drivers reduces portability, the performance benefits obtained from more efficient I/O often outweigh the drawbacks [Koh09] Furthermore, with support for hardware-assisted paging in modern processors, the need to shadow VM's page table updates is no longer needed during normal operations. Hence, the only additional remaining work that the VMM must perform on behalf of a FV VM is emulating special instructions such as 8086 instructions in real-mode. However, in terms of VMM recovery, typically the instruction being emulated at the point of failure can simply be re-executed upon VMM recovery. With this setup, recovering the VMM that is hosting FV VMs is no different than if the VMM is hosting PV VMs.

We evaluate ReHype with the VMM hosting FV AppVMs using the slightly modified 3AppVM configuration described in Section 7.4.1. The same fault injection experiments used for evaluating ReHype in the previous sections are used here. We injected roughly 890 register faults, 400 SW faults, and 360 Code faults, of which 290, 280, and 280 faults

manifested as failures, respectively.

Successful recovery rates are shown in Table 7.13 and the percentages of silent VMM failures are shown in Table 7.14. Results from previous experiments where the target system is run on bare hardware with only PV VMs are re-shown in parenthesis. The results of the confidence test (discussed in Subsection 7.7.1) for the rate of successful recovery and successful recovery with no AppVM failure are the following: Reg: 88% ± 6, 85% ± 2; SW: 83% ± 6, 78% ± 12; Code: 82% ± 4, 80% ± 5. The results of the confidence test for silent single AppVM failure and silent system failure are the following: Reg: 8% ± 4, 14% ± 10; SW: 2% ± 3, 28% ± 10; Code: 4% ± 3, 24% ± 10.

Fault Type	Successful Recovery Rate	Successful Recovery Rate (w/ no AppVM Failure)
Reg	88.2% (90.9%)	85.5% (69.2%)
SW	82.9% (87.6%)	78.4% (72.1%)
Code	82.5% (91.0%)	79.6% (73.0%)

Table 7.13. Recovery success rates of ReHype recovering VMM hosting FV AppVMs across different fault types. Values in parenthesis are success rates of ReHype at recovering VMM hosting PV AppVMs running on bare hardware (copied from Table 7.11). Percentage of successful recoveries are out of detected VMM failures. Target system: 3AppVM (FV) running on bare hardware.

For register injections, the results suggest that the behavior of ReHype hosting FV AppVMs is for the most part similar to when ReHype is hosting PV AppVMs. In fact, due to lesser amount of hypercall handling performed by the VMM on behalf of FV AppVMs in our setup, there is lesser chance a VMM failure can interrupt a hypercall and cause inconsistency. As a direct consequence, there is a significant increase in the percentage of successful recoveries with no AppVM failure for the FV AppVM setup (85.5% vs. 69.2%).

Fault Type	Silent 1 AppVM Failure	Silent System Failure
Reg	7.9% (5.8%)	14.0% (12.6%)
SW	2.1% (1.2%)	28.3% (36.0%)
Code	3.5% (1.3%)	23.9% (22.3%)

Table 7.14. Percentages of manifested faults that result in silent failures. Values in parenthesis are silent failure rates of ReHype at recovering VMM hosting PV AppVMs running on bare hardware (copied from Table 7.12). Target system: 3AppVM (FV) running on bare hardware.

SW and Code injections have markedly lower success rates when the VMM is hosting FV AppVMs than when the VMM is hosting PV AppVMs. This discrepancy is mainly caused by differences in the set of VMM instructions being targeted between the two configurations. As explained in subsection 7.4.4, the instruction targets are chosen based on profiling the VMM and selecting the instructions that belong to functions most frequently executed. Since there are different activities in the VMM when hosting FV and PV VMs, the two configurations yield different VMM execution profiles, and thus, different set of instruction targets. When considering the same set of instruction targets that belong to functions executed by both configurations, the results are more similar between the two configurations (FV AppVMs vs. PV AppVMs): 82.1% vs. 84.1% for SW and 92.8% vs. 91.2% for Code.

With respect to silent failures, except for silent system failures for SW faults, the differences shown in the Table 7.14 are for the most part not significant. The difference between silent system failures for SW faults may have to do with the reduction in hypercalls that must be handled by the VMM as described above.

7.10. Recovery Latency

As part of recovery, Rehype pauses all running VMs on the system. The VMs are unpaused once the VMM has finished being fully rebooted. This interruption can cause failures to applications running in the AppVMs or failures to applications relying on services provided by the applications running in the AppVMs. Hence, the amount of time it takes to fully recover the system from a VMM failure is an important measure to understand and minimize. To measure recovery latency, we use the 3AppVM configuration described in Subsection 7.4.1}. In this experiment, we invoke the VMM crash handler to cause an immediate recovery of the VMM and measure the time it takes for the netbench program running on the PM host to resume receiving/sending packets from/to the netbench program running on the VM host. We analyze the main components of this time and discuss ways to optimize the recovery latency without requiring major modifications.

Operations	Time (no opt.)	Time (with opt.)
CPU initialization: - Initialize and wait for all CPUs to come online	150ms	150ms
Timer/hardware initialization: - Initialize/calibrate platform timer, TSC, I/O APIC, NMI watchdog, etc.	410ms	310ms
Memory initialization: - Record page number of all allocated pages in old heap (Use to preserve content of old heap) - Restore and check consistency of page frame entries - Create and allocate free pages to VMM's heap - Scrub unallocated pages	3210ms 20ms 30ms 200ms 2080ms	250ms 20ms 30ms 200ms 0ms
Other	5ms	5ms
Total	2895ms	715ms

Table 7.15. Breakdown of recovery latency using ReHype with and without optimization.

During normal operations, the ping inter-arrival time of netbench is roughly 1.1ms. When there is VMM recovery, the inter-arrival time at the point of failure is about 2895ms. The bulk of the recovery time is spent by the VMM initializing hardware devices and creating data structures associated with the management of CPUs, memory, platform timers, and interrupt controllers. Table 7.15 shows the recovery time subdivided into major operations performed by the VMM during recovery. The right most column contains time from an optimized VMM bootup sequence that will be further discussed below. As can be seen from this table, the majority of the recovery time is spent performing memory initialization operations with the bulk of that time spent primarily scrubbing (zeroing) all unallocated pages. Scrubbing unallocated memory pages is a security measure that prevents the leaking of data between VMs.

One way to reduce the recovery time is to identify operations that can be safely skipped on a VMM reboot. As will be further discussed below, these operations include the scrubbing of unallocated pages and initialization of the NMI watchdog. The reduced recovery time when these operations are skipped is shown on the right most column of Table 7.15.

Given 7GB of free space on the heap, scrubbing every unallocated memory pages takes about 2080ms to complete. However, since we are microrebooting the VMM, pages are either already scrubbed by the failed VMM or are on a list of pages to be scrubbed. Hence, this step can be skipped entirely during a microreboot. Of course this assumes that the old VMM was still correctly performing memory scrubbing up until VMM failure was detected.

A second optimization that we performed is to remove the delay used for checking the correctness of the NMI watchdog mechanism. Part of this check involves counting the number of NMI interrupts received by the VMM within an interval of 100ms. While this

check can catch cases in which the NMI watchdog fails to work properly after a microreboot, it is our experience that such failure is extremely rare (we have not observed the problem in our experiments). Hence, in the interest of reducing recovery latency, we skip the NMI watchdog check when performing a recovery.

A different approach to optimize recovery time is to modify the VMM boot code to parallelize some of the initialization operations. For example, while waiting for CPUs to come online, entries in the old page frame table can be checked and restored. While this approach can further reduce the recovery time when combined with the approach above, it will also require time consuming engineering effort to refactor the VMM boot code.

With the above improvements, the recovery latency is greatly reduced from 2895ms to 715ms. Approximately 50ms of this 715ms is directly associated with the operations required by ReHype for identifying free pages in the system by walking the old heap and performing consistency check on each page frame structure as discussed in the mechanism *Reset page counter* presented in Section 7.3.

7.11. Summary

The VMM plays a critical role in hosting and isolating AppVMs running in a virtualized system. Failure of the VMM can cause all AppVMs running on the system to fail. In this chapter, we have presented ReHype, a microreboot-based mechanism to tolerate the failure of the VMM while preserving the state of running VMs.

The basic version of ReHype recovered successfully from only 5.6% of detected hypervisor failures. We used fault injection results to guide incremental improvements of

ReHype, leading to a success rate of over 88%. The incremental improvements involved a combination of mechanisms to repair VMM corruption and resolve inconsistencies within the VMM, between the VMM and VMs, and between the VMM and the hardware.

Our evaluation of the final version of ReHype consisted of injecting transient hardware faults into CPU registers while it is executing the VMM as well as faults emulating typical programming errors into VMM code and random single-bit VMM code corruption. We showed that with respect to the three different fault types, recovery failures account for roughly 12% of detected VMM faults. The main cause of recovery failures when faults are injected into CPU registers is due to corruption of critical data structures in the VMM leading to triple faults. Software faults and random VMM code corruption on the other hand cause a significant amount of kernel panics in multiple VMs due to failed hypercall retries after VMM recovery. A significant fraction of manifested (15%-36%) faults result in silent system failures which will require more sophisticated detection mechanisms to prevent.

We validated our main results which used a FV VM to host the target system with results obtained when running the target system directly on bare hardware. In addition, we showed that ReHype, with almost no additional modifications, can be used to effectively recover the VMM hosting FV AppVMs. Furthermore, we discussed the factors that affect the recovery latency using ReHype and showed that sub-second (715ms) recovery latency can be achieved with minimal effort.

Chapter Eight

Resilience to Virtualization Infrastructure Failures

This chapter brings together all the recovery mechanisms that make up our resilient or fault tolerant VI (FTVI) and evaluate how well they work together to prevent the loss of services provided by the AppVMs despite failure of a single VI component [Le12].

The overall framework of our FTVI is presented in Section 8.1. The experimental setup used to evaluate the entire FTVI is presented in Section 8.2. Section 8.3 analyzes the fault injection results showing the effectiveness of our recovery mechanisms. Section 8.4 discusses the implementation complexity of the different recovery schemes.

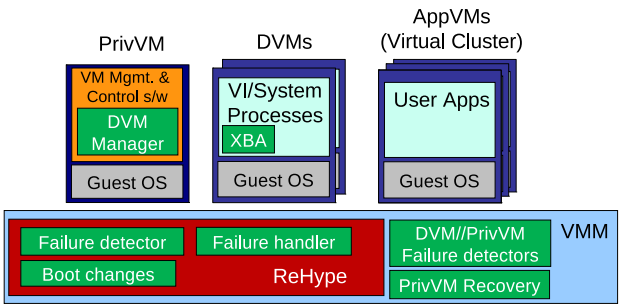


Figure 8.1: Overview of the FTVI showing the main components providing resiliency.

8.1. FTVI Framework

Figure 8.1 shows the main components of our FTVI and a virtual cluster consisting of multiple AppVMs. The VI resiliency enhancements include ReHype for detecting and recovering from VMM failure, two DVMs to enable uninterrupted access to devices for the

AppVMs (see below), a DVM Manager for controlling recovery from DVM failure, DVM failure detectors for detecting and informing AppVMs and the DVM Manager of the DVM failure, and PrivVM failure detectors along with the *Xen Backup Agent* (XBA) and mechanisms in the VMM to microreboot and restore the state of a failed PrivVM.

If the DVM is microrebooted and hardware devices are reset, the duration of the interruption may be on the order of seconds (see Chapter 4). Such long interruptions can result in the failure of the workload running in the AppVMs. Therefore, unlike other VI components, for the evaluation in this chapter, we do not rely on microreboot to recover from DVM failure. Instead, recovery from a DVM failure involves failing over to a redundant DVM with access to separate hardware devices. However, microreboot must still be used to replace the failed DVM so that the fault tolerance capabilities of the system are restored.

8.2. Experimental Setup

To evaluate the resiliency of the FTVI as a whole, faults are injected into CPU registers during the execution of each of the VI components. The target system includes a cluster of 5 AppVMs running a highly reliable web service application. Details of the experimental setup are provided below.

We conducted three fault injection campaigns which differ in when faults are injected: 1) during execution of the VMM, 2) during execution of the DVM (user and kernel-level), and 3) during execution of the PrivVM (user and kernel-level). An injection is triggered after a random time period between 500ms to 6.5s after the AppVMs begin running their benchmarks. To ensure the injection occurs during the execution of the VMM/DVM/PrivVM, a fault is only injected after the designated time has elapsed and 0 to 20,000 VMM, DVM

(user and kernel-level), or PrivVM (user and kernel-level) instructions, depending on the campaign, chosen at random, have been executed. While these injections do not accurately represent all possible faults, they are a good choice since transient hardware faults in CPU logic and memory are likely to be manifested as erroneous values in registers. In addition, based on our results from evaluating ReHype (see Chapter 7), faults into registers can be equally stressful on the recovery mechanisms as software faults and random code corruption.

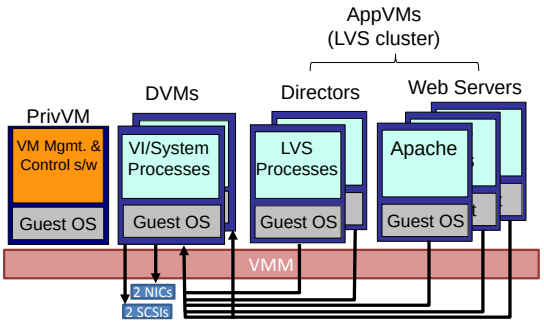


Figure 8.2: Target system configuration for evaluating the resiliency mechanisms of the FTVI. AppVMs access I/O devices through DVMs. Each AppVM has access to two disks, hosted on separate DVMs, which the AppVM uses to form a single RAID level-1 block device. Each AppVM has access to two NICs, with only one NIC active at a time.

Figure 8.2 shows the target system configuration for our experiments in this chapter. The setup consists of a PrivVM, two DVMs, and five AppVMs. In this setup, the PrivVM's root filesystem is in memory and the PrivVM does not access any devices. The AppVMs access their I/O devices through the DVMs. For seamless operation despite failure of a DVM, the AppVM's root disk is configured as a RAID level-1 device with each disk of the RAID hosted on a separate DVM. The AppVM's network device is accessed through a single DVM with the other DVM acting as a hot-spare (*MultiNetIf* scheme, see Subsection 4.3.4). In these experiments, the entire target system runs directly on bare hardware. The five AppVMs

together run a reliable web service workload described below.

The workload used to stress the VI is a deployment of the Linux Virtual Server (LVS [Zhan03, Linu]) on a cluster of VMs (virtual cluster) providing highly-reliable web services. LVS is an open-source load-balancing solution for building highly-scalable and highly-available servers using clusters of servers. It is widely deployed in industry for online services such as web and FTP servers. (Note that the term ‘‘virtual server’’ here is unrelated to system virtualization technology). We use the LVS/NAT configuration shown in Figure 8.3 to provide highly available web service. Clients connect to the virtual server by using a virtual IP address that is owned by the LVS director node. The director forwards clients' requests to one of the real servers in the server cluster. The Apache web server is run on each real server.

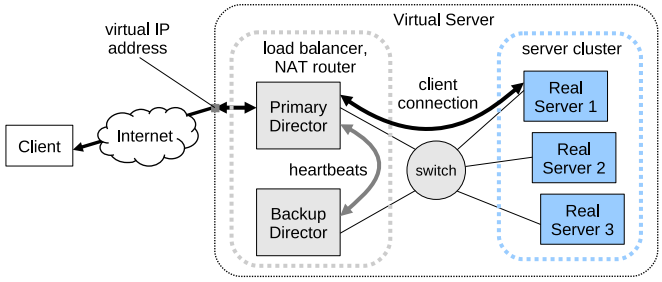


Figure 8.3: Structure of Linux Virtual Server with primary and backup directors and three real servers.

Since the director node can be a single point of failure, LVS configuration typically involves the use of primary and backup directors. We use the open-source *keepalived* project to deploy a primary and one backup director. Periodic heartbeats between the primary and backup directors are used to detect when either director has failed. Upon failure of the primary director, the backup director assumes the role of the primary director. The LVS

director also periodically checks for the liveness of the real servers by forming a TCP connection with each server. Failure to form a TCP connection causes the server to be removed from the pool of servers for handling requests. New or recovered (rebooted) real servers are automatically added to the server pool.

Our virtual server cluster is configured for two directors and three real Apache web servers. The client runs on a remote node outside of the virtualized system. To stress the virtual cluster, the client executes five instances of the Apache *ab* benchmark. Two *ab* instances each sequentially send 3770 HTTPS requests for a static web page and three *ab* instances each sequentially send 365 HTTPS requests for a dynamically generated web page. The Apache web server executes the *blkbench* program to generate a response for each request of a dynamically generated web page.

An injection run begins by booting the VMM, PrivVM, two DVMs, and five AppVMs. After the remote clients begin to generate requests, a single fault is injected into one of the VI components. Mechanisms are added to restart a node (AppVM) the director deemed as failed. If no nodes are restarted during a run, one AppVM is randomly selected to be destroyed and recreated about 50 seconds after a fault is injected to ensure, that if recovery had occurred, the VI is still correctly providing basic functionality.

The LVS cluster is considered to have failed if it is unable to service web requests. However, LVS director and server failover involves terminating client connections. Any time a director or real server fails, existing client connections through the director or with the real server are terminated. Furthermore, in the case that a new director becomes the new primary director, all connections must be terminated and re-formed using the new primary director. Hence, given a fault, regardless if there is a VMM/PrivVM/DVM error detected or not, an

unsuccessful application completion is recorded if a client experiences more than two connection timeouts during a single injection run.

Ultimately, the critical question is whether the system (the entire virtual cluster) continues to operate correctly. If a VMM, DVM, or PrivVM failure is detected, recovery is considered *successful*, no system failure, if (1) the LVS workload completes successfully and (2) the recovery does not lead to the failure of more than one AppVM and the recovered VI is able to continue hosting the remaining VMs as well as create and host new VMs. If the fault does not manifest as a detected VI component failure and one of the two *successful* recovery conditions above is not true, then a ‘‘silent system failure’’ has occurred.

8.3. Effectiveness Results

In total, we injected about 1980 faults during the execution of the VMM, 990 faults during the execution of the DVM, and 940 faults during the execution of the PrivVM. Approximately one third of these faults manifest as failures. These numbers of injections were chosen as they represent the best compromise between stability of the results and time needed to run the campaigns. With these numbers of injections, the recovery success rates obtained when the number of injections are divided in two are within 2% of the success rates reported in this section.

Recovery using microreboot is effective only if single faults do not manifest as errors in multiple components of the system. Based on our analysis of ReHype discussed in Chapter 7, we already know that faults in the VMM can lead to the failure of one or more VMs on the system. To better understand the extent to which faults in the VMM as well as other VI components affect the rest of the system, and ultimately, the ability to successfully perform

recovery, we analyze the likelihood that a fault in one VI component can cause failures in other components of the system. Figures 8.4-8.6 show the distributions of component failures caused by faults injected during the execution of each VI component along with the distributions of successful recoveries from those failures. Fortunately, from these results, it is clear that the vast majority of component failures are confined to the component into which the faults have been injected. In parts due to the high degree of fault isolation exhibited by the Xen VI, a great majority of component failures were successfully tolerated by a combination of our resiliency mechanisms and those of the LVS workload.

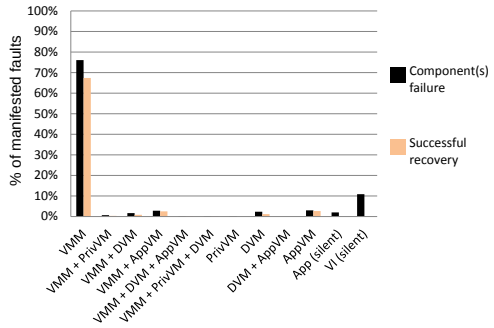


Figure 8.4: Distributions of component failures and successful recovery from those failures when injecting faults into CPU registers during VMM execution.

While uncommon, a single fault in a VI component can cause other components to fail either together with the faulty component or independently. For example, when faults are injected during VMM execution (Figure 8.4), about 4% of detected VI failures cause an AppVM to fail together with the VMM. Not surprisingly, given the privileged nature of the VMM, faults in the VMM cause the highest rate of multiple component failures compared to faults in other VI components. Specifically, close to 9% of detected VI failures caused by faults during the execution of the VMM result in the failure of multiple components. Faults

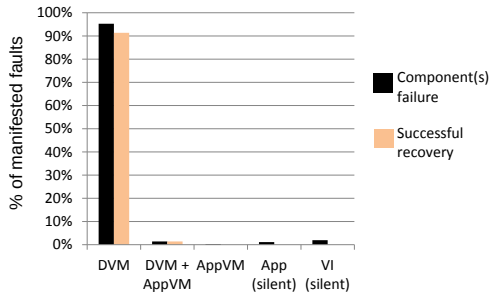


Figure 8.5: Distributions of component failures and successful recovery from those failures when injecting faults into CPU registers during DVM execution.

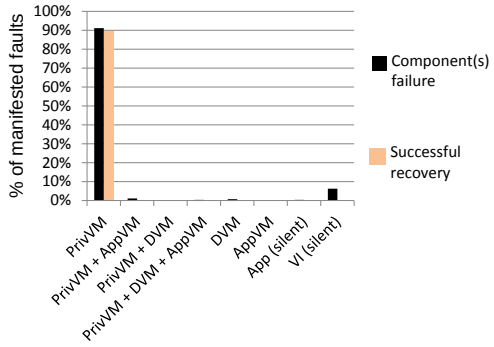


Figure 8.6: Distributions of component failures and successful recovery from those failures when injecting faults into CPU registers during PrivVM execution.

occurring during the execution of the DVM and PrivVM cause failures in multiple components in less than 2% of detected VI failures. Not all failure of multiple components are due to the propagation of errors across components. Some of these failures can be a result of an incomplete recovery leaving the system in an inconsistent state, leading to the failure of other components.

When there is multiple component failures, successful recovery of the system requires recovery to be performed successively on each failed component. In the cases when a VMM failure lead to the failure of a DVM and/or PrivVM (6% of detected VI failures), about half

were successfully recovered from by successively invoking the recovery mechanisms of the DVM and/or PrivVM after having recovered the VMM. The low recovery success rate is most likely due to the incomplete recovery of a component leaving the system in a partially corrupted or inconsistent state, thus causing further recovery to fail.

VI Component	Successful Recovery Rate
VMM	87.5%
DVM	96.0%
PrivVM	96.8%

Table 8.1. Recovery success rates, out of detected failures, for different VI components.

To see the overall effectiveness of our mechanisms, Table 8.1 shows the recovery success rates with respect to detected failures in each of the Xen VI components. Faults in the VMM result in the lowest success rates due to the inherent vulnerability of the recovery approach in reusing data from the failed VMM instance. Despite such drawbacks, due in large parts to our resiliency mechanisms, the LVS workload running on our FTVI can continue to correctly operate a great majority of the time across a failure of a VI component.

Some faults in VI components are not detected by our detection mechanisms but cause the LVS workload to fail (‘‘LVS (silent)’’) or prevent the VI from correctly hosting current AppVMs or creating new AppVMs (‘‘VI (silent)’’). Silent application failures can occur, for example, when faults in the DVM cause files used for servicing a client’s request to become corrupted. Faults in the VMM and PrivVM cause the vast majority of silent VI failures as both components are used to host and create AppVMs. The majority of silent VI failures caused by faults in the VMM are due to failure of the entire system most likely brought on by

triple faults (see Subsection 7.7.1). As these silent failures are the result of corruptions that do not lead to crashes and hangs, to reduce these type of failures will require more sophisticated detection mechanisms than the ones we currently employ, such as the ones proposed in [Tan12, Jeff12]

8.4. Implementation Complexity

Component	Mechanism	User- level	Kernel-level	VMM-level
VMM	Basic	0	0	830
	Final	+0	+0	+70
DVM	Basic	20	285	0
PrivVM	Basic	1730	1770	350
	Final	+575	+0	+15

Table 8.2. Lines of code (LOC) needed to implement the different microreboot mechanisms. The Final mechanism category includes the LOC for all improvements made in *addition* to the Basic mechanism.

To provide insight regarding the engineering effort required to implement microreboot-based recovery for the Xen VI components, Table 8.2 shows the breakdown of the implementation complexity, in terms of lines of code (LOC), for the different microreboot mechanisms. The basic PrivVM microreboot mechanism has the highest LOC count. Most of this code is related to backing up the XenStore and XenStored state. On the other hand, microrebooting the DVM requires the least amount of code. The DVM has no internal state that needs to be maintained and the information needed to boot and reconnect a new DVM instance to existing AppVMs is kept in the PrivVM. Note that the LOC reported for the DVM microreboot mechanism in Table 8.2 refers to the basic RebootNet/Blk scheme discussed in Chapter 4. The LOC used to implement the *MultiNetIf* and *RAID* schemes used in this

chapter is roughly the following: user-level: 780, kernel-level: 1390, and VMM-level: 340. Modifications to the VMM are needed to implement the failure alert mechanism discussed in Subsection 4.3.3. The bulk of the code in user-level is for interacting and updating device information in the XenStore during failover.

Similarly to the PrivVM, the VMM has state that must be maintained across a microreboot. However, unlike the PrivVM, the VMM preserve this state in place, in memory. This reduces the amount of code needed for saving and restoring state. To provide a sense of proportion, the entire Xen VMM has about 140K LOC.

8.5. Summary

This chapter demonstrated the effectiveness of the recovery mechanisms of our FTVI at allowing applications running in the AppVMs to continue providing services despite failure of a component in the VI. When faults are detected in the VMM, the recovery success rate is over 86%. This is the lowest success rate among the three VI components as failure of the VMM is more likely to cause the corruption of other components in the system. When faults are detected in the DVM or PrivVM, the recovery success rate is over 96%. This level of resiliency is achieved with roughly 8.2K LOC added to the Xen VI. A large fraction of this code is for saving and restoring state in the PrivVM.

Roughly 14% and 7% of faults in the VMM and PrivVM, respectively, manifest as silent failures. These failures require more sophisticated mechanisms to detect. If the very first incorrect update of some illegal memory location can be prevented, then the resiliency of the virtualized system can be further improved.

Chapter Nine

Gigan Fault Injector

Software-implemented fault injection (SWIFI) is commonly used for evaluating and characterizing system dependability features [Stot00, Carr98, Hsue97, Carr99]. The work in this dissertation has relied extensively on *Gigan*, a SWIFI tool described in this chapter, to help develop and evaluate the proposed VI resiliency mechanisms. Besides this work, as the use of virtualization has increased, there has been increasing interest in utilizing virtualization to improve system dependability [Bres96, Douc05, Rama07, Jeff12]. It is thus important to have a general SWIFI tool that can be used to evaluate and characterize the dependability features of these virtualized systems [Le08]. This chapter starts by discussing the challenges and opportunities of leveraging system virtualization for fault injection. As a part of this discussion, we present the design and implementation of our Gigan fault injector that leverages the Xen VMM [Barh03] to inject faults into VMs and the VMM. We then present an evaluation of the use of virtualized systems for fault injection.

The challenges in implementing SWIFI include: minimizing the ‘‘intrusion’’ [Arla03] on the behavior of the system under test due to the injection mechanisms or the logging of test results, the need to adapt the injector to different versions of the operating system (OS) of the system under test, and providing the ability to target individual processes or different parts of processes while also enabling the injection of faults into any part of the system.

Virtualization can simplify SWIFI even for evaluating non-virtualized systems [Sieh02, Subk07]. This simplification falls into three key categories: 1) the virtualization layer

between the OS and the hardware can be used to minimize the modifications to the system under test, 2) virtualization simplifies the management of the injection campaign and collection of results, and 3) virtualization provides ‘‘sandboxing’’ that isolates the actions of the system under test, preventing it from harming the host/control environment.

For evaluating complete virtualized systems, a SWIFI tool should be able to inject faults into the individual VMs as well as the VMM. Injection into a VM is used not only for evaluating the VM itself but also for testing the isolation among VMs provided by the VMM and the resiliency of the VMM to arbitrary faulty behavior of guest VMs. The virtualization infrastructure (VI), consisting of the VMM, privileged and driver VMs [Barh03, Fras04], is critical to the operation of a virtualized system. Hence, characterizing the behavior of the VI under faults is particularly important.

This chapter provides:

- A systematic analysis of the interactions between SWIFI and system virtualization as well as of the key challenges with implementing fault injection in a virtualized system and the resolutions to those challenges.
- An evaluation of the use of virtualized systems for fault injection, including: (1) whether fault injection results obtained when the target system runs in a virtual machine are comparable to when the target system runs on bare hardware, (2) the extent of intrusion when fault injection is performed from within the system under test, (3) the impact of logging mechanisms on the outcome of fault injection experiments, and (4) the performance benefits of conducting fault injection in a virtualized system.
- Examples of deploying Gigan to evaluate non-virtualized and virtualized systems,

highlighting the benefits of using virtualization for fault injection.

A brief review of SWIFI is presented in Section 9.1. Section 9.2 discusses the interaction between system virtualization and SWIFI. Key issues related to the design and implementation of the UCLA *Gigan* SWIFI tool are presented in Section 9.3. The experimental setup and evaluation of the use of virtualization for fault injection are presented in Sections 9.4 and 9.5, respectively. Practical examples of deploying *Gigan* to evaluate different systems are presented in Section 9.6.

9.1. Software-Implemented System Fault Injection

SWIFI requires mechanisms for: 1) triggering and performing injections, 2) logging system events and application outputs used to analyze the impact of injections, and 3) running injection campaigns, such as restarting a failed system under test so that the campaign can proceed.

SWIFIs are typically implemented in a kernel module to enable access to privileged system state. A fault injection can be triggered by breakpoints or various hardware counters, such as CPU cycles, instructions retired, cache misses, etc [Carr98]. Injection targets include registers and memory of individual processes, the kernel, or the system as a whole.

When fault injection is used to evaluate a system, logs containing the parameters of the injection as well as the outcomes from the benchmarks and error messages from the OS must be saved in a ‘‘safe’’ place, where the information cannot be corrupted. The challenge is to generate and reliably collect these logs without perturbing the normal operation of the system and skewing the injection results.

System fault injection campaigns require automating the fault injection process, log files collection, and refreshing the target system state [Gu03]. A particularly difficult step to automate is the reboot of the system on a crash or hang and restoration of the system to a clean state before the next injection. Tools such as Kdump [Goya05] and LKCD [Wild02] help automate kernel crash logging and reboot.

9.2. Virtualization and Fault Injection

This section describes how virtualization interacts with the implementation and deployment of SWIFI. It includes design choices and tradeoffs regarding how, for evaluating non-virtualized systems, the simplifications outlined in Section 9 impact the key SWIFI mechanisms (Section 9.1). While there are important advantages to using virtualization for SWIFI, there are also some non-trivial challenges that are outlined. The SWIFI mechanisms needed and challenges faced when evaluating *virtualized* systems are also presented.

9.2.1. Virtualization Facilitating SWIFI

When the goal is to evaluate a non-virtualized system, the system under test is run in a VM. The mechanism to trigger and inject faults can be run within the VM, as it does when injection is done without using virtualization (Section 9.1). An alternative is to trigger and perform the injection from the VMM. Figure 9.1 shows the basic setup of injection in a VM, illustrating a modified VM (VM1) for implementing injection from within the VM, an unmodified VM (VM2) that can be injected into from the VMM, a VMM that supports injection, and a privileged VM that controls the fault injection campaign.

There are many options for implementing injection from inside a VM. These are

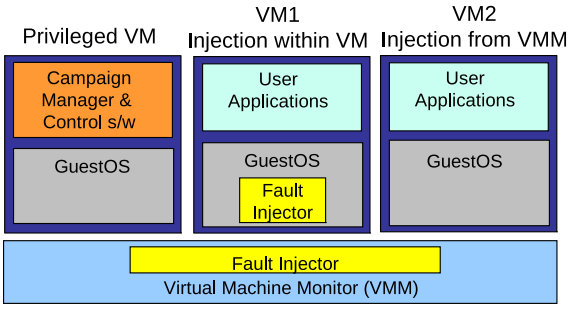


Figure 9.1: Fault injection in a virtualized system

essentially the same options as for implementing injection in a non-virtualized system [Hsue97, Stot00]. A common approach, that minimizes overhead and maximizes flexibility, uses dedicated kernel modules and hardware performance/debug registers [Carr98]. This requires detailed knowledge of the OS and porting effort for each target OS and even for different kernel versions of the same OS. On the other hand, with virtualization, there is the option of performing the injection from the VMM in an OS-agnostic fashion, without any modifications to the kernel running in the VM, thus eliminating the effort of porting the injector to different OSs. Furthermore, since no changes are required to the target VMs, the intrusion caused by the fault injector (impact of the fault injector on the behavior of the target system) is minimized.

Another alternative for injecting into VMs is to perform fault injection from the PrivVM [Oyam11]. VM introspection tools such as VMITool [Payn07] can also be extended to support such an approach. At first glance, this approach has all the benefits of injecting from the VMM without having to modify the VMM. However, this is not the case as the VMM must ultimately be modified to support the setting of injection triggers and synchronization between the target VM and the PrivVM. An additional drawback of this

approach is that it cannot be used to inject faults directly into the VMM when running on bare hardware, which is crucial for evaluating virtualized systems discussed in Subsection 9.2.4. Henceforth, we do not discuss using this approach to inject faults into VMs.

Without virtualization, logging system events and application outputs typically requires a connection to a different physical system over a network and/or a direct connection to an I/O device (e.g., a disk) where results are written. This can impact the injection results — for example, logging information by the kernel can cause a system crash that would not have otherwise happened. If information is logged to a local device, the faulty system may corrupt the log, making further analysis impossible. Mechanisms such as Kdump [Goya05], can help save the kernel state following a crash for subsequent analysis of the effects of the fault that caused the crash. However, Kdump work only if the *crash kernel* can be started following a crash. In some cases, the effects of a fault can prevent the crash kernel from starting, leading to the permanent loss of the required diagnostic information.

Virtualization can solve many of the problems discussed above by enabling the implementation of light-weight mechanisms for logging from the system under test to another VM on the same physical machine. Such mechanisms minimize the probability that the logging process itself changes the behavior of the system (intrusion). The isolation (‘‘sandboxing’’) among VMs provided by the VMM protects the logging results.

Virtualization also facilitates diagnosing the impact of faults. For example, from the VMM it is easy to determine whether the VM kernel is continuing to perform context switches among processes in the VM or whether the VM kernel has halted all of its CPUs, and thus provide more information than that the system has simply ceased to function/respond. While similar diagnosis is possible with a non-virtualized system running

on a bare machine [Wang06], without virtualization the diagnosis mechanism itself and the results of the diagnosis are vulnerable to corruption.

Fault injection campaigns often involve many injection runs in order to obtain statistically significant results. Each run requires starting with a pristine system, unaffected by any prior faults. It also requires the ability to continue running the experiment after the system under test crashes or hangs. Without virtualization, accomplishing these tasks is more complex and may require specialized hardware support. For example, if a fault can cause the system to crash or hang, without virtualization it is necessary to use a hardware mechanism, such as IPMI [Slai02], that allows a remote system to reset the system under test. On the other hand, with virtualization, a simple script, running on the host, can destroy the VM with the system under test and create a new VM for the next run. Similarly, without virtualization, before every run, pristine disk images may have to be copied across the network for the next run. This may require booting the system through a network boot, just to restore the disk contents, and then booting the system again to perform the next run. With virtualization, the equivalent task is easier to implement since the system under test can operate from disk images stored as files on the host. A simple script on the host can restore those images to a pristine state for each run.

9.2.2. Fault Injection Performance

As discussed above, it is often necessary to run fault injection campaigns for days and perform thousands of injections. Hence, the performance of a fault injector is critical to the overall usefulness of the tool. As discussed above, in a fault injection campaign, a key step in each run is the restoration of the target system to a pristine state. While the previous

subsection briefly explained that the functionality of restoring the system to a pristine state is simpler to implement with virtualization, the focus of this subsection is on performance issues.

Faults in a privileged component, such as an OS kernel, can corrupt any software or hardware components in the system. Restoring the system to a pristine state includes the state on hard disks and the states of any other hardware devices. A pristine disk state needs to be obtained from an image located in a ‘‘safe’’ place, not writable during the previous run. To restore other hardware devices to a pristine state, a system reboot is typically required. Thus, the factors that affect the fault injection run time include the time to: (1) restore disk state, (2) boot the system, (3) run the workload, (4) inject a fault, and (5) collect injection logs.

With respect to the restoration of the disk state of the target system, there are two main issues to consider: (i) the safety of the pristine disk images used for updating the system’s disk state, and (ii) the speed at which the restoration can be performed. Regarding the first issue, in a virtualized system, the virtualization layer guarantees that pristine disk images, stored in the host system, cannot be corrupted by a faulty VM. On a bare machine, unless there is a hardware mechanism to write-protect a local storage device, the only way to ensure that new disk images are pristine is to copy them across the network from another host. As to the second issue, a possible advantage of the virtualized system over a bare machine is that, with typical simple hardware setups, copying the disk images locally on the virtualized system is likely to be faster than copying across the network. Also, as discussed above, on bare hardware an additional system boot may be required, causing significant additional delay.

The time required to boot a system depends on the following: the speed at which the BIOS performs POST, the speed at which the BIOS finds and loads the OS and file system

images, the number of hardware devices that the OS probes and configures during boot, and the number of services the OS starts upon system startup. A VM is typically configured with fewer and simpler devices than a typical physical machine. This difference is not inherent and is likely to be correctable with the right hardware and software configurations. However, as a practical matter, the VM is likely to boot faster. In addition, with a virtualized system, the entire boot process can be skipped by starting the system from a checkpoint [Subk07]. With sufficient effort, a similar mechanism can be set up with bare hardware. However, with a virtualized system, this capability is typically already implemented in the existing VI.

Regarding the workload execution time, the advantage, if any, is with bare hardware. This advantage is highly dependent on the characteristics of the workload. For example, the virtual system may be significantly slower if the workload performs many page table manipulations and the host hardware does not include support for page table virtualization [AMD08, VMwa08].

Typically, the time to inject a single fault in either setup is negligible compared to the other factors. The time to *safely* store injection logs is likely to be shorter with a virtualized system. With a virtualized system, the logs from the target system are simply copied to storage on the host that is not accessible by the target. On bare hardware, the information may have to be transmitted to a different physical host using a network or even a serial connection, which is likely to be slower.

In summary, without specialized hardware and software configurations, it can be expected that fault injection campaigns can be performed significantly faster using virtualized systems than with bare hardware. Our experience with this issue is briefly discussed in Subsection 9.5.5.

9.2.3. Implementation Challenges

While there are clear advantages to using virtualization for fault injection into non-virtualized systems, there are also significant challenges: 1) accurate virtualization of the hardware cycle counter and performance monitoring/debug registers used for measurements and injection, 2) the need, in some campaigns, to target specific processes or specific parts of a process/kernel address spaces, and 3) accurate emulation of errors in a virtualized system without compromising the safety of the host system.

Modern processors include registers holding the current cycle count (tickstamp) and registers dedicated to performance evaluation and debugging. Fault injectors often use these registers for triggering injection — for example, a breakpoint in the code or some number of elapsed cycles [Carr98]. These registers may also be used for measuring events such as crash latency. The VMM must virtualize these registers if they are used to measure activities or trigger an injection in a particular VM. It is critical for the validity of injection results obtained using virtualization for the performance counter register virtualization to be accurate, measuring *only* VM activity, despite the frequent transitions between the VM and VMM that occurs during normal operation. Accurate virtualization of tickstamps in Gigan is discussed in Section 9.3.4.

Fault injection campaigns often involve targeting specific processes or parts of processes (e.g., the stack segment) as well as specific parts of the kernel address space. This is relatively easy to do when the injection is performed from inside the VM. In order to trigger and perform targeted injection from the VMM, or alternatively from the PrivVM [Oyam11, Payn07], the injector must possess detailed knowledge of the internal data structures and operation of the kernel running in the VM under test. Thus, while injection into the entire

VM is best performed from the VMM, in most cases targeted injection should still be done with injectors within the VM, enhanced by logging and campaign management features provided by the virtualized host.

In general, SWIFI's ability to emulate errors caused by hardware faults is limited by the processor's software interface [Carr98]. For example, SWIFI cannot directly inject faults into caches or registers that are not accessible to software. These limitations are not eliminated when SWIFI is deployed in a virtualized system. Furthermore, there are challenges that are unique to using SWIFI to emulate hardware errors in VMs. These challenges can be generalized as the need to accurately emulate errors that conflict with VMM/CPU mechanisms designed to protect the system from ''misbehaving'' VMs. In this context, ''accuracy'' refers to the extent to which the behavior due to a fault when the system under test is in a VM is similar to the behavior when the system under test is running on bare hardware. The rest of this subsection is focused on these challenges.

VMMs and hardware support for virtualization are designed to prevent a ''misbehaving'' VM from harming the VMM or other VMs. The mechanisms used to enforce this isolation tend to reduce the accuracy of the behavior that results from faults. In particular, faults affecting structures that interact with the mechanisms that maintain isolation (e.g., page tables, control registers, etc.) can result in different behavior from the same faults on a bare machine. This is due to: (1) for some VMs, updates to page tables are done differently than on a bare machine, and (2) there are more consistency checks performed on a VM's virtual CPU (VCPU) state than on a bare machine's physical CPU state.

To maintain isolation, virtualization mechanisms do not allow VMs unrestricted access to the address spaces of the VMM or other VMs. Memory accesses are controlled by utilizing

nested page tables or prohibiting direct modifications of page tables by VMs. With the system under test in a VM, the accuracy of emulating errors caused by faults affecting page tables depends on how the VM's page tables are managed. For example, if the VMM shadows the VM's page tables (page table installed in hardware is different from the page table accessed by the VM), then simply injecting faults into the page tables accessed by the VM will not accurately emulate the errors caused by corresponding faults on a bare machine. This is because the actual page tables being used by the hardware for that VM would not be affected by the fault. Details of how to accurately inject into page tables are discussed in Section 9.3.2.

Ensuring that the VCPU state is ''safe'' requires, for example, ensuring that reserved bits in registers contain correct values and that there is no selection of unsupported/undefined CPU mode of operations. For this safety, when hardware virtualization is used, the processor checks the consistency of a VM's VCPU state during VM entry and also whenever the VM writes to protected state. VMs that fail these consistency checks are prevented from being run by the CPU and are terminated by the VMM. Hence, an injected fault in a VM, that changes the state of a VCPU to an unsupported/undefined state, is likely to violate some consistency checks and result in the termination of the VM before the fault can manifest as it would on a bare machine. For example, on some Intel CPUs, virtual memory (paging) must be enabled before VMs can be executed. A fault that can cause paging to be disabled on a bare machine and result in memory corruption will not get a chance to even manifest in a VM as the consistency checks will prevent the VM from running.

The problem discussed in the previous paragraph can also be explained as a consequence of the fact that VMMs do not typically accurately emulate all possible faulty VM states. Fully

accurate emulation would be difficult to do and would not be useful for normal operation. Instead, VMMs often perform a consistency check on the state of a VM and, if the state is identified as being erroneous, the VMM simply terminates the VM instead of attempting to accurately emulate what bare hardware would do with that state. One example of such a state is the result of a fault that sets the VM-8086 bit in the system flags register of an FV VM running on an Intel CPU. To be consistent with Intel's specification, when this bit is set, the limit field of the segment registers must be set to a specified value. If the segment registers are not properly set and the injection is performed from the VMM, when VM entry is attempted, it fails and the VMM terminates the VM. There is no similar consistency check on a bare machine. Hence, on bare hardware, a similar fault is likely to manifest differently, most likely as memory corruption or a page fault.

9.2.4. Evaluating Virtualized Systems

As outlined in Section 9, in order to evaluate complete virtualized systems, the SWIFI tool must be able to inject into the individual VMs as well as the VMM. Injection into individual VMs is useful for evaluating the extent to which the VMM enforces isolation (protection of the entire system from arbitrary VM actions) as well as for evaluating mechanisms implemented by the VMM to tolerate VM failures [Bres96, Douc05, Rama07]. Injection into the components of the VI (VMM, PrivVM, DVM) is useful for understanding the impact of faults on these critical components.

The previous two subsections, which dealt with injection to a system in a VM, also cover most of the issues related to injection into VMs in order to evaluate the isolation properties of the VMM as well as any mechanisms it includes for tolerating VM failures. However, there

are some fault injections to individual VM that are relevant here but are not relevant when the goal is simply to evaluate non-virtualized system. These have to do with interactions between VMs and the VMM that are specific to virtualized systems. Two key examples are explicit synchronous calls (*hypercalls*[Barh03]) from PV VMs to the VMM and accesses to ‘‘I/O pages’’ that correspond to accesses to device controllers in a non-virtualized systems.

Hypercalls provide a way for a VM to make explicit requests from the VMM in the same way that system calls provide a way for normal processes to make requests from an OS kernel. A faulty or malicious VM can make arbitrary hypercalls to the VMM and it is the responsibility of the VMM to ensure that it is not damaged by these calls. There has been extensive prior work on testing the resiliency of OS kernels by perturbing system call parameters [Fabr99, Carr98]. A SWIFI tool for evaluating VMMs must be able to similarly perturb hypercall parameters.

In a non-virtualized system, device drivers in the kernel may read/write from/to addresses that are mapped to device controllers. With an FV VM, the same addresses (‘‘I/O pages’’) are mapped such that accesses cause traps to handlers in the VMM that emulate the behavior of the device with respect to the VM and may cause the appropriate operations on host devices to be performed. As mentioned in the previous subsection, if injection is performed from within the VM to an I/O page, the behavior of the real system will be correctly emulated. However, when injection is performed into the VM’s address space from the VMM, the injector must explicitly identify accesses to the VM’s I/O regions and invoke the appropriate handlers in the VMM.

Fault injection into VI components suffer from the same difficulties described earlier in this section with respect to injection into non-virtualized systems running on bare physical

machines. In particular, there are difficulties with logging of experimental results and managing injection campaigns. As discussed in Subsection 9.2.1, some of these difficulties can be handled by using virtualization. This requires running the VI under test inside a VM. With hardware support for virtualization[Uhli05], it is possible to run an unmodified VMM in an FV VM. While current hardware support does not allow the guest VMM to run FV VMs, by adopting nested virtualization techniques proposed in [Ben-10], it is possible to overcome this problem and allow FV VMs to run inside the guest VMM.

9.3. Fault Injector Design and Implementation

We have implemented, *Gigan*, a fault injector for the Xen virtualized system that can inject a variety of faults into VMs and the VMM. Gigan incorporates SWIFI enhancements that make use of system virtualization. This section describes the design and implementation of critical components of our fault injector, focusing on key implementation challenges outlined in Section 9.2.

9.3.1. Basic Capabilities

Operations in Gigan are based on *triggers* and *actions*. *Triggers* are set to fire after some threshold has been reached or some event has occurred in the target machine. Triggers can fire based on timers, instruction breakpoints, process creation/termination, and performance monitoring events (e.g., CPU cycle count) [Carr98]. Associated with each trigger is a set of one or more *actions* to be performed at the time the trigger is fired. To maximize flexibility, an action either injects a fault or sets another trigger. This ability to chain triggers increases the precision and range of when and where to perform the injection. For instance, an injection

can be set to occur 2000 cycles after the function *foo* has been invoked.

Gigan incorporates two approaches to fault injection: the VMM-level injector and the kernel-level injector. With the first approach, the injector is implemented completely outside of the VMs. To target user-level processes or specific VM kernel data structures, the kernel-level injector is used.

9.3.2. Page Table Injection Fidelity

Accurately implementing memory fault injection in a virtualized system is challenging since a random fault can occur in areas containing page tables. The fault injector must be able to emulate errors that occur on a bare machine, but guarantee that the errors injected can only affect the target VM and not the entire system. Incorrectly implementing memory injection can erroneously give a VM under test access to the VMM's or another VM's address space.

On a real machine, the available page frames are contiguous. In a virtualized system, however, machine page frames assigned to a VM are not necessarily contiguous in the host physical memory. Hence, another level of mapping is used in addition to the page tables. The VM maps its virtual pages to contiguous *guest page frames* and uses this mapping in its memory management operations. During normal execution, when the VM accesses memory, the virtual addresses it generates must be translated into physical memory addresses, often referred to as *machine addresses*. Hence, at some point, memory management must involve not only the translation of *virtual page numbers* (VPNs) to *guest frame numbers* (GFNs), but also the translation of GFNs to *machine frame numbers* (MFNs). As will be explained below, how these translations are performed greatly affect the mechanism that is required for correctly injecting faults into page tables.

A VM can use page tables that have been paravirtualized (PV) (changes to the page tables require making a hypercall) or fully-virtualized (FV). The main issue when injecting into a page table page is how to reflect changes made to the VM's view of the page table to the actual page table while maintaining correctness in terms of isolation and overall system stability. A page table entry contains a page frame number (PFN) and several control bits. We have implemented page table injection to the PFN and read/write control bits. We have yet to implement support for injecting into all control bits of a page table entry. The mechanisms to correctly and safely inject faults into a page table is different for PV and FV page tables. Hence, focusing on PFNs, the following subsections discuss how to address the main issue separately for PV and FV page tables.

9.3.2.1. Para-virtualized Page Tables

With PV page tables, the VM accesses the actual page table used by the processor. The VM can read this page table normally but can update it only through the VMM, using hypercalls. VMs using PV page tables maintain a second table, henceforth called the *GtoM table*, that maps *guest page frames* to *machine page frames*. VM page table management operations use the *GtoM table* to translate the GFNs to MFNs.

Directly modifying the PFN in a PV page table entry does not reflect the behavior that would occur on a bare machine. This is because the PFNs that are installed in the page tables represent the host's view of memory, and not the VM's view of memory. Hence, it is possible that an injection into the PFN may result in a PFN that belongs to the VMM or to a different VM. To replicate the behavior that would occur on a bare machine, the injector must first map the target PFN into a GFN, which represents the VM's view of its memory layout. The

injection is done by modifying the GFN, converting the modified GFN back into an MFN, and then installing the MFN in the VM's page table.

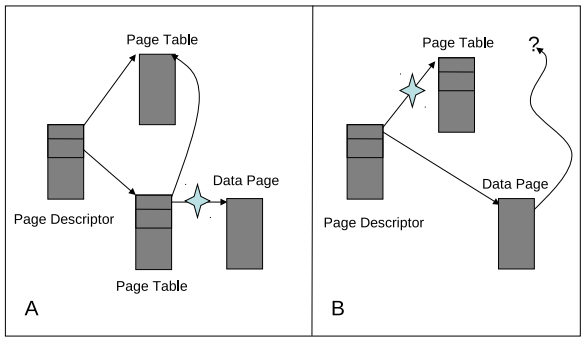


Figure 9.2: Page table injection scenarios violating fault isolation. Star means overwritten mapping.

On a bare machine, a bit flip in the page table may map a virtual page to a non-existent physical page. We have found that, on a bare machine, accesses to such pages do not result in traps. Instead, stores effectively become NOPs and loads return some fixed value. Hence, for a VM, these cases can be emulated by using an MFN that is outside the physical memory range of the host.

There are additional scenarios where injecting faults into page tables or page directories can lead to fault isolation violation. Figure 9.2 depicts these scenarios. In scenario *A*, an injection into the PFN part of a page table entry for a writable data page (*write* bit is set) can change it to map to a page table page, thus giving the VM write permission to that page table. This can potentially give the VM access to any page in the system's memory. Scenario *B* depicts the case where an injection into a page directory entry causes the entry to point to a normal data page instead of to a page table page. This data page will now act as page table and random data in that page will now be interpreted as mappings to arbitrary pages in

memory.

One way to prevent the safety problems presented in the previous paragraph is for the injector to include special checks for these scenarios and avoid performing such injections. This solution prevents some injections that can be performed when the system under test runs on bare hardware from being performed in a VM. Allowing such injections in a VM requires more complex mechanisms. For Scenario *A*, the *write* bit in the page table entry must be cleared. This causes subsequent writes to trap to the VMM, where the appropriate emulation or redirection can be performed. For Scenario *B*, the *present* bit in the page directory entry must be cleared, so that any attempt to use it traps to the VMM. As with Scenario *A*, the trap handler in the VMM must perform appropriate emulation or redirection.

Injecting into page table pages of a VM require the ability to safely and accurately emulate the effects of changes to page table entry control bits, as well as the page frame number. Unfortunately, there are some control bits in page tables for which faults are very difficult to accurately emulate. For example, a fault causing the setting of the *global page bit* in a page table entry on a bare machine can result in the incorrect translation of some memory references. However, on a virtualized system, the effect of this fault may be modified if the TLB entry containing this global mapping is evicted due to the activity of other VMs running on the same CPU. Correct emulation of the error would require mechanisms to restore the error when the target VM is scheduled to run, i.e., restore the incorrect global mapping for that VM.

9.3.2.2. Fully-virtualized Page Tables

With FV page tables, the translation of GFNs to MFNs is performed transparently to the VM by either the CPU or the VMM. In order for the CPU is perform the translation, it must have support for page table virtualization (Nested/Extended page tables) [AMD08, VMwa08]. For these systems, with respect to the safety of the VMM and other VMs, no special operations are needed when injecting faults into the memory of a VM using FV page tables. As mentioned at the end of this subsection, there are still be problems with respect to injection fidelity for some control bits.

There are still CPUs that do not provide such support and require special software mechanisms (shadowing) for implementing paging for VMs using FV page tables. With shadowing, there are two page tables for each VM. The VM accesses a *guest page table* but the actual page mapping is performed using the *shadow page table*, which is maintained by the VMM. Reflecting the injection when the shadow paging mode is used is much simpler than when PV page tables are used. When injection is performed from within the VM, the page table shadowing mechanism automatically reflects the changes from the guest page table to the shadow page table, while maintaining safety. Injection fidelity is maintained for many, but not all, the control bits. If injection is performed from the VMM, the injector must identify accesses to page table pages and explicitly invoke the shadowing mechanism to reflect the changes in the shadow page tables.

As with PV page tables discussed in the previous subsection, with FV page tables it is also not possible to accurately emulate the effects of faults in all the page table entry control bits. This is the case even if nested page tables are used. A good example of a problematic

control bit is the *global page bit* described in the previous subsection. As with PV page tables, the effects of a fault that sets this bit may also be modified if the TLB entry containing the global mapping is evicted due to the activity of other VMs.

9.3.3. Logging in a Virtualized System

Our fault injector includes a light-weight, low-overhead logging mechanism (*Comm Driver*), implemented using shared memory between VMs. We utilize Xen's split driver API [Fras04] to construct the Comm Driver frontend and backend. The backend is placed in the PrivVM while the frontend is placed in the target VM, either PV or FV. Before a fault injection run, the frontend allocates memory for logs and shares them with the backend. During an injection run, logs are generated directly into the shared memory instead of through the filesystem or network, thus minimizing injector intrusion. This has the added benefit of capturing logs as they are being generated and being able to immediately store them in a reliable place, without having to rely on a possibly faulty target VM to store the logs.

9.3.4. Measuring Latency in VMs

In the context of fault injection, accurate measurement of latency, e.g., crash latency, is often important. As explained in Subsection 9.2.1, it is desirable to run the system under test in a VM even if the goal is to evaluate system behavior on bare hardware. If this is done, the latency measurements need to be in terms of VM execution time. There are difficulties when trying to measure latency inside both PV and FV VMs. Since FV VMs most faithfully represent the environment on bare hardware, we focus the discussion on techniques for accurately measuring latency, in cycles, in FV VMs. The technique for measuring latency in

PV VMs is simpler and is discussed towards the end of this subsection. In the rest of this subsection we first discuss the problem with latency measurement in a VM that relies on Xen's existing mechanisms. Our more accurate mechanism for latency measurement within a VM is then presented.

On bare Intel hardware, latency can be accurately measured using either the time-stamp counter (TSC) or the performance monitoring counter (PMC). The TSC counts the number of elapsed clock cycles in the CPU. The PMC can be configured to count the number of elapsed non-idle CPU cycles. When the target system is run in a VM, measuring latency with respect to VM execution time requires that the TSC and/or PMC be virtualized.

As currently implemented in Xen, the TSC of a VCPU (virtual TSC) is incremented even when the VCPU is not running. This is to ensure that time keeping in the VM, which uses the virtual TSC, is consistent with wall-clock time. Hence, the virtual TSC cannot be directly used to measure latency with respect to VM execution time.

Intel's Virtualization Technology (VT) provides a general mechanism that allows the processor to be configured so that, upon exit from a VM, the values of certain machine specific registers (MSRs) are saved in memory and new values are loaded from memory to those MSRs. Upon a subsequent VM entry, the saved values are loaded from memory to those MSRs. This mechanism is used by Xen to virtualize the PMC. Specifically, this is used for a PMC control MSR that contains a bit that controls whether the PMC continues to count. Thus, upon exit from a VM, the PMC stops incrementing and VMM software can save its value. This saved value can be loaded to the PMC before a subsequent entry to that VM. We have found that, with this mechanism, the latency measurements obtained are too high since the PMC continues to be incremented after exit is initiated and begins incrementing before

entry completes (see below).

We have conducted a simple experiment to demonstrate the inaccuracy of latency measurements using Xen's existing PMC virtualization mechanism. We used a test program that consists of two nested loops: the inner loop simply adds two values together, and the outer loop executes the inner loop and then executes a single instruction (*rdtsc*) that, when executed in a VM, causes a trap to the VMM. This program roughly models the typical behavior of a VM — periods of execution strictly within the VM interrupted by exits to the VMM. Both the TSC and the PMC were used to measure the execution time of the test program on bare hardware. Sixty measurements were taken with each. The median of both sets of measurements was 455M cycles. Running the same program in a VM, measuring the latency in the same way using the PMC, yielded a result of 617.7M cycles.

In recent Intel processors there is a new mechanism that directly controls whether the PMC control MSR is loaded from memory upon VM exit and entry. We have found that the accuracy of latency measurements within a VM can be significantly improved using this newer mechanism to stop and start the cycle counting by the PMC upon VM exit and entry, respectively. Specifically, using this mechanism to virtualize the PMC, the measurement of the execution time of our test program yielded a result of 478.5M cycles.

While using the newer hardware-assisted PMC virtualization technique significantly improves the accuracy of latency measurements in a VM, there is still a discrepancy between those measurements and what is obtained with bare hardware. A key reason for this remaining discrepancy is that entering and exiting the VM is not instantaneous. The time measured in the VM still includes part of this entry/exit time (VM state load/save time). As described below, we have added to Xen a mechanism to compensate for this VM entry/exit

time and thus yield more accurate latency measurement results.

VM exits are caused by asynchronous events, such as interrupts, as well as synchronous events, such as page faults. We have found that the time for VM exit/entry depends on the cause of the exit, since the hardware performs different operations for different exit causes. We have added to the Xen VMM a mechanism to automatically measure, for each VM exit cause, the duration in cycles of the exit and corresponding VM entry. After Xen is booted, this is done for each VM exit cause, the first time a VM exit due to this cause is encountered. As the system executes, the mechanism also records the number of VM exists for each exit reason. When a VM time-stamp is needed (e.g., to mark the time of crash), the mechanism adjusts the cycle count obtained from the PMC to compensate for the VM exit/entry times, using the recorded VM exit/entry latencies and the number of VM exits.

With the improved latency measurement mechanism described above, measurement of the execution time of the test program yielded 462.9M cycles. We believe that the remaining discrepancy between this measurement and the measurement on bare hardware is due, in part, to cache and TLB pollution caused by executing VMM code. An additional source of inaccuracy of latency measurements within the VM is that these measurements do not include the time to execute instructions that cause traps to the VMM, such as *rdtsc*, *hlt*, or I/O instructions. On bare hardware, such instructions do increase the measured latency.

For PV VMs, using the TSC to measure latency has the same problems as using the TSC to measure latency in FV VMs discussed above. Instead we use the PMC directly, without having to virtualize it across VM entries/exits, to measure latency in PV VMs. This is possible since PV VMs run in a privilege level that is separate from the VMM and the PMC can be set to count non-VMM cycles.

9.4. Experimental Setup

To evaluate the impact of using system virtualization for fault injection on the fidelity of injection results, we analyze and compare fault injection results obtained when the target system is hosted on a VM and bare hardware, when injection is performed from without and from within the VM, and when different logging mechanisms are used. The following paragraphs describe the system setup and fault injection approach used for our evaluation.

We use Xen version 3.3 and Linux version 2.6.18.8 in our experiments. Our test platforms consist of 2.2GHz Intel Nehalem dual quad-core based processors with 8GB of memory. The platforms are equipped with a NIC and a BMC card using IPMI [Slai02] to allow the automation of the reboot of the system following a crash or hang.

We inject faults during the execution of the Xen VMM in some experiments and during the execution of the Linux kernel in other experiments. The details of the injection and target system configurations differ depending on the experiments, and are thus described in later subsections, where specific experiments are discussed. A common workload is used to stress the target system which consists of a subset of programs in UnixBench [Unix] with minor modifications to improve logging and failure detection.

When fault injection is performed on bare hardware, logs from the VMM and kernel are directed to the serial console where they are captured by a separate physical machine. Outputs from the workload are captured using the Comm Driver and are sent across the network to a separate physical machine before the next injection run. When performing injection on a virtualized system, the Comm Driver is used as the logging mechanism. Logs from the target system are copied from memory to separate files in the PrivVM before the

next injection run.

All the injected faults are single bit flips. We inject into the CPU's EAX register, as a representative general purpose register, as well as the stack (ESP) and instruction (EIP) pointers, and the system flags register (EFL). We inject into all non-reserved bits of the EFL, except for the VM-8086 bit (see Subsection 9.2.3). We also inject into the VMM's and Linux kernel's stack segments. Register and stack segment injections are triggered a random number of instructions (VMM instructions for experiments targeting the VMM or Linux kernel instructions for experiments targeting the kernel) after the workload (UnixBench) has been started.

We also inject single bit flips into the VMM and Linux kernel's code segments. Code injection is triggered by a breakpoint on the target instruction at a random time after starting the workload. All faults injected into the code are immediately activated. For code injection, we profiled the VMM running the workload to collect the most frequently used functions and selected their instructions as injection targets [Gu03]. The same was done to select instructions for the Linux kernel code injection.

The outcome of each injection is classified as either a crash, an externally induced crash (ExtCrash), a hang, an incorrect result, or not manifested. A crash occurs when the VMM or Linux kernel explicitly dies or stops working due to a system panic caused by a fatal exception. We detect a crash by analyzing the output of the panic handler. An ExtCrash can only occur when the target system is run in a VM. This is a scenario where the VMM explicitly crashes the VM with the target system due to fatal corruption of VM state. Hang occurs when the VMM or Linux kernel stops responding with no explicit report of a crash. To detect hangs of the VMM, we rely on the Xen VMM's built-in watchdog-based hang detector

(see Subsection 7.4.5). To detect hangs of the Linux kernel, we implemented a hang detector in the VMM that periodically checks if process scheduling is being performed by the VM's kernel (see Subsection 9.6.1). If the VM is no longer performing scheduling, a hang is identified. An incorrect result is identified when the logs from the benchmark are missing or they do not match their expected contents and there is no indication of a crash (including externally induced) or hang. Not manifested means that no errors were observed.

Crashes are the result of a majority of manifested faults. Hence, detailed information regarding crashes is particularly important. Thus, for faults that result in crashes, we also record the cause of the crash, such as a null pointer dereference or general protection fault.

The time between the occurrence of a fault and the resulting system failure is often useful for system design [Li08] and is thus an important measure to obtain from fault injection campaigns [Gu03]. Since crashes are the result of a majority of manifested faults and hang detection latency greatly depends on the check interval of the hang detector, we focus on measuring crash latencies. The crash latency we report is measured from the time the fault is injected to the invocation of the software exception handler triggered by the fault. With code injection, instructions are modified by the fault and are then immediately executed. Hence, for code injection, crash latency is measured from fault *activation* to invocation of the software exception handler triggered by the fault [Gu04].

9.5. Evaluating the Fault Injector

In this section, we analyze five aspects of performing fault injection in virtualized systems. First, we discuss the extent to which the Xen VMM can provide fault isolation guarantees to support fault injection. Second, we analyze the similarities and differences

between results obtained when the target system runs inside a VM and when the target system runs on bare hardware. Third, we analyze the degree of intrusion caused when running the fault injector inside the target system. Fourth, we analyze the extent to which using our simple, low intrusion logging mechanism can increase the fidelity of fault injection results. Finally, we analyze the impact on performance of fault injection when using virtualization.

9.5.1. Fault Isolation Enabling Fault Injection

To use system virtualization for fault injection, the VI must provide good ‘‘sandboxing’’ capability, so that faults in the target system (VM) cannot corrupt or interfere with the testing infrastructure. Based on years of experience using the Xen VI with fault injection campaigns consisting of tens of thousands of injections into VMs, we find that the vast majority of faults injected into the target VMs only cause those VMs to fail while leaving the rest of the virtualized system in working order. We uncovered only two cases where faults into VM registers caused the VMM to fail.

In the first case, a corrupted I/O state flag accessible by the FV VM caused Xen to crash due to a bug in the way Xen’s VM crash code interacts with bad VM I/O requests/replies. In the second case, a corrupted instruction pointer in the kernel of a FV VM caused a re-execution of the CPU initialization code used during VM boot. In this code the Advanced Programmable Interrupt Controller, which is virtualized by Xen, sends an Inter-Processor Interrupt to initialize all Virtual CPUs (VCPUs) including the sender VCPU. This resulted in the current VCPU trying to reinitialize itself, which the code in Xen does not handle correctly, causing a hang of the VMM. Despite these two fault isolation violations, overall, we find that the Xen VMM provides good fault isolation guarantees to enable the hosting of fault injection

campaigns.

9.5.2. VM vs. Bare Hardware

As discussed earlier, there are significant advantages to using virtualization to evaluate systems that will ultimately run on bare hardware. A cause for concern with this approach is that the virtualized hardware presented to the system under test is likely to be different from the bare hardware that will ultimately host the system when it is deployed. Differences in hardware platforms affect not only what code gets executed in the system but also when the code gets executed due to variations in the interleaving of asynchronous events. These differences can potentially affect how faults are manifested in the system and make results obtained from injecting faults into systems running inside VMs not representative.

This subsection is focused on whether the ways faults are manifested change when the target system is in a VM vs. on bare hardware. Given that the interaction with hardware is at the heart of the differences between the two ways of running the target system, injecting faults into components that interact with hardware is most likely to reveal differences in the ways faults manifest. Hence, our experiments consist of injecting faults into the Xen VMM and the Linux kernel. In both cases the injector is running within the target system: a VMM-level injector for the first case and a kernel-level injector for the second.

For the Xen VMM experiments, the target system is a single-CPU system that hosts the VMM, PrivVM (Dom0), and two single-CPU Application VMs (AppVMs), each running the UnixBench workload. We run the target system in an FV VM and on bare hardware. The injection targets are described in Section 9.4. The number of injections performed varies depending on the target. This number is set so that, for each target, the number of injected

faults manifested as crashes exceeds 100. We have found that, with this sample size, in results from additional samples the count of each of the outcomes is within 2% of the total number of injections from the reported count. Based on the above criteria, on bare hardware, the number of injections range from 300 into the EIP register to 2,700 into the EAX register. With the target system in a VM, since injection runs are significantly faster (see Subsections 9.2.2 and 9.5.5), the number of injections range from 750 into the EIP register to 3,600 into the EAX register.

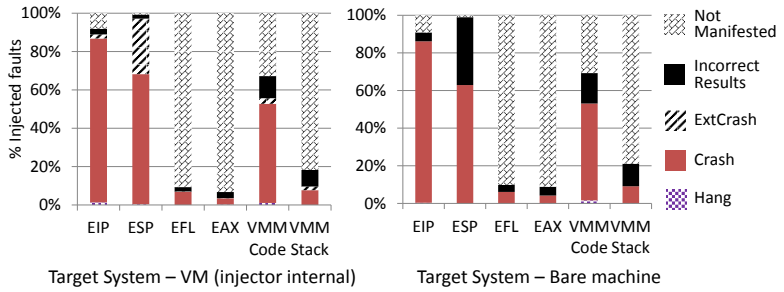


Figure 9.3: Injection outcome distributions. Target is the VMM: in an FV VM vs. on bare hardware. VMM-level injector inside the target system.

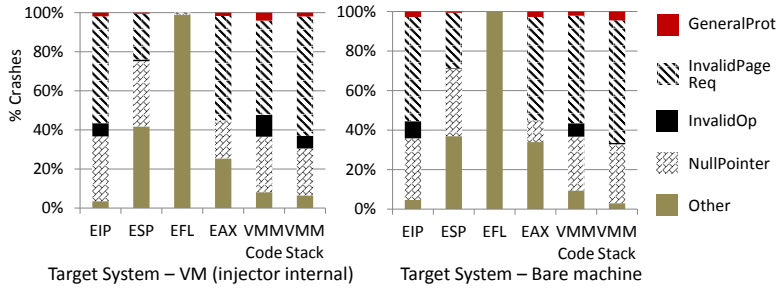


Figure 9.4: Crash cause distributions. Target is the VMM: in an FV VM vs. on bare hardware. VMM-level injector inside the target system.

Figures 9.3 and 9.4 show breakdowns of injection outcomes and crash causes from injections into the VMM. Overall, the effects of faults when the target system runs inside a

VM are very similar to when the target system runs on bare hardware. The few differences in the results shown are not due to significant differences in the effects of faults with the two platforms. Instead, as explained below, these differences are mostly due to differences in the logs that are obtained with the two platforms.

As shown in Figure 9.3, the main difference in injection outcomes between the VM and bare hardware setups are the results from injecting into the ESP register. Specifically, roughly 35% of faults are classified as incorrect results when the target system is run on bare hardware but as ExtCrashes when the system is run in a VM. The reason for this difference is that some faults can corrupt the VMM (e.g., corrupt the interrupt descriptor table) to the extent that the VMM is unable to even invoke its panic handler. With bare hardware, no useful diagnostic information is logged, but the benchmark outputs are missing or incomplete. When the target system runs inside a VM, the host VMM detects and logs such VM crashes (e.g., triple fault), thus providing more information on the cause of the system hang or spontaneous reboot. This issue is likely to also be the cause of the small differences in outcomes with EIP, VMM code, and VMM stack injections.

There are, potentially, additional causes for minor differences in the injection outcomes between the VM and bare hardware setups. For instance, the VM setup has 3GB of memory while the bare hardware setup has 8GB of memory. With the 32-bit versions of the VMM and kernel we are using, this difference in setups requires the system to execute different code for certain operations. In addition, some operations (e.g., timer handling) are executed more often in the VM setup than the bare hardware setup due to the increased execution time (wall-clock) of the workload. Therefore, these routines are more likely to be injected in the VM setup than on bare hardware.

Cycles	EIP		ESP		EFL		EAX		VMM Code		VMM Stack	
	V	B	V	B	V	B	V	B	V	B	V	B
<1M	100.0	99.6	100.0	98.7	98.6	99.1	99.2	97.2	94.9	93.7	97.4	99.2
<100K	99.8	99.2	94.6	92.3	98.1	99.1	99.2	97.2	92.9	90.1	96.7	99.2
<10K	98.1	96.1	92.8	81.3	39.6	6.4	99.2	95.4	91.4	85.9	52.3	54.5
<3K	95.2	92.9	82.6	73.6	1.5	4.6	98.3	95.4	88.3	82.2	32.0	38.6
<2K	92.5	90.6	44.1	50.8	0.5	0.0	94.1	94.5	83.8	79.6	21.6	32.6
<1.5K	88.8	89.4	30.2	48.2	0.0	0.0	90.7	94.5	79.2	78.0	17.0	27.3
<1K	43.5	67.7	0.2	16.7	0.0	0.0	48.3	81.7	45.2	56.5	3.3	12.9
<5H	0.0	6.3	0.2	1.3	0.0	0.0	0.0	23.9	0.0	3.7	0.0	3.8

Table 9.1. Cumulative distributions of crash latency measurements. Target is the VMM: in an FV VM (V) vs. on bare hardware (B). VMM-level injector inside the target system. Example: for 44.1% of crashes, latency less than 2K cycles for injection into ESP in a VM.

We use our VM time-stamp counter mechanism (Subsection 9.3.4) to measure crash latencies. Table 9.1 shows the cumulative distribution of the crash latency for both platforms. For crash latencies of 3K cycles and higher, measurements with the two platforms yield very similar results (with the exception of the EFL target). This indicates that the fidelity of crash latency measurements within a VM is good for latencies above a few thousand cycles. However, across all injection targets, with bare hardware a significantly higher fraction of the measured crash latencies are less than 1K cycles. This discrepancy may be due to variations in the VM exit/entry times (Subsection 9.3.4) that we have observed in experiments not discussed in this paper.

For the Linux kernel experiments, the target is a Linux system running UnixBench. Our experiments are limited to running this system in a single-CPU FV VM and injecting faults into the kernel code segment. This campaign consists of approximately 10,000 fault injections. We compare these results to prior work that performed similar injections, running

the Linux kernel on bare hardware [Gu03, Gu04].

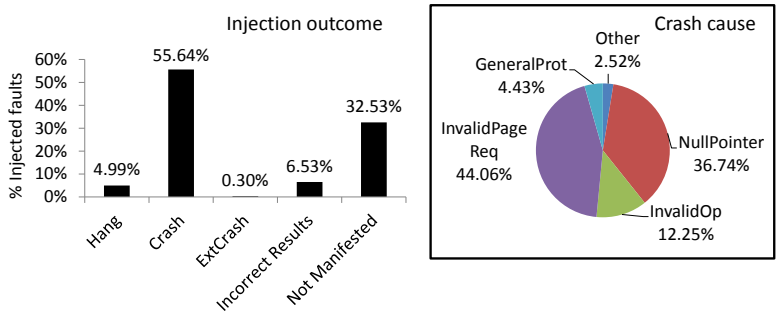


Figure 9.5: Injection outcome and crash cause distributions. Target is the Linux kernel in an FV VM. Kernel-level injector inside the target system.

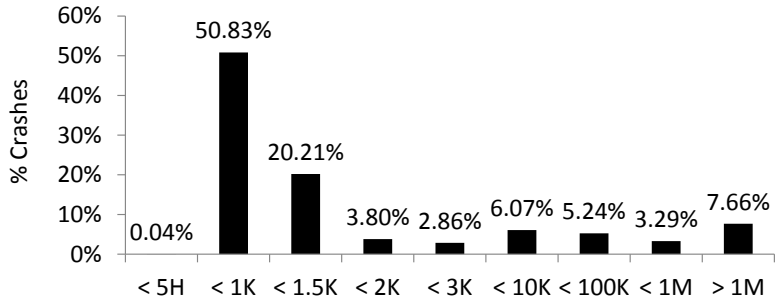


Figure 9.6: Crash latency distribution. Target is the Linux kernel in an FV VM. Kernel-level injector inside the target system. For example: 20.21% of all crashes occur within 1K-1.5K cycles.

The results from our Linux kernel fault injection experiments are shown in Figures 9.5 and 9.6. Note that Figure 9.6 shows the distribution of the crash latencies as in [Gu03, Gu04] and not the cumulative distribution as in Table 9.1. The distributions of injection outcomes, crash causes, and crash latencies when running the target system in a VM are comparable to the results from similar experiments, performed on bare hardware, by Gu et al.[Gu03, Gu04]. The few differences may be attributed to our use of a newer version of the Linux kernel (2.6 instead of 2.4) and/or to the reduced intrusiveness of the logging mechanism enabled by

running the system under test in a VM (see Subsection 9.5.4).

9.5.3. Impact of Injector in Target System

Ideally, to prevent intruding on normal system operations and skewing injection results, fault injection should not require running any additional code on the target system. However, as discussed in Subsection 9.2.3, if the goal is to target individual processes or data structures in the system under test, a fault injector that resides inside the target system may be the best option. Hence, the focus of this subsection is on whether injection results are skewed when the injection is performed from within the target system.

For injections performed from within the target system, the target OS kernel or the target VMM must be modified. Hence, any impact of the injector on the behavior of the target system is most likely to be exposed in the manifestation of faults injected into the the target OS kernel or the target VMM. Thus, for both the Linux kernel and the Xen VMM, we compare results obtained from injections from within the target VM to results obtained from injections into the VM from the host VMM, which eliminates the intrusion problem. This is done in four sets of injection campaigns: kernel/VMM target, internal/external injector. The target system configurations for the kernel and VMM experiments, workload, and fault injection targets are the same as described in the previous subsection.

For the Linux kernel experiments, the number of fault injections varied from 600 into the EIP register to 10,000 into the kernel code. The injection outcomes and crash causes distributions are shown in Figures 9.7 and 9.8, respectively. The results are, essentially, identical.

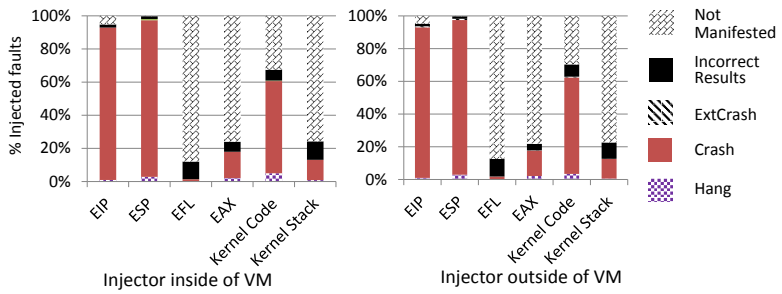


Figure 9.7: Injection outcome distributions. Target is the Linux kernel in an FV VM. Injection from within the target system vs. from the hosting VMM.

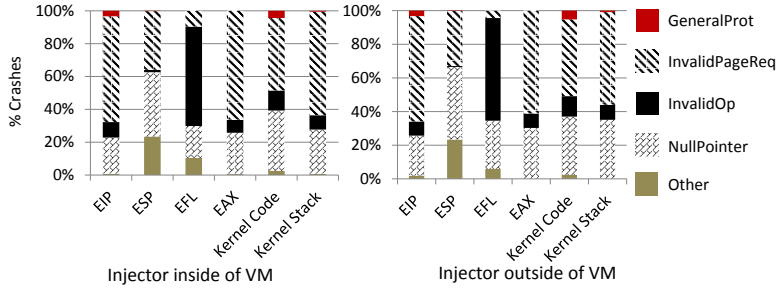


Figure 9.8: Crash cause distributions. Target is the Linux kernel in an FV VM. Injection from within the target system vs. from the hosting VMM.

For the VMM experiments, with entire target system running within a VM, the results from injections within the target system are shown in left halves of Figures 9.9 and 9.10. These are the same results shown in the left halves of Figures 9.3 and 9.4. The same injections are performed from outside the target system, with the injector in the host VMM. These results are shown in the right halves of Figures 9.9 and 9.10. In this case, the number of injections varies from 800 into the EIP register to 4,000 into the EFL register. As with the injections into the Linux kernel, the results with the external injector are, essentially, identical to the results with the internal injector.

Tables 9.2 and 9.3 show the crash latency measurements for injections into the Linux

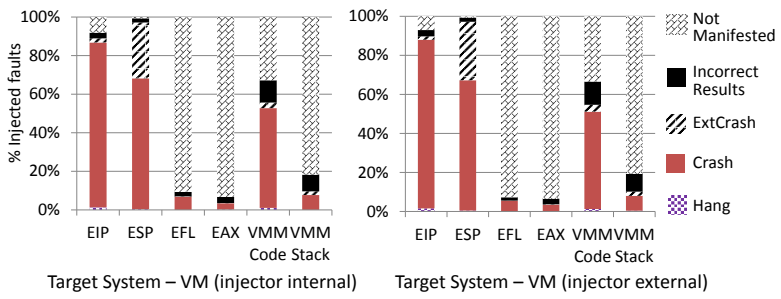


Figure 9.9: Injection outcome distributions. Target is the VMM in an FV VM. Injection from within the target system vs. from the hosting VMM. Figures showing injection from within the target system are copies from the left hand side of Figure 9.3.

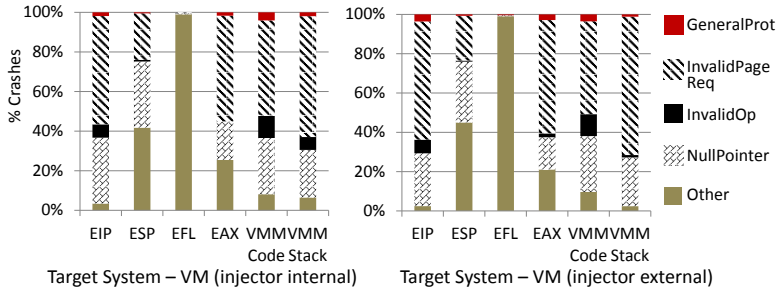


Figure 9.10: Crash cause distributions. Target is the VMM in an FV VM. Injection from within the target system vs. from the hosting VMM. Figures showing injection from within the target system are copies from the left hand side of Figure 9.4.

kernel and the VMM. For injections into the Linux kernel, crash latency measurements with the two injection modes yield very similar results for crash latencies greater than 1.5K cycles (with the exception of the kernel stack target). Across all injection targets, with the external injector a significantly lower fraction of the measured crash latencies are less than 1K cycles. This discrepancy may be caused by cache and TLB misses in the target VM after executing the injector in the host VMM.

In terms of a comparison between internal and external injections, injections into the VMM yield crash latency measurements that have very similar characteristics to the

Cycles	EIP		ESP		EFL		EAX		Kernel Code		Kernel Stack	
	V	H	V	H	V	H	V	H	V	H	V	H
<1M	97.5	96.3	91.8	91.8	97.1	95.6	97.2	95.0	92.3	93.3	96.3	98.2
<100K	96.7	94.4	81.2	82.0	97.1	92.6	95.6	92.6	89.0	90.0	94.2	95.9
<10K	93.8	90.5	52.3	59.2	66.7	66.9	93.4	88.1	83.8	83.7	69.7	73.4
<3K	89.6	86.4	45.8	45.9	51.4	50.7	91.2	85.6	77.7	75.4	50.8	37.3
<2K	87.7	82.5	43.3	43.8	39.0	39.0	90.1	83.2	74.8	69.0	38.9	22.5
<1.5K	83.8	73.1	37.9	37.8	21.9	23.5	86.2	74.3	71.0	48.6	27.6	7.4
<1K	53.5	20.5	16.1	11.4	12.4	7.4	47.0	12.9	50.8	0.5	9.2	0.6
<5H	0.0	0.0	3.8	4.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 9.2. Cumulative distributions of crash latency measurements. Target is the Linux kernel in an FV VM. Injection is performed from within the target VM (V) and from the host VMM (H). Example: for 43.3% of crashes, latency less than 2K cycles for injection into ESP with injector from inside target VM (V).

Cycles	EIP		ESP		EFL		EAX		VMM Code		VMM Stack	
	V	H	V	H	V	H	V	H	V	H	V	H
<1M	100.0	99.8	100.0	100.0	98.6	98.6	99.2	98.1	94.9	95.4	97.4	100.0
<100K	99.8	99.4	94.6	95.3	98.1	97.7	99.2	97.1	92.9	93.9	96.7	98.8
<10K	98.1	98.8	92.8	93.9	39.6	49.8	99.2	97.1	91.4	91.4	52.3	58.6
<3K	95.2	96.5	82.6	86.1	1.5	1.0	98.3	91.3	88.3	87.8	32.0	32.0
<2K	92.5	92.5	44.1	55.7	0.5	1.0	94.1	81.7	83.8	78.7	21.6	18.3
<1.5K	88.8	78.3	30.2	34.4	0.0	1.0	90.7	69.2	79.2	58.9	17.0	6.5
<1K	43.5	22.9	0.2	12.0	0.0	0.5	48.3	42.3	45.2	1.5	3.3	0.0
<5H	0.0	0.0	0.2	0.2	0.0	0.0	0.0	1.9	0.0	0.0	0.0	0.0

Table 9.3. Cumulative distributions of crash latency measurements. Target is the VMM in an FV VM. Injection is performed from within the target VM (V) and from the host VMM (H). Example: for 44.1% of crashes, latency less than 2K cycles for injection into ESP with injector from inside target VM (V).

measurements obtained from injections into the Linux kernel. The exceptions are the ESP and EFL targets, for which, with the external injector, there are *higher* fractions of latencies less than 1K cycles and 10K cycles, respectively. With ESP, during the measured interval,

relative to the external injector, the internal injector executes additional code, possibly causing additional TLB/cache misses whose latency is included in the measurement. With EFL, injections are uniformly distributed across 18 bits. However, injections to different bits cause most of the crashes with injections to the VMM vs. to the Linux kernel. For the VMM, over 90% of the injections to the EFL that result in a crash are to the Nested Task (NT) bit. For injections to the Linux kernel, injections to the NT bit are responsible for less than 5% of the crashes. The specific impact of a modified NT bit is likely responsible for the VMM vs. kernel differences.

9.5.4. Impact of Logging Mechanism

This subsection focuses on the effects of the fault injection logging mechanism on the fidelity of the injection results. We run fault injection campaigns using two setups: 1) using the Comm Driver (Subsection 9.3.3) for logging kernel and workload outputs from the target VM, and 2) using a virtual serial console to log the kernel outputs and a hard disk to log workload outputs. In both cases, the target system is an FV VM hosting the Linux kernel, running UnixBench. Faults are injected from the host VMM into CPU registers, while the CPU is executing kernel code in the VM.

The results from the first setup are discussed in Subsection 9.5.3. We compare register injection results from those experiments to results obtained using the second setup described above. With the second setup, between 400 and 2300 faults are injected into each of the registers: EIP, ESP, EFL, and EAX. With respect to the fault outcome distribution, for the EIP, EFL, and EAX registers, the results are essentially the same. However, with respect to the ESP register, there are some differences. Specifically, comparing the first and second

setups, the percentages of crashes are 94.6% and 88.5%, respectively, and the percentages of hangs are 2.9% and 10.3%, respectively.

The differences between the two setups in terms of outcomes from injections to the ESP register are likely due to the lower complexity of the Comm Driver, which made it more likely for the crash information from the target kernel to be successfully recorded with the first setup. With the second setup, the crash information were lost, leading to the classification of the outcome as a hang. Thus, this simple example demonstrates a benefit of using the low-intrusive logging enabled by virtualization.

9.5.5. Fault Injection Performance

The key issues that impact injection performance on virtualized systems vs. bare hardware are discussed in Subsection 9.2.2. To evaluate these issues in practice, we set up and measure a simple fault injection run in a virtualized system and on a bare machine.

The bare machine is configured as described in Section 9.4. The target system has a 250MB root partition. On the bare machine setup, the root partition resides on a local hard disk and the pristine root image used for restoration resides on a separate local hard disk. With this setup, there is, obviously, a small risk that the pristine root image will be corrupted by previous runs. In the VM setup, the VM's root partition is a disk image stored as a file in the PrivVM. The pristine root image is stored as a separate file in the PrivVM, not accessible by the VM. The workload is UnixBench running on top of the Linux kernel. Mechanisms for injection and collecting logs are described in Section 9.4.

Table 9.4 shows the time breakdown of a fault injection run on a virtualized system and

Operations	VM	Bare hardware
Restore disk state	0.5	7.3
Boot system	9.4	45.0
Run workload and inject	1.9	0.9
Collect logs	0.3	0.5
Total	12.1	53.7

Table 9.4. Time breakdown, in seconds, of operations in a single injection run. Target is the Linux kernel: in an FV VM vs. on bare hardware. Kernel-level injector inside the target system.

on bare hardware. As the time measurements shown are very specific to our system configuration, these results only serve to demonstrate the contributions of the five factors (Subsection 9.2.2) affecting the performance of a fault injection run.

As shown in Table 9.4, the time to boot the system dominates the overall time of performing a single injection run. On our bare machine setup, the BIOS POST stage takes 20s and 15s are used by the OS to probe and configure the devices on the system. Restoring hard disk state is faster in the virtualized system due to the ability of the host PrivVM's OS to completely cache the disk image files in memory.

9.6. Gigan Deployments

We have used Gigan to evaluate both non-virtualized and virtualized systems. In this section we discuss a few different fault injection campaigns conducted and explain how virtualization helped simplify or enable key aspects of those campaigns. We provide key sample results exemplifying the type of analysis that can be performed.

9.6.1. Evaluating Non-virtualized Systems

As an example of how system virtualization can be used to facilitate injection into non-virtualized systems, we describe a fault injection campaign used for evaluating the fault-tolerance mechanisms of *Ghidrah* [Li01, Hsu10], a fault-tolerant distributed cluster management middleware (CMM) developed at UCLA. In this campaign we used Gigan's abilities to target both the OS kernel and user-level processes. This campaign highlights the use of virtualization to facilitate the evaluation of distributed applications [Hsu10].

UCLA's Ghidrah CMM: With clusters for parallel tasks, the CMM performs critical functions, such as: allocating resources to user tasks, scheduling tasks, reporting task status, and coordinating fault handling for tasks [Li01]. Ghidrah supports running multiple parallel applications and is designed to maintain the basic cluster functionality despite the failure of any single cluster node.

CMM campaign description: While Ghidrah was designed to run on bare hardware, we ran the Ghidrah campaign on a cluster of four VMs. The application workload consisted of two instances of the IS (integer sort) benchmark from the NAS Parallel Benchmark Suite [Bail91], both running across four nodes.

To cause arbitrary failures in the VMs (cluster nodes), we injected single bit-flip faults into CPU registers while the CPU was executing Ghidrah middleware processes or kernel code. To facilitate targeting Ghidrah processes, we used the kernel-level injector inside each cluster node. This injector was also used to ‘‘inject’’ the immediate termination of specific Ghidrah processes by sending them kill signals.

Benefits of using virtualization: All the previously discussed benefits of using

virtualization for fault injection were applicable to the evaluation of Ghidrah. An additional critical benefit was the reduction in the hardware resources required. Instead of requiring a separate physical host for each cluster node, Ghidrah was run on *virtual clusters* [Le11a], where each cluster node was in a separate VM. This allowed four 8-core physical hosts to simultaneous run four separate Ghidrah clusters, each consisting of four cluster nodes.

In this campaign, kernel injections sometimes caused a cluster node to crash or hang. When this happened, a *campaign manager* process, running in the PrivVM of the host, restarted the cluster node by simply destroying the VM containing the failed node and creating a new VM instance.

As previously discussed, virtualization simplified the restoration of cluster node disk images to a pristine state. This was done using disk images kept in the PrivVM. To minimize the time to restore these disk images, we avoided the time-consuming operation of copying large disk images. This was done using a union mount in each VM [Okaj, Hsu10] for the root file system. The union mount ‘‘merges’’ a read-only branch with a read-write branch. With the union mount, only the read-write image of the file system had to be refreshed. In our setup, the size of the root file system read-only branch was 2.5GB, while the size of the read-write branch file system was 50MB. Copying the 50MB image took at most one second, while copying the 2.5GB image would have taken up to one minute. The file system images for the union mount were stored in the PrivVM, protected by virtualization. Without virtualization, it would not have been possible to reliably enforce read-only access to the read-only file system without special hardware.

Results: Of the 43,607 fault injections that caused errors, 37 exposed a total of 11 unique flaws in Ghidrah. Some flaws took tens of hours of injections to uncover,

demonstrating the importance of a fault injection campaign that can be automated to run for a long time. Of the 11 flaws, one flaw was exposed primarily by injection into the kernel. This flaw demonstrated the utility of injections targeting the kernel, even when the goal is to expose flaws in user-level programs (in this case, the Ghidrah processes).

9.6.2. Evaluating Virtualized Systems

The fault injection experiments conducted in this dissertation are good examples of the use of fault injection to evaluate virtualized systems. In this dissertation, we have designed, implemented, and evaluated mechanisms that increase the resiliency of the Xen VI to the failure of any VI component. These VI resiliency mechanisms detect and recover a failed VI component while allowing VMs running on the system to continue executing correctly with minimal interruption. For over four years, we have been using Gigan's ability to inject faults into all the Xen VI components and collect detailed logs of the results to evaluate and enhance our resiliency mechanisms. The rest of this subsection briefly discusses the benefits of using virtualization to conduct some of the fault injection campaigns described in this dissertation.

Benefits of using virtualization: All the previously discussed benefits of using virtualization for fault injection were applicable to the evaluation of our resilient VI. Of particular importance was the ability to use our *Comm Driver* (Subsection 9.3.3), which leverages shared memory on a virtualized system to collect outputs from the target system. This allowed us to collect, with minimal intrusion, detailed information regarding the operation of the system at critical stages after a fault was injected. For example, when faults were injected during VMM execution and our VMM recovery mechanism failed, this detailed information allowed us to identify and correct deficiencies with our mechanism (see Chapter

7). In addition, the Comm Driver was also used as a light-weight signaling mechanism, to coordinate activities in a campaign. For example, this facilitated initiating injection only after the target system started executing its workload. Without virtualization, such coordination would have required using the network, thus significantly increasing intrusion.

9.7. Summary

We have explored the interaction between fault injection and virtualization: using virtualization to facilitate the evaluation of non-virtualized systems and deploying SWIFI for evaluating virtualized systems. We have identified and demonstrated key benefits to using virtualization for fault injection, including: facilitating low intrusion, low overhead logging; simplifying campaign automation and target system restoration; and minimizing modifications to target system. Challenges and associate solutions to using virtualization for fault injection were discussed, including mechanisms to accurately emulate memory errors (in page tables) and accurately measuring latencies in VMs.

We have shown that fault outcome and crash cause results obtained when the target system is in a VM are very similar to the results obtained when the target system runs on bare hardware. For the few differences found (e.g., stack pointer faults), the results when the target system is in a VM actually appear to be *more* accurate and informative. This result strongly reinforces the desirability of fault injection with the system under test in a VM by demonstrating that the fidelity of the results with respect to outcomes and causes matches or exceeds the fidelity of results obtain with bare hardware.

We have developed a mechanism that improves the accuracy of latency measurements within a VM. Using this mechanism, we have shown that, for latencies of a few thousand

cycle and above, crash latencies measured when the target system is in a VM closely match the results when the target system is on bare hardware. For shorter latencies, there are significant inaccuracies with measurements within a VM.

We have also shown that executing the fault injector inside the target system does not significantly skew injection outcome and crash cause results. Hence, injection from inside the system under test is an attractive option for performing targeted injections into specific user-level processes or data-structures. Our results demonstrate that the simple, light-weight logging mechanism, enabled by virtualization, can reduce the skewing of injection outcomes due to logging. Our results also show that, while not perfect, Xen's fault isolation mechanisms protect the host from the misbehavior of the VM running the target system in almost all cases, thus demonstrating that the Xen VI is a suitable platform for fault injection. We discussed the performance benefits of using virtualization for fault injection and demonstrated those benefits in a practical example. Finally, we described examples of practical deployments of our Gigan fault injector that highlight the benefits from and need for a fault injection infrastructure that takes advantage of virtualization and can also target virtualized systems.

Chapter Ten

Summary and Conclusions

Server consolidation using system virtualization improves resource utilization but at the cost of reduced reliability since the VI is a single point of failure — a single hardware or software fault may affect the VI, causing all VMs running on the virtualized system to fail. This dissertation has presented software-based mechanisms that enhance the resiliency of the VI to transient hardware and software faults. These mechanisms are based on the microreboot technique and allow applications running in the VMs to continue providing correct services despite failure of a VI component. All the proposed resiliency mechanisms have been implemented on top of the widely-used Xen virtualized system, providing resilience with respect to faults in all three Xen VI components: the virtual machine monitor (VMM), driver VM (DVM), and privileged VM (PrivVM).

With respect to DVM recovery, this dissertation has systematically explored different techniques to tolerate device driver failure for virtualized systems using the IDVM architecture. A taxonomy for classifying the different techniques has also been developed to facilitate the discussion. While the effectiveness of the different resiliency schemes are very similar (>93% successful recoveries out of all detected failures), the latency of recovery can vary drastically across schemes. Schemes that make use of redundant devices can recover very fast, often less than 1ms, while schemes that do not make use of device redundancy may require multiple seconds to recover due to the time it takes to perform a full device reset. Techniques for reducing recovery latency without using redundant devices has been presented

for network device drivers. In that scheme, recovery latency is reduced from a few seconds to about 10ms. Unfortunately, to achieve this amount of reduction, the VMM must be modified along with device drivers for network devices that have long device reset time.

Unlike recovering the DVM, recovering the PrivVM requires maintaining state that must be restored after recovery. These states, including the XenStore and states maintained by the XenStored process, are replicated in another VI component (DVM) and are used to recover PrivVM. The main obstacle to successful PrivVM recovery is the inconsistencies that arise between the recovered PrivVM state and other VMs on the system. Fixing these problems by transactionalizing VM management operations and ensuring responses are generated for every pending requests, allow recovery success rate to increase from 36% to over 96% of detected PrivVM failures.

The VMM plays a critical role in hosting and isolating AppVMs running in a virtualized system. In this dissertation, we have presented ReHype, a mechanism to tolerate the failure of the VMM while preserving the state of running VMs. The basic version of ReHype recovered successfully from only 5.6% of detected hypervisor failures. We used fault injection results to guide incremental improvements of ReHype, leading to a success rate of over 88%. The incremental improvements involved a combination of mechanisms to repair VMM corruption and resolve inconsistencies within the VMM, between the VMM and VMs, and between the VMM and the hardware. We also discussed the factors that affect the recovery latency using ReHype and showed that sub-second (715ms) recovery latency can be achieved with minimal effort.

To assess how well all the mechanisms discussed in this dissertation work together to provide resiliency to VI component failures, we deployed and evaluated the effectiveness of

our fault tolerant VI (FTVI) on bare hardware. To stress the virtualized system, we ran a distributed highly-reliable Web service application on top of it. We then injected faults into CPU registers during the execution of different VI components. The results reveal that the combined resiliency mechanisms of the FTVI allow for the Web service application running in the VMs to correctly provide services a great majority of the time despite failure of a VI component. Specifically, for faults detected in the VMM, over 86% of those faults result in successful recovery. Given the better isolation of the DVM and PrivVM compared with the VMM, over 96% of detected faults in the DVM and PrivVM result in successful recovery.

A software-implemented fault injector, called *Gigan*, has been developed to evaluate the effectiveness of the above resiliency mechanisms. The injector is built to take advantage of facilities provided in the virtualized systems to reduce the intrusion of fault injection and log collection while increasing the performance of the injection campaign. As part of the work on the fault injector, an analysis of the differences in fault representativeness and the effects of faults between performing fault injection into systems running in a virtual machine versus on bare hardware is performed. Mechanisms to accurately measure latency using CPU cycles for VMs are also developed and evaluated.

While the mechanisms proposed in this work are able to tolerate a significant amount of detected failures in VI components, there are still a relatively large fraction of failures that were not detectable by our simple detection mechanisms. These failures are the result of silent data corruption. Direction for future work include developing better detection mechanisms to reduce the number of faults not detectable by our crash and hang detectors. Developing better ways to detect data corruption combined with recovery mechanisms proposed in this work can greatly improve the overall resiliency of the virtualized system.

Bibliography

[AMD08] AMD, ‘‘AMD-V Nested Paging,’’ *developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf* (2008).

[Arla03] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, ‘‘Comparison of Physical and Software-Implemented Fault Injection Techniques,’’ *IEEE Transactions on Computers*, vol.52, no.9, pp.1115-1133 (September 2003).

[Aviz85] A. Avizienis, ‘‘The N-Version Approach to Fault-Tolerant Software,’’ *Transactions on Software Engineering* vol.SE-11, no.12, (December 1985).

[Bail91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, ‘‘The NAS Parallel Benchmarks,’’ *Int. Journal of High Performance Computing Applications*, vol.5, no.3, pp.63-73 (Sep. 1991).

[Barh03] 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*, pp.164-177 (October 2003).

[Ben-06] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. V. Doorn, A. Mallick, J. Nakajima, and E. Wahlig, ‘‘Utilizing IOMMUs for Virtualization in Linux and Xen,’’ *Proceedings of the Linux Symposium* (July 2006).

[Ben-10] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har’El, A. Gordon, A.

Liguori, O. Wasserman, and B.-A. Yassour, ‘‘The Turtles Project: Design and Implementation of Nested Virtualization,’’ *9th Symposium on Operating Systems Design and Implementation* (October 2010).

[Bres96] T. C. Bressoud and F. B. Schneider, ‘‘Hypervisor-Based Fault-Tolerance,’’ *Computer Systems*, vol.14 , no.1, pp.80-107 (February 1996).

[Buch01] K. Buchacker and V. Sieh, ‘‘Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects,’’ *6th Int. Symposium on High-Assurance Systems Engineering* , pp.95-105 (October 2001).

[Buch03] K. Buchacker, M. D. Cin, H. J. Hoxer, V. Sieh, O. Tschache, and M. Waitz, ‘‘Hardware Fault Injection with UMLinux,’’ *International Conference on Dependable Systems and Networks, Fast Abstracts*, p.670 (June 2003).

[Cand04] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, ‘‘Microreboot - A Technique for Cheap Recovery,’’ *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pp.31-44 (December 2004).

[Carr98] J. Carreira, H. Madeira, and J. G. Silva, ‘‘Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers,’’ *IEEE Transactions on Software Engineering*, vol.24, no.2, pp.125-136 (February 1998).

[Carr99] J. Carreira, D. Costa, and J. G. Silva, ‘‘Fault Injection Spot-Checks Computer System Dependability,’’ *IEEE Spectrum*, vol.36, no.8, pp.50-55 (August 1999).

[Chou01] 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*, pp.73-88 (October 2001).

[Corp10] I. Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3A*. (2010).

[Cui09] W. Cui, D. Ma, T. Wo, and Q. Li, ''Enhancing Reliability for Virtual Machines via Continual Migration,'' *Proceedings of the 15th International Conference on Parallel and Distributed Systems* (December 2009).

[Cull08] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, ''Remus: High Availability via Asynchronous Virtual Machine Replication,'' *5th USENIX Symposium on Networked Systems Design and Implementation*, pp.161-174 (April 2008).

[Depo10] A. Depoutovitch and M. Stumm, " ''Otherworld" - Giving Applications a Chance to Survive OS Kernel Crashes,'' *Proceedings of the 5th ACM European Conference on Computer Systems*, pp.181-194 (2010).

[Dist08] T. Distler, R. Kapitza, and H. P. Reiser, ''Efficient State Transfer for Hypervisor-Based Proactive Recovery,'' *Proceedings of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems* (2008).

[Dong06] Y. Dong, S. Li, A. Mallick, J. Nakajima, K. Tian, X. Xu, F. Yang, and W. Yu, ''Extending Xen with Intel Virtualization Technology,'' *Intel Technology Journal* vol.10, no.3, (2006).

[Dong09] 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).

[Douc05] J. R. Douceur and J. Howell, ‘‘Replicated Virtual Machines,’’ *Technical Report MSR TR-2005-119, Microsoft Research* (September 2005).

[Erli06] 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*, pp.75-88 (2006).

[Fabr99] J.-C. Fabre, F. Salles, M. R. Moreno, and J. Arlat, ‘‘Assessment of COTS Microkernels by Fault Injection,’’ *Seventh IFIP Working Conference on Dependable Computing for Critical Applications*, pp.25-44 (January 1999).

[Fras04] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, ‘‘Safe Hardware Access with the Xen Virtual Machine Monitor,’’ *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS) (ASPLOS)* (October 2004).

[Gana08] 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*, pp.168-178 (March 2008).

[Goir10] I. Goiri, F. Julia, J. Guitart, and J. Torres, ‘‘Checkpoint-based Fault-tolerant Infrastructure for Virtualized Service Providers,’’ *Proceedings of 12th Network Operations and Management Symposium*, pp.455-462 (April 2010).

[Goya05] V. Goyal, E. Biederman, and H. Nellitheertha, ‘‘Kdump, A Kexec-based

Kernel Crash Dumping Mechanism,'' *lse.sourceforge.net/kdump/documentation/ols2oo5-kdump-paper.pdf* (2005).

[Gray86] J. Gray, ''Why Do Computers Stop and What Can Be Done About It?,'' *5th Symposium on Reliability in Distributed Software and Database Systems*, pp.3-12 (January 1986).

[Gupt08] 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* (December 2008).

[Gu03] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, ''Characterization of Linux Kernel Behavior Under Errors,'' *International Conference on Dependable Systems and Networks*, pp.459-468 (June 2003).

[Gu04] 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*, pp.887-896 (June 2004).

[Herd07] 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*, pp.41-50 (June 2007).

[Herd09] 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*, p.xx (June 2009).

[Hsue97] M. Hsueh, T. Tsai, and R. Iyer, ''Fault Injection Techniques and Tools,'' *IEEE*

Computer, vol.30, no.4, pp.75-82 (April 1997).

[Hsu10] I. Hsu, A. Gallagher, M. Le, and Y. Tamir, ‘‘Using Virtualization to Validate Fault-Tolerant Distributed Systems,’’ *International Conference on Parallel and Distributed Computing and Systems*, pp.210-217 (November 2010).

[Inte11] Intel Corporation, *PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology*. (January 2011).

[Inte09] Intel, ‘‘PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer’s Manual,’’ *http://download.intel.com/design/network/manuals/8254x_GBe_SDM.pdf* (Accessed March 2009).

[Jeff12] C. M. Jeffery and R. J. Figueiredo, ‘‘A Flexible Approach to Improving System Reliability with Virtual Lockstep,’’ *IEEE Transactions on Dependable and Secure Computing*, vol.9, no.1, pp.2-15 (2012).

[Jo10] 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*, vol.59, no.11, pp.1466-1479 (November 2010).

[Kang09] A. Kangarlou, P. Eugster, and D. Xu, ‘‘VNsnap: Taking Snapshots of Virtual Networked Environments with Minimal Downtime,’’ *International Conference on Dependable Systems and Networks* (June 2009).

[Kell10] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, ‘‘NoHype: Virtualized Cloud Infrastructure without the Virtualization,’’ *Proceedings of the 37th International Symposium on Computer Architecture* (June 2010).

[Kivi07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, ‘‘KVM: The Linux

Virtual Machine Monitor,'' *Ottawa Linux Symposium*, pp.225-230 (July 2007).

[Koh09] Y. Koh, C. Pu, Y. Shinjo, H. Eiraku, G. Saito, and D. Nobori, ''Improving Virtualized Windows Network Performance by Delegating Network Processing,'' *8th IEEE International Symposium on Network Computing and Applications*, pp.203-210 (July 2009).

[Kour07] K. Kourai and S. Chiba, ''A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines,'' *International Conference on Dependable Systems and Networks*, pp.245-255 (June 2007).

[Kour10] K. Kourai and S. Chiba, ''Fast Software Rejuvenation of Virtual Machine Monitors,'' *IEEE Transactions on Dependable and Secure Computing* (May 2010).

[Le08] 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* (April 2008).

[Le09] 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*, pp.195-202 (July 2009).

[Le11a] M. Le, I. Hsu, and Y. Tamir, ''Resilient Virtual Clusters,'' *17th IEEE Pacific Rim International Symposium on Dependable Computing*, pp.214-223

(December 2011).

[Le11b] M. Le and Y. Tamir, ‘‘ReHype: Enabling VM Survival Across Hypervisor Failures,’’ *7th ACM Int. Conf. on Virtual Execution Environments*, pp.63-74 (March 2011).

[Le12] M. Le and Y. Tamir, ‘‘Applying Microreboot to System Software,’’ *IEEE Int. Conf. on Software Security and Reliability*, pp.11-20 (June 2012).

[LeVa04] 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* (September 2004).

[Li01] M. Li, D. Goldberg, W. Tao, and Y. Tamir, ‘‘Fault-Tolerant Cluster Management for Reliable High-Performance Computing,’’ *Int. Conf. on Parallel and Distributed Computing and Systems*, pp.480-485 (August 2001).

[Li08] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, ‘‘Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design,’’ *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.265-276 (March 2008).

[Linu09] Linux Foundation, *Bonding*, http://www.linuxfoundation.org/collaborate/workgroups/networking/bonding (2009).

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

[Meno05] 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).

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

[Mukh05] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, ‘‘The Soft Error Problem: An Architectural Perspective ,’’ *International Symposium on High-Performance Computer Architecture*, pp.243-247 (February 2005).

[Murr08] D. Murray, G. Milos, and S. Hand, ‘‘Improving Xen Security Through Disaggregation,’’ *International Conference on Virtual Execution Environmnent*, pp.151-160 (March 2008).

[Naga07] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, ‘‘Proactive Fault Tolerance for HPC with Xen Virtualization,’’ *Proceedings of the 21st Annual International Conference on Supercomputing*, pp.23-32 (2007).

[Nguy13] 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).

[Ng99] W. T. Ng and P. M. Chen, ‘‘The Systematic Improvement of Fault Tolerance in the Rio File Cache,’’ *30th Fault Tolerant Computing Symposium*, p.76 (June 1999).

[Ng01] W. T. Ng and P. M. Chen, ‘‘The Design and Verification of the Rio File Cache,’’ *IEEE Transactions on Computers*, vol.50, no.4, pp.322-337 (April

2001).

[Okaj] J. R. Okajima, ‘‘aufs - another unionfs,’’ *http://aufs.sourceforge.net/aufs.html* .

[Oyam11] Y. Oyama and Y. Hoshi, ‘‘A Hypervisor for Injecting Scenario-Based Attack Effects,’’ *IEEE Annual Computer Software and Applications Conference*, pp.682-687 (July 2011).

[Payn07] B. Payne, M. Carbone, and W. Lee, ‘‘Secure and Flexible Monitoring of Virtual Machines,’’ *23rd Annual Computer Security Applications Conference* (December 2007).

[Pfif03] A. Pfiffer, ‘‘Reducing System Reboot Time With kexec,’’ *devresources.linuxfoundation.org/andyp/kexec/whitepaper/kexec.pdf* (April 2003).

[Pham11] C. Pham, D. Chen, Z. Kalbarczyk, R. Iyer, S. Sarkar, and R. Hosn, ‘‘CloudVal: A Framework for Validation of Virtualization Environment in Cloud Infrastructure,’’ *International Conference on Dependable Systems and Networks*, pp.189-196 (June 2011).

[Poty07] S. Potyra, V. Sieh, and M. D. Cin, ‘‘Evaluating Fault-Tolerant System Designs Using FAUmachine,’’ *Workshop on Engineering Fault Tolerant Systems* (September 2007).

[Rama07] H. V. Ramasamy and M. Schunter, ‘‘Architecting Dependable Systems Using Virtualization,’’ *Workshop on Architecting Dependable Systems* (June 2007).

[Reis06] H. P. Reiser, F. J. Hauck, R. Kapitza, and W. Schroder-Preikschat, ‘‘Hypervisor-based Redundant Execution on a Single Physical Host,’’

Proceedings of the 6th European Dependable Computing Conference, Supplemental Volume , pp.67-68 (October 2006).

[Reis07a] H. P. Reiser and R. Kapitza, ‘‘Hypervisor-based Efficient Proactive Recovery,’’ *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems,* pp.83-92 (October 2007).

[Reis07b] H. P. Reiser and R. Kapitza, ‘‘VM-FIT: Supporting Intrusion Tolerance with Virtualisation Technology,’’ *Proceedings of the 1st Workshop on Recent Advances on Intrusion-Tolerant Systems (in conjunction with Eurosys 2007),* pp.18-22 (March 2007).

[Rose05] M. Rosenblum and T. Garfinkel, ‘‘Virtual Machine Monitors: Current Technology and Future Trends,’’ *IEEE Computer*, vol.38, no.5, pp.39-47 (May 2005).

[Sega88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Dancey, A. Robinson, and T. Lin, ‘‘FIAT — Fault Injection Based Automated Testing Environment,’’ *18th Fault-Tolerant Computing Symposium*, pp.102-107 (June 1988).

[Sieh02] V. Sieh and K. Buchacker, ‘‘UMLinux - A Versatile SWIFI Tool,’’ *4th European Dependable Computing Conference,* pp.159-171 (October 2002).

[Silv07] L. M. Silva, J. Alonso, P. Silva, J. Torres, and A. Andrzejak, ‘‘Using Virtualization to Improve Software Rejuvenation,’’ *IEEE International Symposium on Network Computing and Applications* (July 2007).

[Slai02] T. Slaight, ‘‘Platform Management IPMI Controllers, Sensors, and Tools,’’

Intel Developer Forum (September 11, 2002).

[Stei10] U. Steinberg and B. Kauer, ‘‘NOVA: A Microhypervisor-Based Secure Virtualization Architecture,’’ *Proceedings of the 5th European Conference on Computer systems* (April 2010).

[Stot00] D. T. Stott, B. Floering, D. Burke, Z. Kalbarcypk, and R. K. Iyer, ‘‘NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors,’’ *4th International Computer Performance and Dependability Symposium*, pp.91-100 (March 2000).

[Subk07] M. Subkraut, S. Creutz, and C. Fetzer, ‘‘Fast Fault Injection with Virtual Machines,’’ *International Conference on Dependable Systems and Networks, Fast Abstracts* (June 2007).

[Sull91] M. Sullivan and R. Chillarege, ‘‘Software Defects and Their Impact on System Availability: A Study of Field Failures in Operating Systems,’’ *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, pp.2-9 (June 1991).

[Sun10] M. H. Sun and D. Blough, ‘‘Fast, Lightweight Virtual Machine Checkpointing,’’ *Technical Report, Center for Experimental Research in Computer Systems, Georgia Institute of Technology, GIT-CERCS-10-05* (2010).

[Swif04] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, ‘‘Recovering Device Drivers,’’ *6th Symposium on Operating Systems Design and Implementation* (December 2004).

[Swif05] M. M. Swift, B. N. Bershad, and H. M. Levy, ‘‘Improving the Reliability of Commodity Operating Systems,’’ *ACM Transactions on Computer Systems*, vol.23, no.1, pp.77-110 (February 2005).

[Tamu08] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, ‘‘Kemari: Virtual Machine Synchronization for Fault Tolerance using DomT,’’ *Xen Summit* (2008).

[Tan12] C. Tan, Y. Xia, H. Chen, and B. Zang, ‘‘TinyChecker: Transparent Protection of VMs against Hypervisor Failures with Nested Virtualization,’’ *International Conference on Dependable Systems and Networks Workshops*, pp.1-6 (June 2012).

[Tech14] S. Technologies, ‘‘Stratus - everRun,’’ *http://www.stratus.com/Products/Software/everRun* (February 2014).

[Uhli05] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, ‘‘Intel Virtualization Technology,’’ *IEEE Computer*, vol.38, no.5, pp.48-56 (May 2005).

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

[Vall06] G. Vallee, T. Naughton, H. Ong, and S. Scott, ‘‘Checkpoint/Restart of Virtual Machines Based on Xen,’’ *High Availability and Performace Computing Workshop* (October 2006).

[Vall08] G. Vallee, K. Charoenpornwattana, C. Engelmann, A. Tikotekar, C. Leangsuksun, T. Naughton, and S. Scott, ‘‘A Framework for Proactive Fault Tolerance,’’ *Proceedings of the 3rd International Conference on Availability,*

Reliability and Security, pp.659-664 (2008).

[VMwa08] VMware, ‘‘Performance Evaluation of Intel EPT Hardware Assist,’’ *http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf* (2008).

[VMwa10] VMware, ‘‘VMware High Availability,’’ *http://www.vmware.com/products/high-availability/index.html* (November 2010).

[Wang06] L. Wang, Z. Kalbarcyzk, W. Gu, and R. K. Iyer, ‘‘An OS-Level Framework for Providing Application-Aware Reliability,’’ *12th Pacific Rim International Symposium on Dependable Computing*, pp.55-62 (December 2006).

[Wang10] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar, ‘‘Checkpointing Virtual Machines Against Transient Errors,’’ *IEEE 16th International On-Line Testing Symposium* (July 2010).

[Warf08] A. Warfield, *Private Communication* (July 2008).

[Wild02] D. Wilder, ‘‘LKCD Installation and Configuration,’’ *http://lkcd.sourceforge.net/* (2002).

[Zhai08] E. Zhai, G. D. Cummings, and Y. Dong, ‘‘Live Migration with Pass-through Device for Linux VM,’’ *Ottawa Linux Symposium*, pp.261-267 (July 2008).

[Zhan03] W. Zhang and W. Zhang, ‘‘Linux Virtual Server Clusters,’’ *Linux Magazine* vol.5, no.11, (November 2003).

[Zhou06] 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*, pp.45-60 (November 2006).