Keeping an Eye on Virtualization: VM Monitoring Techniques & Applications

Validation
Why Is this Important?

- Providing a higher level of reliability and availability is one of the biggest challenges of Cloud computing
  - Amazon’s S3 outage (7/20/2008) due to a single bit error in the gossip message
    - offers customer 10% credit if S3 availability drops below 99.9% in a month
  - Google AppEngine’s partial outage (6/17/2008) due to a programming error
  - Microsoft Azure’s outage (3/17/2009) for 22 hours due to the malfunction in the hypervisor
Why Is this Important (cont)?

- **Virtualization** plays a role of enabling technology for the Cloud core architecture, e.g.,
  - Amazon’s EC2 uses Xen hypervisor,
  - Microsoft’s Azure uses Windows Azure Hypervisor
  - IBM cloud employs KVM hypervisor

- **Accidental failures and malicious attacks** can result in significant down time (lower availability) of systems and applications

- Important to characterize the **resiliency** of different cloud infrastructures
Outline

• Validation of virtualization environment in cloud infrastructure using fault injection

• Failure Diagnosis using message flow reconstruction and targeted fault injection
CloudVal: A Framework for Validation of Virtualization Environment in Cloud Infrastructure
Approach

Use a software implemented fault injection (SWIFI) framework (CloudVal) to automate the process of conducting fault injection-based experiments for black box testing and reliability/security evaluation of virtualized environments.
Fault Injection Framework

- Design principles:
  - Automatable, repeatable experiment
  - Realistic, representative, extendable fault models
  - Architecture independent, applicable for actual systems

- Extend NFTAPE fault injection framework

Fault Models

• Guest system misbehavior
• Soft errors
• Performance faults
• Maintenance faults (e.g., turn off a certain part of the hardware to reduce the power consumption)
FM 1: Guest system misbehavior

• Rationale
  – Test failure isolation between the guest system and the host system

• Simulation of the incident
  – Randomly corrupt memory of the kernel of the guest OS to generate failures in a guest VM
FM 2: Soft errors

• Rationale
  – Test resiliency to single and multi-bit memory errors
  – Today’s 1GB ECC protected memory has the same uncorrectable error rate as yesterday’s 32MB parity protected memory

• Simulation of the incident:
  – Inject multi-bit errors into virtualization layers (user and kernel) and host OS (for hypervisor type 2)
FM 3: Performance faults

• Rationale
  – Virtualization facilitates sharing resources
  – Sharing can cause race condition
  – Simulate I/O delay, cpu exhaustion

• Simulation of the incident
  – Insert delay time in application threads
FM 4: Maintenance faults

• Rationale
  – Online replacement of hardware components due to hard errors
  – Turn off a certain part of the hardware to reduce the power consumption

• Simulation of the incident
  – CPU hot-plug/unplug
  – Memory hot-add, hot-swap
Steps to conduct experiments

1. Identify targets
2. Define fault models
3. Setup automated experiments
4. Run experiment and collect data
5. Analyze experiment result
Identify targets

- Hypervisor: KVM (Xen – not depicted)
- Management system:
  - Virt-manager (http://virt-manager.et.redhat.com/)
Identify fault models

- Guest misbehavior
- Soft errors in User mode
- Performance faults
- Soft errors in Kernel mode
- Maintenance events: - CPU core ON/OFF
Experiment Set up

- ApacheBench VMs create connections and send HTTP GET requests to HTTP server VM.
- HTTP server VM accepts connections and replies requested pages.
## Experimental Results

<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>
</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</td>
<td>Guest kernel crashes</td>
<td>No fault manifestation from guest to host is observed</td>
</tr>
<tr>
<td></td>
<td>Xen</td>
<td>XEN</td>
<td></td>
<td></td>
<td>Data from [3]: Found two cases that the failures manifested from the guest to the host</td>
</tr>
<tr>
<td>Turn one CPU core ON</td>
<td>KVM</td>
<td>Physical CPU core</td>
<td>10/10</td>
<td>N/A</td>
<td>10 kernel crashes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU core in guest VM</td>
<td>10/10</td>
<td>No change</td>
<td>No Change</td>
</tr>
<tr>
<td></td>
<td>Xen</td>
<td>CPU core in Dom0</td>
<td>10/10</td>
<td>No change</td>
<td>No change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU core in guest VM</td>
<td>10/10</td>
<td>Guest kernel returns error, but executes normally</td>
<td>N/A</td>
</tr>
<tr>
<td>Soft errors (user space)</td>
<td>KVM</td>
<td>Data segment, stack segment and register of QEMU-KVM PROCESS</td>
<td>496/500 faults</td>
<td>N/A</td>
<td>120 qemu-kvm crashes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 qemu-kvm defunct state) state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No Kernel crashes</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>
</tr>
<tr>
<td></td>
<td></td>
<td>xenstored</td>
<td>100/100</td>
<td>No change</td>
<td>No change</td>
</tr>
</tbody>
</table>
# Experimental Results

<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>
</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</td>
<td>Guest kernel crashes</td>
<td>No fault manifestation from guest to host is observed</td>
</tr>
<tr>
<td></td>
<td>Xen</td>
<td>XEN Data from [3]: Found two cases that the failures manifested from the guest to the host</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Turn one CPU core ON</td>
<td>KVM</td>
<td>Physical CPU core</td>
<td>10/10</td>
<td>N/A</td>
<td>10 kernel crashes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU core in guest VM</td>
<td>10/10</td>
<td>No change</td>
<td>No Change</td>
</tr>
<tr>
<td></td>
<td>Xen</td>
<td>CPU core in Dom0</td>
<td>10/10</td>
<td>No change</td>
<td>No change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU core in guest VM</td>
<td>10/10</td>
<td>Guest kernel returns error, but executes normally</td>
<td>N/A</td>
</tr>
<tr>
<td>Soft errors (user space)</td>
<td>KVM</td>
<td>Data segment, stack segment and register of QEMU-KVM PROCESS</td>
<td>496/500 faults</td>
<td>N/A</td>
<td>120 qemu-kvm crashes 26 qemu-kvm defunct (zombie) state No Kernel crashes</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>
</tr>
<tr>
<td></td>
<td></td>
<td>xenstored</td>
<td>100/100</td>
<td>No change</td>
<td>No change</td>
</tr>
</tbody>
</table>
## Experimental Results (2)

<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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Soft errors</strong></td>
<td>KVM</td>
<td>Data segment, code segment and register of KVM KERNEL MODULE</td>
<td>380/1000</td>
<td>N/A</td>
<td>94 kernel crashes</td>
</tr>
<tr>
<td></td>
<td>Xen</td>
<td>Crashing Dom0 by inserting a faulty kernel module</td>
<td>20/20</td>
<td>All guest VMs stopped</td>
<td>20 kernel crashes</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>KVM</td>
<td>Threads in QEMU-KVM PROCESS</td>
<td>399/400 faults</td>
<td>DoS during injected fault</td>
<td>16 kernel crashes</td>
</tr>
<tr>
<td>faults</td>
<td>Xen</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**ECE ILLINOIS**
# Fault Analysis: Multi-threads delay

**Observation:**
- Non-deterministic behavior
  - kernel crashes only when injecting delay to 3-4 threads
- Kernel call traces show the involvement of KVM module

**Potential cause:** Race condition in KVM module

<table>
<thead>
<tr>
<th>Run No.</th>
<th>Trigger point</th>
<th># injected thread</th>
<th>Delay time</th>
<th>Duration time</th>
<th>Activated/Injected faults</th>
<th>Number of kernel crashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sampled addresses</td>
<td>1, 2, 3, or 4</td>
<td>rand(100) seconds</td>
<td>rand(200) seconds</td>
<td>399/400</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Only one address that crashed the kernel in run 1</td>
<td>1 or 2</td>
<td>rand(100) seconds</td>
<td>rand(100) seconds</td>
<td>80/80</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Same address as run 2</td>
<td>3 or 4</td>
<td>rand(100) seconds</td>
<td>rand(100) seconds</td>
<td>100/100</td>
<td>3</td>
</tr>
</tbody>
</table>
Summary

• Fault injection based evaluation of VM environments
  – failure isolation mechanisms, maintainability, and completeness of the implementation of the two hypervisors and the management system
  – fault/error models: guest system misbehavior, soft errors, performance faults, and maintenance faults

• Key results
  – virtual machine ≠ physical machine: a guest system may behave differently in different virtualization environments
  – Turning on CPUs/cores may lead to the hypervisor failure (e.g., crash of KVM-hypervisor when turning on a CPU core)
  – Bugs in the hot-plugging mechanism of the virtual CPU (e.g., unable to turn on the virtual CPU in Xen) can make system vulnerable to malicious attacks
Failure Diagnosis using Message Flow Reconstruction and Targeted Fault Injection
Approximate Fault Localization: Concept

Assumption: Similar failure profiles imply similar faults
Approximate Fault Localization: Approach

• Upon a failure in a system collect a failure profile
  – e.g. in terms of sent and received messages

• Process failure profile to reconstruct an end-to-end processing flow corresponding to the failure
  – a sequence of system events across distributed components invoked to process a user/application request

• Use the reconstructed processing flow to query against a pre-constructed failure profiles stored in Failure Profiles Database
  – Use “string edit distance” metric to identify similar flows and “pinpoint” the fault location
Message Flow Reconstruction and Comparison

• Need to represent event flows so to enable fast identification of similar flows

• Event flows translated into event strings
  – an event in a string represented as a letter that corresponds to the source component of this event, e.g., BBABCABCB
  – event order based on timestamps

• Compare flows using *String Edit Distance*
  – the minimum number of insertion, deletion, or replacement of a letter required for changing one string into the other
Enabling Techniques

- **Distributed Events Tracing**: record system events (e.g., syscall, library call) in distributed systems

- **Message Flow Reconstruction and Comparison**: quantify the dissimilarity between failure profiles

- **Targeted Fault Injection**: deterministically inject faults at exact locations in the execution flow of a distributed system
Evaluation

- **Target:** OpenStack, an open source distributed cloud management system

- **Validation**
  - Do similar failure profiles imply similar faults?

- **Evaluation of AFL Accuracy**
  - Identification of fault type and affected component(s)
  - *Fault distance* between the determined fault location and the actual injected fault (Top-K nearest faults)
    - fault distance measured as the number of *libc* calls between the determined approximate fault location and the actual fault location in the end-to-end flow of fault-free execution
Construction of Failure Profile Database (FPDB)

- The FPDB is constructed for VM Provision (nova boot) requests
- Five failure profiles collected for each fault
- Fault models:
  - Contained faults: *Process crash, deadlock* (within a process)
  - Propagated faults: *Message corruption*

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Location Type</th>
<th>F</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Crash</td>
<td>All monitored libc calls</td>
<td>23323</td>
<td>116589</td>
</tr>
<tr>
<td>Message Corruption</td>
<td>All <em>read, write, send,</em> and <em>recv</em> libc calls</td>
<td>18221</td>
<td>91092</td>
</tr>
<tr>
<td>Deadlock</td>
<td>All thread and lock related libc calls</td>
<td>2143</td>
<td>10702</td>
</tr>
</tbody>
</table>
Do Similar Failure Profiles Imply Same faults?

More than 80% of all the injected faults, across all three fault models, result in less than 4% of the failure profile variation.
Accuracy of AFL: Top-K Nearest Faults for Known Faults

50% of the Top-1 query results contain the exact fault locations, i.e., fault distance is zero.
**Accuracy of AFL: Top-K Nearest Faults for Unknown Faults**

- **K=1**
- **K=5**
- **K=10**
- **K=20**

Two orders of magnitude better than OpenStack’s error reporting mechanism
Evaluation of FPDB against Real Bugs

- Use information on bug fixes included in the OpenStack Grizzly 2013.1 released in April 2013.
- Reproduce each bug and capture the bug traces in testing environment
  - 14 bugs (out of 766) selected
- Information (from bug reports) on bug locations as the ground truth to compare against the locations returned from querying FPDB

- Measures: use Top-10 queries to find injections that generated closest traces to each reproduced trace of selected bugs
- Compute fault distance (in terms of the number of LibC function invocations) between each returned injected location and the locations where the code was fixed for each bug
Query Results

- Reproduced six VM provision-, four VM resize-, and four VM migration-related bugs
- 8 out of 14 queries returned at least one trace that had fault distance within 100 LibC invocations
- Identified fault-to-failure propagation paths provided useful indicators for locating and fixing actual bugs

<table>
<thead>
<tr>
<th>Request type</th>
<th>VM Provision</th>
<th>VM Resize</th>
<th>VM Migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest fault distance (in LibC calls)</td>
<td>2  14  18  221  236  565</td>
<td>14  56  84  245</td>
<td>34  45  342  442</td>
</tr>
</tbody>
</table>
Example of Real Bug Report

LibvirtBridgeDriver crashes when spawning an instance with NoopFirewallDriver

Bug #1050433 reported by on 2012-09-13

Bug Description

I am trying the LibvirtHybridOVSBridgeDriver, with the up-to-date devstack and the vlan manager.

Required system configuration

I've created one network for the demo project:
nova-manage network create net2 --fixed_range_v4=172.16.2.0/24 --
um_networks=1 --network_size=256 --vlan=1001

I boot an instance:
nova boot --flavor l --image 000f2e5b-e1d1-4bc0-a9d1-3e07d527f7f1 vm_1

the instance can't be launched due to the following error:

2012-09-13 15:14:06 ERROR nova.compute.manager [req-a10f2d0d-4992-4b50-a297-b7b188ab98a1 demo demo] [instance: b6fab81d-d700-4a4a-bb3f-88422e3bc0f] Instance failed to spawn

OpenStack error messages
Example of Localizing a Real Bug

- **Reported Bug:** LibvirtBridgeDriver crashed when spawning an instance with NoopFirewallDriver

- **Localization using FPDB:** closest failure profile to that bug generated by message corruption fault to *write()* LibC function invoked by *nova-compute process*.
  - write() sends network filter configuration from the nova-compute to the socket of libvirtd (executs VM provisions)

- Injected fault corrupted the name of the network filter

```
Original value:
nova-compute 7fad40b08700 6208 write (15, AF_FILE =>/var/run/libvirt/libvirt-sock) = 74: <filter name='nova-instance-instance-ID-MACADDRESS' chain='root'>

New value (the 33rd character is corrupted):
nova-compute 7fad40b08700 6208 write (15, AF_FILE =>/var/run/libvirt/libvirt-sock) = 74: <filter name='nova-instance-instance-ID-MACADDRESS' chain='root'>
```
Example of Localizing a Real Bug

• Tracing back data flow of that message content in the execution of nova-compute found that
  – configuration for a network filter is generated regardless of whether the NoopFirewallDriver option is selected
  – implementation of the NoopFirewall driver is not intended to recognize a network filter configuration
  – an exception prevents VM provision

• Network configuration function had to be fixed to validate the NoopFirewallDriver option before generating a network filter configuration
Limitations

• Not effective for failures that do not alter the processing flow, e.g., a bug that only causes a performance degradation

• No weights assigned to different parts of the processing flow.

• Developers need to select the right answer based on the ranking (trace distance) provided by the localization method.

• Approximate bug locations indicate a context, in which a bug might occur

• Overhead of tracing in production systems
Summary

- Develop low-cost method for the approximate fault localization
  - reduce the cost of fault diagnostic while providing precision close to the methods used for the exact fault localization
  - support large complex distributed environments such as the cloud computing
- Demonstrate effectiveness of the prototype implementation on the OpenStack
  - effective in determining (approximate) fault/error locations
  - accurate in identifying the failure types and the affected components
Recommended Readings


A. Stern, “Update from Amazon Regarding Friday’s S3 Downtime,” CenterNetworks (Feb. 2008).


Recommended Readings


