First International Workshop on
Virtualization Performance: Analysis, Characterization, and Tools
Austin, Texas, April 2008.

Challenges and Opportunities with Fault Injection in Virtualized Systems

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

Abstract

We analyze: (i) use of virtualization to facilitate fault injection into non-virtualized systems, and (ii) use of fault injection to evaluate the dependability of virtualized systems. With the Xen Virtual Machine Monitor (VMM) as a test case, for (i), we injected thousands of faults into the code, memory, and registers of paravirtualized and fully-virtualized Virtual Machines (VMs) from within the VM and from the VMM. We describe and resolve challenges related to the implementation of result logging, page table injection, and the use of performance counters in VMs. For (ii), we injected into the VMM, privileged/driver VM, and unprivileged VMs. We used multiple types of fault injections in VMs to evaluate the isolation among VMs and demonstrate the value of fault injection for VMM validation.

1. Introduction

Software-implemented fault injection (SWIFI) is commonly used for evaluating and characterizing system dependability features [24, 9, 17, 10]. The challenges in implementing SWIFI include: minimizing the ''intrusion'' [4] on the behavior of the system under test due to the injection mechanisms or the logging of test results, the need to adapt the injector to different versions of the operating system (OS) of the system under test, and providing the ability to target individual processes or different parts of processes while also providing the ability to inject faults into any part of the entire system.

Virtualization technology allows multiple Virtual Machines (VMs), each with its own Operating System (OS), to run on a single physical machine [22, 5]. A software layer, called a Virtual Machine Monitor (VMM), manages the execution of VMs on the physical hardware. Virtualization is commonly used for server consolidation, software development and deployment, and system security and dependability research. As the use of virtualization has increased, there has been increasing interest in utilizing virtualization to improve system dependability [6, 12, 21]. It is thus important to have SWIFI tools for evaluating and characterizing the dependability features of virtualized systems.

Even when the goal is to evaluate non-virtualized systems, virtualization can be used to simplify SWIFI [23, 25]. This simplification falls into three key categories: 1) the virtualization layer between the OS and the hardware can be used to minimize the modifications to the system under test, 2) virtualization simplifies the management of the injection campaign and collection of results, and 3) virtualization provides ''sandboxing'' that isolates the actions of the system under test, preventing it from harming the host/control environment.

For evaluating complete virtualized systems, the SWIFI tool must be able to inject faults into the individual VMs as well as the VMM. Injection into a VM is used not only for evaluating the VM itself but also for testing the isolation among VMs provided by the VMM and the resiliency of the VMM to arbitrary faulty behavior of guest VMs. The VMM and the privileged/driver VM [5] are critical to the operation of a virtualized system. Hence, characterizing the behavior under faults of the VMM and the privileged VM is particularly important.

This paper presents a systematic analysis of the interactions between SWIFI and system virtualization. To the best of our knowledge, such analysis has not been previously presented. As a test case, we use an experimental setup consisting of the Xen [5] VMM and VMs running Linux. For evaluating non-virtualized systems, we explore the potential benefits of virtualization when the fault injection is done from within the system under test and compare this setup with injection performed from the VMM. We compare injection into paravirtualized VMs to injection into fully-virtualized VMs. We describe key difficulties with implementing injection due to the virtualization mechanisms and show how these difficulties can be overcome. We present results from stressing the isolation properties of the VMM, demonstrating the value of such tests by uncovering bugs in the VMM. We present results from injecting faults into the VMM and compare them with injection into the Linux kernel.

Brief reviews of SWIFI and system virtualization are presented in Sections 2 and 3, respectively. Section 4 discusses the interaction between SWIFI and system virtualization. Key issues related to the design and implementation of the UCLA *Gigan* SWIFI tool are presented in Section 5. The experimental setup is discussed in Section 6 and the results are presented in Section 7. Related work is discussed in Section 8.

2. Software-Implemented System Fault Injection

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

A fault injection can be triggered by breakpoints or various hardware counters, such as CPU cycles, instructions retired, cache misses, etc [9]. Injection targets include registers and memory of individual processes, the kernel, or the system as a whole.

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

System fault injection campaigns require the ability to automate the fault injection process, collect log files, and refresh the target system state [15]. The most difficult step to automate is the reboot of the system on a crash or hang and restore the system image to a clean state before running the next injection. Tools such as Kdump [14] and LKCD [29] help automate kernel crash logging and reboot.

SWIFIs are typically implemented in a kernel module so that access to privileged system state is allowed. As a kernel module, information about all user processes and critical kernel data structures are available and can be used for targeted fault injection. In addition, hardware resources such as debug registers and performance counter mechanisms can be used to trigger fault injection.

3. System Virtualization

There are two types of virtual machines: paravirtualized (PV [22]) and fully-virtualized (FV) [11]. The key difference between the two is that PV VMs are aware that they are running within a virtual system while FV VMs are not. PV VMs are modified (paravirtualized) to incorporate this knowledge and the modification must be carried over to newer generations of the kernel. Dedicated hardware mechanisms are required to support FV VMs [27]. While hardware support for virtualization is becoming common [27], a PV VM is still needed to serve as a special Privileged VM that interacts with the VMM and manages the entire virtualized system. The Privileged VM is also used to access shared devices on behalf of the VM. Furthermore, PV VMs can be made to run on hardware that is not fully virtualizable and can be more efficient since privileged operations can be batched [2].

A critical functionality of the VMM is to isolate VMs from each other so that activities in one VM cannot affect another VM [21]. Isolation is maintained by controlling accesses to shared hardware resources such as CPU, memory, and I/O devices. Maintaining correct control of the memory resources is challenging since valid memory mappings for each VM must be maintained and every memory mapping update through the page tables must be validated to make sure no VMs can have access to other VM's or VMM's memory regions.

4. Fault Injection and Virtualization

This section describes how the implementation and deployment of SWIFI interacts with virtualization. It includes design choices and tradeoffs regarding how, for evaluating non-virtualized systems, the simplifications outlined in Section 1 impact the components of SWIFI deployment (Section 2). While there are key advantages to using virtualization to implement SWIFI, there are also some non-trivial challenges that are outlined. The SWIFI mechanisms needed and challenges faced when the goal is to evaluate *virtualized* systems are also presented.

4.1. Virtualization Facilitating SWIFI

When the goal is to evaluate a non-virtualized system, the system under test is run in a VM. The mechanism to trigger and inject faults can be run within the VM, as it does when injection is done without using virtualization (Section 2). An alternative is to trigger and perform the injection from the VMM. Figure 1 shows the basic setup of injection in a VM, illustrating a modified VM for implementing injection from within the VM, an unmodified VM that can be injected into from the VMM, a VMM that supports injection and a privileged VM that controls the campaign.

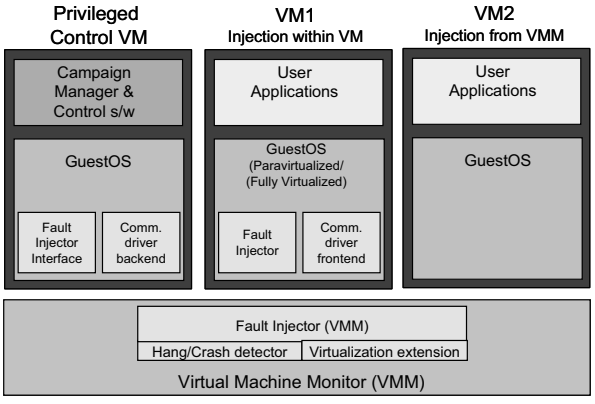

Figure 1: Fault injection in a virtualized system

There are many options for implementing injection from inside a VM. These are essentially the same options as for implementing injection in a non-virtualized system running

on a bare machine [17, 24]. The most common approach that minimizes overhead and maximizes flexibility involves the use of dedicated kernel modules and taking advantage of hardware performance/debug registers [9]. This requires detailed knowledge of the OS and porting effort for each target OS and even for different kernel versions of the same OS. For example, when we ported our fault injection kernel module from version 2.4 of the Linux kernel to version 2.6, we found that significant effort was required due to differences in the details of how interrupt handlers are managed. On the other hand, with virtualization there is the option of performing the injection from the VMM in an OS-agnostic fashion. For example, injection into registers and memory of a VM can be done without any modifications to the kernel running in the VM. Injection to different operating systems can thus be done without any knowledge of the internal structures of the operating systems and without any porting effort.

Without the use of virtualization, logging system events and application outputs typically requires a connection to a different physical system over a network and/or a direct connection to an I/O device (e.g., a disk) where results are written. This can impact the injection results — for example, use of the kernel for logging test information can actually cause a system crash that would not have otherwise happened. Furthermore, with some mechanisms, such as logging to a local disk, the faulty system may actually corrupt the log, making further analysis impossible. Virtualization creates an opportunity to implement light-weight mechanisms for logging information from the system under test to another VM on the same physical machine. Such mechanisms minimize the probability that the logging process itself changes the behavior of the system (intrusion). The isolation (‘‘sandboxing’’) among VMs provided by the VMM protects the logging results.

Virtualization also facilitates diagnosing the impact of faults on the system under test. For example, from the VMM it is easy to determine whether the VM kernel is continuing to perform context switches among processes in the VM, and thus provide more information than that the system has simply ceased to function/respond. While similar diagnosis is possible with a non-virtualized system running on a bare machine [28], without virtualization the diagnosis mechanism itself and the results of the diagnosis are vulnerable to corruption.

Running fault injection campaigns often involves repeating experiments thousands of times in order to obtain statistically significant results. Each experiment requires starting with a pristine system, unaffected by any prior faults. It also requires the ability to continue running the experiment after the system under test crashes or hangs. Without virtualization, accomplishing these tasks is quite complex — for example, it may require a hardware mechanism to reset a system that hangs or time-consuming operations such as copying complete disk images across the network before every experiment. With virtualization, the system under test operates from disk images and it is simple and fast to start with a new disk image every time. The ability to restart a VM from a checkpoint can be used to speed up testing by skipping over the system boot delay for every experiment [25]. With virtualization, using software without any non-standard hardware devices, it is easy to detect crashes and hangs of the system under test, force a cleanup of the previous experiment and restart a new experiment.

4.2. Implementation Challenges

As discussed above, there are clear advantages to using virtualization for fault injection into non-virtualized systems. However, there are also significant challenges. These challenges fall into four main categories: 1) the need to accurately emulate errors that conflict with VMM mechanisms designed to protect the system from ‘‘misbehaving’’ VMs, 2) the need to avoid compromising the security of the VMM and other VMs in the system, 3) the need to ‘‘virtualize’’ hardware performance/debug registers used for injection and measurements, and 4) the need in some campaigns to target specific processes or parts of processes in the system.

The VMM is designed to prevent a ‘‘misbehaving’’ VM from harming the VMM or other VMs. This is often referred to as performance isolation [19] and fault isolation. The mechanisms used to enforce isolation often make it difficult for errors caused by fault injection in VMs to accurately emulate errors caused by corresponding faults on a bare machine. For example, in order to preserve isolation, the VMM must prevent the VM from accessing the memory of other VMs or of the VMM itself. This is typically done by preventing the VM from directly modifying the page table used by the hardware. When the VM attempts to change a page table, the VMM maps (*virtualized*) the change so that the proper functionality is provided without compromising the integrity of other VMs. Similarly, if memory fault injection hits page tables of the VM, the effects of the injection must be *virtualized* in the sense that the possible consequences (errors) must correspond to what can happen in a non-virtualized system running on a bare machine. For example, a fault that affects the page frame number stored in a page table of the VM, should be able to cause future accesses in that VM to map to different pages of the VM or to nonexistent pages, but not to pages of other VMs or the VMM. When the goal is to evaluate non-virtualized systems, it is not meaningful for a fault in the VM to affect other VMs or the VMM. Details of this for our test case is discussed in Section 5.

The challenges to accurate emulation of errors described above are tightly coupled with the need to avoid compromising the security of the VMM and other VMs. As

a practical matter, in order to run fault injection campaigns, the stability of the host system must be maintained. When the goal is to evaluate non-virtualized systems, allowing the fault in the VM under test to corrupt other VMs or the VMM can compromise the experiment. When injection is done from within a VM, with a properly designed VMM, the injection cannot harm the virtualized system hosting the injection. However, when injection is done from the VMM, there is no such protection. An example of this is memory injection. If a VMM-based injector randomly modifies any page that is part of the memory of the VM, it may modify a page table entry, thus allowing the VM to later access memory that belongs to other VMs or to the VMM. This could be the case in a paravirtualized system, where the page tables used by the VM are readable but not writable by the VM.

In order to avoid the security problems described above, the injector needs to be aware of the semantics of sensitive structures that are targets of injection. Specifically, if the injection is into a page table, a ''random bit flip'' may have to be *virtualized* by making sure that any possible error is properly mapped to affect only the behavior of the VM under test. Otherwise, such faults can potentially crash the host. Details of this for our test case are discussed in Section 5.

Commercial processors include registers dedicated to performance evaluation and debugging. Fault injectors often use these registers for triggering injection — for example, a breakpoint in the code or some number of elapsed cycles [9]. These registers may also be used for measurements — for example, measuring the number of clock cycles between the injection of a fault and its manifestation as a system crash. Conventional VMMs currently do not virtualize these registers, i.e., there is a single set of registers and they measure all activity on the host. In order to use these registers for injection into a particular VM, the VMM must save and restore them at every exit from or entry to the VM.

Fault injection campaigns often involve targeting specific processes or parts of processes (e.g., the stack segment) as well as specific parts of the kernel address space. This is relatively easy to do when the injection is performed from inside the VM. However, as discussed above, injection from inside the VM has significant disadvantages. In order to perform targeted injection from the VMM, the injector must posses detailed knowledge of the internal data structures and internal operation of the kernel running in the VM. Hence, it is unlikely that implementing such targeted injection from the VMM is a good choice. Thus, while injection into the entire VM is best performed from the VMM, targeted injection should still be done with traditional injectors within the VM, enhanced by logging and campaign management features provided by the virtualized host.

4.3. Evaluating Virtualized Systems

As outlined in Section 1, in order to evaluate complete virtualized systems the SWIFI tool must be able to inject into the individual VMs as well as the VMM. Injection into individual VMs is useful for evaluating the extent to which the VMM protects itself from arbitrary behavior of VMs as well as for evaluating mechanisms implemented by the VMM to tolerate VM failures [6, 12, 21]. Injection into the VMM or into a privileged/driver VM (e.g., Dom0 in Xen[5]), is useful for understanding the impact of faults on these critical components.

The discussion in the previous two subsections covers most of the issues related to injection into VMs in order to evaluate the isolation properties of the VMM as well as any mechanisms it includes for tolerating VM failures. However, there are some fault injections to individual VM that are relevant here but are not relevant when the goal is simply to evaluate non-virtualized system. These have to do with interactions between VMs and the VMM that are specific to virtualized systems. Two key examples are explicit synchronous calls (*hypercalls*[5]) from paravirtualized VMs to the VMM and access to ''I/O pages'' that correspond to accesses to device controllers in a non-virtualized systems.

Hypercalls provide a way for a VM to make explicit requests from the VMM in the same way that system calls provide a way for normal processes to make requests from an OS kernel. A faulty or malicious VM can make arbitrary hypercalls to the VMM and it is the responsibility of the VMM to ensure that it is not damaged by these calls. There has been extensive prior work on testing the resiliency of OS kernels by perturbing system call parameters [13, 9]. A SWIFI tool for evaluating VMMs must be able to similarly perturb hypercall parameters.

Access to ''I/O pages'' is a second example of VM-specific injection that is not directly relevant when the goal is to evaluate non-virtualized systems (unless the evaluation includes faulty hardware in the device controller). In a non-virtualized system, device drivers in the kernel may read/write from addresses that are mapped to device controllers. With a fully-virtualized VM, the same addresses are mapped such that accesses cause traps to handlers in the VMM that emulate the behavior of the device with respect to the VM and may cause the appropriate operations on host devices to be performed. If injection is performed from within the VM, this setup does not cause any problems — the behavior of the real system will be correctly emulated. With injection performed from the VMM, this scenario requires additional modifications. In particular, injections into the VMs address space that map to such I/O pages must be identified by the injector and mapped to invocations of the appropriate handlers in the VMM.

Fault injection into the VMM itself as well as into the privileged or driver VM suffer from the difficulties described

earlier in this section with respect to injection into non-virtualized systems running on bare physical machines. In particular, there are difficulties with logging of experimental results and managing injection campaigns. As discussed in Subsection 4.1, some of these difficulties can be handled by using virtualization. This requires running the VMM under test inside a VM running on a host VMM. With hardware support for virtualization [27] it is possible to run an unmodified VMM in a fully-virtualized VM. Unfortunately, the current hardware support does not allow the guest VMM to run fully-virtualized VMs. Thus, the use of a host VMM to facilitate the evaluation using SWIFI of a guest VMM is restricted to the case where the guest VMM is running only paravirtualized VMs.

5. Implementation Details

This section describes the implementation of critical components of our fault injector, focusing on key implementation challenges outlined in Section 4.

5.1. Page Table Injection Fidelity

Implementing memory fault injection in a virtualized system is challenging due to the existence of page and descriptor tables used by the processor to access physical memory. A random fault can occur anywhere in memory including into areas containing page and descriptor tables. Hence, the fault injector must be able to emulate errors that occur on a bare machine, but guarantee that the errors injected can only affect the target VM and not the entire system. Incorrectly implementing memory injection can erroneously give a VM access to the VMM's or another VM's memory area. We focus the following discussion on page tables as the solution for descriptor tables is similar.

The VMM virtualizes memory by controlling accesses to the VM's page tables. Some recent CPUs provide special support for memory mapping in virtualized systems [3]. However, most current CPUs do not provide such support and require special software mechanisms for implementing paging for VMs. For PV VMs, the VM accesses the actual page table used by the processor. The VM can read this page table normally but can update it only through the VMM, using hypercalls. We refer to such page tables as *guarded page tables*. The PV VM maintains a second table, henceforth called the *GtoM table*, that maps *guest page frames* to *machine page frames*. With full virtualization, there are two page tables for each VM. The FV VM accesses a *guest page table* but the actual page mapping is performed using the *shadow page table*, which is maintained by the VMM. We note that while PV VMs can also use shadow page tables for page-table virtualization, this is not the standard supported paging mode used for PV VMs in Xen, our test platform.

In a real machine, the available page frames are contiguous. In a virtualized system, however, machine page frames assigned to a VM are not necessarily contiguous in the host physical memory. Hence, another level of mapping is used. The VM maps its virtual pages to contiguous *guest page frames* and uses this mapping in its memory management operations. Page table management operations require translating the guest frame numbers (GFNs) to machine frame numbers (MFNs). With PV VMs, this is done using the GtoM mapping. FV VMs rely on shadowing, which transparently maps GFNs to MFNs.

The main issue when injecting into a page table page is how to reflect changes made to the VM's view of the page table to the actual page table while maintaining correctness in terms of isolation and overall system stability. A page table entry contains a page frame number (PFN) and a control bit field. Focusing on PFNs, the following paragraphs discuss each issue in turn for PV and FV VMs.

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

Reflecting the injection when the shadow paging mode is used is conceptually similar to the guarded page tables. Injecting into the PFN must be done from the perspective of the VM's memory layout before reflecting the injection into the shadow page table. Xen already has a shadow page table management mechanism to safely and correctly reflect changes by a VM to the VM's page table to the shadow page table. This mechanism is re-used to reflect the changes made by fault injection into the guest page table to the shadow page table.

Correctness and system stability when performing page table injection is maintained by ensuring that the resulting PFN obtained after fault injection will not give the VM access to memory that belongs to another VM or the VMM. For VMs that use the guarded page tables, these safety checks must be implemented as part of the injector. As mentioned above, for VMs that use shadow paging mode, the injector can leverage the shadow page table management code of Xen to perform the validation.

On a bare machine, a bit flip in the page table may map a virtual page to a non-existent physical page. These cases can be emulated by using a page frame number that is also outside the physical memory range of the host.

There are additional scenarios where injecting faults into page tables or page directories can lead to fault isolation violation. Figure 2 depicts these scenarios. In scenario *A*, an injection into a page table entry that once contained a writable mapping to a data page can change it to map to a page table page, thus giving the VM writable permissions to that page table. This can potentially give the VM access to any page in the system's memory. Scenario *B* depicts the case where an entry in a page directory is injected into so that it now points to a normal data page. This data page will now act as page table and random data in that page will now be interpreted as mappings to arbitrary pages in memory. To fully resolve these problems, a shadow must be kept of the injected page. All writes to this shadowed page will be intercepted by Xen to do appropriate emulation or redirection. We plan to implement this in future work.

Besides the page frame number field, each bit in the control bit field of a page table entry must be examined to make sure no isolation issues exist. For example, on x86, the global page and super page bits cannot be safely injected into since they can affect the behavior of the VMM itself. We have yet to implement support for injecting into all control bits of a page table entry. We have implemented page table injection to the PFN and read/write control bits.

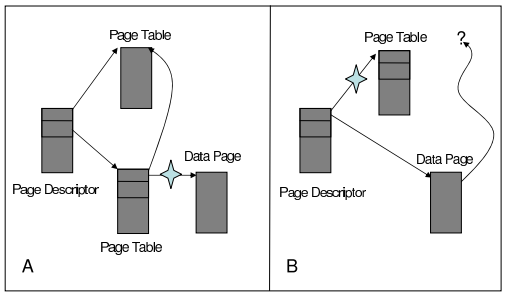

Figure 2: Page table injection scenarios violating fault isolation. Star means overwritten mapping.

5.2. Logging in a Virtualized System

Our fault injector includes a light-weight, low-overhead logging mechanisms (*Comm Driver*), implemented using shared memory between VMs. We utilize Xen's split driver API to construct the Comm Driver frontend and backend. The backend is placed in the privileged VM while the frontend is placed in the target VM, either PV or FV. Before a fault injection run, the frontend uses the grant table mechanism of Xen to set up shared pages with the backend. During an injection run, logs are generated directly into the shared memory instead of through the filesystem or network which can be intrusive. This has the added benefit of capturing logs as they are being generated and immediately storing them into a reliable place, without having to rely on a possibly faulty target VM to store the logs.

5.3. Tickstamps and Performance Counters

In order to measure crash latency, we use the tickstamp and performance counter registers. There are difficulties when trying to measure active CPU cycles inside both PV and FV VMs. In PV VMs, reading the tickstamp to calculate elapsed CPU cycles will also include the execution time of the VMM unless the tickstamps are virtualized. Instead we use performance counters to count CPU cycles for a PV VM when measuring crash latency since the performance counters can be set to count non-VMM ring levels. For FV VMs, Xen virtualizes tickstamps only to maintain consistency with the rate of timer interrupt delivery. Since timer interrupt delivery in Xen is not necessarily proportional to VM running time, tickstamps can not accurately measure FV clock cycles. To achieve higher fidelity tickstamp measurements, we implemented a separate tickstamp virtualization mechanism in Xen by accumulating the differences between VM entry and exit times.

Performance counters (PMCs) are also used to trigger fault injections. We virtualized the PMCs by having Xen save/restore the PMC registers and the associated control and event-selector registers across VM context switch, to provide the VM and their kernel-level fault injectors the functionality of a bare hardware.

6. Experimental Setup

Our fault injection campaigns yielded three classes of results: (1) characterization of non-virtualized systems under faults (using virtualized systems), (2) evaluation of performance and fault isolation among VMs and between a VM and the VMM when one of the VMs is subjected to faults, and (3) evaluation of the VMM and the privileged VM under faults. It should be noted that the results presented in this paper are only intended to highlight the usefulness of our tool. We plan to do a more thorough evaluation in future work.

Faults were injected into the code, memory, and registers of PV and FV VMs from both a kernel-level SWIFI and a VMM-level SWIFI. The impact on the injected VM were recorded, yielding Class 1 results. Furthermore, to evaluate the impact of logging techniques on fault injection analysis, we compared a typical logging setup consisting of the network, filesystem, and serial console with our Comm Driver. These experiments also yielded some Class 2 results — evaluation of the impact of faults in a VM on the VMM. Additional Class 2 results were obtained by running and monitoring a ‘‘control’’ VM with an I/O-intensive benchmark while injecting faults into hypercall arguments of another VM or, in a separate experiment, injecting faults into the kernel code of the other VM. For Class 3 results, faults were injected into the code, memory, and registers of the VMM and into the registers of the privileged/driver VM.

We used Xen 3.1.0 and ran Linux 2.6.18 for all our VMs

on Intel x86 machines. For Class 1 and 3 results, we ran a subset of programs in UnixBench [1] as our benchmark in the target VM with minor modifications to improve logging and failure detection. The selected programs were chosen for their ability to stress the Linux kernel. In experiments for Class 2 results, there was a target VM and control VM. The target VM benchmark ran UnixBench. The control VM ran a filesystem and network stressing benchmark.

A single injection run consisted of: restoring the VM filesystem, launching the VM, running the benchmark, injecting a single fault, collecting logs, and destroying the VM. The privileged VM controlled the fault injection experiments.

The outcome of each injection experiment was classified as either a crash, a hang, an incorrect result, or not manifested. A crash occurs when the observed component of a system explicitly dies or stops working. Hang occurs when the observed component of a system stops responding with no explicit report of a crash. An incorrect result occurs when the logs from the benchmark are corrupted or missing. Not manifested means no errors were observed.

We also measured the crash latency, which is the time between the injected fault and system crash, not including the time to execute the kernel's crash handler. In addition, we recorded the cause of system crash, such as a null pointer dereference or general protection fault. This information along with the failure distribution was used to characterize the behavior of the system under fault.

Our fault model is a single-bit flip. Breakpoints were used to trigger code injection. Memory and register injections were triggered after some random number of kernel or VMM instructions. For code injection, we profiled the kernel running the benchmark to collect the most frequently used functions and selected their instructions as injection targets [15]. The same was done to select instructions for VMM code injection.

Memory injection targeted a random memory location of a VM or VMM. Register injection targeted the general purpose, stack, instruction pointer, and system flags registers. We only report results for a single general purpose register, as we found very similar results from injection into different general purpose registers. We have not yet experimented with injection into other system registers, such as the control registers, segment selectors, descriptor table registers, and model-specific registers. While we do not anticipate major problems with such injection for FV targets, with PV targets injecting into these registers can affect the entire (host) system. For example, injecting into the Protection Enable bit in the CR0 register which disables/enables protected mode can affect the behavior of the VMM itself.

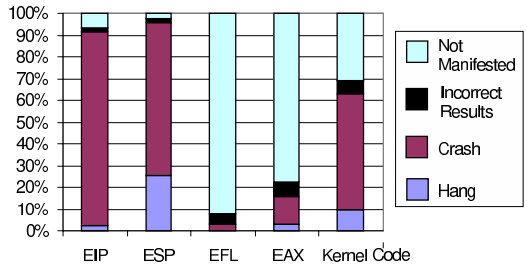

Figure 3: Failure distribution - register and code using VMM-level injector into FV VM.

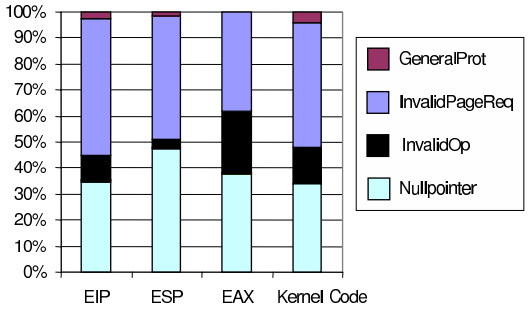

Figure 4: Crash distribution - register and code using VMM-level injector into FV VM.

Table 1. Crash latency - register and code using VMM-level injector into FV VM. Shows the crash latency percentage within an interval. For example: EIP - 55% crash between 5H-1K cycles.

	5H	1K	2K	3K	10K	100K	1M	10M	100M	1B	>1B
EIP	19%	55%	11%	3%	3%	0%	1%	0%	3%	5%	0%
ESP	21%	31%	16%	2%	3%	8%	7%	4%	2%	2%	3%
EAX	40%	30%	10%	4%	6%	4%	4%	1%	1%	1%	1%
Kernel Code	11%	44%	19%	3%	4%	4%	6%	3%	2%	2%	3%

7. Results and Evaluation

7.1. Evaluating Non-virtualized Systems

We found that the results from fault injection were similar regardless of whether we used a kernel-level or VMM-level injector and regardless of whether the target was a PV VM or an FV VM (see below regarding differences between injection to a PV VM vs. an FV VM when the target was the stack pointer register). Hence, in this section we only show and discuss results for VMM-level fault injection into FV VMs. Random injection into physical memory of a VM resulted in only 1% failures, which is consistent with results from [23]. Hence, we do not yet have sufficient results from memory injection for meaningful detailed analysis.

Figure 3 shows the failure distribution with a VMM-level fault injector injecting into the registers (1200 fault injections per register) and code segment (30,000 injections)

of an FV VM. Figure 4 and Table 1 show the crash cause and latency distribution, respectively. We do not include EFLAGS (EFL) in the crash analysis since the number of crashes for EFL is too small for evaluation. The key results are as follows:

- ESP (stack pointer register) injections with a PV VM cause more crashes than ESP injections with an FV VM (95% vs. 70%) since Xen takes a more active role in the operations of PV VMs. Xen performs checks to verify the consistency of the PV VM state, and if these checks fail, Xen proactively crashes the VM.
- The failure distribution, crash latency, and crash causes for code injection are comparable to [15, 16], which performed similar fault injection experiments on bare machines. There are minor differences which could be attributed to our experimental setup using a 2.6 kernel instead of 2.4. This at least shows that the results for kernel code injection using a virtualized system are comparable to those on a bare machine.

Other Linux kernel injection results are:

- Faults in EIP and ESP are very likely to lead to crashes.
- EFL has many unused bits which can explain the high percentage of non-manifested outcome.
- The major reason for crashes caused by register and code injection is illegal memory access.
- High percentage of InvalidOp crashes for EAX could be assertion violations. Assertions are implemented with an invalid opcode instruction in the Linux kernel.
- Other crash causes such as divide-by-zero or iret exception are rare.
- Crashes for EAX and EIP occur within 2K cycles as opposed to the ESP register which depends on the frequency of stack access.
- Code injection cause the most crashes within 1K-2K cycles. Cases where latency is much longer than 2K might be attributed to changing, for example, the branch condition of the instruction which might not cause an immediate crash.

7.2. Impact of Logging Mechanism

We evaluated the effect of logging overhead on the fidelity of the injection results. We ran two experiments: one using the Comm Driver for logging results from the target VM, and another using a virtual serial console and external filesystem image for logging results. In both cases we injected into registers in FV VMs.

We found that while the failure outcomes show similar numbers for incorrect results and not-manifested, the Comm Driver reported a significantly greater percentage of crashes 70% vs 50% and smaller percentage of hangs 25% vs 44% for the ESP register. We determined that fewer crash logs were recovered by the virtual serial console than with the Comm Driver, leading to misclassification of the outcomes by our analysis scripts. This is due to the lower complexity of the Comm Driver, which made it less likely to be corrupted than the virtual serial console driver. These experiments demonstrate the need for low-intrusive logging as complex logging techniques can skew injection results.

7.3. Performance and Fault Isolation

As part of an evaluation of the resiliency of the VMM, it is important to determine whether faults in one VM can significantly degrade the performance of another VM. It is even more important to determine whether faults in one VM can cause a non-faulty VM or the VMM to behave incorrectly. As discussed in Section 6, we injected faults into one VM and observed the impact on the VMM and a control VM.

Although our experiments were by no means exhaustive, we discovered two cases where simply injecting faults into a VM caused the VMM to fail. In the first case, register injection corrupted an I/O state flag used and accessible by the FV VM and Xen for I/O emulation. The fault caused Xen to crash due to a bug in the way Xen's VM crash code interacts with bad VM I/O requests/replies. In the second case, a corrupted instruction pointer in the kernel of a FV VM caused a re-execution of the CPU initialization code used during VM boot. In this code the Advanced Programmable Interrupt Controller, which is virtualized by Xen, sends an Inter-Processor Interrupt to initialize all Virtual CPUs (VCPUs) including the sender VCPU. This resulted in the current VCPU trying to reinitialize itself, which the code in Xen does not handle correctly, causing a hang of the VMM.

The two fault isolation violations described above demonstrate the importance of being able to systematically evaluate the resiliency of a virtualized system using a fault injector.

As discussed in Section 6, we ran some experiments with both a target VM, into which faults were injected, and a control VM. In one set of experiments we performed 30,000 kernel code injections into the target VM while the control VM executed an Apache server. For each request, the web server performed an RSA encryption of a static web page and then delivered in its response an MD5 checksum of the encrypted page. In a second set of experiments, we injected random single bit flips into random hypercall arguments of the target VM. In this case, the control VM executed a program that stressed the file system by reading in large chunks of data and writing them back out. In each experiment of this set, injection commenced after a random duration starting from experiment initiation. Injection continued until the target VM either crashed or hanged. We performed tens of thousands of hypercall injections, focusing

on the ten most-used hypercalls in Xen.

Results from both sets of experiments described above did not show any performance degradation in the control VMs despite faults being injected into the target VM. In addition, even with such a high rate of injection into hypercalls, we did not find any major errors in the VMM. However, we did discover a minor bug in Xen activated only when asserts are enabled. In the *do_iret* hypercall, Xen updates a new value for the saved guest code segment register from the guest stack prior to verifying its validity. When asserts are enabled, Xen sees that its view of the guest state is corrupted and crashes itself. For this set of experiments, the VMM does a good job at isolating the faulty VM from affecting the performance of the other VMs.

7.4. VMM and Privileged VM Injection

The target system in these experiments was a dual-CPU FV VM running an ‘‘inner’’ instance of the Xen VMM. In other words, we ran Xen on top of Xen. The ‘‘inner’’ Xen executed two single-CPU PV VMs each running UnixBench. The privileged VM (Dom0) running on the ‘‘inner’’ Xen ran a looper program to ensure VM scheduling code was being exercised in the target VM (‘‘inner’’ Xen). We used the VMM-level fault injector located in the ‘‘inner’’ Xen to inject faults into the ‘‘inner’’ Xen’s code and memory areas as well as into the ‘‘inner’’ Xen’s Dom0’s registers. When injecting into the ‘‘inner’’ Xen’s registers, we used a fault injector located in the ‘‘outer’’ host Xen.

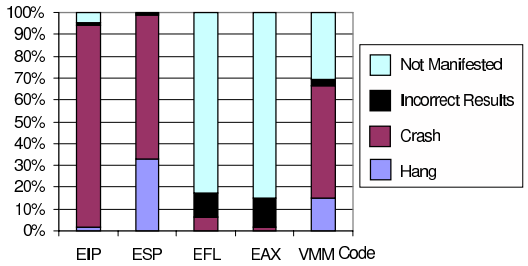

Figure 5: Xen register and code injection.

Figure 5 shows the results from fault injection into Xen’s code and registers. A comparison between Xen’s failure distribution for register and code injection with the Linux kernel show similarities, despite Xen being less complex. We do not show the details of the VMM’s memory injection because our random memory injection experiments have yielded only a small percentage of failures, 2%. The failure distribution of register injection into the ‘‘inner’’ Xen’s Dom0 show 18% incorrect results and 82% not-manifested. The incorrect results for Dom0 is much higher than other injection experiments since targeting Dom0 can cause drivers to fail and cause the benchmark to be unable to log injection results. Crashes and hangs are at 0% because in this experiment, we are only interested in Xen’s behavior when Dom0 is faulted. Further investigation is needed to fully understand the behavior of the Xen VMM under faults.

8. Related Work

There is extensive prior work on software-implemented fault injection [10, 17]. Fault injection into the Linux kernel is discussed in some detail in [16, 15, 18]. However, there is relatively little prior work focused on using virtualization for fault injection or injecting into virtualized systems.

Starting in 2001, a sequence of related papers proposed and expanded on the use of VMs to facilitate low-cost fault injection for evaluating non-virtualized systems[7, 23, 8, 20]. These papers discussed the design, implementation, and use of a hardware emulator to run a VM into which faults targeting devices, CPU registers, and memory can be injected. Benefits of using virtualization for fault injection such as the ease of automation, and reduced intrusion and interference of the fault injector on the target system are also presented. However, these papers did not discuss the challenges of accurately emulating errors in page tables and system control registers when performing fault injection in virtualized systems. They also did not analyze the impact of the logging mechanisms on the fidelity of the injection results nor the use of virtualization to facilitate low intrusion logging. Furthermore, these papers did not discuss the tradeoffs among the different fault injector configurations possible in a virtualized system.

Subkraut, et al. discussed the benefits of using fast snapshot and rollback mechanisms in VMs to decrease the time to perform a single fault injection run when evaluating the robustness of software APIs [25]. Swift, et al., used VMs to facilitate fault injection automation and to speedup fault injection experiments for evaluating OS reliability [26]. These papers did not focus on the challenges of implementing fault injection in virtualized systems.

The effectiveness of different VMMs in providing performance isolation between multiple hosted VMs is analyzed by Matthews, et al. in [19]. They did not use fault injection to evaluate the performance and fault isolation between VMs and the VMM.

9. Conclusions and Future Work

This paper explored the interaction between fault injection and virtualization: using virtualization to facilitate the evaluation of non-virtualized systems and deploying SWIFI as a method for evaluating the resiliency of virtualized systems. We used our VMM-level and kernel-level injectors to inject faults into PV and FV VMs, the privileged VM, and the VMM itself. Key results and conclusions include: (1) mechanisms for logging injection campaign outcomes may affect the results and virtualization facilitates low intrusion, low overhead logging; (2) fault injection into VMs from kernel-level and VMM-level yield similar results; (3) due to (2) above, kernel-level injectors are

needed only for injections that target specific VM kernel internals; (4) outcomes from injection to the stack pointer demonstrate that with PV VM injection there is greater opportunity for undesirable impact of the VMM on the results than with FV VM injection; (5) error isolation testing of the VMM is important, as our results demonstrate multiple violations in the Xen VMM; and (6) when the goal is to accurately emulate the effects of faults in page tables on a bare machine, correct implementation of injection from the VMM level involves significant complexity with PV as well as FV VMs.

Future work will include: 1) injections into virtualized systems running FV VMs, 2) targeted injections to critical locations of the VMM, 3) more targeted injections into the privileged VM, and 4) injections into a virtualized system on a bare machine.

Acknowledgments

Work related to virtualization at the UCLA Concurrent Systems Laboratory is supported, in part, by a gift from HP Laboratories. Comments from Yoshio Turner and the anonymous referees helped us improve this paper.

References

1. ‘‘UnixBench,’’ *www.tux.org/pub/tux/benchmarks/System/unixbench*.

2. K. Adams and O. Agesen, ‘‘A Comparison of Software and Hardware Techniques for x86 Virtualization,’’ *12th International Conference on Architectural Support For Programming Languages and Operating Systems*, San Jose, CA, pp. 2-13 (October 2006).

3. AMD, ‘‘AMD64 Architecture Tech Docs Volume 3,’’ *www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf* (September 2007).

4. J. Arlat et al., ‘‘Comparison of Physical and Software-Implemented Fault Injection Techniques,’’ *IEEE Transactions on Computers* **52**(9), pp. 1115-1133 (September 2003).

5. P. Barham et al., ‘‘Xen and the Art of Virtualization,’’ *Nineteenth ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, pp. 164-177 (October 2003).

6. T. C. Bressoud and F. B. Schneider, ‘‘Hypervisor-Based Fault-Tolerance,’’ *Computer Systems* **14** (1), pp. 80-107 (February 1996).

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

8. K. Buchacker et al., ‘‘Hardware Fault Injection with UMLinux,’’ *International Conference on Dependable Systems and Networks, Fast Abstracts*, San Francisco, CA, p. 670 (June 2003).

9. J. Carreira et al., ‘‘Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers,’’ *IEEE Transactions on Software Engineering* **24**(2), pp. 125-136 (February 1998).

10. J. Carreira et al., ‘‘Fault Injection Spot-Checks Computer System Dependability,’’ *IEEE Spectrum* **36**(8), pp. 50-55 (August 1999).

11. Y. Dong et al., ‘‘Extending Xen with Intel Virtualization Technology,’’ *Intel Technology Journal* **10**(3) (2006).

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

13. J.-C. Fabre et al., ‘‘Assessment of COTS Microkernels by Fault Injection,’’ *Seventh IFIP Working Conference on Dependable Computing for Critical Applications*, pp. 25-44 (January 1999).

14. V. Goyal et al., ‘‘Kdump, A Kexec-based Kernel Crash Dumping Mechanism,’’ *lse.sourceforge.net/kdump/documentation/ols2oo5-kdump-paper.pdf* (2005).

15. W. Gu et al., ‘‘Characterization of Linux Kernel Behavior Under Errors,’’ *International Conference on Dependable Systems and Networks*, San Francisco, CA, pp. 459-468 (June 2003).

16. W. Gu et al., ‘‘Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors,’’ *International Conference on Dependable Systems and Networks*, Florence, Italy, pp. 887-896 (June 2004).

17. M. Hsueh et al., ‘‘Fault Injection Techniques and Tools,’’ *IEEE Computer* **30**(4), pp. 75-82 (April 1997).

18. T. Jarboui et al., ‘‘Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques,’’ *International Conference on Dependable Systems and Networks*, Washington, D.C., pp. 331-336 (2002).

19. J. N. Matthews et al., ‘‘Quantifying the Performance Isolation Properties of Virtualization Systems,’’ *Workshop on Experimental Computer Science*, San Diego, CA, p. 5 (June 2007).

20. S. Potyra et al., ‘‘Evaluating Fault-Tolerant System Designs Using FAUmachine,’’ *Workshop on Engineering Fault Tolerant Systems*, Dubrovnik, Croatia (September 2007).

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

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

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

24. D. T. Stott et al., ‘‘NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors,’’ *4th International Computer Performance and Dependability Symposium*, Chicago, IL, pp. 91-100 (March 2000).

25. M. Subkraut et al., ‘‘Fast Fault Injection with Virtual Machines,’’ *International Conference on Dependable Systems and Networks, Fast Abstracts*, Edinburgh, UK (June 2007).

26. M. M. Swift et al., ‘‘Improving the Reliability of Commodity Operating Systems,’’ *ACM Transactions on Computer Systems* **23**(1), pp. 77-110 (February 2005).

27. R. Uhlig et al., ‘‘Intel Virtualization Technology,’’ *IEEE Computer* **38**(5), pp. 48-56 (May 2005).

28. L. Wang et al., ‘‘An OS-Level Framework for Providing Application-Aware Reliability,’’ *12th Pacific Rim International Symposium on Dependable Computing*, pp. 55-62 (December 2006).

29. D. Wilder, ‘‘LKCD Installation and Configuration,’’ *http://lkcd.sourceforge.net/* (2002).