Fault Injection in Virtualized Systems – Challenges and Applications

Michael Le and Yuval Tamir, *Member, IEEE*

Abstract—We analyze the interaction between system virtualization and fault injection: (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. We explore the benefits of using virtualization for fault injection and discuss the challenges of implementing fault injection in virtualized systems along with resolutions to those challenges. For experimental evaluation, we use a test platform that consists of the *Gigan* fault injector, that we have developed, with the Xen Virtual Machine Monitor. We evaluate the degree to which fault injection results obtained from running the target system in a virtual machine are comparable to running the target system on bare hardware. We compare results when injection is done from within the target system vs. from the hosting hypervisor. We evaluate the performance benefits of leveraging system virtualization for fault injection. Finally, we demonstrate the capabilities of our injector and highlight the benefits of leveraging system virtualization for fault injection by describing deployments of Gigan to evaluate both non-virtualized and virtualized systems.

Index Terms—Fault injection, virtualization, hypervisors, reliability

1 Introduction

Developing and evaluating mechanisms for enhancing the resilience of computer systems to faults requires an ability to induce faults and monitor their effects. Fault injection techniques and tools provide this ability[19]. With software-implemented fault injection (SWIFI), this can be done without specialized expensive hardware and with the capability to target specific system components [38], [9], [19], [8]. The challenges in implementing SWIFI include: minimizing the "intrusion"[1] on the behavior of the System Under Test (SUT) 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 SUT, and providing the ability to target individual processes or different parts of processes while also enabling the injection of faults into any part of the system.

System virtualization allows multiple Virtual Machines (VMs), each with its own OS, to run on a single physical machine [34], [3]. A software layer, called a Virtual Machine Monitor (VMM), manages the execution of VMs on the physical hardware. System 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 [5], [12], [33], [21]. It is thus important to have SWIFI tools for evaluating and characterizing the dependability features of virtualized systems.

Virtualization can simplify SWIFI even for evaluating non-virtualized systems [36], [39]. 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 SUT, 2) virtualization simplifies the management of the injection campaign and collection of results, and 3) virtualization provides "sandboxing" that isolates the actions of the SUT, preventing it from harming the host/control environment.

For evaluating complete virtualized systems, a SWIFI tool should be able to inject faults into the individual VMs as well as the VMM. Injection into a VM is used not only for evaluating the VM itself but also for testing the isolation among VMs provided by the VMM and the resiliency of the VMM to arbitrary faulty behavior of guest VMs. The virtualization infrastructure (VI), consisting of the VMM, privileged and driver VMs [3], [14], is critical to the operation of a virtualized system. Hence, characterizing the behavior of the VI under faults is particularly important.

The main contribution of this work is the 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 practical embodiment of the ideas discussed, we present the design and implementation of a fault injector, called *Gigan*[18], that leverages the Xen VMM[3] to inject faults into VMs and the VMM. The key contributions of this paper are:

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

Brief reviews of SWIFI and system virtualization are pre-

• *M. Le and Y. Tamir are with the Concurrent Systems Laboratory, Computer Science Department, UCLA.*
E-mail: {mvle,tamir}@cs.ucla.edu

Manuscript received October 6, 2013; revised June 11, 2014.

sented in Sections 2 and 3, respectively. Section 4 discusses the interaction between system virtualization and SWIFI. Key issues related to the design and implementation of the UCLA *Gigan* SWIFI tool are presented in Section 5. The experimental setup and evaluation of the use of virtualization for fault injection are presented in Sections 6 and 7, respectively. Practical examples of deploying *Gigan* to evaluate different systems are presented in Section 8. Related work is presented in Section 9.

2 SWIFI

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

SWIFIs are typically implemented in a kernel module to enable access to privileged system state. A fault injection can be triggered by breakpoints or various hardware counters, such as CPU cycles, instructions retired, cache misses, etc[9]. Injection targets include registers and memory of individual processes, the kernel, or the system as a whole.

When fault injection is used to evaluate a system, logs containing the parameters of the injection as well as the outcomes from the benchmarks and error messages from the OS must be saved in a "safe" place, where the information cannot be corrupted. The challenge is to generate and reliably collect these logs without perturbing the normal operation of the system and skewing the injection results. A related challenge is obtaining the state of the SUT at failure time for postmortem analysis — failure may make much of this state (e.g., memory) inaccessible.

System fault injection campaigns require automating the fault injection process, log files collection, and refreshing the target system state[16]. A particularly difficult step to automate is the reboot of the system on a crash or hang and restoration of the system to a clean state before the next injection. Tools such as Kdump[15] and LKCD[45] help automate kernel crash logging and reboot.

3 System Virtualization

System-level virtualization allows multiple VMs, each with its own OS, to run on a single physical computer[34]. The *virtualization infrastructure* (VI) consists of all the software components involved in multiplexing hardware resources among VMs.

The Xen[3] VI is composed of three components: virtual machine monitor (VMM), privileged VM (PrivVM/Dom0), and driver VM (DVM). The VMM isolates VMs from each other so that activities in one VM cannot affect another VM[33]. Isolation is maintained by controlling accesses to shared hardware resources such as CPU, memory, and I/O devices. The PrivVM is used to perform system management operations such as creating, destroying, and checkpointing VMs. Special dedicated DVMs are used to directly control devices and provide VMs with shared device access[14].

There are two types of virtual machines: paravirtualized (PV)[34] and fully-virtualized (FV)[11]. 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[41]. While hardware support for virtualization is becoming common[41], PV VMs are still needed to serve as the PrivVM and DVMs.

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

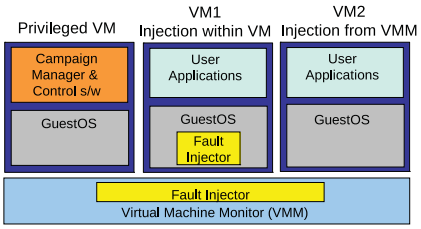

Fig. 1. Fault injection in a virtualized system.

4 Virtualization and Fault Injection

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

4.1 Virtualization Facilitating SWIFI

When the goal is to evaluate a non-virtualized system, the SUT is run in a VM. Triggering and injection of faults can be from within the VM, as is the case without virtualization. Alternatively, it can be done from the VMM. Figure 1 shows a modified VM (VM1) for injection from within the VM, an unmodified VM (VM2) that can be injected into from the VMM, a VMM that supports injection, and a privileged VM that controls the fault injection campaign.

The options for implementing injection from inside a VM are essentially the same as in a non-virtualized system [19], [38]. A common approach is based on dedicated kernel modules and 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. On the other hand, with virtualization, injection can also be performed from the VMM into an unmodified VM, in an OS-agnostic fashion, thus avoiding porting the injector to different OSs. Furthermore, since the target VM is unmodified, the intrusion caused by the fault injector (impact of the fault injector on the behavior of the target system) is minimized.

Injection into unmodified VMs can also be performed from the PrivVM[31]. This requires VMM modifications to

support the setting of injection triggers and synchronization between the target VM and the PrivVM. A key drawback of this approach is that it cannot be used to inject faults directly into the VMM when running on bare hardware, which is crucial for evaluating virtualized systems (Subsection 4.4).

Without virtualization, logging system events and application outputs typically requires a network connection to a different physical system and/or a connection to an I/O device (e.g., a disk), where results are written. This can impact the injection results. For example, logging information by the kernel can cause a system crash that would not have otherwise happened. If information is logged to a local device, the faulty system may corrupt the log, making further analysis impossible. Virtualization can mitigate this problem by enabling light-weight mechanisms for logging from the SUT 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 among VMs protects the logging results.

There are mechanisms, such as Kdump[15], that can help save the kernel state following a crash for postmortem analysis. However, without virtualization, the state may be lost since system corruption may prevent the crash kernel from starting, or interfere with the retrieval or storage of the desired state even if the crash kernel does start. If the SUT is in a VM, the SUT state can be preserved by suspending the VM and then analyzed from, for example, the PrivVM, without relying on any services from the failed SUT.

Virtualization also facilitates diagnosing the impact of faults. For example, from the VMM it is easy to determine whether the VM kernel is continuing to perform context switches[24] or whether it has halted all of its CPUs. Hence, it is possible to obtain 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[44], without virtualization the diagnosis mechanism itself and its outputs are vulnerable to corruption.

Computer systems are not completely deterministic since they are affected by the interleaving of asynchronous events (e.g., interrupts). Hence, repeated injection of the same fault can yield different results. Thus, to obtain statistically-significant results, fault injection campaigns often require many injection runs. Each run starts with a pristine system, unaffected by any prior faults. The system state includes the state on hard disks and the states of any other hardware devices. A pristine disk state needs to be obtained from a "safe" place, not writable during the previous run. To restore other hardware devices to a pristine state, a system reboot is typically required. Thus, each run consists of: (1) restoring disk state, (2) booting the system, (3) running the workload, (4) injecting a fault, and (5) collecting injection logs.

Campaigns consisting of multiple injection runs require the ability to continue after the SUT crashes or hangs. Without virtualization, this may require specialized hardware support. For example, IPMI[37] may be needed to allow a remote system to reset the SUT. On the other hand, with virtualization, a simple script, running on the host, can destroy the VM with the SUT and create a new VM for the next run.

In a non-virtualized system, without specialized hardware to write-protect local storage, safe restoration of disk state requires copying disk images across a network prior to each run. This copying step is likely to involve an extra system boot (network boot). With virtualization, the SUT can operate from disk images stored as files on the host. The virtualization layer guarantees that pristine disk images, stored in the host system, cannot be corrupted by a faulty VM. Hence, a simple script on the host can restore the disk images used by the SUT to their pristine states with a local file copy.

4.2 Fault Injection Performance

Since it is often necessary to run fault injection campaigns for days and perform thousands of injections, the performance of a fault injector is critical to the overall usefulness of the tool. This performance is affected by each of the five steps described in the previous subsection.

The disk state restoration step is likely to be significantly faster when the SUT is in a VM since: A) a local copy of a disk image file is likely to be faster than a copy of a disk image across the network, and B) the extra network boot required in the non-virtualized system is avoided.

The time to boot the system depends on: the speed at which the BIOS performs POST and then finds and loads the OS and file system images, the number of hardware devices that the OS probes and configures during boot, and the number of services the OS starts. A VM is usually configured with fewer and simpler devices than a typical physical machine. This difference is not inherent and is correctable with the appropriate hardware and software configurations. However, as a practical matter, the VM is likely to boot faster. In addition, with a virtualized system, the entire boot process can be skipped by starting the system from a checkpoint[39]. With sufficient effort, a similar mechanism can be set up with bare hardware. However, with a virtualized system, this capability is typically already implemented in the existing VI.

Regarding the workload execution time (step (3)), the advantage, if any, is with bare hardware and is dependent on workload characteristics. For example, the virtual system may be significantly slower if the workload performs many page table manipulations and the host hardware does not include support for page table virtualization[43].

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

In summary, without specialized hardware and software configurations, it can be expected that fault injection campaigns can be performed significantly faster using virtualized systems than with bare hardware. Our experience with this issue is briefly discussed in Subsection 7.4.

4.3 Implementation Challenges

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

Modern processors include registers useful for performance evaluation and debugging, containing counts of various system events, breakpoint addresses, etc. 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 collecting measurements, such as crash latency. To obtain accurate results when the SUT is in a VM, the performance counter registers must be accurately virtualized by the VMM — allow the measurement of VM activity *only*, despite the frequent transitions between the VM and VMM that occurs during normal operation. Accurate virtualization of tickstamps (cycle count) in Gigan is discussed in Section 5.4.

Fault injection campaigns often target 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 SUT. However, such targeted injection requires detailed knowledge of the internal data structures and operation of the kernel running in the SUT. Hence, it is more complex to implement such injection from the VMM or PrivVM.

SWIFI's ability to emulate errors caused by hardware faults is limited by the processor's ISA[9]. For example, SWIFI cannot directly inject faults into caches or registers that are not accessible to software. These limitations are not eliminated when the SUT is in a VM. In fact, when the SUT is in a VM, there are unique challenges with the use of SWIFI to accurately emulate error due to hardware faults. These challenges are generally caused by conflicts with VMM/CPU mechanisms designed to protect the system from "misbehaving" VMs. In this context, "accuracy" refers to the extent to which the behavior due to a fault when the SUT is in a VM is similar to the behavior when the SUT is running on bare hardware. The rest of this subsection is focused on these challenges.

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

With the SUT in a VM, the accuracy of emulating errors caused by faults affecting page tables depends on how the VM's page tables are managed. For example, if the VMM shadows the VM's page tables (page table installed in hardware is different from the page table accessed by the VM), then simply injecting faults into the page tables accessed by the VM will not accurately emulate the errors caused by corresponding faults on a bare machine. This is because the actual page tables being used by the hardware for that VM would not be affected by the fault. Details of how to accurately inject into page tables are discussed in Section 5.2.

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

The problem discussed in the previous paragraph can also be explained as a consequence of the fact that VMMs do not typically accurately emulate all possible faulty VM states. Fully accurate emulation would be difficult to do and would not be useful for normal operation. One example of such a state is the result of a fault that sets the VM-8086 bit in the system flags register of an FV VM running on an Intel CPU. To be consistent with Intel's specification, when this bit is set, the limit field of the segment registers must be set to a specified value. If the segment registers are not properly set and the injection is performed from the VMM, when VM entry is attempted, it fails and the VMM terminates the VM. There is no similar consistency check on a bare machine. Hence, on bare hardware, a similar fault is likely to manifest differently, most likely as memory corruption or a page fault.

4.4 Evaluating Virtualized Systems

In order to evaluate complete virtualized systems, the SWIFI tool must be able to inject into VMs as well as the VMM. Injection into individual VMs is useful for evaluating the extent to which the VMM enforces isolation (protection of the entire system from arbitrary VM actions) as well as for evaluating mechanisms implemented by the VMM to tolerate VM failures [5], [12], [33]. Injection into the components of the VI (VMM, PrivVM, DVM) is useful for understanding the impact of faults on these critical components.

The previous two subsections, which dealt with injection to a system in a VM, also cover most of the issues related to injection into VMs in order to evaluate the isolation properties of the VMM as well as mechanisms for tolerating VM failures. However, there are some fault injections to individual VMs that are relevant here but are not relevant

when the goal is 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*[3]) from PV VMs to the VMM and accesses to "I/O pages" that correspond to accesses to device controllers in a non-virtualized systems.

VMs use hypercalls to make explicit requests from the VMM in the same way that processes use system calls for requests from an OS kernel. The VMM must protect itself from hypercalls that may be issued by faulty or malicious VMs. There has been extensive work on testing the resiliency of OS kernels by perturbing system call parameters [13], [9]. A SWIFI tool for evaluating VMMs should be able to similarly perturb hypercall parameters.

In a non-virtualized system, device drivers in the kernel may read/write from/to addresses that are mapped to device controllers. With an FV VM, the same addresses ("I/O pages") are mapped such that accesses cause traps to handlers in the VI 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 to an I/O page, the behavior of the real system will be correctly emulated. However, when injection is performed into the VM's address space from the VMM, the injector must explicitly identify accesses to the VM's I/O regions and invoke the appropriate handlers.

Fault injection into VI components face the same difficulties described earlier in this section with respect to injection into non-virtualized systems 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 VI under test inside a VM. With hardware support for virtualization[41], it is possible to run an unmodified VMM in an FV VM. Current hardware support does not allow the guest VMM to run FV VMs. However, using nested virtualization techniques proposed in[4], it is possible to overcome this problem and allow FV VMs to run inside the guest VMM.

5 Injector Design and Implementation

We have implemented, *Gigan*, a fault injector for the Xen virtualized system that can inject a variety of faults into VMs and the VMM. Gigan includes SWIFI enhancements that utilize system virtualization. This section describes the design and implementation of critical components of Gigan, focusing on the challenges outlined in Subsection 4.3.

5.1 Basic Capabilities

Operations in Gigan are based on *triggers* and *actions*. *Triggers* are set to fire after some threshold has been reached or some event has occurred in the target machine. Triggers can fire based on timers, instruction breakpoints, process creation/termination, and performance monitoring events (e.g., CPU cycle count)[9]. Associated with each trigger is a set of one or more *actions* to be performed at the time the trigger is fired. To maximize flexibility, an action either injects a fault or sets another trigger. This ability to chain triggers increases the precision and range of when and where to perform the injection. For instance, an injection can be set to occur 2000 cycles after the function *foo* has been invoked.

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

5.2 Page Table Injection Fidelity

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

Injection into memory containing paravirtualized page tables is discussed in[23]. For FV VMs, with CPUs that provide hardware support for page table virtualization (nested page tables)[43], no special operations are needed when injecting faults into VM memory that may contain page tables. However, some CPUs do not provide such support and require software mechanisms for implementing paging for FV VMs. Specifically, shadow page tables is a commonly-used software technique for virtualizing page tables.

With shadow page tables, the VM accesses a *guest page table* but the actual page mapping is performed using a *shadow page table*, maintained by the VMM. When injecting into a VM's page table, the changes made must be reflected in the shadow page table, while maintaining correctness in terms of isolation and host system stability. With respect to the basic information in the page table, such as the page frame number (PFN) and protection bits, when injection is performed from within the VM, the page table shadowing mechanism reflects the changes from the guest page table to the shadow page table in a correct and safe manner. However, this is not automatically done if injection is performed from the VMM. In this latter case, the injector identifies accesses to page table pages and explicitly invokes the shadowing mechanism to reflect the changes in the shadow page tables.

For some control bits in page tables, the effects of faults are difficult to accurately emulate, even if nested page tables are used. An example of this is the *global page bit* that can cause TLB entries to not be flushed on a context switch. A fault that sets this bit may result in incorrect address mapping. If the SUT is in a VM, the manifestations of the fault may be different from those on bare hardware since the relevant TLB entry may be evicted due to the activity of other VMs.

5.3 Logging in a Virtualized System

Gigan includes a light-weight, low-overhead logging mechanism (*V2V Driver*), implemented using memory shared

between VMs. We utilize Xen's split driver API[14] to construct the V2V Driver backend and frontend, located, respectively, in the PrivVM and target VM (PV or FV). The frontend allocates memory for logs and shares it with the backend. During an injection run, logs are generated directly into the shared memory instead of through the filesystem or network, thus minimizing injector intrusion. This has the added benefit of capturing logs as they are being generated and being able to immediately store them in a reliable place, without having to rely on a possibly faulty target VM to store the logs. The risk of a fault causing corruption of the logs in the shared memory pages exists, but our experience indicates that such corruption is rare.

5.4 Measuring Latency in VMs

In the context of fault injection, accurate measurement of latency, e.g., crash latency, is often important. As explained in Subsection 4.1, it is desirable to run the SUT in a VM even if the goal is to evaluate system behavior on bare hardware. If this is done, the latency measurements need to be in terms of VM execution time. Since FV VMs most faithfully represent the environment on bare hardware, we focus the discussion on techniques for accurately measuring latency, in cycles, in FV VMs. Techniques for measuring latency in PV VMs can be found in[23]. In the rest of this subsection we first discuss the problem with latency measurement in a VM that relies on Xen's existing mechanisms. Our more accurate mechanism for latency measurement within a VM is then presented.

On Intel hardware, latency can be accurately measured using either the time-stamp counter (TSC) or the performance monitoring counter (PMC). The TSC counts the number of elapsed clock cycles in the CPU. The PMC can be configured to count the number of elapsed non-idle CPU cycles. When the target system is run in a VM, measuring latency with respect to VM execution time requires that the TSC and/or PMC be virtualized. It should be noted that the desired latency measurement is strictly of the execution time within the VM and does not include the time to execute software handlers invoked after the VM exit, outside of the VM.

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

With Intel's Virtualization Technology (VT), using "VM-Exit Controls for MSRs" and "VM-Entry Controls for MSRs"[20], the processor can be configured so that, upon VM exit, the values of certain machine specific registers (MSRs) are saved in memory and replaced by new values loaded from memory. Upon a subsequent VM entry, the saved values are loaded from memory to those MSRs. This mechanism is used by Xen to virtualize the PMC. Specifically, this is used for a PMC control MSR that contains a bit that controls whether the PMC continues to count. Thus, upon exit from a VM, the PMC stops incrementing and VMM software can save its value. This saved value can be loaded to the PMC before a subsequent entry to that VM. We have found that, with this mechanism, the latency measurements obtained are too high since the PMC continues to be incremented after exit is initiated and begins incrementing before entry completes (see below).

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

In recent Intel processors, "VM-Exit Controls" and "VM-Entry Controls" contain a bit that directly controls whether the PMC control MSR is loaded from memory upon VM exit and entry, respectively[20]. We have found that using this new mechanism, in place of the mechanism used by Xen, significantly improved the accuracy of the latency measurement in a VM. Specifically, the experiment described above yielded a result of 478.5M cycles.

A key reason for the remaining discrepancy between latency measurement within a VM and on bare hardware is that entering and exiting the VM is not instantaneous. The time measured in the VM still includes part of this entry/exit time (VM state load/save time). As described below, we have improved the accuracy of latency measurements by adding to Xen a mechanism to compensate for this VM entry/exit time.

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

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

instructions do increase the measured latency.

6 Motivation and Setup for Evaluation

Two of the important conclusions from Subsections 4.1 and 4.3, are, respectively: 1) there are significant advantages to running fault injection campaigns with the SUT in a VM, and 2) when the campaign needs to target specific parts of the SUT, it is advantageous to perform the injection from within the SUT. However, these advantages would be largely moot if, respectively: A) the results from such campaigns when the SUT is in a VM are significantly different from when the SUT is running on bare hardware, and B) the results when the injection is performed from within the SUT are (due to intrusion) significantly different from the results when injection is performed from outside the SUT.

Our experimental evaluation addresses the two issues above. We also evaluate the fault isolation provided by the VMM, the impact of logging, and injection performance. In total, these evaluation provide critical information and insight regarding the extent to which combining SWIFI with virtualization is practical and useful. To the best of our knowledge, prior works have not reported results addressing these issues. The rest of this section describes the experimental setups used for our evaluations.

The types of faults injected and the analysis of the outcomes were guided by the choices typically made in the relevant literature. SWIFI is commonly used for evaluating and characterizing system behavior under faults. This can be used in the development of mechanisms for improving system resiliency. The most commonly-used faults are bit flips in registers, data memory, and code. Key results of such analysis are breakdowns of the rates at which faults are not manifested or manifested as crashes, hangs, or silent corruption, as well as the latency of the manifestation of faults [7], [8], [13], [16], [17], [21]. The focus of our experimental evaluation is on analyzing different injection configurations, not on characterizing the behavior of any particular system under faults.

We use Xen version 3.3 and Linux version 2.6.18.8 in our experiments. Our test platforms consist dual quad-core 2.2GHz Intel Nehalem processors with 8GB of memory. The platforms are equipped with a NIC and a BMC card using IPMI[37] to allow the automation of the reboot of the system following a crash or hang.

Faults are injected during the execution of the Xen VMM in some experiments and during the execution of the Linux kernel in other experiments. Figure 2 shows the different target system configurations used in these experiments. Each target system executes on a single CPU. This single CPU setup is used to simplify the crash latency measurements discussed below. For the VMM target system, two PV Application VMs (AppVMs) are used in order to exercise the scheduling functionalities of the VMM. The VMM and Linux target systems are used in separate experiments. The different fault injection setups are shown in Figure 3.

The same user-level workload is used in all setups. This workload consists of a subset of programs from UnixBench[42], few of which are CPU-intensive programs with a majority being programs that exercise different OS

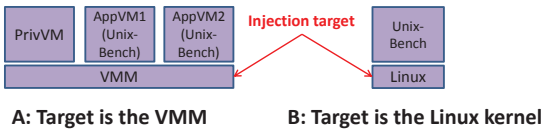

Fig. 2. Target system configurations.

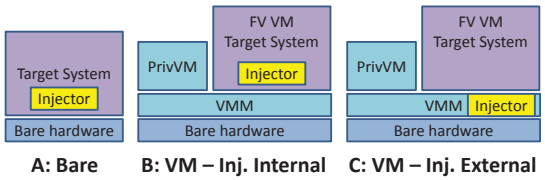

Fig. 3. Injection setup configurations. When the injector resides within the target system, for VMM target, the injector is in the VMM, for Linux target, the injector is in the Linux kernel.

subsystems such as process creation, system calls, and file system operations. These programs are slightly modified to improve logging and failure detection.

With the SUT on bare hardware (Fig. 3-A), kernel/VMM logs are sent through the serial console to a separate physical machine. User-level workload outputs are stored within the SUT. At the end of an injection run, if the SUT survives, workload outputs are sent across the network to a separate physical machine. When the SUT runs in a VM (Fig. 3-B/3-C), the V2V Driver is used for kernel/VMM logs and user-level workload outputs — even if the SUT crashes or hangs, the workload outputs as well as the kernel/VMM logs are accessible to the host at the end of the injection run.

Injection is performed into the CPU's EAX register (a representative general purpose register), stack (ESP) and instruction (EIP) pointers, and the system flags register (EFL). For EFL, injection is into all non-reserved bits, except for the VM-8086 bit (see Subsection 4.3). Faults are also injected into the VMM's and Linux kernel's stack segments. Register and stack segment injections are triggered a random number of instructions (VMM instructions when targeting the VMM or Linux kernel instructions when targeting the kernel) after the workload (UnixBench) has been started.

Single bit flip faults are also injected into the VMM and Linux kernel's code segments. Injection is triggered by a breakpoint on the target instruction at a random time after starting the workload. Hence, these code faults are activated immediately. The injection target instructions are randomly selected in functions that are identified by profiling as the most frequently executed functions in the VMM (Fig. 2-A) or kernel (Fig. 2-B)[16].

The outcome of each injection is classified as either a crash, an externally induced crash (ExtCrash), a hang, an incorrect result, or not manifested. A crash occurs when the VMM or Linux kernel explicitly dies or stops working due to a system panic caused by a fatal exception. Crashes are identified by analyzing the output of the panic handler. An ExtCrash can only occur when the SUT is run in a VM (Fig. 3-B/3-C). This is a scenario where the VMM explicitly crashes the VM with the SUT due to fatal corruption of VM state. A hang occurs when the VMM or Linux kernel stops responding with no explicit report of a crash. Hangs in the VMM (Fig. 2-A) are detected by the Xen VMM's built-

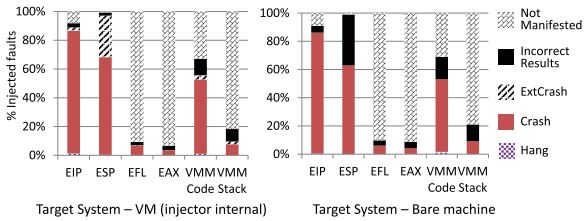

Fig. 4. Injection outcome distributions. Target is the VMM in FV VM (Fig. 3-B) vs. bare hardware (Fig. 3-A).

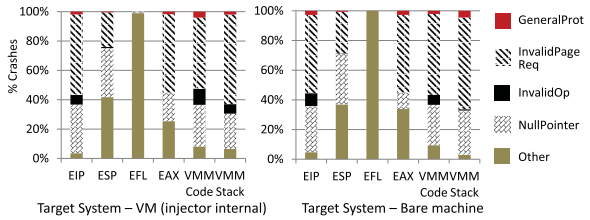

Fig. 5. Crash cause distributions. Target is the VMM in FV VM (Fig. 3-B) vs. bare hardware (Fig. 3-A).

in watchdog-based hang detector[26]. Hangs in the Linux kernel (Fig. 2-B), with the SUT running in a VM (Fig. 3-B/3-C), are detected by the host VMM when the VM is no longer performing scheduling[24]. Incorrect results are identified when there is no indication of a crash or hang and the user-level workload outputs are missing or do not match their expected contents. The injection outcome is classified as "not manifested" if no errors are observed.

Since the majority of manifested faults result in crashes, it is particularly useful to analyze the causes of crashes. Thus, we identify the main crash causes, such as null pointer dereference or general protection fault.

The time between the occurrence of a fault and the resulting system failure is often useful for system design[28] and is thus an important measure to obtain from fault injection campaigns[16]. Since the majority of manifested faults result in crashes, and hang detection latency greatly depends on the check interval of the hang detector, we focus on measuring crash latencies. The crash latency is measured from the time the fault is injected to the invocation of the software exception handler triggered by the fault. Since with code injection instructions modified by a fault are immediately executed, crash latency is measured from fault *activation* to the invocation of the handler triggered by the fault[17].

7 Fault Injection Evaluation

In this section, we analyze five aspects of performing fault injection in virtualized systems. First, we analyze the similarities and differences between results obtained when the SUT runs inside a VM and when the SUT runs on bare hardware. Second, we analyze the degree of intrusion caused when running the fault injector inside the SUT. Third, we analyze the extent to which using our simple, low intrusion logging mechanism affects fidelity of fault injection results. Fourth, we analyze the impact of using virtualization on the performance of fault injection. Finally, we discuss the extent to which the Xen VMM provides the fault isolation guarantees needed to support fault injection.

7.1 VM vs. Bare Hardware

As discussed in Section 4, there are significant advantages to using virtualization to evaluate systems that will ultimately run on bare hardware. A possible concern with such evaluation is that the virtualized hardware is likely to be different from the physical hardware on which the system will ultimately be deployed. Differences in hardware platforms affect not only what code is executed but also when the code is executed, due to variations in the interleaving of asynchronous events. These differences can potentially affect how faults are manifested and thus reduce the usefulness of results obtained when the SUT is in a VM.

This subsection is focused on whether the ways faults are manifested are significantly affected by whether the SUT runs in a VM vs. on bare hardware. Since the interaction with hardware is at the heart of the differences between the two injection configurations, injecting faults into SUT components that interact with hardware is most likely to reveal differences in the ways faults manifest. Hence, our experiments consist of injecting faults into the Xen VMM (Fig. 2-A) and the Linux kernel (Fig. 2-B). Specifically, the comparison is between results when the SUT in an FV VM, as shown in Fig. 3-B, and when the SUT runs on bare hardware (Fig. 3-A).

The injection targets are described in Section 6. The campaign is designed to obtain meaningful numerical results while keeping the number of injections manageable. Since the fraction of injected faults that are manifested as errors varies significantly depending on the injection target, the above campaign design goal results in a different number of injections for different targets. For each injection target and configuration, the number of injections is set so that the outcome distribution is stable. Specifically, when the same number of injections are repeated, the results are within two percentage points of the results obtained from the original set of injections. Based on this criteria, on bare hardware, the number of injections range from 300 into the EIP register to 2,700 into the EAX register. With the SUT in a VM, since injection runs are faster (see Subsections 4.2 and 7.4), for increased accuracy, the number of injections range from 750 into the EIP register to 3,600 into the EAX register.

Figures 4 and 5 show breakdowns of injection outcomes and crash causes from injections into the VMM (Fig. 2-A). The effects of faults when the SUT runs in a VM are similar to when the SUT runs on bare hardware. The few differences in the results shown are not due to significant differences in the effects of faults with the two platforms. Instead, as explained below, these differences are mostly due to differences in the logs that are obtained with the two platforms.

As shown in Figure 4, the main difference in injection outcomes between the VM and bare hardware setups are the results from injecting into the ESP register. Specifically, roughly 35% of faults are classified as manifested as incorrect results when the SUT runs on bare hardware, but as ExtCrashes when the SUT runs in a VM. The reason for this difference is that some faults can corrupt the VMM (e.g.,

TABLE 1
Cumulative distributions of crash latency measurements. Target is the VMM. FV VM (V) vs. bare hardware (B) — Fig. 3-B vs. Fig. 3-A. Example: for 44.1% of crashes, latency less than 2K cycles for injection into ESP in a VM.

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

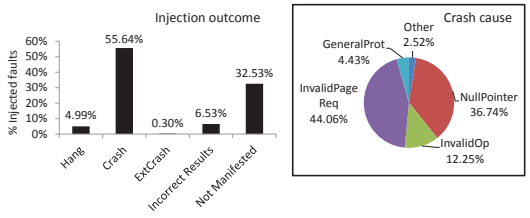

Fig. 6. Injection outcome and crash cause distributions. Target is the Linux kernel in an FV VM (Fig. 3.B).

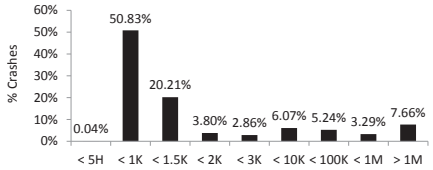

Fig. 7. Crash latency distribution. Target is the Linux in an FV VM (Fig. 3.B). For example: 20.21% of all crashes occur within 1K-1.5K cycles.

corrupt the interrupt descriptor table) to the extent that the VMM is unable to even invoke its panic handler. With bare hardware, no useful diagnostic information is logged, but the benchmark outputs are missing or incomplete. When the SUT runs in a VM, the host VMM detects and logs such VM crashes (e.g., triple fault), thus providing more information regarding the cause of the system hang or spontaneous reboot. This issue is likely to also be the cause of the small differences in outcomes with EIP, VMM code, and VMM stack injections.

There are additional possible causes of minor differences in the injection outcomes between the VM (Fig. 3-B) and bare hardware (Fig. 3-A) setups. For instance, the VM (virtual) hardware components, such as disk controllers, are likely to be different from those on the bare machine, requiring the system to execute different code for certain operations. Another example is that some operations are executed at a slightly different rate relative to SUT progress when the SUT is in a VM vs. on bare hardware. This is due to the performance overhead of virtualization (progress with respect to wall-clock time). As a specific example, timer events are triggered based on wall-clock time. Thus, the rate of execution of timer handling routines relative to SUT instruction count is higher with the SUT in a VM vs. on bare hardware. Hence, if injection is triggered based on an instruction count (e.g., injection into registers and stack), injection is more likely to happen while timer handling routines are being executed with the SUT in a VM vs. on bare hardware.

We use our VM cycle counter mechanism (Subsection 5.4) to measure crash latencies. Table 1 shows the cumulative distributions of crash latencies for both platforms (Fig. 3-B and 3-A). For crash latencies of 3K cycles and higher, measurements with the two platforms yield very similar results (with the exception of the EFL target). This indicates that the fidelity of crash latency measurements within a VM is good for latencies above a few thousand cycles. However, across all injection targets, with bare hardware a significantly higher fraction of the measured crash latencies are less than 1K cycles. This discrepancy may be due to variations in the VM exit/entry time within a single exit reason (Subsection 5.4) that we have observed in our experiments. Without our more accurate VM cycle counter mechanism, the differences in the 3K range and less would be more pronounced as the inaccuracy due to VM exit/entry time overhead would be a larger fraction of the total crash latency. Maintaining an accurate VM cycle counter has negligible impact on system performance as it involves just incrementing a counter on each VM exit.

Fault injection into the Linux kernel (Fig. 2-B) is performed with the SUT in an FV VM (Fig. 3-B). This campaign consists of 10,000 fault injections into the kernel code segment. Our results are shown in Figures 6 and 7 and include injection outcomes, crash causes, and crash latencies. We compare these results to prior work by Gu et al. [16], [17], who performed similar injections, running the Linux kernel on bare hardware. Note that Figure 7 shows the distribution of the crash latencies as in that prior work, and not the cumulative distribution as in Table 1.

The experimental results obtained in [16], [17] are generally comparable to our results, demonstrating the validity of conclusions derived from measurements with the SUT in a VM. Specifically, the injection outcome distribution in that prior work is: Hang - 15.2%, Crash - 49.5%, Incorrect results - 1.9%, Not manifested - 33.4%. The distribution of crash causes in the prior work is: InvalidPageReq - 35.7%, GeneralProt - 5.6%, Null pointer - 23.9%, InvalidOp - 33.2%, Other - 1.6%. Finallly, with respect to crash latency, roughly 50% of crashes in their experiments occur in less than 1K cycles. The differences in the results between this prior work and ours may be attributed to our use of a newer version of the Linux kernel (2.6 instead of 2.4) and/or to the reduced intrusiveness of the logging mechanism enabled by running the system under test in a VM (see Subsection 7.3).

7.2 Impact of Injector in Target System

Ideally, to prevent intruding on normal system operations and skewing injection results, fault injection should not require running any additional code on the target system. However, as discussed in Subsection 4.3, if the goal is to target individual processes or data structures in the SUT, a fault injector that resides inside the target system may be the best option. Hence, the focus of this subsection is on

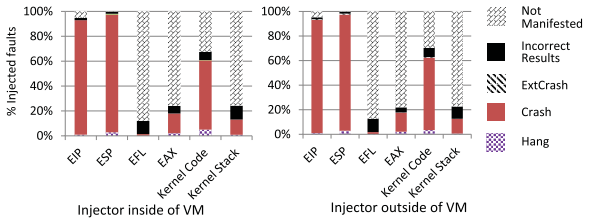

Fig. 8. Injection outcome distributions. Target is the Linux kernel in an FV VM. Injection from within the target system (Fig. 3-B) vs. from the hosting VMM (Fig. 3-C).

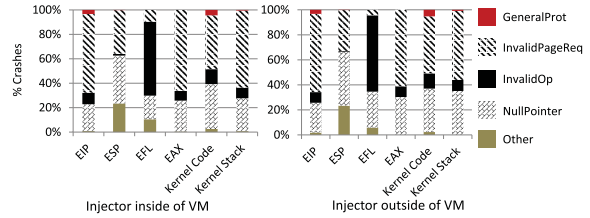

Fig. 9. Crash cause distributions. Target is the Linux kernel in an FV VM. Injection from within the target system (Fig. 3-B) vs. from the hosting VMM (Fig. 3-C).

whether injection results are skewed when the injection is performed from within the target system.

For injections performed from within the target system, the target OS kernel or the target VMM must be modified. Hence, any impact of the injector on the behavior of the target system is most likely to be exposed in the manifestation of faults injected into the the target OS kernel or VMM. Thus, for both the Linux kernel and the Xen VMM, we compare results obtained from injections from within the target VM to results obtained from injections into the VM from the host VMM, which eliminates the intrusion problem. This is done in four sets of campaigns: Linux/VMM target system (Fig. 2-B/2-A), internal/external injector (Fig. 3-B/3-C).

When the target is the Linux kernel (Fig. 2-B), the number of fault injections varies from 600 into the EIP register to 10,000 into the kernel code. The injection outcomes and crash causes distributions are shown in Figures 8 and 9, respectively. The results are, essentially, identical.

When the target is the VMM (Fig. 2-A), the results from injections performed within the target system (Fig. 3-B) are shown in left halves of Figures 4 and 5. The same injections are also performed from outside the target system, with the injector in the host VMM (Fig. 3-C). In this case, the number of injections varies from 800 into the EIP register to 4,000 into the EFL register. As with the injections into the Linux kernel, the results with the external injector are, essentially, identical to the results with the internal injector. Hence, due to limited space, we do not show these results here.

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

When the target is the VMM, the results of comparing crash latency outcomes of internal vs. external injections have similar characteristics to the results of the corresponding comparison with the Linux kernel target. However, when the injection targets are the ESP and EFL registers, with the external injector, there are *higher* fractions of latencies less than 1K cycles and 10K cycles, respectively (i.e., lower crash latencies). For ESP injections, during the measured interval, relative to the external injector, the internal injector executes additional code, possibly causing additional TLB/cache misses whose latencies are included in the measurement. With EFL, injections are uniformly distributed across 18 bits. However, injections to different bits cause most of the crashes with injections to the VMM vs. to the Linux kernel. For the VMM, over 90% of the injections to the EFL that result in a crash are to the Nested Task (NT) bit. For injections to the Linux kernel, injections to the NT bit are responsible for less than 5% of the crashes. The specific impact of a modified NT bit is likely responsible for the VMM vs. kernel differences.

7.3 Impact of the Logging Mechanism

This subsection focuses on the effects of the fault injection logging mechanism on the fidelity of the injection results. We run fault injection campaigns, where the SUT is in an FV VM (Fig. 3-C), in two setups: 1) using the V2V Driver (Subsection 5.3) for logging kernel and workload outputs from the target VM, and 2) using a virtual serial console to log the kernel outputs and a hard disk to log workload outputs. In both cases, the target is the Linux kernel (Fig. 2-B) and faults are injected from the host VMM (Fig. 3-C) into CPU registers, while the CPU is executing kernel code.

The first setup results are discussed in Subsection 7.2. We compare register injection results from those experiments to results obtained using the second setup. With the second setup, between 400 and 2300 faults are injected into each of the registers: EIP, ESP, EFL, and EAX. With respect to the fault outcome distribution, for the EIP, EFL, and EAX registers, the results are essentially the same. However, with respect to the ESP register, there are some differences. Specifically, comparing the first and second setups, the percentages of crashes are 94.6% and 88.5%, respectively, and the percentages of hangs are 2.9% and 10.3%, respectively.

The differences between the two setups in terms of outcomes from injections to the ESP register are likely due to the lower complexity of the V2V Driver, which made it more likely for the crash information from the target kernel to be successfully recorded with the first setup. With the second setup, the crash information were lost, leading to the classification of the outcome as a hang. Thus, this simple example demonstrates a benefit of using the low-intrusive logging enabled by virtualization.

7.4 Fault Injection Performance

The key issues that impact injection performance on virtualized systems vs. bare hardware are discussed in Subsection 4.2. To evaluate these issues in practice, we set up and measure a simple fault injection run in a virtualized system and on a bare machine.

TABLE 2
Time breakdown, in seconds, of operations in a single injection run. Target is the Linux kernel: in an FV VM (Fig. 3-B) vs. on bare hardware (Fig. 3-A).

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

The bare machine is configured as described in Section 6. The target system has a 250MB root partition. On the bare machine setup, the root partition resides on a local hard disk and the pristine root image used for restoration resides on a separate local hard disk. With this setup, there is, obviously, a small risk that the pristine root image will be corrupted by previous runs. In the VM setup, the VM's root partition is a disk image stored as a file in the PrivVM. The pristine root image is stored as a separate file in the PrivVM, not accessible by the VM. The workload is UnixBench running on top of the Linux kernel (Fig. 2-B). Mechanisms for injection and collecting logs are described in Section 6.

Table 2 shows the time breakdown of a fault injection run on a virtualized system and on bare hardware. As the time measurements shown are specific to our system configuration, these results only serve to demonstrate the contributions of the five factors (Subsections 4.1/4.2) affecting the performance of a fault injection run.

As shown in Table 2, the time to boot the system dominates the overall time of performing a single injection run. On our bare machine setup, the BIOS POST stage takes 20s and 15s are used by the OS to probe and configure the devices on the system. Restoring hard disk state is faster in the virtualized system due to the ability of the host PrivVM's OS to completely cache the disk image files in memory.

7.5 Fault Isolation Enabling Fault Injection

To use system virtualization for fault injection, the VI must provide good "sandboxing" capability, so that faults in the target system (VM) cannot corrupt or interfere with the testing infrastructure. Based on years of experience using the Xen VI with fault injection campaigns consisting of tens of thousands of injections into VMs, we find that the vast majority of faults injected into the target VMs only cause those VMs to fail while leaving the rest of the virtualized system in working order. We uncovered only two cases where faults into VM registers caused the VMM to fail[23]. In the first case, a corrupted I/O state flag accessible by the FV VM caused Xen to crash due to a bug in the way Xen's VM crash code interacts with bad VM I/O requests/replies. In the second case, a corrupted instruction pointer caused re-execution of the VM CPU initialization code which Xen does not handle correctly, causing a hang of the VMM. Despite these two fault isolation violations, overall, we find that the Xen VMM provides good fault isolation guarantees to enable the hosting of fault injection campaigns.

8 Examples of Gigan Deployments

We have used Gigan to evaluate both non-virtualized and virtualized systems. Fault injection was used to stress the resilience mechanisms of the different SUTs, in order to debug and enhance these mechanisms. In this section we discuss a few different fault injection campaigns conducted and explain how virtualization helped simplify or enable key aspects of those campaigns. We provide key sample results exemplifying the type of analysis that can be performed. The purpose of this section is to highlight the benefits of *conducting* the campaigns using virtualization, not on analyzing the injection outcomes or the systems that were evaluated. The latter information can be found in the cited publications.

8.1 Evaluating Non-virtualized Systems

As an example of how system virtualization can be used to facilitate injection into non-virtualized systems, we describe a fault injection campaign used for evaluating the fault-tolerance mechanisms of *Ghidrah* [29], [18], a fault-tolerant distributed cluster management middleware (CMM) developed at UCLA. In this campaign, we used Gigan's abilities to target both the OS kernel and user-level processes. This campaign highlights the use of virtualization to facilitate the evaluation of distributed applications[18].

UCLA's Ghidrah CMM: With clusters for parallel tasks, the CMM performs critical functions, such as: allocating resources to user tasks, scheduling tasks, reporting task status, and coordinating fault handling for tasks[29]. Ghidrah supports running multiple parallel applications and is designed to maintain the basic cluster functionality despite the failure of any single cluster node.

CMM campaign description: While Ghidrah was designed to run on bare hardware, we ran the Ghidrah campaign on a cluster of four VMs. The application workload consisted of two instances of the IS (integer sort) benchmark from the NAS Parallel Benchmark Suite[2], both running across four nodes.

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

Benefits of using virtualization: All the previously discussed benefits of using virtualization for fault injection were applicable to the evaluation of Ghidrah. An additional critical benefit was the reduction in the hardware resources required. Instead of requiring a separate physical host for each cluster node, Ghidrah was run on *virtual clusters*[25], where each cluster node was in a separate VM. This allowed four 8-core physical hosts to simultaneous run four separate Ghidrah clusters, each consisting of four cluster nodes.

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

As previously discussed, virtualization simplified the restoration of cluster node disk images to a pristine state. This was done using disk images kept in the PrivVM. To minimize the time to restore these disk images, we avoided the time-consuming operation of copying large disk images. This was done using a union mount in each VM [30], [18] for the root file system. The union mount "merges" a read-only branch with a read-write branch. With the union mount, only the read-write image of the file system had to be refreshed. In our setup, the size of the root file system read-only branch was 2.5GB, while the size of the read-write branch file system was 50MB. Copying the 50MB image took at most one second, while copying the 2.5GB image would have taken up to one minute. The file system images for the union mount were stored in the PrivVM, protected by virtualization. Without virtualization, it would not have been possible to reliably enforce read-only access to the read-only file system without special hardware.

Results: Of the 43,607 fault injections that caused errors, 37 exposed 11 unique flaws in Ghidrah. Each of these flaws was a bug in the Ghidrah implementation that we were able to identify and fix. Some flaws took tens of hours of injections to uncover, demonstrating the importance of a fault injection campaign that can be automated to run for a long time. Of the 11 flaws, one flaw was exposed primarily by injection into the kernel. This flaw demonstrated the utility of injections targeting the kernel, even when the goal is to expose flaws in user-level programs (in this case, the Ghidrah processes).

8.2 Evaluating Virtualized Systems

Failure of a single VI component (see Section 3) can cause the entire virtualized system to fail [25], [32]. In order to improve the reliability of virtualized systems, we have designed, implemented, and evaluated mechanisms that increase the resiliency of the Xen VI to the failure of any VI component [24], [26], [25], [27]. These mechanisms detect and recover a failed VI component, while allowing VMs on the system to continue executing correctly with minimal interruption. For over four years, we have been using Gigan's ability to inject faults into all the Xen VI components and collect detailed logs of the results to evaluate and enhance our resiliency mechanisms. The rest of this subsection is a brief overview of this use of Gigan, as an example use of fault injection to evaluate virtualized system.

Campaign description: We stressed the Xen VI using two different workloads: (1) a synthetic workload consisting of three AppVMs — one running a user-level network ping application, one running UnixBench (see Section 6), and one running a file system intensive application; and (2) the Linux Virtual Server (LVS)[46] — two AppVMs as directors, three AppVMs running the Apache web server, and five clients sending HTTPS requests from a remote machine. For each workload, five fault injection campaigns were performed. In three campaigns, faults were injected into CPU registers: 1) while the CPU executed VMM code, 2) while the CPU executed DVM code (user and kernel-level), and 3) while the CPU executed PrivVM code (user and kernel-level). The fourth campaign injected software faults[40] into the VMM code segment. The fifth campaign injected software faults in code segments of device drivers (network and disk controllers) in the DVM.

Faults in the VMM or PrivVM often cause the entire system to crash or hang. Hence, to simplify the automation of resetting the system upon a crash or hang (Subsection 4.1), for campaigns targeting the VMM or PrivVM, the entire target system (our resilient VI along with all the AppVMs) ran in an FV VM. In these campaigns, in order to minimize intrusion, faults were injected from the host VMM. Campaigns targeting the DVM ran on bare hardware, relying on the VMM isolation mechanisms to prevent corruption of the rest of the system. Injection was performed from the VMM. For these campaigns, only the state of the faulty DVM was restored after every injection run, thus speeding up the campaigns.

Benefits of using virtualization: All the previously discussed benefits of using virtualization for fault injection were applicable to the evaluation of our resilient VI. Of particular importance was the ability to use our *V2V Driver* (Subsection 5.3), which leverages shared memory on a virtualized system to collect outputs from the target system. This allowed us to collect, with minimal intrusion, detailed information regarding the operation of the system at critical stages after a fault was injected. For example, when faults were injected during VMM execution and our VMM recovery mechanism failed, this detailed information allowed us to identify and correct deficiencies with our mechanism[26]. In addition, the V2V Driver was also used as a light-weight signaling mechanism, to coordinate activities in a campaign. For example, this facilitated initiating injection only after the target system started executing its workload. Without virtualization, such coordination would have required using the network, thus significantly increasing intrusion.

Results: Altogether about 24,700 faults were injected into components of the Xen VI. As discussed above, we used fault injection results to guide enhancements of our *ReHype* VMM recovery mechanism[26]. The initial version of ReHype had a recovery success rate of 6%. After the final improvement, an injection campaign consisting of 7,600 faults showed successful recovery from 90% of detected VMM failures. With respect to recovery from DVM failures, a campaign consisting of 15,000 manifested faults in the DVM, demonstrated a recovery success rate of 95%. For the last VI component, the PrivVM, a campaign consisting of 2,100 injected faults showed successful recovery from 90% of the faults manifested as PrivVM failures.

9 Related Work

SWIFI, without virtualization, has been used for decades to evaluate non-virtualized systems [35], [19], [8], [38]. This has included injection into OS kernels [9], [16], [17]. As discussed further in the rest of this section, in addition to our work with Gigan [18], [24], [26], [25], [27], there has been some work using fault injection in conjunction with system virtualization to evaluate non-virtualized systems [7], [36], [6], [39], [40], [31] as well as the use of SWIFI to evaluate virtualized systems [21], [32]. However, to the best of our knowledge, no prior work has provided a systematic analysis and evaluation of the *interactions* between virtualization and fault injection, including the benefits (e.g., result

logging) and challenges (e.g., crash latency measurement) with this combination. Prior work has not provided experimental comparison of the results from injections with the SUT in a VM vs. on bare hardware, or of injections from inside or outside the SUT.

Early work on using SWIFI with virtualization to evaluate non-virtualized systems was based on a custom-built User Mode Linux (UMLinux) [7], [36], [6] (different from UM Linux[10]). With UMLinux, the OS of the SUT runs in user mode, as opposed to privileged mode in non-virtualized systems or FV VMs. The "device drivers" do not interact with controllers that are similar to real device controller. Furthermore, with UMLinux, the interactions between processes and the OS kernel in the SUT (system calls, traps, interrupts) are different from the way these interactions occur in non-virtualized systems and in FV VMs. Despite the significant differences between real hardware and the virtualized system under UMLinux, there is no evaluation of the impact of these differences on injection results.

In addition to the issues discussed above, the performance overhead of UMLinux is *much* higher than with conventional system-level virtualization. For example, for a kernel build, UMLinux incurred an overhead of 378%[7], compared with only 3% with Xen[3]. The UMLinux works briefly mention, without evaluation, the potential benefits of using virtualization for fault injection with respect to ease of automation and reduced intrusion.

Subkraut et al.[39] propose using VM checkpoints to speed up fault injection, but do not describe any implementation or evaluation. Swift et al.[40] run their SUT as well as a fault injector in a VM. They mention that virtualization is used to ease and speed up the automation of a campaign consisting of thousands of independent injection runs. They do not provide any analysis of the interactions between SWIFI and virtualization. Oyama and Hoshi[31] use virtualization to inject security "attack effects" into an SUT in a VM. Their injector runs in the PrivVM. They briefly discuss the advantages provided by using virtualization to inject from outside the SUT, with respect to minimizing intrusion and ensuring accessibility to injection results. They do not provide numerical experimental results.

Jeffery and Figueiredo[21] use an injector in the KVM hypervisor[22] to inject faults into an SUT in a VM to evaluate their VM replication mechanism. They do not discuss or evaluate the interactions between fault injection and virtualization. Pham et al.[32] present a fault injection framework, based on kernel-level injectors, for evaluating virtualization environments. With both KVM and Xen, they inject faults from within different VMs into those VMs. They also inject faults into the KVM hypervisor, which is implemented as a kernel module for Linux. They refer to our unpublished workshop presentation[23] regarding injection into the Xen hypervisor. They mention challenges when the SUT is in a VM with respect to incomplete virtualization of some special registers and sensitive instructions. They do not present solutions to those problems nor experimental results that are equivalent or similar to ours.

We have used Gigan to evaluate non-virtualized distributed systems[18] and our resiliency enhancements to the Xen VI [24], [26], [25], [27]. In an unpublished workshop presentation[23] we introduced our fault injector and a preliminary discussion of the the interactions between system virtualization and SWIFI.

10 Conclusion and Future Work

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

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

We have developed a mechanism that improves the accuracy of latency measurements within a VM. We have shown that, for latencies of a few thousand cycles and above, crash latencies measured when the target system is in a VM closely match the results when the target system is on bare hardware. For shorter latencies, there are significant inaccuracies with measurements within a VM.

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

Future work will include using nested virtualization[4] to facilitate injection into systems consisting of a hypervisor and multiple FV VMs, as well as improving the accuracy of latency measurements when the target system is in a VM, especially when the target is a multiprocessor VM.

REFERENCES

[1] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Trans. on Computers*, vol. 52, no. 9, pp. 1115–1133, Sep. 2003.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks," *Int. J. of High Performance Comp. Applications*, vol. 5, no. 3, pp. 63–73, Sep. 1991.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *19th ACM Symp. on Operating Systems Principles*, pp. 164–177, 2003.

[4] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The Turtles project: Design and implementation of nested virtualization," *9th Symp. on Operating Syst. Design and Implementation*, 2010.

[5] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault-tolerance," *Computer Systems*, vol. 14, no. 1, pp. 80–107, Feb. 1996.

[6] K. Buchacker, M. D. Cin, H. J. Hoxer, V. Sieh, O. Tschache, and M. Waitz, "Hardware fault injection with UMLinux," *Int. Conf. on Dependable Systems and Networks, Fast Abstracts*, p. 670, 2003.

[7] K. Buchacker and V. Sieh, "Framework for testing the fault-tolerance of systems including OS and network aspects," *6th Int. Symp. on High-Assurance Systems Engineering*, pp. 95–105, 2001.

[8] J. Carreira, D. Costa, and J. G. Silva, "Fault injection spot-checks computer system dependability," *IEEE Spectrum*, vol. 36, no. 8, pp. 50–55, Aug. 1999.

[9] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. on Software Engineering*, vol. 24, no. 2, pp. 125–136, Feb. 1998.

[10] J. Dike, "A user-mode port of the linux kernel," *4th Annual Linux Showcase & Conf.*, pp. 63–72, 2000.

[11] Y. Dong, S. Li, A. Mallick, J. Nakajima, K. Tian, X. Xu, F. Yang, and W. Yu, "Extending Xen with Intel virtualization technology," *Intel Technology Journal*, vol. 10, no. 3, 2006.

[12] J. R. Douceur and J. Howell, "Replicated virtual machines," *Technical Report MSR TR-2005-119, Microsoft Research*, Sep. 2005.

[13] J.-C. Fabre, F. Salles, M. R. Moreno, and J. Arlat, "Assessment of COTS microkernels by fault injection," *7th IFIP Working Conf. on Dependable Computing for Critical Applications*, pp. 25–44, 1999.

[14] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the Xen virtual machine monitor," *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS) (ASPLOS)*, 2004.

[15] V. Goyal, E. Biederman, and H. Nellitheertha, "Kdump, A kexec-based kernel crash dumping mechanism," lse.sourceforge.net/kdump/documentation/ols2oo5-kdump-paper.pdf, 2005.

[16] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of Linux kernel behavior under errors," *Int. Conf. on Dependable Systems and Networks*, pp. 459–468, 2003.

[17] W. Gu, Z. Kalbarczyk, and R. K. Iyer, "Error sensitivity of the linux kernel executing on PowerPC G4 and Pentium 4 processors," *Int. Conf. on Dependable Systems and Networks*, pp. 887–896, 2004.

[18] I. Hsu, A. Gallagher, M. Le, and Y. Tamir, "Using virtualization to validate fault-tolerant distributed systems," *Int. Conf. on Parallel and Distributed Computing and Systems*, pp. 210–217, 2010.

[19] M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[20] Intel Corporation, "Intel 64 and IA-32 architectures software developer's manual: Volume 3c," 2014.

[21] C. M. Jeffery and R. J. Figueiredo, "A flexible approach to improving system reliability with virtual lockstep," *IEEE Trans. on Dependable and Secure Computing*, vol. 9, no. 1, pp. 2–15, Jan. 2012.

[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," *Linux Symp.*, pp. 225–230, 2007.

[23] M. Le, A. Gallagher, and Y. Tamir, "Challenges and opportunities with fault injection in virtualized systems," *1st Int. Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, 2008.

[24] M. Le, A. Gallagher, Y. Tamir, and Y. Turner, "Maintaining network QoS across NIC device driver failures using virtualization," *8th IEEE Int. Symp. on Network Computing and Applications*, pp. 195–202, 2009.

[25] M. Le, I. Hsu, and Y. Tamir, "Resilient virtual clusters," *17th IEEE Pacific Rim Int. Symp. on Dependable Computing*, pp. 214–223, 2011.

[26] M. Le and Y. Tamir, "Rehype: Enabling VM survival across hypervisor failures," *7th ACM Int. Conf. on Virtual Execution Environments*, pp. 63–74, 2011.

[27] ——, "Applying microreboot to system software," *IEEE Int. Conf. on Software Security and Reliability*, pp. 11–20, 2012.

[28] M.-L. Li, P. Ramchandran, S. K. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," *13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 265–276, 2008.

[29] M. Li, D. Goldberg, W. Tao, and Y. Tamir, "Fault-tolerant cluster management for reliable high-performance computing," *Int. Conf. on Parallel and Distributed Computing and Systems*, pp. 480–485, 2001.

[30] J. R. Okajima, "aufs – another unionfs," *http://aufs.sourceforge.net/aufs.html*.

[31] Y. Oyama and Y. Hoshi, "A hypervisor for injecting scenario-based attack effects," *IEEE Annual Computer Software and Applications Conf.*, pp. 682–687, 2011.

[32] C. Pham, D. Chen, Z. Kalbarczyk, R. Iyer, S. Sarkar, and R. Hosn, "Cloudval: A framework for validation of virtualization environment in cloud infrastructure," *Int. Conf. on Dependable Systems and Networks*, pp. 189–196, 2011.

[33] H. V. Ramasamy and M. Schunter, "Architecting dependable systems using virtualization," *Workshop on Architecting Dependable Systems*, 2007.

[34] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *IEEE Computer*, vol. 38, no. 5, pp. 39–47, May 2005.

[35] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Dancey, A. Robinson, and T. Lin, "FIAT – fault injection based automated testing environment," *18th Fault-Tolerant Computing Symp.*, pp. 102–107, 1988.

[36] V. Sieh and K. Buchacker, "UMLinux - A versatile SWIFI tool," *4th European Dependable Computing Conf.*, pp. 159–171, 2002.

[37] T. Slaight, "Platform management IPMI controllers, sensors, and tools," *Intel Developer Forum*, Sep. 2002.

[38] D. T. Stott, B. Floering, D. Burke, Z. Kalbarcypk, and R. K. Iyer, "NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors," *4th Int. Computer Performance and Dependability Symp.*, pp. 91–100, 2000.

[39] M. Subkraut, S. Creutz, and C. Fetzer, "Fast fault injection with virtual machines," *Int. Conf. on Dependable Systems and Networks, Fast Abstracts*, 2007.

[40] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," *ACM Trans. on Computer Systems*, vol. 23, no. 1, pp. 77–110, Feb. 2005.

[41] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *IEEE Computer*, vol. 38, no. 5, pp. 48–56, May 2005.

[42] UnixBench, "Unixbench," www.tux.org/pub/tux/benchmarks/System/unixbench, 2013.

[43] VMware, "Performance evaluation of Intel EPT hardware assist," *http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf*, 2008.

[44] L. Wang, Z. Kalbarcyzk, W. Gu, and R. K. Iyer, "An OS-level framework for providing application-aware reliability," *12th Pacific Rim Int. Symp. on Dependable Computing*, pp. 55–62, 2006.

[45] D. Wilder, "LKCD installation and configuration," *http://lkcd.sourceforge.net/*, 2002.

[46] W. Zhang and W. Zhang, "Linux virtual server clusters," *Linux Magazine*, vol. 5, no. 11, Nov. 2003.

Michael Le received the PhD degree in Computer Science from UCLA in 2014. His research interests are in computer systems.

Yuval Tamir received the PhD degree in Electrical Engineering and Computer Sciences from UC Berkeley in 1985, and has been a faculty member in the UCLA Computer Science Department since then. His research interests are in computer systems and computer architecture.