CLOUDVAL: A FRAMEWORK FOR FAILURE VALIDATION OF VIRTUALIZATION ENVIRONMENT IN CLOUD INFRASTRUCTURE

BY

CUONG MANH PHAM

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Ravishankar K. Iyer
We present CloudVal, a framework to validate the reliability of virtualization environment in cloud computing infrastructure. A case study, based on injecting faults in the KVM hypervisor and XEN hypervisor, was conducted to show the viability of the framework. The study shows that due to the architectural differences between KVM and XEN, a direct comparison of the two virtualization systems is not feasible. In order to confidently weigh error resiliency of virtualization systems, more comprehensive studies are required. We believe, however, that the fault injection approach and the fault models proposed in this thesis are a good starting point towards designing and implementing a benchmark which would enable the assessment of different virtualization infrastructures in a common manner.
To my family and friends
ACKNOWLEDGMENTS

I would like to thank my advisers, Professor Ravishankar K. Iyer and Professor Zbigniew Kalbarczyk, for their great guidance and support. I would like to thank my labmates: Daniel Chen and Nitthin Nakka for their advice, collaboration and support. I also thank Soumitra (Ronnie) Sarkar, Rafah Hosn, and Murthy Devarakonda from IBM T. J. Watson Research Center for many useful discussions in the course of the work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Cloud Computing: Growing Trend of Interest and Reliability Challenges</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The Need for a More Effective Testing for Cloud Computing Infrastructures</td>
<td>2</td>
</tr>
<tr>
<td>1.3 CloudVal Approach</td>
<td>3</td>
</tr>
<tr>
<td>CHAPTER 2 FAULT INJECTION FRAMEWORK</td>
<td>5</td>
</tr>
<tr>
<td>2.1 The Injector</td>
<td>6</td>
</tr>
<tr>
<td>2.2 The Control Host</td>
<td>7</td>
</tr>
<tr>
<td>2.3 The Process Manager</td>
<td>12</td>
</tr>
<tr>
<td>2.4 The Data Extractor</td>
<td>12</td>
</tr>
<tr>
<td>CHAPTER 3 EXPERIMENT WITH VM AND XEN HYPERVISOR BASED CLOUDS</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Experimental Clouds</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Fault Models</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Experiment Setup</td>
<td>24</td>
</tr>
<tr>
<td>CHAPTER 4 EXPERIMENTAL RESULTS</td>
<td>28</td>
</tr>
<tr>
<td>4.1 Result of KVM-Based Cloud Experiment</td>
<td>28</td>
</tr>
<tr>
<td>4.2 Result of XEN-Based Cloud Experiment</td>
<td>32</td>
</tr>
<tr>
<td>4.3 Discussion</td>
<td>33</td>
</tr>
<tr>
<td>CHAPTER 5 RELATED WORK</td>
<td>35</td>
</tr>
<tr>
<td>5.1 Cloud Outages</td>
<td>35</td>
</tr>
<tr>
<td>5.2 Dependability of Virtual Machines</td>
<td>37</td>
</tr>
<tr>
<td>5.3 Fault Injection for Cloud Computing</td>
<td>41</td>
</tr>
<tr>
<td>CHAPTER 6 CONCLUSION</td>
<td>43</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>44</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

1.1 The growing trend of cloud computing (from Google Insight for Search) ........................................ 2

2.1 CloudVal components .............................................. 5
2.2 NFTAPE Control Host GUI overview ........................... 7
2.3 Campaign script state machine ................................. 10
2.4 Example of the DWARF tree structure of entries ............ 13
2.5 CloudVal workflow with DWARF components (shaded color) . 14

3.1 KVM architecture .................................................. 18
3.2 XEN architecture in hardware-assisted virtualization (HVM) . 19
3.3 Example of performance fault. Step 1: set the breakpoint at the third instruction of ioctl(). Step 2 (when breakpoint is reached): execute the injection code. Step 3 (inside the injection code): execute sleep command. Step 4: return to the normal execution of ioctl() .................................................. 23
3.4 A single testbed running four VMs (one VM running an Apache server, and three VMs running the ApacheBench workload) .................................................. 26
CHAPTER 1

INTRODUCTION

1.1 Cloud Computing: Growing Trend of Interest and Reliability Challenges

Cloud computing is an emerging paradigm in the IT industry with many large enterprises, such as IBM, Microsoft, Google, and Amazon, participating as cloud providers. It is a paradigm adopted by an increasing population with dynamic computing requirements. According to the research firm Gartner [1], Software-as-a-Service (SaaS) – one type of cloud-based service – grew 18% from $6.4 billion in 2008 to $7.7 billion in 2009, and this trend will persist through 2013. For customers, the cloud is viewed as a potential solution to replace costly in-house server farms and self-managed data centers. Paying for computing resources on an as-needed basis is the feature that differentiates cloud computing from traditional IT infrastructures. This mitigates the risk of over-provisioning or under-provisioning in IT management. These advantages promise the stable growth of cloud computing.

However, providing a higher level of availability is one of the major challenges [2] that cloud computing needs to address in order to attract more customers. Customers who have migrated their services to a cloud should not have to encounter problems such as Amazon’s S3 outage (7/20/2008) due to a single bit error in the gossip message [3], Google AppEngine’s partial outage (6/17/2008) due to a programming error [4], Microsoft Azure’s outage (3/13/2009) for 22 hours due to the malfunction in the hypervisor [5], or the case where data was lost due to a system failure in Microsoft Sidekick service [6].

Figure 1.1 is an image from Google Insight for Search showing the increasing attention to cloud computing from 2007 to 2011. Along with the increasing trend of interest, we use the dots to mark the publicly reported
outages that occurred during this time period. The data was collected from three major cloud providers: Amazon, Microsoft and Google. The glowing dots are the major outages, some of them are highlighted above. Chapter 5.1 describes in details several outstanding outages.

1.2 The Need for a More Effective Testing for Cloud Computing Infrastructures

Approaches involving the use of multiple providers [2] or running several instances [5] are typically not preferred by either cloud service providers or cloud customers. Therefore, cloud providers need to seek better solutions to gain the confidence of their customers and, conversely, customers need to see evidence of high reliability and availability in the services they purchase. A desirable long-term solution is to have (i) a highly reliable implementation for every single instance of a dynamically provisioned (possibly virtual) machine, (ii) an effective monitoring and real-time management of the problems, and (iii) a rigorous testing process for cloud deployments. In this solution, testing plays an important role in evaluating the performance and error resiliency of the services offered by the cloud.

In addition to traditional testing techniques (which mainly focus on functional testing), fault injection is widely recognized as an effective way to (i) study the complex interactions between faults and fault-handling mecha-
nisms, (ii) assess the efficiency of protection mechanisms, and (iii) quantify system/application availability and/or reliability [7, 8]. Fault injection techniques can create failure scenarios for which normal testing techniques do not usually account, such as hardware transient faults, perturbations in the operating system (OS), or drivers.

1.3 CloudVal Approach

In this research, we focus on assessing the reliability of cloud’s virtualization infrastructure given that virtualization plays an important role as the enabling technology behind the growth of cloud computing. In fact, most clouds use virtual machines (VMs) as an integral part of their core architecture. To name a few, Amazon’s EC2 is powered by XEN hypervisor, Microsoft’s Azure uses Windows Azure Hypervisor (WAH) as the underlying infrastructure, and an IBM cloud offering includes a version of the Kernel-based Virtual Machine (KVM). Virtualization technology facilitates dynamic partitioning and sharing resources in the cloud, leading to better resource utilization, and flexibility based on dynamic workload migration, etc.

We present CloudVal, a software-implemented fault injection (SWIFI) framework to automate the process of conducting fault injection-based experiments for black box testing and reliability evaluation of virtualized environments. The framework supports injecting not only basic fault models such as soft errors, but also more complex fault models such as fault mimicking delayed I/O operations or maintenance events. Moreover, the design of the framework allows new fault models to be implemented and easily added to the experimental suite.

As a case study, the framework is applied to characterize error behavior of two experimental cloud environments: (i) a KVM hypervisor [9] based cloud and (ii) a XEN hypervisor [10] based cloud. Virt-manager [11] is used as the management system of both clouds. Note that to gain insights into XEN’s error behavior, we combine the results from our experiments with the findings reported in [12]. KVM and XEN hypervisors are chosen for this study because both are open sources. Having access to the source code allows us to conduct an in-depth interpretation of the measurements.

The key contributions of this thesis are:
• Implementation of a framework to inject different types of faults (e.g. maintenance faults, performance faults) using debugger-based techniques. The design is extensible to add other fault models.

• A proposed set of representative fault models, including soft error, guest system misbehavior, performance faults, and maintenance faults, to evaluate the virtualization system.

• Demonstration of the proposed framework as a viable method to test and evaluate error behavior of KVM and XEN virtual environments, and the libvirts-based management system.

The proposed fault injection framework and the fault models are a good starting point toward designing and implementing a reliability benchmark which would enable the assessment of different virtualization infrastructures in a common manner.

The rest of the thesis is organized as follows: Chapter 2 describes the overview of the CloudVal framework; Chapter 3 shows how the framework is used to evaluate KVM and XEN hypervisor-based clouds; Chapter 4 discusses important observations from the experiments and a further discussion on the implication of the results; Chapter 5 further motivates our thesis by surveying the related work; and finally Chapter 6 concludes the thesis.
We extended the NFTAPE fault injection framework [13] to support injecting various types of fault models. We call the new framework CloudVal.

The extended framework consists of four components: Control Host, Process Manager, Data Extractor, and Fault Injector. Figure 2.1 illustrates this structure. The Process Manager, the Data Extractor, and the Fault Injector reside on target machines, which run target programs or the target operating systems. The Fault Injector is in charge of inserting fault to the software system of the target machines. In order to do that, it is implemented as an integral software component (i.e., a kernel module) of the target machines. The Process Manager creates a communication channel between the Fault Injector and the Control Host. The Control Host, which often resides on a separate machine called Control Host Machine, is interactively controlled by users to (1) specify fault injection targets, (2) create fault injection campaigns, and (3) automatically conduct fault injection experiments. The Data Extractor is a new improvement of CloudVal compared to NFTAPE. This component allows CloudVal to create a much more user-friendly interface for the users. Specifically, the Data Extractor is capable of translating symbolic
information, such as a variable's name or a function's name, input by the users to low-level target machine-specific information, such as register names or virtual addresses.

The next sections provide further details of each component.

2.1 The Injector

Our Injector uses a technique called breakpoint-based injection. This is a Software Implemented Fault Injection (SWIFI) technique that takes advantage of the debugging feature in modern processors to perform fault injection with low performance overhead. The basic idea of the breakpoint-based technique is using hardware debug registers to set breakpoints in target applications under execution. When a target program reaches a breakpoint, the injection code is executed to insert errors to the target program.

This technique has several advantages, namely (i) it allows a target program to execute at full speed until the breakpoint location is reached; (ii) neither source code instrumentation nor re-compilation is required for the target program, allowing the evaluation to be conducted on the same system as it is deployed in the field; and (iii) the injector executes in the kernel context, thus, it has access to a wide range of targets, from kernel space to application space.

The breakpoint-based injector component is implemented as a loadable kernel module on Target Machines. Using the debugger-based technique, the Injector is responsible for (i) setting each triggering location as a breakpoint location, and (ii) injecting the fault when the breakpoint location is reached. A processor debug register is used to set the breakpoint, and the fault injection code is implemented in the breakpoint interrupt handler. The breakpoint determines when to inject the fault; and the breakpoint interrupt handler (the injection code) determines how to inject the fault.

This design gives the injector both the required capabilities as well as the extensibility. On the one hand, the injection code executes, as part of the kernel, and hence, can access the entire kernel address space, the user address space of every running process, and the physical registers available to the operating system. On the other hand, using the basic breakpoint triggering mechanism, the injector can be extended to support new fault
models by adding new injection code to the breakpoint interrupt handler (see Section 3.2 for the descriptions of different fault models implemented in the Injector).

2.2 The Control Host

![NFTAPE Control Host GUI overview](image)

The Control Host is implemented as a Java program running on Control Host Machines. It provides a GUI interface for the end user to (i) specify the parameters of the experiment, (ii) generate a script to automate the experiment based on the parameters provided, (iii) execute the script to perform the experiment, and (iv) log the data that is the output of the experiment. The Control Host communicates with the Process Manager through a TCP connection to control the experiment and collect the experimental data from the target machine.
Table 2.1: Control Host GUI’s Components

<table>
<thead>
<tr>
<th><strong>Component</strong></th>
<th><strong>Support for Users</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Browser Pane</td>
<td>Manage target programs and target machines</td>
</tr>
<tr>
<td>Target Browser Pane</td>
<td>Display target application structure to enable users to select injection locations (e.g., global function, global variable, class members, etc.)</td>
</tr>
<tr>
<td>Workspace Pane</td>
<td>Create custom fault models</td>
</tr>
<tr>
<td>Edit Campaign Script Tab</td>
<td>View and edit the auto-generated campaign script</td>
</tr>
<tr>
<td>Injection Location View Pane</td>
<td>Display the created injection locations</td>
</tr>
<tr>
<td>Output Pane</td>
<td>Display logging information during experiments running</td>
</tr>
</tbody>
</table>

2.2.1 Control Host GUI Overview

Table 2.1 describes Control Host GUI’s components. The two most important components are the Target Browser pane and the Workspace pane. The Target Browser pane allows users to view the symbolic structure of target program and/or target operating system. Users can also select the injection locations (e.g., a function or a local variable) directly from this pane. The Control Host obtains this information from the Data Extractor. See Section 2.4 for further details on how the Data Extractor obtains this information.

The Workspace pane allows users to create custom faults. There are two areas in this pane, the upper area is to specify injection locations, such as trigger locations and injection locations. A trigger location is a location in the target program, where the execution of this location will trigger the fault. For example the trigger location is the first instruction of a function, when this function is called, the fault will be injected. An injection location is a location in the target program, to which the fault will be injected to. For example the injection location is a local variable, when the trigger location is reached, this local variable will be corrupted as specified by the fault model. The fault model is created in the lower area of the Workspace pane. In this area, users need to specify how the fault should be performed. For example, a fault model would corrupt one single bit (as the soft error model presented in Section 3.2.1) in the injection location, or simply just delay the execution of the triggered thread for a pre-defined amount of time (as the performance fault presented in Section 3.2.3).
2.2.2 Experiment Parameters

The experiment parameters include:

**Target object**: provides information about the target machine and the target software. The target machine is identified by its IP address and port number on which the Process Manager listens. The target process (executing the software) is identified by its location in the target machine. The target process can be a user process (e.g., VM or an application) or the operating system. Control Host uses this parameter to create connection to the target machine.

**Number of injections**: the number of injections that the Control Host needs to generate for the experiment.

**Trigger location**: a trigger can be located within a function or specified by a range of addresses within the code segment or data segment of the target object.

**Injection location/object**: specifies the location where the fault should be injected. An injection location can be given as a memory address, a register, or an offset from a base register.

**Fault model**: specifies the fault type to be injected. It can range from bit-flips and value overwrite, to thread hang and processor on/off.

2.2.3 Automation Scripts

The script generated by the Control Host describes a finite state machine, as shown in Figure 2.3, for running the injection experiment in an automated way. It describes the actions to be taken to initialize the experiment, perform the injection, verify the injection result, and repeat this process until all the specified injections are performed.

2.2.4 General Usage of Control Host

The purpose of the Control Host GUI (referred to as GUI hereafter) is to help the user manage the injection targets, specify parameters for the injector, create and execute the injection campaigns, and collect the injection result. This is closely related to the steps the user goes through to perform evaluation of a target application/system using NFTAPE. The steps for conducting an
evaluation include (1) specifying injection targets, (2) designing campaigns, (3) running injection campaign and (4) analyzing logged information. In the following, we explain the purpose of each step.

**Step 1: Specifying injection targets**

The first step in creating an injection campaign is to provide the GUI with the information about the target system and target application. The information is specified in the Target Browser tab of the GUI and includes the IP address or hostname of the target system, the type of target system platform, the location of the target system, and the location of the verifier (if one is available). Based on this information, the GUI extracts information about the target application (e.g., function/variable address) and the target system (e.g., network devices), and displays them on the GUI to assist the user to specify the injection locations in the next step. The target application and target system information can be saved to a target profile for the later use.

**Step 2: Designing injection campaign**

The second step is to tell the GUI WHEN the fault should be triggered, WHERE the fault should be injected, and HOW to perform the corruption.

First, the user designs the campaign by selecting the injection locations (WHEN and WHERE information), and the fault model (HOW information). GUI provides interfaces for the user to specify this information in the Create Injection tab in the Workspace pane. Under the Create Injection tab there are four sub-tabs which correspond to four types of injection TEXT, STACK,
DATA, and REGISTER. The detailed functionality of each sub-tab will be explained in the next sections.

Second, the user requests GUI to generate the campaign. A campaign contains (1) a campaign script and (2) an injection command list.

(1) The campaign script defines the precise activities that automatically execute a fault-injection experiment. It defines the sequence of steps, and data, needed to run a specific experiment. This includes the various timing, scheduling, and dependency constraints that determine the execution of a typical experiment.

The model used in structuring the progress of a campaign is that of a state machine. At any given moment, the campaign is in a single unique state. This state, in turn, reflects the current state of all processes involved in the experiment. State transitions occur when an event of significance to the current campaign happens. Such an event might be the report that a particular process has started executing, that it has terminated, or that it has crashed.

(2) The injection command list is automatically generated by GUI based on the injection locations and fault models that the user has specified in the previous step.

Because the target application resides on the target machine, GUI needs to invoke the Data Extractor module, which also resides on target machine to extract the requested information, such as breakpoint addresses. The rule of the Data Extractor module is to search within the given range (“whole program” or “a specific function”) and randomly picks the breakpoints that meet the user’s requirements. Text injection requires that selected breakpoints must correspond to an instruction. Stack injection requires that instructions at selected breakpoints must access memory in the stack (note: access here means both read and write; “memory accessed” could be any location on the stack, not only the target location). Register injection requires that instructions at selected breakpoints must access the target register.

Step 3: Running injection campaign
Select “Run campaign” button to start the campaign.

Step 4: Analyzing log information
The first part of the log file contains the summary of the campaign. The second part contains the information about the activities of NFTAPE during the time the campaign is running. Each message line has the same form as:
The following message elements are defined:

- \textbf{Inj \# \{number\}}: the sequence number of the current injection
- \textbf{[Run \#\{number\}]}: the sequence number of the running campaign
- \textbf{(Timestamp)}: timestamp of the current message
- \textbf{\{sys/out/in\}}: the source and destination of the message
  - \textbf{sys}: means this is an information message generated by GUI
  - \textbf{out}: means this is the command that GUI sends to target machines
  - \textbf{in}: means this message is a return message from target machines to GUI

\section*{2.3 The Process Manager}

The Process Manager is a daemon running on the Target Machine which is responsible for (i) executing the commands received from the Control Host on the Target Machine and (ii) passing the experimental data back to the Control Host. The Process Manager is automatically started at boot time to allow the Control Host to communicate with it when the injected faults cause the target machine to crash and reboot.

\section*{2.4 The Data Extractor}

The injector, which operates as an interrupt handler, can only understand the low-level description of locations, e.g., virtual addresses or register base addresses. In order to inject faults at precise timing and location on high-level programming language objects, the user has to manually calculate the mapping from source-code-level locations to binary-level addresses. Moreover, recent computer architectures and compiler optimization techniques allow an object to change its location throughout its lifetime; this makes location calculation a non-trivial task.
Responding to this drawback, we add the Data Extractor component to CloudVal to allow the user to create injection experiments from the source-code-level abstraction, providing, for example, capability to select local variables for a particular function. Specifically, the Data Extractor uses debugging information to automatically derive injection timing and location, so that the user does not have to deal with this semantic gap between high-level and binary-level descriptions.

2.4.1 Debugging Information

![DWARF tree structure of entries](example.png)

Figure 2.4: Example of the DWARF tree structure of entries

Debugging information is designed for debuggers to coordinate with compilers. A compiler generates information in a format that is understandable by a debugger. The debugger then uses this information to provide developers features such as setting breakpoints at the source code level and printing values of local variables at the time of a breakpoint triggered.

Among debugging formats, DWARF is a mature standard and widely supported by major compilers (e.g., GCC) and debuggers (e.g., GBD). DWARF is designed as an architecture-independent standard to support a variety of procedural languages, such as C, C++, FORTRAN, and Modula2.

DWARF represents the structure of an executable file in a tree of Debugging Information Entries (DIE). Figure 2.4 depicts the DIE tree structure of a simple application, which consists of compilation unit (simple_app.c). This
compilation unit owns a `main()` function, which is described by the following set of attributes: name (DW_AT_name), return type (DW_AT_type), code range address (DW_AT_low_pc and DW_AT_high_pc) and location of stack base pointer (DW_AT_frame_base). Function `main()` has two child nodes describing two local variables `lv` and `plv`.

DWARF location descriptions hold locations of program objects. In the above example, the DW_AT_location attribute describes the location of the `lv` variable, which is DW_OP_fbreg:-10. This description means the address of `lv` is a -10 byte offset from the stack frame base pointer. Tracing back to the parent node, the stack frame base pointer is described as the DW_AT_frame_base attribute of the main function. The attribute value DW_OP_breg4 8 means the stack frame base pointer is an 8 byte offset from the register 4 (in IA32 architecture, reg4 is the ESP register). Combining these descriptions gives the address of the `lv` variable, which is -2 (accumulation of all offsets: 8+(-10)) byte offset from the reg4 register.

### 2.4.2 Integrate DWARF to CloudVal Framework

![Figure 2.5: CloudVal workflow with DWARF components (shaded color)](image)

The process of conducting a fault injection campaign is demonstrated in Figure 2.5. An executable file when compiled with the debug option contains several debug sections in binary form. Our DWARF Parser processes these sections to form a condensed DWARF data structure. A GUI tool is developed to import this data and construct a tree that represents the target program structure.

The Control Host GUI allows the user to visually select triggering conditions/locations and injection locations. A trigger location can be a specific
instruction, a lexical block, a function or the whole code segment. A trigger condition allows the user to specify more precisely when the injection is triggered. For example, a trigger condition can be a specific instruction type, a read/write operation on a specific data object, or when a certain data object matches a certain value. These selected locations are then translated to virtual addresses to form the injection commands. Inject commands are batched into a campaign script. The campaign script automatically invokes the injector to inject each fault.
CHAPTER 3

EXPERIMENT WITH VM AND XEN HYPervisor BASED CLOUDS

In this experiment, we apply CloudVal to study the behavior of (i) KVM-based and (ii) XEN-based clouds in the presence of injected faults. Virt-manager is used as the monitoring and management system of both clouds. The goal is to evaluate the failure isolation mechanisms, the maintainability, and the completeness of the implementation of the two hypervisors and the management system.

Specifically, this chapter describes the experiment in the following five steps:

- **Step 1: Analyzing the target system** to understand its high-level architecture: what the main components of the system are and how they interact to each other; specifically, Section 3.1 analyzes the architectures of KVM and XEN hypervisors - the subjects of our experiments.

- **Step 2: Selecting candidate fault models** for each component based on the understanding about the target system gained from the first step; the selected fault models need to be representative and implementable by software fault injector. Specifically, Section 3.2 describes four fault models that we use in the experiments, namely soft error, guest system misbehavior, performance fault, and maintenance fault.

- **Step 3: Performing pre-injection analysis** to identify the triggering and injecting locations within each component for each fault model. This is an important step to ensure the efficiency of each injection: maximize the activation rate of the fault. Since we want to evaluate the system in the actual execution scenarios, the set of identified locations usually associates with a certain set of input or workload for the system. Specifically, we apply two injection strategies: stress-based and path-based fault injections.
• **Step 4:** **Setting up the experiment** involves creating an experimental environment that mimics the actual deployment of the system under evaluation. One of the important factors of this environment is the workload, which is used to exercise the system during the fault injection experiment. This workload also needs to be representative for the typical workload in operational system. Specifically, Section 3.3 describes the KVM and the XEN experimental environments used in our experiments.

• **Step 5:** **Analyzing the experiment result** is the final step that involves processing the logged data of the injection tool. The next chapter, Chapter 4, is dedicated to present our analysis.

3.1 Experimental Clouds

3.1.1 KVM-Based Cloud

The experimental cloud on which the experiments were conducted is built on the KVM hypervisor included with the Redhat Enterprise Linux (RHEL) 5.4 distribution (Linux kernel version 2.6.18-164.el5). Virt-manager 0.6.1, an open source virtual machine management tool developed by RedHat based on libvirt API, was used as the management system. Virt-manager supports basic virtual machine administrative operations, such as provisioning of VMs, modifying VM configurations, cloning VMs, installing guest OSes, and monitoring resources online. A single virt-manager instance can manage multiple hypervisors across physical machines.

3.1.1.1 The KVM Hypervisor

KVM (the Linux Virtual Machine Monitor) is a kernel extension, which, after loading, turns the Linux kernel into a virtual machine monitor or hypervisor.

Figure 3.1 depicts the KVM architecture. The hypervisor consists of a KVM kernel module and one qemu-kvm user process for each VM (or guest system). The KVM kernel module leverages hardware virtualization provided by recent x86 processors (e.g., Intel VT and AMD-V), to emulate virtual
CPUs (vcpu). This module is also responsible for entering the guest mode and handling memory management of the VM. After entering the guest mode, the guest code, including both the guest OS and the guest applications, is executed natively, rather than using emulation or binary translation, until it needs I/O access or receives incoming interrupts. In KVM architecture, all IO operations are forwarded to user mode, which is the hardware emulator qemu-kvm. A qemu-kvm handles all the I/O accesses of a VM. This is a multi-threaded process, which creates one thread for each vcpu and one thread to simulate other devices such as a NIC (network interface card) controller and disk controller.

3.1.1.2 Fail-Stop Model of KVM

In a cloud environment, the providers often do not control the user’s workload running in the guest VM. However, they need to take care of the software and hardware layers running below the guest OS. In this case, they are qemu-kvm processes, the KVM kernel module, the host OS, the management system, and the physical hardware. Ideally, a failure should not propagate (i) from guest mode to user mode or kernel mode; (ii) from user mode to kernel mode and (iii) from hypervisor to the management system.
3.1.2 XEN-Based Cloud

This experimental cloud is built on XEN hypervisor 3.0, which is included with the RHEL 5.4 distribution, and Virt-manager 0.6.1.

A XEN virtualization system is powered by the XEN hypervisor, which is the most privileged software layer operating right on top of the hardware. On top of the XEN hypervisor, one or more guest operating systems can be hosted. After the hypervisor boots, it automatically loads the first guest operating system (Dom0). Dom0 has special management privileges with respect to other VMs, called DomU. By default, Dom0 has direct access to the physical hardware.

In order to separate the mechanism and policy, XEN hypervisor exports control interface to Dom0. Application-level management software running in Dom0 utilizes this interface to manage the system’s resources. For example, xenstored is used to store information about the domains during their execution and to create and control domU devices. In order to support the unmodified guest OS, XEN also uses a customized version of qemu (qemu-dm) to simulate virtual hardware. Similar to KVM, each VM is coupled with one qemu-dm process running in Dom0. In the context of this thesis, XEN hypervisor and the user-application management software are the targets for fault injection based analysis. Figure 3.2 depicts XEN architecture in hardware-assisted virtualization (HVM) mode.

Figure 3.2: XEN architecture in hardware-assisted virtualization (HVM)
Table 3.1: Soft Error Fault Model

<table>
<thead>
<tr>
<th>Fault</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit flip</td>
<td>Flip single or multiple bits</td>
<td>Flip one bit of original value 0xabcdef98; new value is 0xabcdedefb8</td>
</tr>
<tr>
<td>Add</td>
<td>Increment/decrement value of the target</td>
<td>Increment the original value 0xabcdef98; new value is 0xabcdedef99</td>
</tr>
<tr>
<td>Swap</td>
<td>Interchange the lowest bits and highest bits of a byte or word</td>
<td>Swap the byte of the original value 0xabcdef98; new value is 0xef98abcd</td>
</tr>
<tr>
<td>Overwrite</td>
<td>Write a specific value to the target</td>
<td>Write 0x00000001 to target memory</td>
</tr>
<tr>
<td>NOP</td>
<td>Write NOP instruction to the target in code segment</td>
<td>Original instruction is 0xabcdef98; new value is 0x90909090</td>
</tr>
</tbody>
</table>

3.2 Fault Models

The following description introduces fault models selected for this experiment. For each fault model we discuss the real situation mimicked by the fault, the rationale behind its selection, and the implementation of the fault.

3.2.1 Soft Error

This fault model mimics soft or transient errors occurring in memory, registers, or execution and control units of the CPU. The error is represented as a single-bit or multiple-bits flip. Table 3.1 details the types of soft errors supported by CloudVal.

3.2.1.1 Rationale

Soft errors are well-known phenomena caused by “glitches” in semiconductor devices that occur mainly because of radiation particles (e.g., alpha particles and atmospheric neutrons, which are by-products of cosmic rays) and power spikes during system operation. The problem tends to worsen since the error rate increases when manufacturers reduce the voltage and the memory cell size but increase the memory speed and capacity of each new generation of memory technology. Current memory technologies have been embodied with Error Correcting Code (ECC) to detect and correct soft errors. However, the
standard hamming ECC is only able to correct single-bit errors (SEC) and detect double-bit error (DED) due to the substantial space overhead (23.4% for 64-bit triple bit correction).

3.2.1.2 Implementation

The injector sets a breakpoint at the target process, such as the qemu-kvm process as specified by the user. When the breakpoint is triggered, the injection code first removes all the breakpoints to avoid repeating the injection, then corrupts the target object (a register or a memory location). Table 3.1 lists all the data corruption actions that the user can select.

3.2.2 Guest System Misbehavior

This fault model mimics the misbehavior of the guest system which may lead to system state corruption or CPU exceptions. Ultimately this unusual behavior could result in crashing or hanging the target guest system.

3.2.2.1 Rationale

This fault model is used as a black box testing of the failure isolation between the guest system and the host system. All failures ideally should be confined within the target guest system.

3.2.2.2 Implementation

A modified version of the injector is installed as a kernel module in the guest OS to inject this type of fault. As a kernel module, the injector can identify the current running process in the guest system and then:

- **randomly corrupt a state of a process**: flip one or multiple bits in data, stack, code or register of that process; or

- **raise a CPU exception**: the exception is randomly chosen from among MCE (machine check exception), divide by zero, and invalid instruction.
3.2.3 Performance Fault

This fault model mimics the situation where one or more threads are experiencing delay due to the blocking during I/O access, CPU exhaustion, or being interrupted by events, such as a scheduling event.

3.2.3.1 Rationale

The central idea behind a VM is to abstract the hardware of a single physical machine for sharing between many guest OSs. However, if accesses to these shared resources are not properly synchronized, it may result in race conditions, which eventually cause unexpected system behaviors. In addition, due to the time-dependent and non-deterministic nature of the problem, detecting this type of bug is difficult, though it is important.

Since a race condition is timing sensitive, it is more likely to be exposed in high load situations. For instance, one class of race condition bugs is the time-of-check-to-time-of-use (TOCTTOU) flaw. This flaw is more likely to be exposed if the TOCTTOU is prolonged because of, for example, a heavy workload. Therefore, this fault model is specifically designed to exercise the implementation of the hypervisor in the situation where several threads are blocked at a certain point for a certain period of time.

3.2.3.2 Implementation

First, the user specifies a breakpoint address, a number of threads to be injected, and a delay time. The injector then sets a breakpoint at the given address in all running threads of the target program. When the breakpoint is triggered in a thread, the injector removes the breakpoint in that particular thread to avoid repeating the fault injection step in the same thread. Note that the breakpoints in other threads are still kept active. After the given number of threads are triggered, the injector removes the breakpoint in all the threads to make sure that fault is injected only to those specific threads. The injection code, after being triggered, simply puts the thread to sleep for the given delay time.

For example, the injector inserts 52 seconds of delay in three threads when executing the third instruction in the ioctl() function, the virtual address
Figure 3.3: Example of performance fault. Step 1: set the breakpoint at the third instruction of ioctl(). Step 2 (when breakpoint is reached): execute the injection code. Step 3 (inside the injection code): execute sleep command. Step 4: return to the normal execution of ioctl() of which is 0x3f7b0cc647. To perform this injection, a breakpoint is set at virtual address 0x3f7b0cc647 in all running qemu-kvm processes. However, only the first three threads which reach the breakpoint will be subjected to the 52 seconds delay. Figure 3.3 depicts the above example.

Due to the time constraint, we were able to develop this fault model for KVM systems only.

3.2.4 Maintenance Fault

This fault model mimics the situation where a certain part (e.g. a CPU core or a memory bank) of the hardware needs to be turned off for replacement or power management.

3.2.4.1 Rationale

This fault model is intended to evaluate the maintainability and manageability of the hypervisor. Modern system architecture and the current version of the Linux kernel allow CPU hot-plugging [14] for online maintenance, upgrading capacity on demand, and power saving [15] without affecting the Reliability, Availability and Serviceability (RAS) of the system. It is important that cloud components retain this feature of the base infrastructure.
3.2.4.2 Implementation

Current implementation of Linux supports CPU hot-plugging, which is the ability to turn a CPU core on and off dynamically. A user can change the state of a CPU by modifying the value in /sys/devices/system/cpu/cpuX/online, which in turn invokes a kernel function that updates the CPU state. The injector changes the target CPU state using the same kernel function. The user first specifies the breakpoint location and the target CPU to be turned on or off. The injector sets a breakpoint at the breakpoint location. When the breakpoint location is reached, the injector calls the kernel function that manages the CPU state, with the target CPU as a parameter, to change the CPU state.

3.3 Experiment Setup

3.3.1 Selecting Fault Models

This section describes our selection of fault models for each component based on the knowledge about the target systems.

3.3.1.1 KVM Virtualization System

- **Guest system**: We use the guest system misbehavior fault model to verify whether the injected faults are contained in the targeted VM or not.

- **qemu-kvm**: Since this multi-threaded process is responsible for multiplexing shared resources between VMs, we introduce performance fault into this process to evaluate its resource sharing implementation. In addition, a set of soft errors is also used to validate the fail-stop assurance of this user-mode component.

- **KVM kernel module**: We use soft errors to evaluate the robustness and fail-stop assurance of this kernel module against random hardware transient fault.
• **Linux kernel of the host system**: Since this is the layer that directly works with the hardware components, we apply the maintenance fault here to evaluate the system’s maintainability, as well as to compare the behavior of the guest system against the hardware configuration changes.

### 3.3.1.2 XEN Virtualization System

• **Guest system**: fault injection results reported in [12] are used to characterize the implication of errors that impact the guest system.

• **qemu-dm and xenstored**: these two user-level components provided by XEN enable proper creation and control of guest systems. Soft errors are injected in the two processes to evaluate the error impact on the reliability of the XEN system.

• **Linux kernel of Dom0**: as Dom0 is able to manage the physical hardware, we test it with the maintenance fault. In addition, to evaluate the criticality of Dom0 for the reliability of the XEN system, we test the impact of Dom0 crashes.

• **XEN hypervisor**: fault injection results reported in [12] are used to characterize the implication of errors that impact XEN hypervisor.

### 3.3.2 Testbed Setup

Figure 3.4 depicts the workload setup of the testbed, in which a KVM hypervisor hosts four VMs: three VMs run ApacheBench, a HTTP server benchmarking tool, to intensively send HTTP requests to the Apache server running in the other VM. This KVM hypervisor and all four VMs are managed by a virt-manager (which is not depicted in Figure 3.4) running in a separate machine via an SSH connection.

The XEN-based testbed has the same configuration and executes the same workloads as the KVM-based testbed described previously.
3.3.3 Pre-injection Analysis

Pre-injection analysis is performed to identify the triggering and injecting locations within each component for each fault model. Since the injection space is large, this is an important step to ensure the efficiency of each injection: maximize the activation rate of the fault. As we want to evaluate the system in the actual execution scenarios, the set of identified locations is usually associated with a certain set of input or workload for the system.

3.3.3.1 Stress-Based Fault Injection

Stress-based fault injection [16] prioritizes the introduction of faults into the most heavily used components of the target. We used Oprofile [17], a Linux profiler that utilizes hardware performance counters, to accumulate statistical information on how much time is spent by the program in each function. This information is used to identify the functions that are most often used by the target program and thereby generate a fault injection campaign to target these functions.

3.3.3.2 Path-Based Fault Injection

The idea of path-based fault injection [16] is to record the sequence of instructions executed by the target program under a certain input. Instead
of using the technique presented in [16] to record the instruction sequence, we used a Linux feature that allows reading the online status of the running processes via /proc file system. For each running process, Linux generates one /proc/PID/stat file (PID is the value of the process ID, e.g. 1010) containing the status information of the specified process (by PID) at the time the user reads its contents. One important piece of information this file provides is the current value in the current instruction pointer (EIP register) of the process. Our script periodically reads this file to sample the locations on the execution path of the target program. The injector uses these locations as fault targets.
CHAPTER 4
EXPERIMENTAL RESULTS

This chapter discusses the results and observations from the conducted fault injection experiments. Table 4.1 summarizes these results.

4.1 Result of KVM-Based Cloud Experiment

Observation 1.1: In KVM, no fault propagation occurred from the guest to host system during the four weeks experiment.

In contrast to [12], which found cases that the injected faults propagated from the guest system to the host system in XEN hypervisor, we did not find any similar case in the KVM hypervisor during a period of four weeks (two machines x two weeks each) of continuously injecting faults. It could be that we have not executed the test cases that uncover the potential bugs; or it could be that the KVM hypervisor’s implementation does not have the same defects as the tested XEN hypervisor.

When setting up fault injection in the guest kernel to generate the failures in the VM, we realized that KVM does not allow the guest OS to modify the content of debug registers of the VM. While the injector could successfully read the value of any debug registers, its value remained unchanged when the injector tried to write to those registers. This indicates that debug registers are not fully implemented in KVM guest systems. In order to overcome this problem, we modified the injector to directly inject the error to the memory or register, without waiting for the triggering event. Therefore, we could not count the number of activated faults in this experiment (this information is missing in Table 4.1, in the guest misbehavior row), because this number is the count of triggered faults when using the breakpoint-triggering mechanism. However, this information is not important in this experiment because its main purpose is to generate failures in the guest VM.
<table>
<thead>
<tr>
<th>Fault type</th>
<th>Hypervisor</th>
<th>Target</th>
<th># Activated / Injected faults</th>
<th>Guest behavior</th>
<th>Hypervisor behavior</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest misbehavior</td>
<td>KVM</td>
<td>Code segment, data segment, stack segment and register of guest OS kernel</td>
<td>14,000 injected faults (2 machines x 2 weeks)</td>
<td>Guest kernel crashes</td>
<td>No fault manifestation from guest to host is observed</td>
<td>Debug registers are not fully implemented in guest machines</td>
</tr>
<tr>
<td>Turn one CPU core OFF</td>
<td>KVM</td>
<td>Physical CPU core</td>
<td>100/100</td>
<td>No change</td>
<td>No change</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>XEN</td>
<td>CPU core in guest VM</td>
<td>100/100</td>
<td>No change</td>
<td>No change</td>
<td>N/A</td>
</tr>
<tr>
<td>Turn one CPU core ON</td>
<td>KVM</td>
<td>Physical CPU core</td>
<td>100/100</td>
<td>No change</td>
<td>100 kernel crashes</td>
<td>Kernel crashes only when KVM module is loaded</td>
</tr>
<tr>
<td></td>
<td>XEN</td>
<td>CPU core in Dom0</td>
<td>100/100</td>
<td>No change</td>
<td>No change</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>XEN</td>
<td>CPU core in guest VM</td>
<td>100/100</td>
<td>No change</td>
<td>N/A</td>
<td>Cannot turn CPU ON by command</td>
</tr>
<tr>
<td>Soft error (in user address space)</td>
<td>KVM</td>
<td>Data segment, stack segment and register of qemu-kvm process</td>
<td>496/500</td>
<td>Guest VM stopped when qemu-kvm crashed</td>
<td>120 qemu-kvm crashes. 26 become defunct (zombie) state. No kernel crash.</td>
<td>When a qemu-kvm process becomes defunct, virt-manager hangs (some libvirt APIs do not return)</td>
</tr>
<tr>
<td></td>
<td>XEN</td>
<td>qemu-dm</td>
<td>100/100</td>
<td>Guest VM stopped when qemu-dm crashed</td>
<td>55 qemu-dm crashes. 12 defunct (zombie) state. No kernel crash.</td>
<td>Virt-manager loses track of the crashed VM. Need to manually call the destroy command to release the VM before being able to restart it.</td>
</tr>
<tr>
<td></td>
<td>XEN</td>
<td>xenstored</td>
<td>100/100</td>
<td>No change</td>
<td>No change</td>
<td>Virt-manager loses control of this XEN hypervisor</td>
</tr>
<tr>
<td>Soft error (in kernel address space)</td>
<td>KVM</td>
<td>Data, code segment and register of KVM kernel module</td>
<td>380/1000</td>
<td>N/A</td>
<td>94 kernel crashes</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>XEN</td>
<td>Crashing Dom0 by inserting a faulty kernel module</td>
<td>100/100</td>
<td>All guest VMs stopped</td>
<td>20 kernel crashes</td>
<td>N/A</td>
</tr>
<tr>
<td>Performance fault (threads delayed)</td>
<td>KVM</td>
<td>Threads in qemu-kvm process</td>
<td>399/400</td>
<td>DoS during injected fault</td>
<td>16 kernel crashes</td>
<td>Kernel crashes. Could be due to race conditions</td>
</tr>
<tr>
<td></td>
<td>XEN</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Observation 1.2: In KVM, the kernel hangs when a CPU core is turned on.

Listing 4.1: Kernel call trace printed after turning one CPU core on with the KVM module loaded when no VM is running

Listing 4.2: Kernel call trace printed after turning one CPU core on with the KVM module loaded when one VM is running
In the experiment, we observed that turning off a physical CPU core in the hypervisor does not affect the availability of the running VMs and other services. However, whenever we turned that CPU core on, the KVM kernel module caused the whole host system to hang.

We also noticed that the host system hung even when there was no VM running. However, when we unloaded the KVM kernel module from the host system, we could successfully turn on and off any CPU core (except core 0).

Interestingly, when turning on one CPU core in the hypervisor that was hosting no VM, the kernel message repeatedly printed the call trace showing that KVM was hung in the kvm_cpu_hotplug() function (see Listing 4.1 and Listing 4.2). The name of the function suggested that KVM kernel module did consider this CPU hot-plugged situation; however, it did not perform correctly in this corner case. The function might have not been tested thoroughly.

**Observation 1.3: The Qemu-kvm process becomes a zombie, and the management system based on libvirt hangs.**

We injected bit-flip error to random registers in the qemu-kvm process. In some cases, the bit-level error caused the qemu-kvm process to become a zombie process (transition to defunct state); the management system hung when querying the state of the VM corresponding to the defunct qemu-kvm process.

We discovered that this occurred because of bugs in some API in libvirt used by the virt-manager. That function does not return a value when the target is a defunct VM. This is another corner case that appears not be tested fully in libvirt.

**Observation 1.4: The kernel crashes when performance faults are injected into the qemu-kvm.**

In this experiment, we used the path-based method to sample the breakpoint addresses. The number of injected threads is randomly selected between 1 and 4, after which the delay time to inject the fault is also randomly selected from 1 to 100 seconds.

Among 399 activated faults, we observed 16 faults causing the kernel to crash during the experiment. This behavior is interesting because while the faults were injected in user-level processes, they caused the kernel to crash.

In order to understand these kernel crashes, we set up another experiment to inject the delay time into only one breakpoint location that crashed the
kernel in the previous experiment. The result of the experiment is shown in Table 4.2. We did not observe any kernel crash when inserting a delay in only 1 or 2 threads. However, we found three kernel crashes among 100 activated faults, which inserted delays in three or four threads.

Even though we did not determine the root cause of the crash, this symptom is similar to one caused by a race condition error. The crash only happens when the delay is inserted in multiple threads and the result is non-deterministic, meaning that the crash occurs for a few injection actions only.

### 4.2 Result of XEN-Based Cloud Experiment

Beside the two cases reported in [12] of error propagation from the guest VM to the hypervisor, our experiments reveal the following additional problems in XEN hypervisor and virt-manager:

**Observation 2.1: Cannot turn on CPU in domUs.**

In domUs, the CPU core can be turned OFF, but cannot be turned back ON: the guest OS throws an error message while still operating normally. The only way to bring the CPU back is to reboot the guest OS. This problem could be due to the incomplete implementation of the virtual CPU in XEN. Note that dom0 is able to handle these events correctly.

**Observation 2.2: The qemu-dm crashes are not detected by virt-manager automatically.**
When the qemu-dm process crashes or becomes a defunct state, virt-manager loses track of the crashed VM – lists the VM as “no state” – until the user manually executes a destroy command to force the release of this VM. The “no state” state means that the virt-manager still considers the crashed VM as an active one and hence, it does not allow VM restart.

Observation 2.3: Xenstored crashes cause the virt-manager management system to lose control.

Although crashing xenstored does not affect the availability of the current running VMs, it causes the management system to lose control of the whole hypervisor until the hypervisor is properly restarted. Changing xenstored from a statefull design to the stateless design might help to avoid this problem.

4.3 Discussion

The use of the framework. Our experiment shows that the proposed fault injection framework can be used for black box testing of the virtualization system, since it does not require deep understanding of the target system. Techniques such as path-based and stress-based fault injection can automate the process of generating and executing the fault injection experiment without requiring knowledge of the internal implementation/structure of the program.

This is a starting point toward building a common reliability benchmark to enable comparing between cloud implementations: the fault injection results could be used to compare the error sensitivity between each implementation against a certain set of fault models. This information could be a good reference for the customers to consider when they select a specific cloud for their application.

Virtual machine vs. bare metal machine. From our experience of conducting debugger-based fault injection, hardware debug registers are fully implemented in VMWare hypervisors. However, the guest misbehavior experiment in KVM shows that hardware debug registers are not fully emulated in guest systems. Besides, it is commonly known that some sensitive instructions do not function correctly in most virtualized systems (e.g. CPUID instruction, performance counters). This behavior suggests that a virtu-
alized system does not completely represent the bare metal system. VM implementations are different across hypervisors. Therefore, a guest system may behave differently in different virtualization environments, especially for low-level system operations.

In addition to KVM’s lack of low-level debugging capability, our experiments also point out that the CPU hot-plugging capability is not correctly simulated in XEN’s guest system. Even though this problem is not likely to affect the normal usability of XEN, a malicious user might exploit such system-specific information leakage to access the XEN underlining virtualization system of the cloud and exploit XEN vulnerabilities (e.g., bugs reported in [12]) to crash the VMs of other users sharing the same hypervisor.
CHAPTER 5
RELATED WORK

In order to motivate the demand for a sound assessment method for cloud computing, we start our discussion by highlighting representative examples of cloud outages. Next, we generalize the issues which are challenging cloud providers in order to provide a higher level of availability for their services. Finally, we present a survey of fault injection techniques and studies that are relevant to the context of cloud computing.

5.1 Cloud Outages

These outages did not only affect the cloud availability, they also forced cloud providers to make important design changes in the cloud infrastructure. These are the valuable lessons for constructing the next generation of highly available and secure clouds.

5.1.1 Microsoft Azure

During a routine operating system upgrade on March 13, 2009, the deployment service within Windows Azure began to slow down due to networking issues. This caused a large number of servers to time out and fail.

Applications running only as a single instance (i.e., without replication) shut down when the corresponding server went down. Very few applications running multiple instances failed, although some were degraded due to one instance being down. In addition, the ability to perform management tasks from the web portal appeared unavailable for many applications due to the Fabric Controller being loaded with work during the serialized recovery process.

To prevent such occurrences in the future, Microsoft has been fixing the
network issues. It is refining and tuning recovery algorithms to ensure handling malfunctions quickly and gracefully. For continued availability during upgrades, application owners are encouraged to deploy their application with multiple instances. The second instance of an application is not counted against quota limits to allow customers to run two instances of each application.

5.1.2 Amazon S3

The 8-hour outage of Amazon services on July 20, 2008, was caused by a single bit error in messages communicated (using a gossip protocol) between the servers. In their postmortem analysis, some Amazon system engineers determined that a handful of messages had a single bit corrupted in such a way that the message remained intelligible, but the system information was incorrect. MD5 checksums were used throughout the system to prevent, detect, and recover from the corruption that can occur during receipt, storage, and retrieval of objects from customers. However, the system did not have the same protection to detect whether this internal state information had been corrupted. Therefore, corruption was not detected when it occurred and it spread throughout the system [3].

Amazon decided to add one more layer of checksum to protect the stored internal state information.

5.1.3 Bitbucket

On October 3, 2009, BitBucket (https://bitbucket.org/) experienced 16+ hours of downtime due to two consecutive DDoS attacks targeted at the network interfaces on Amazon EBS (Elastic Block Store) service for storage used with EC2 instances. Such a problem could have been quickly resolved if the incidents were promptly diagnosed given sufficient visibility to the network traffic. The real issue is the multilevel administration of cloud services.

For BitBucket system administrators, the network traffic on the physical servers is a black box. Consequently, the administrators cannot do anything at the layers they cannot reach. The only solution is to rely on Amazon’s support. However, after six hours, even with urgent request tickets and phone
calls, Amazon’s best advice was that EBS is a shared network resource and therefore performance would vary. Only afterward did Amazon acknowledge the problems with the service and work with BitBucket to resolve the problem.

It was determined that BitBucket was under a massive-scale DDoS attack using UDP packets. Amazon simply blocked the UDP traffic to resolve the problem. On the next day, another DDoS using TCP packets targeted BitBucket. But this time it took only two hours for Amazon and BitBucket to resolve the problem.

5.1.4 How Can Research Help?

Failure patterns similar to the S3 failure were observed in an error-injection-based experimental analysis [18] of the Ensemble Group Communication System (GCS), a robust communication layer for distributed dependable applications. The study shows that about 5-6% of application failures are due to an error escaping the GCS error-containment mechanism and manifesting as silence data corruption. It is important to note that although the percentage of the observed silence data corruption is relatively small, such errors do constitute an impediment to achieving high dependability because recovery from these failures can involve significant system downtime.

Validation of such large-scale cloud computing deployments is always challenging, yet it is an interesting opportunity for the research community.

5.2 Dependability of Virtual Machines

The use of virtual machine (VM) based systems introduces the hypervisor between the operating system and the hardware. The relationship between the hypervisor, also called the virtual machine monitor (VMM), and the guest operating system is analogous to the traditional relationship between the operating system and the application processes running on it.
5.2.1 Virtual Machine-Based Dependability Techniques

5.2.1.1 Fault and Failure Detection

VMs provide a software layer between OS and hardware and enable monitoring of the behavior of the guest system. Vigilant [19] is an initial effort that applies machine learning to detect VM failures based on the correlation of events generated by monitors at the hypervisor layer. Intrusion detection systems (IDS) are moving toward out-of-host implementations, in which the hypervisor layer becomes an attractive option [20].

Monitoring at the hypervisor layer enables failure/attack isolation and hence, independent reporting of the observed incidents from the outside, without the possibility of being corrupted/manipulated by the failing guest system.

5.2.1.2 Recovery

Virtualization encapsulates each complete guest system into a virtual machine and provides a convenient way to capture snapshots of the system state. Therefore, checkpoint and rollback are the primary recovery mechanisms in virtualization environment. Table 5.1 lists the existing mechanisms of VM checkpointing.

In the first category of VM checkpoint, the VM is stopped completely to save its state in persistent storage, and then the VM resumes. This approach incurs a large system downtime during the checkpoint.

In the second category (e.g., CEVM [21] and VNsnap [22]), VM live migration and copy-on-write are employed to create replica images of VMs with low overhead. Then the image is written to disk in the background or by the separate physical node. This disk-based VM checkpointing is not scalable, as it stresses the storage system when many VM checkpoints need to be written at the same time. In addition, the checkpoint is susceptible to corruption due the low-frequency updating.

The last two categories are the high-frequency VM checkpointing based on live migration (e.g., Remus [23]) and incremental checkpoint in main memory (e.g., VM-μCheckpoint [24]). These checkpoint schemes cannot tolerate latent errors, since the checkpoint might contain dormant faults. A
Table 5.1: Categories of Existing Mechanisms for VM Checkpointing

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Brief Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop-and-Save</td>
<td>Stops a VM completely, and saves its state to persistent storage</td>
<td>• Large system downtime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Provided by all major VMM systems</td>
</tr>
<tr>
<td>Low-Freq (interval &gt;1h) based on live migration (e.g., CEVM, VNsnap)</td>
<td>Creates a VM replica on a remote node via live migration, then the remote node writes the replica to disk</td>
<td>• Significant recomputation during recovery, as checkpoint frequency is low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Large overhead (maintain full replicas for a protected VM)</td>
</tr>
<tr>
<td>High-Freq (interval 10-1000 ms) based on live migration (e.g., Remus)</td>
<td>Maintains a VM replica on a separate physical node via live migration, and fails-over upon a failure</td>
<td>• Large overhead while migrating latest updates to the remote node continuously (~50% overhead for 50 ms checkpoint interval)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fail-stop assumption</td>
</tr>
<tr>
<td>High-Freq (interval 10-1000 ms) based on incremental checkpointing (e.g., VM-μCheckpoint)</td>
<td>Maintains high frequency (intervals &lt;1s) incremental checkpoint of dirty pages in main-memory; recovers from the stored checkpoint in the same process context</td>
<td>• Small overhead (6.3% for SPEC06, 17.5% for Apache)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Reduced likelihood of checkpoint corruption</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• High recovery overhead when suffering latent faults (must recover from the disk-based checkpoint)</td>
</tr>
</tbody>
</table>
hierarchical architecture combining high-frequency, incremental checkpointing and low-frequency, disk-based checkpointing is proposed in [24]. This approach reduces checkpointing corruption and performance overhead, but it still achieves latent error tolerance.

5.2.2 Challenges of Using Virtualization in the Cloud

Although there has been substantial progress in improving VM checkpointing, it is still challenging to minimize checkpoint performance overhead, checkpoint corruption, and checkpoint inconsistency in a seamless way in the cloud infrastructure.

The non-uniform, dynamic geographic distribution of the nodes in the current cloud-computing environment violates the assumptions of traditional distributed systems regarding communication overhead. Legacy techniques such as synchronous and asynchronous checkpointing already incur significant overhead and cannot be applied naively in the new scenario without investigation. Added to that are the high and nondeterministic costs that result from the dynamic nature of the distributed system.

The major technical challenge in implementing monitoring and recovery mechanisms at the hypervisor level is the semantic gap between the guest system and the low-level hypervisor. The semantic gap prevents the hypervisor from interpreting the behavior of the guest system from the set of observable events and states at the hypervisor’s point of view. To address this problem, virtual machine introspection techniques [25] have been introduced. The current techniques use the knowledge of the internal structure of the guest system to extract its behavior. For example, Linux kernels manage processes by maintaining a list of task_struct data structures, which can be used to obtain the list of the running processes in the guest system.

The fundamental limitation of this approach is its dependence on the invariants of the guest system, mostly of the guest operating system. Therefore, the implementation of this approach must be adjusted whenever the used invariants change. For example, the location and structure of the task_struct vary in different versions of the Linux kernel, so the introspection tools have to be customized accordingly to operate correctly. In addition, significant effort is often required to obtain the meaningful OS invariants. The develop-
opers and maintainers of these tools need to have a deep understanding of
the internal implementation of the interested systems. Unfortunately, this is
almost impossible in case of a closed-sourced OS like any version of Windows
OS.

In order to motivate the demand for a sound assessment method for the
cloud, we start our discussion by highlighting representative examples of
cloud outages. These outages did not only affect the cloud availability, they
also forced cloud providers to make important design changes in the cloud
infrastructure. These are the valuable lessons for constructing the next gen-
eration of highly available and secure clouds.

5.3 Fault Injection for Cloud Computing

Fault injection techniques have been widely used to evaluate the dependabil-
ity of computer systems. Since both hardware and software are sources of
system failures [8], various fault injection tools [13, 26, 27, 28, 29, 30, 31, 32]
are developed to support the evaluation of computing systems by injecting
software and hardware fault models.

Fault injection tools can be categorized into three types based on their
implementations on simulation, hardware, or software. A simulation-based
fault injection is conducted in early design stages on a simulation platform,
where the design is being tested. In order to evaluate a prototype of the
developed system, injection tools must be implemented in either hardware
or software (or both of the above [33]) of the target system. A hardware-
based tool, such as pin-level injection [34] or laser fault injection (LFI) [35],
often incurs high cost to build and exhibits inflexibility to use. The idea
of using software implemented fault injection (SWIFI) was originated in the
late 1980s with the introduction of FIAT [28], a tool that added functions to
test trigger conditions and inject faults at compile time. The SWIFI method
shows many advanced features over the hardware-based method, such as low
cost to build and operate, wide range of simulated faults, and flexibility to
control. Our CloudVal injection mechanism falls into the SWIFI category.

Fault injection has been adopted in the context of cloud computing due to
the increasing demand for high availability. Studies [36, 37] present PreFail
and Fate tools for efficient injection of failures into cloud software systems,
such as HDFS. PreFail [36] provides a programmable failure abstraction, which allows users to write policies to prune down large spaces of multiple-failure combinations. Fate [37] is a framework for cloud recovery testing. Fate specifically aims to solve the problem of massive combinatorial explosion of failure scenarios by implementing a smart exploration strategy which prioritizes failure scenarios that result in distinct recovery actions. The idea of Failure as a Service (FaaS) was generalized in [38] and [39]. Both studies aim at developing general frameworks to perform failure drills on various components of cloud infrastructures, such as storage, network, and compute nodes. The proposed failure models include node failures, message losses, and resource exhaustion. The authors of [39] also present a case study on the impact of injected failures on Hadoop applications.

Many fault injection experiments have been conducted to measure the reliability of the OS [40, 41]. However, only a few systematically apply this technique to evaluate the cloud virtualization environment. For example, M. Le et al. in [12] study XEN hypervisors error behavior by injecting faults to both the hypervisor and the guest system. This work reveals some bugs in XEN’s implementation, which allows injected faults to propagate from the guest system to the hypervisor. However, this experiment used only hardware transient fault (or soft error) and had to modify the hypervisor to inject fault to the guest VM.

In this study, we propose CloudVal, a fault injection framework for thoroughly evaluating the virtualization layer of cloud computing infrastructures.
CHAPTER 6

CONCLUSION

We have presented CloudVal, a fault injection framework that supports injecting different types of fault models. The experiments demonstrate the use of this framework in KVM and Xen virtualization systems (managed by a virt-manager virtual machine manager) by using soft error, guest misbehavior, performance fault, and maintenance fault models. The experiment results show that the presented fault injection mechanism and design of fault models are a good starting point to develop a common benchmark for assessing cloud virtualization infrastructures.
REFERENCES


