© 2016 Mengjia Yan

PERFORMANCE EVALUATION OF VM-LEVEL RECORD-AND-REPLAY TECHNIQUES AND APPLICATIONS

BY

MENGJIA YAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Josep Torrellas

ABSTRACT

Virtual machine level record and replay can be used for complex system debugging and analysis, fault-tolerance replication and forensic analysis. Previous work on performance evaluation of RnR frameworks are not complete enough due to their narrow focuses. RnR related projects either focus on performance evaluation of plain record and replay mechanisms or specifically target the effectiveness of the functionality RnR supports.

In order to identify the performance bottlenecks in the complicated RnR system and its various applications, this thesis conducts a thorough evaluation and analysis on 3 different modes of RnR, that is, record, replay with checkpointing and replay with VMI analysis. Both RnR system developer and users can benefit from our work. With our evaluation results, system developer can propose more efficient design accordingly, and RnR users can configure the system properly to achieve expected performance.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First of all, this work would not be possible without the vision of my advisor, Professor Josep Torrellas. Josep offered me Research Assistant position in iacoma group and guided to this exciting research topic, using deterministic record and replay to design novel hardware architecture for system security. From him, I learnt two key characteristics of successful researcher, that is, passion and persistence. Research is hard. Therefore, one has to devote a lot of love to his work, and keep working on it to reach its perfect.

Equally important, this work would not happen without the help from my great partner, Yasser Shalabi. It is he who first started this project and explore this field in our research group. The project is quickly developed with his enthusiasm, inspiration and hard work. His knowledge and experience on hardware-assisted virtualization technology helps me get through the initial stage of the research, when I have to difficulties in understanding the complicated design of Intel extensions. He also encourages me a lot with his optimistic and passionate attitude towards life.

Moreover, Nima Honarmand plays an uniquely important role in this project as an adivsor and partner. As a senior researcher in the field of record and replay, he provides valuable insight and precise criticism of the project. Without him, the project would not reach its current state. Nima taught me the big lesson about data presentation. It is the basic requirement for a researcher to present well-explained data logically and concisely.

I would like to thank everyone in iacoma group, for their valuable support and feedback on this project. Bhargava Gopi Reddy, Wooil Kim and Raghavendra Pradyumna Pothukuchi always shared their experience and vision on computer architecture with me, and gave me very useful tips about graduate student life. Thomas Shull and Dimitrios Skarlatos created the harmonious environment of the research lab, where I had unforgettable memory. Tanmay Gangwani and I co-worked on the related course project, where I

got solid understanding of record and replay techniques for multi-threading. Jiho Choi and I shared our hard experience in working on unfamiliar topics and gave each other encouragements.

Many people provided their help, love and support during the first three years of my PhD. My mother and father gave me great mental support and shielded me from various problems, so that I can completely focus on my course work and research. As a senior researcher, my boy friend, Xuhao, helped me to build self-confidence by sharing me with his own experience, and encouraged me to do valuable research and make contributions to this exciting field. With someone fully understand my situations and concerns, I am full of courage, no matter how many difficulties ahead.

Finally, I would like to thank my committee members for agreeing to participate in this effort. I truly appreciate their diverse opinions, unbiased criticism, and insightful suggestions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Virtual machine level deterministic record and replay has been well studied in the past decade. A number of complete implementations [1, 2, 3] of deterministic replay systems are available for different production platforms. Moreover, numerous applications of RnR have been suggested for complex system debugging and analysis [4, 5, 6, 7, 8], fault-tolerant replication [9, 10], security and forensic analysis [11, 1, 12, 13, 14], etc. However, despite flourishing study in academia, we haven't seen deterministic RnR be incorporated in current production virtualization systems or software stacks. There are various reasons for it, and performance overhead is one of them.

Previous work on performance evaluation of RnR frameworks and applications is not complete enough. Different projects have different focuses. RnR design paper conducts evaluation for plain record and replay execution overhead, while evaluation in RnR application paper focuses on effectiveness and robustness of the targeted functionality. Despite the existence of abundant research results, we find it still difficult to obtain a complete view of the complicated RnR system and identify performance bottlenecks within the framework and its different applications.

After surveying a large number of RnR application projects, we summarize the usages of RnR in two categories as follows. First, checkpointing mechanism is commonly used together with RnR to either support fault-tolerant replication or provide initial consistent system state for debugging. Second, various VMI analysis techniques are applied at replay stage, to monitor system execution or conduct forensic analysis.

In this thesis, we carry out a thorough analysis and evaluation on a virtual machine level record and replay framework, as well as its two major applications, i.e. replay with checkpointing and replay with VMI analysis. Our work is useful from two perspectives. On one hand, with the major sources of overhead being identified, developers can accordingly propose more effi-

cient design. On the other hand, with a detailed performance analysis, RnR users can have a more reasonable performance expectations, and customize applications to achieve better performance.

CHAPTER 2

BACKGROUND

2.1 Virtualization

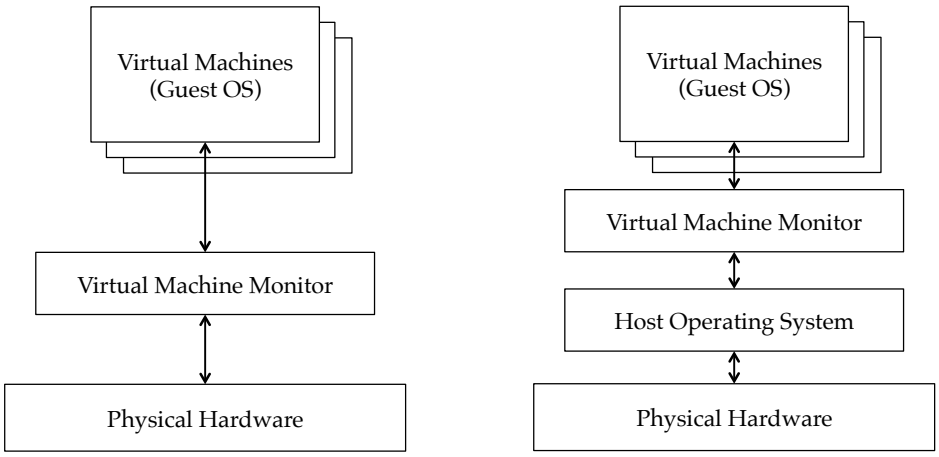
Over the last two decades, thanks to the rapid developing virtualization techniques, data centers have turned into multi-tennant, dynamically provisionable resources, and cloud computing has been widely adopted.

The concept of virtualization is not new, which can be dated back to late 1960s, when IBM developed the first system to support concurrent and interactive access to a mainframe computer. Modern virtualization indicates techniques that enable multiple operating environments execute on the same hardware at the same time. In this section, we briefly discuss two main types of virtual machine monitors and several commonly used virtualization techniques.

2.1.1 Types of virtual machine monitor

A virtual machine(VM) is an software abstract of physical machine. An operating system can be installed in the virtual machine and executes as if it runs on a physical machine. Such operating system is called as "Guest OS". The software layer that provides the interface between virtual machine and underlying hardware is called virtual machine monitor(VMM) or hypervisor. There are two major types of virtual machine monitors, named as type 1 and type 2 according to [15].

Figure 2.1 shows different architecture of the two types of VMM. Type 1 VMM, also named as native or bare-metal hypervisor, runs directly on host hardware. The hypervisor self can be considered as a specialized operating system, which is able to manage CPU, memory, devices, etc. and provide interface for virtual machine. Example of type 1 VMM includes



(a) Type 1 VMM **(b) Type 2 VMM**

Figure 2.1: Types of Virtual Machine Monitor

Xen [16], VMWare ESX and Microsoft Hyper-V [17]. Type 2 VMM, also named as hosted hypervisor, runs on conventional operating system as a normal process, and leverages the host operating system to interact with underlying hardware. Type 2 systems include VMWare Workstation, Virtual Box, QEMU [18].

2.1.2 Hardware-assisted Virtualization

Full virtualization is used to emulate a complete hardware environment for virtual machine, so that unmodified operating system can execute in complete isolation. It can be implemented via binary translation and software emulation of devices, which suffer significant performance overhead. Hardware-assisted virtualization, also called accelerated virtualization, is a platform virtualization approach to enable efficient full virtualization with the help of extended hardware capabilities. In 2005 and 2006, Intel and AMD introduces new processor extensions to support hardware virtualization, named as Intel VT-x and AMD-V, which simplified virtualization software design but offered little speed benefit. Later more features are added to improve memory management virtualization and IO virtualization via extented page table(EPT) and virtualized IO.

One of the difficult in vitalization for x86 system is that, certain privileged instructions can not be directly executed in guest system. Hardware

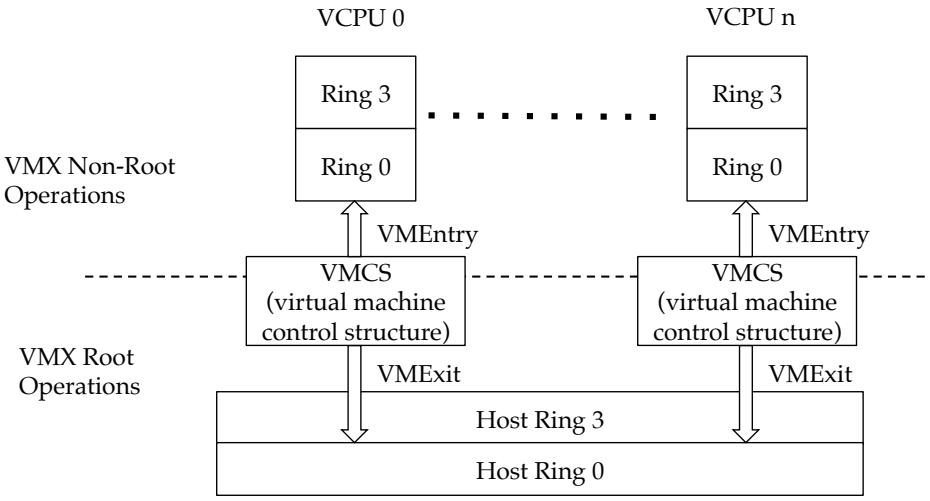


Figure 2.2: Intel VTx Technology: VMX Mode Transition

assisted virtualization techniques adds extra modes to processors to avoid de-privileging rings for guest OS, Taking Intel VT-x for example, the processor supports two kinds of operations: VMX root operation and VMX non-root operation. In general, VMM executes in root operation, and guest software runs in non-root operation. Privileged instructions in VMX non-root operation will be trapped into VMM. Transition between VMX root operation and VMX non-root is presented in Figure 2.2. Transition from VMX root operation to non-root is called VMEntry, while the reverse transition is called VMExit.

2.2 Virtual Machine Introspection

Virtual machine introspection (VMI) is a technique to enable monitoring guest virtual machine state at hypervisor layer. It is first proposed by Garfinkel and Rosenblum [19] as a hypervisor-level Intrusion Detection System (IDS). VMI is considered a promising introspecting technique to ensure security policy enforcement within the untrustworthy operating system for two reasons. First, the smaller code base of hypervisor makes it a smaller attack surface compared to extremely complex monolithic operating system. Second, by operating at a higher privilege level, the hypervisor is isolated and decoupled from the untrusted guest virtual machine. VMI has become a relatively mature research topic, with numerous projects.

However, without OS abstraction and higher level semantics, the hypervisor views the guest memory as raw bits or bytes. This problem is known as "semantic gap" Various techniques have been proposed to "bridge the semantic gap" by reconstructing guest operating system data structures.

VMI techniques can be divided into two categories depending on whether it is triggered by interposition. Passive VMI techniques relies on polling VM state for information with whitelisting or blacklisting. Examples of this type of VMI include libvmi [20] and VIX [21]. Active VMI revolves around monitoring checks triggered by certain events (usually in hardware), such as specific hardware registers or memory regions being accessed. For instance, Hypertap/Hprobes [22], Livewire [19], Lares [23], SIM [24], Antfarm [25], Lycosid [26] are commonly used ones.

2.3 Record and Replay

Record and deterministic Replay (RnR) of workloads is a popular architectural technique [27, 28, 29, 30, 31, 32, 1, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 8, 52, 53, 54, 55, 56, 57, 58, 59, 14, 60]. As a workload runs, RnR records all the non-deterministic events that can affect the execution and stores them in a log (*Record*). Later, in a potentially different platform, the workload is re-executed. At this time, the system injects the recorded events at the correct times, enforcing a deterministic execution (*Replay*). Recent proposals for RnR typically consider two classes of non-deterministic events: inputs to the workload and the memory-access interleaving of parallel processors.

RnR can be done at different abstraction layes—e.g., program-level RnR records and replays one or more programs in isolation, while VM-level RnR records and replays an entire VM. In this work, we use VM-level RnR [29, 32, 1, 33, 37]. Moreover, we consider uniprocessor hardware. As a result, the only relevant sources of non-determinism comprise interrupts raised and data copied by virtual devices into the guest machine. We also assume the widely used model of hypervisor-mediated I/O, as used in Xen [16] or Qemu [18]. In this case, the hypervisor interposes on all the I/O operations of the guest. This makes it possible to implement input recording in the hypervisor and use software-only mechanisms for recording. On the other hand, if the VM

directly interacted with the hardware (using recent virtualization technologies such as Intel VT-d [61]), input recording would require hardware support similar to hardware-assisted RnR (e.g., FDR [54] and DeLorean [41]).

Almost all research papers on RnR list security as one of its principal use-cases, but there are only a few that investigate use of RnR in a security-related scenario [1, 11, 13, 32, 14, 60]. ReVirt [1] shows an example of using VM-level RnR for post-facto offline analysis of a time-of-check to time-of-use race condition in the Linux kernel that could be exploited to compromise the kernel. IntroVirt[11] explored using VM-level RnR to determine if systems were previously exploited once zero-day attacks are discovered. Speck[13] explored using a combination of OS-level speculation and program-level RnR to remove security checks from the critical path of a program, and allow multiple such checks to run in parallel. ParanoidAndroid [14] and Secloud [60] explored the possibility of maintaining replicas of mobile devices in the cloud, and then using program-level RnR to perform heavy analysis of the program executions in the cloud.

CHAPTER 3

ANALYZING RNR FRAMEWORKS AND APPLICATIONS

In this chapter, we carry out a thorough analysis and evaluation on a virtual machine level record and replay framework as well as its two major applications, i.e. replay with checkpointing and replay with VMI analysis. Efficient checkpointing mechanism is commonly used together with RnR to support fault-tolerant replication or provide initial consistent system state for debugging. Various VMI analysis techniques are applied at replay stage to offload heavyweight security checks from original execution, to monitor system execution and conduct forensic analysis. The analysis in this Chapter is based on the system design, and quantitative evaluation results are shown in Chapter 5. With the major sources of overhead identified in different system components, RnR developers can propose more efficient design accordingly.

This chapter first describes the overall design of "insight", then follows a detailed discussion of performance implications of each component.

3.1 RnR Framework

"Insight" [3] is a Qemu/Kvm based virtual machine record and replay framework. By leveraging hardware-assisted virtualization technology, the baseline system without RnR can achieve good performance for most workloads.

"Insight" adds the functionality of record and replay to Virtual Machine Monitor by modifying Linux KVM hypervisor and the device emulator component in QEMU. Figure 3.1 presents the architecture of "insight", highlighting the components modified by "insight" to a traditional virtual machine monitor. Specifically, "Insight" configures VMCS(virtual machine control structure) [62] to trigger VMExits on all these non-deterministic events, and records or re-injects input values according to which mode it executes on. According to the origin of inputs, non-deterministic events can be categorized

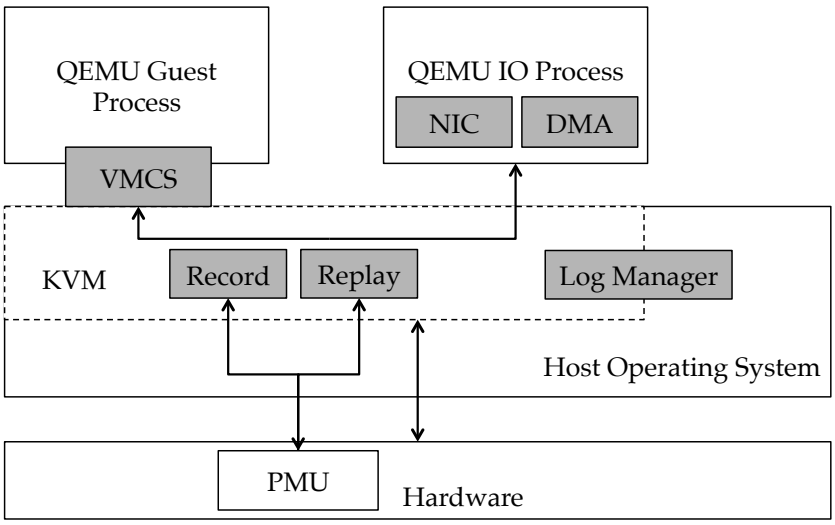


Figure 3.1: Insight Architecture

into three types: synchronous events, asynchronous events and combined events. Different types of events vary in the amount of information required to assist correct replay.

3.1.1 Synchronous Events

Synchronous events are triggered by instructions or functions executed by the guest system, that will return non-deterministic results. For example, read timer stamp counter(*rdtsc*) and get random number(*rdrnd*). More complicated example is read programmed IO and memory mapped IO. The main feature of such event is that even though it accesses non-deterministic data, it is triggered by *deterministic code*. Therefore, synchronous events will be faithfully reproduced, if replay follows the original execution path.

In "Insight", VMCS is configured that *rdtsc* and *rdrnd* always trigger VMExits into hypervisor. During record, the hypervisor logs the return variable; while during replay, corresponding register or memory address is overwritten with the logged variable.

3.1.2 Asynchronous Events

Asynchronous events are triggered by external devices to the guest system, including interrupts from devices such as keyboard, mouse, disk etc., and up-

dates to guest system memory from emulated devices via either NIC(network interface card) or DMA. Since QEMU IO thread is running in parallel with guest thread, asynchronous events need to be carefully scheduled and injected to ensure being replayed at the exact same execution point, at which the events occurred during record.

"Insight" uses $\langle branch\_counter, instruction\_pointer, ECX\_value \rangle$ as system timestamp, which is an identical representation of program execution point [57]. In record, asynchronous events as well as its timestmap is logged. In replay, "Insight" leverages PMU (performance monitor unit) to trap the the guest system into hypervisor at correct execution point. Concretely speaking, the number of instructions to next asynchronous events injection point is calculated as x, and the instruction counter is set up in a way that it will overflow after executing x instructions. Meanwhile, VMCS is configured that VMExit will be triggered once PMU counters overflow. Then in hypervisor, the replay engine will regenerate the interrupts or update memory using the logged values.

3.1.3 Combined Events

Combined events happen when an asynchronous event triggers execution of certain functions or instructions in the guest system, leading to several synchronous events. It is the common case for asynchronous events. For example, on receiving an network packet, the NIC will generate an interrupt and copy the content of the packets to the piece of memory address it maps to. Later, the guest system is preempted and consume the packet by issuing pio or mmio instructions. DMA-based disk IO event is another example.

Since synchronous events require much simpler logic in terms of both record and replay, "Insight" deals with combined events efficiently by delaying the asynchronous event until its following synchronous event.

We show the design details with an example of network traffic events handling as following.

Network IO Events

Figure 3.2 shows a network IO example to illustrate different operations applied during record and replay.

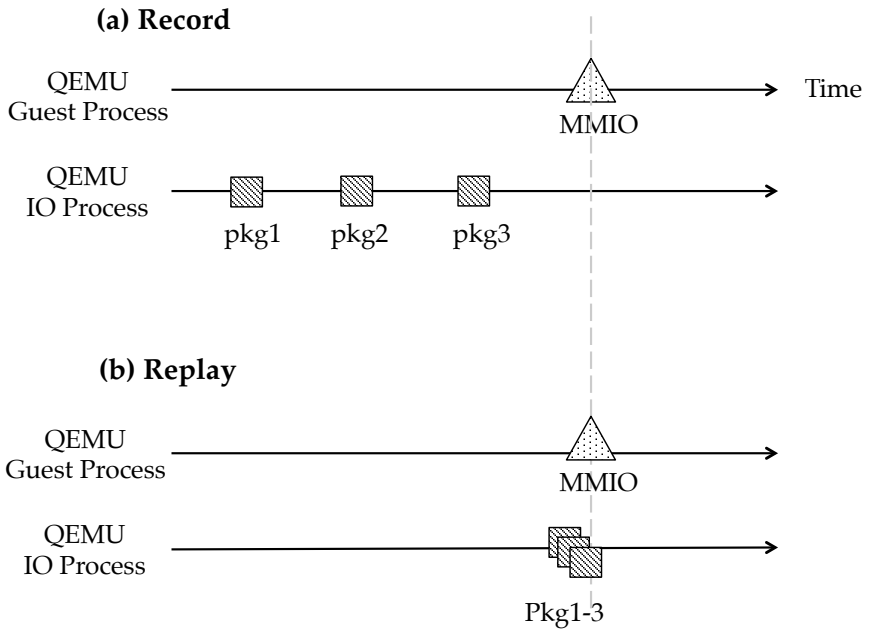


Figure 3.2: Combined Events Example: Network IO Events

During record, 3 packets arrive consecutively before a MMIO event issued from guest system. "Insight" modifies *eepro100* virtual device driver that, upon arrival of network packets, content of the packets (pkg1, pkg2 and pkg3 in this case) as well as the associated timestamps are recorded and saved to logs. The order between the 3 packets is invisible to the guest system.

To ensure correct replay, the happens-before relationship between following MMIO events and the receipt of network packets should be preserved. During replay, as a normal synchronous event, the MMIO event will trap the guest system into the hypervisor. In the hypervisor, before simply replaying the MMIO event, number of undelivered network events are checked, packets content are retrieved from logs, and corresponding memory update are applied. From guest's perspective, the arrival of 3 packets and related state updates are atomic operations during both record and replay.

3.2 RnR Applications

As an effective fault-tolerance solution, several virtual machine checkpointing algorithms and mechanisms have been proposed, and micro-checkpointing has been merged to the main stream of QEMU. A valid checkpointing is

required to contain consistent cpu, memory and external device states so that the replication can be later used for continuous execution.

Compared to general virtual machine checkpoints, the major difference of replay checkpointing is that there is no need to freeze and save device states, which simplifies checkpointing mechanism to some extent. All the non-deterministic inputs from external devices are injected from replay logs instead of QEMU emulated devices. The only exception is disk devices. "Insight" leverages the Copy-On-Write (COW) based disk format provided QEMU to avoid recording extremely huge amount of disk access return data. All disk reads and writes are redirected from/to a temporary image file backed by original disk image, which is created before system is booted up. In this case, the initial sate of disk file is kept intact and can be used for multiple numbers of replays. In addition, only disk access operations as well as their timestamps need to be recorded, resulting in less log storage overhead.

To efficiently take checkpoints of the system memory, differential checkpoints are commonly used if the checkpointing interval is around 10s of milliseconds as follows.

- Guest virtual machine is suspended. Guest memory is set as write protected.
- CPU state is copied to persistent storage as part of checkpointing and a snapshot of disk image is created.
- After guest virtual machine is resumed, any write to memory page will trigger write-protection fault. The accessed page will be copied to the checkpointing and then reset to writable, so that later modification can take effect.

There exists large variance in the size of memory checkpoints. The checkpoint may only take several megabytes when the guest system is mostly idle during that period. Note that it can go extremely large to several gigabytes, which is close to the size of main memory, if heavy workloads actively execute.

3.3 Analysis of Overhead

In this section, we carry out a detailed analysis on record mode, replay mode with checkpointing and VMI analysis one by one.

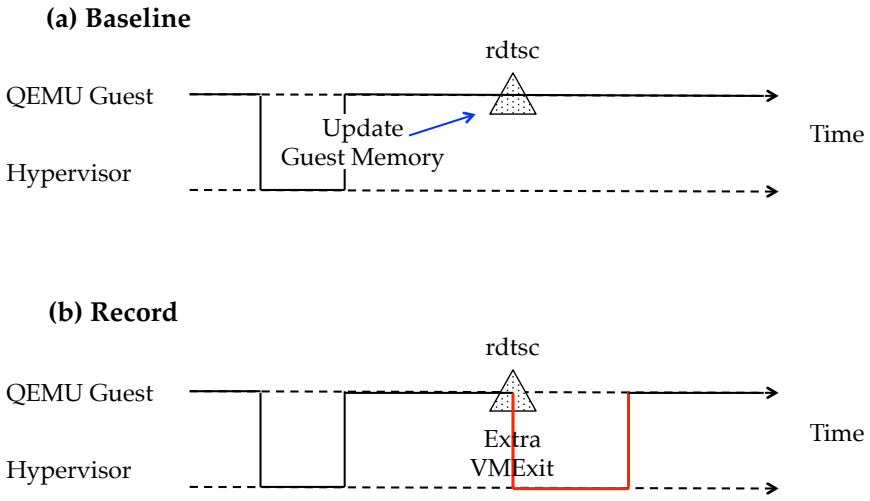


Figure 3.3: Record Overhead Example: *rdtsc*

3.3.1 Record Overhead

Compared to the baseline virtual machine execution, RnR adds three kinds of overhead at the record side.

Firstly, in order to capture all non-deterministic inputs and log them within hypervisor, RnR enforces guest virtual machine to generate extra VMExits, which can be avoided with certain optimizations though. As discussed before, synchronous events, which return non-deterministic results, have to be trapped into hypervisor, leading to the increase of VMExits.

For example, modern kvm leverages "kvmclock", a paravirtualized clock source, to enable *rdtsc* to obtain system-wide clock timer but only executing within the guest system. Specifically, the guest system set up a range of shared memory, and asks the hypervisor to write system clock timer to that memory space at every VMEntry. Figure 3.3 illustrates execution time spent in guest and hypervisor. We can see in baseline, *rdtsc* uses the value from last VMEntry to compute system timer, leading to no VMExits. While, to ensure deterministic replay, RnR has to record the return value of *rdtsc*. One VMExit inevitably needs to be inserted at every read TSC instructions. Each VMExit takes approximately 5000 cycles round trip, mainly due to context switch between supervisor mode and guest mode.

The second overhead comes from log management, including log file write operations. Generally file IO operations take relatively longer time than computation and memory accesses, and will easily block on the critical path.

To batch file writes and reduce the number of IO operations, "Insight" saves log entries and network packets content to a pre-allocated in-memory buffer, with adjustable size. File writes are only triggered when either the buffer is full or complicated QEMU user space operation is required. Later evaluation shows that it induces modest overhead for network non-intensive benchmarks.

The last component of overhead comes from the recording operation itself, which is copy value from registers or memory to target buffer. This overhead is ignorable, due to the lightweight memory copy operation it requires.

3.3.2 Replay Overhead

Deterministic replay enforces the guest system to repeat original execution by repeating all non-deterministic inputs to the system from record and re-inject them at the exact same point of execution. While in modern CPUs, it is not trivial to determine the execution point of the guest system. As discussed before, "Insight" utilizes the hardware performance counters to implement accurate instruction counting logic. Unfortunately, the imprecision of performance counter complicates the design and introduces significant overhead.

The hardware performance monitor interface provides support to preempt guest system execution when a certain performance event occurs, such as counter overflow. Specifically, "Insight" intends to configure the PMU to generate an interrupt when instruction retired counter overflows, and the interrupt then causes VMExit in guest virtual machine, to transfer control to the hypervisor. However, the deliver of performance counter interrupts can be delayed for several to hundreds cycles depending on the type of machines. To ensure replay correctness and robustness, a modified algorithm is designed to tolerate such imprecision as follows. First the branch retired counter is used instead of instruction counter, since the former one turns out to be more reliable. The branch counter is configured to overflow early enough to account for the non-deterministic length of delivery delay of the interrupt. After receiving the interrupt, the guest machine is configured to VMExit in a single-step way until the proper point of execution is reached.

As a result, a large number of VMExits are inserted to achieve precise instruction count. Therefore replay overhead is closely related to the number

of asynchronous events during the execution.

Besides, the other overhead during replay is caused by file operations. Similar buffer approach is applied to reduce the frequency of file operations that large number of log entries are read together into in-memory buffer. Thus log management takes small part in replay overhead.

Checkpointing Overhead

Efficient checkpointing (discussed in Section 3.2) implementation defers memory copying after virtual machine restart. It reduces the guest machine downtime and enables parallel execution of guest code and memory copying operations. Performance overhead due to checkpointing comes from two parts.

The first part is the overhead introduced during pre-copy stage. Before memory copying, virtual machine is suspended. Meanwhile, all pages belong to guest virtual machine are set as write-protection, and snapshot of disk image is taken. Different applications suffer similar amount of pre-copy overhead.

The second part comes from the post-copy stage. After the guest virtual machine is resumed, its execution is interrupted every time when memory writes are issued to read-only pages, leading to write protection faults. These software faults block the guest system execution until the copying of that page is finished.

The amount of post-copy overhead depends on checkpointing frequency and workload characteristics. On one hand, with longer checkpointing duration, more dirty pages need to be saved to the checkpointing, leading to higher checkpointing overhead. On the other hand, post-copy overhead varies with different system workloads. If the system is idle or executes applications with small memory footprint, checkpointing has relatively slight influence on system performance, by benefiting from the parallel copy operations and infrequent write-protection faults. On the opposite, there will be relatively higher performance deviation for systems running heavy workloads.

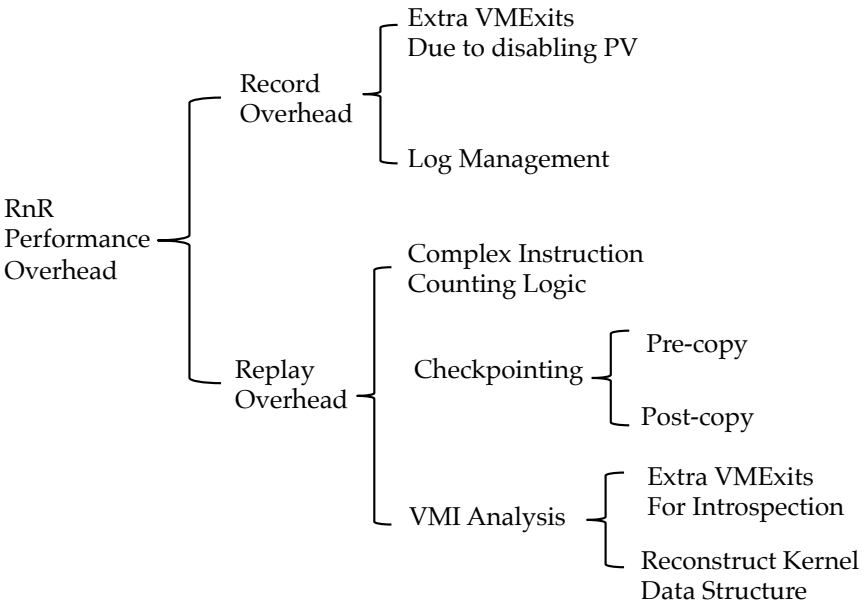


Figure 3.4: RnR Overhead Summary

VMI Analysis Overhead

In this thesis, we use synchronous VMI mechanism on commodity hardware to detect violations of system security invariance. Compared to other asynchronous mechanisms or approaches requiring hardware modifications, inline checks provide a more consistent view of the whole system for analysis and support both passive and active introspection. In spite of the higher performance overhead caused by the synchronous mechanism, it is still reasonable to be adopted in replay stage by offloading heavy weight analysis out of the execution critical path.

One restriction of VMI analysis in replay stage is that, the analysis should not interfere the guest virtual machine state, leading to replay divergence. Therefore only threat monitoring and verification are allowed. While Operations such as killing malicious processes are disallowed. Applying VMI analysis in replay introduces two kinds of overhead.

Firstly, common VMI analysis suffers from "semantic gap", which requires non-trivial techniques to reconstruct kernel data structures for analysis. Modern operating system is extremely complex, containing thousands of different data structure types and a large number of instances of each type. For example, a typical running instance of Linux kernel was found to have 231 different types of data structures and 29,488 instances in total, to

enable scheduling, memory management and IO support [63]. Before applying analyses accordingly, it is required to accurately extract data structures information and correctly distinguish different system objects.

A simple operation, to retrieve the process id of current running process, need to first locate *task_struct*, the data structure for process descriptor, and then retrieve the process id at the correct offset within the descriptor. More complicated information retrieval operations may requires traversing several nested structure pointers.

The other source of overhead comes from the extra VMExits enforced for the introspection point of interest. Transient malwares violate security policies temporally, and may slip through the cracks between two consecutive coarse-grained VMI checks. Therefore, hypervisor needs to enforce VMExits upon execution of certain operations to capture useful sensitive information of the system. Such introspection of interest is pretty useful to apply vulnerability-specific verifications. In IntroVirt [11], Joshi et al listed examples of predicates to check buffer overflow, data race condition and missing authentication. All these require security checks being applied at particular execution points and their corresponding memory states. VMExits are inserted to transfer control to hypervisor.

In summary, Figure 3.4 illustrates different kinds of overhead introduced by the RnR framework to the baseline system. On the record side, RnR causes extra VMExits by disabling paravirtualization of certain instructions and functions. Moreover, log management adds modest overhead due to time-consuming file IO operations. On the replay side, the major overhead comes from complex instruction counting logic, which is an essential component to ensure deterministic replay. In addition, pre-copy and post-copy overhead from checkpointing both slow down the guest virtual machine execution. Virtual machine introspection further increases the number of VMExits in replay. Meanwhile, another major overhad comes from the big effort paid to "bridge the semantic gap" by reconstructing kernel data structures.

CHAPTER 4

EXPERIMENT SETUP

We evaluates the performance of our recording and replaying modes. For this, we use Insight [3], a VM RnR tool based on a modified Linux KVM hypervisor and modifed QEMU devices. Since the KVM hypervisor can leverage Intel VTx extensions to virtualize the processor in hardware, the performance numbers from this setup are representative of real-world machines. We do not use Para-Virtualized (PV) drivers because they are Non-Deterministic (ND).

Host machine	
CPU: Xeon E3-64bit,4-cores,3.1GHz	Memory: 8 Gbytes
OS: Ubuntu, Linux kernel 2.6.38-rc8	
Guest machine	
CPU: uniprocessor	Memory: 1 Gbyte
OS: Debian, Linux kernel 3.19.0	Disk: 32 Gbytes

Table 4.1: System configuration for performance evaluation

Benchmark	Parameters
apache	-n100000 -c20
fileio	–file-total-size=6G –file-test-mode=rndrw –file-extra-flags=direct –max-requests=10000
make	linux-4.0 config with all-no
mysql	–test=oltp –oltp-test-mode=simple –max-requests=500000 –table-size=4000000
radiosity	-p1 -bf 0.005 -batch -largeroom

Table 4.2: Benchmarks executed

4.1 Record and Replay Setup

We use the same record setup as "Insight" [3], and augment replay with two functionality, i.e. checkpointing and VMI analysis. Our two replay setups are discussed as follows.

4.1.1 Replay With Checkpointing

To evaluate the overhead of replay with checkpointing, we reuse the Linux copy-on-write implementation used during fork system calls. Virtual memory belonging to the VM is allocated within a user-space QEMU process running on the host machine. By issuing "fork" system call on QEMU process, all the memory belongs to that process, including guest system main memory, is configured with copy-on-write.

4.1.2 Replay with VMI Analysis

Performance of VMI analysis is closely related with the frequency and complexity of the analyses applied. The more frequent and complex analysis triggeres higher overhead. We evaluate an analysis to list available processes at every context switch boundary to understand their quantitative effect on overall performance.

Unfortunately, current Intel VTx extensions do not support exiting on specified guest execution point. Hence, to measure the performance impact of replay with VMI analysis, we manually instrument Linux kernel source code by inserting a debug exception before kernel context switches. The debug exception is a single byte opcode (0xCC) used to trap instructions by raising debug exceptions. The VMCS is configured to cause VMExits on debug exceptions. The analysis itself takes approximately 10^6 cycles.

4.2 Time Measurement Mechanism

There are two instances of timer within virtualized platform, where one uses physical clock source and is located in host operating system, and the other is emulated by virtual machine monitor and mainly used by guest operating

system. We use the host system wall clock timer to accurately measure execution time of the guest system. To ensure the timer is started and stopped at the same system execution point, we manually insert debug interrupt before and after application execution and write a magic number into register *EAX*. Meanwhile, the VMCS is configured to trap every debug interrupt into hypervisor and the corresponding register value is checked whenever a debug interrupt is received.

CHAPTER 5

EVALUATION RESULTS

5.1 Record

The recording scheme records all non-deterministic inputs and generates the log. In addition, it cannot use para-virtualization (PV). We call the scheme *Rec*. Figure 5.1 compares *Rec*'s execution time to two other setups: no recording with para-virtualization (PV) (*NoRecPV*) and no recording and no PV (*NoRecNoPV*). The bars for each benchmark are normalized to *NoRecNoPV*.

We see that disabling PV increases the execution time of these benchmarks by 25-150%. This is because we now have VMExits due to clock reads, disk IO, and network activity. Apache and fileio are hit the most of the overhead, while mysql is not impacted as it avoids disk accesses by caching recently-accessed tables in memory. We need to record these events to be able to replay deterministically.

Recording (*Rec*) takes, on average, 28% longer than *NoRecNoPV*. To understand their overhead sources, Figure 5.2 shows the slowdown of *Rec* over *NoRecNoPV* and breaks it down into their sources, namely recording timer reads (*rdtsc*), port and memory-mapped I/O accesses (*pio/mmio*), interrupts, and network packet contents, and saving/restoring the RAS (*RAS*).

We see that the dominant overhead across all benchmarks is due to recording *rdtsc*. This event occurs very frequently, especially in fileio and mysql, where the application itself issues many timer reads to measure transaction speed. In addition, fileio issues disk command and control signals using pio. It also has DMA activity, which causes interrupt events to signal file access completion. Apache receives network packets and uses mmio accesses to the NIC to retrieve the packets. The more computation-intensive benchmarks (make and radiosity) have little overhead. Finally, saving/restoring the RAS induces only 4% overhead on average.

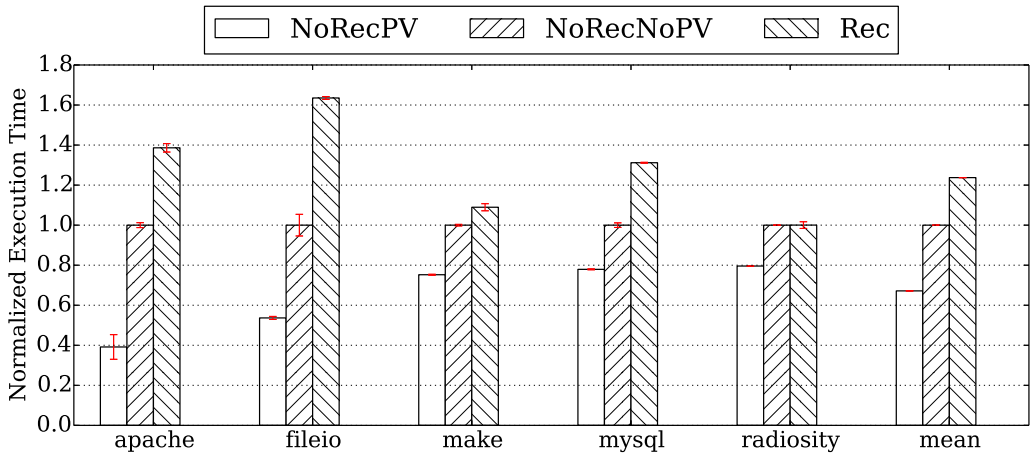


Figure 5.1: Execution time of recording setups

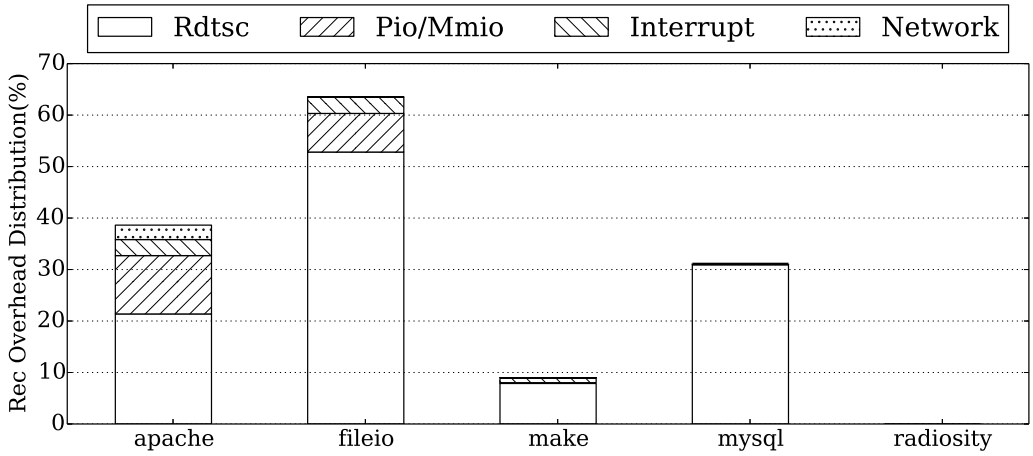


Figure 5.2: Breakdown of the *Rec* overhead over *NoRecNoPV*

Figures 5.3 show the input log generation rate for all our benchmarks. We do not compress the data. We see that the rates are typically low. Apache has the highest input log rate (4 MB/s) because it records packet contents.

5.2 Replay with Checkpoints

Figure 5.4 compares the execution time of various checkpointing replay setups to recording (*Rec*). The replay setups use no checkpointing (*RepNoChk*) or checkpoint every 5, 1, or 0.2 seconds (*RepChk5*, *RepChk1*, and *RepChk02*). The bars are nomalized to *Rec*. From the data, we see that checkpointing every 1 second (*RepChk1*) increases the execution time over *Rec* by 59%.

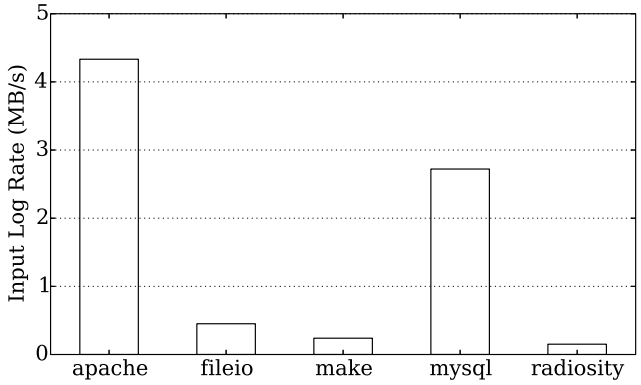


Figure 5.3: Input log generation rate

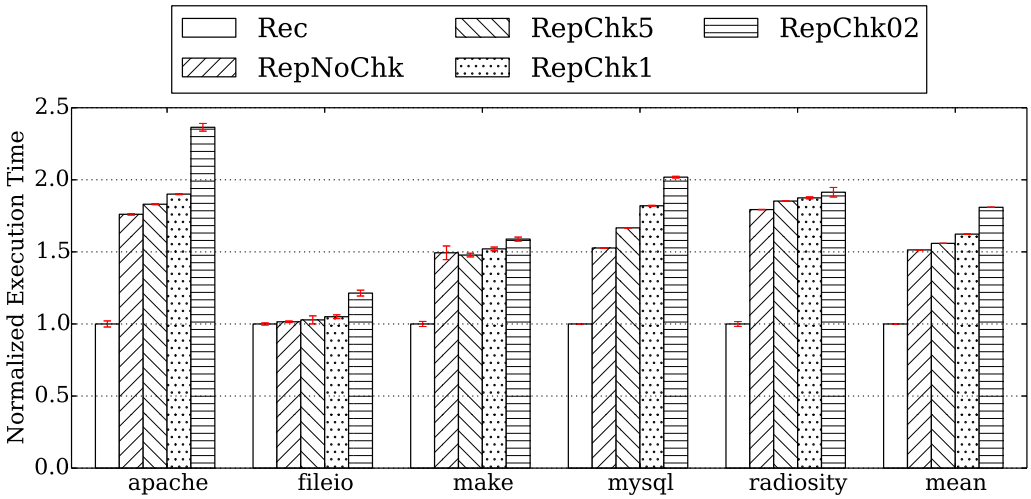


Figure 5.4: Execution time of checkpointing replay setups

These results show that checkpointing replay runs at a speed that is roughly comparable to that of recording. As a result, checkpointing replay can be on all the time. While checkpointing replay is a bit slower, it can catch up with recording because busy machines are rarely 100% utilized — they are often waiting for multiple reasons. During that time, recording slows down but replay can continue. If the replay gets significantly behind, we can use backpressure to temporarily stall recorded execution.

The figure also shows that increasing or decreasing the checkpoint period changes the speed a bit. Interestingly, even without checkpointing, replay already takes on average 48% longer that *Rec*.

To understand these effects, Figure 5.5 shows again the slowdown of *RepChk1* over *Rec* and breaks it down into their sources. The sources are those during

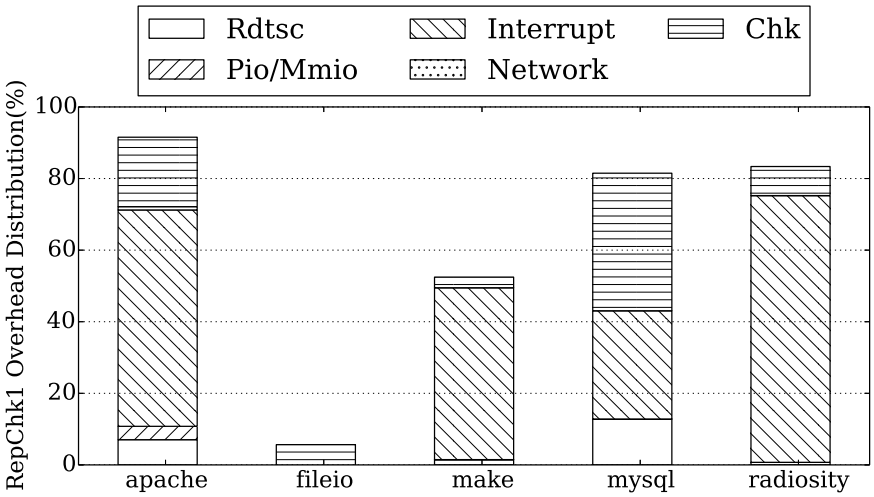


Figure 5.5: Breakdown of the *RepChk1* overhead over *Rec*

recording plus creating checkpoints (*Chk*).

The breakdown in the figure shows that creating checkpoints contributes only modestly, on average, to the total overhead. This is why replaying without checkpointing (*RepNoChk*) does not save much time over *RepChk1*. The actual overhead depends on the memory write characteristics of workload; poor memory locality causes more page copies, increasing checkpointing overhead.

Interestingly, we see that interrupt overhead dominates. The reason is that interrupts are asynchronous events, while *rdtsc*, *pio/mmio*, and *network* are synchronous. Identifying the instruction that should get the asynchronous interrupt injected during replay is time consuming. As indicated in Section **??**, it requires single-stepping VMExits over several instructions. This is the reason for the overhead of Figure 5.5.

5.3 Replay with VMI Analysis

Finally, Figure 5.6 compares the execution time of replay with VMI analysis (*RepVMI*) to previously shown environments: checkpointing replay (*RepChk1*) and recording (*Rec*). The bars are normalized to *Rec*. Replay with VMI analysis needs to trap on every context switch operation. Hence, the slowdown of this mode directly relates to how many context switches were executed when running specific workload. We see that replaying apache with VMI analysis

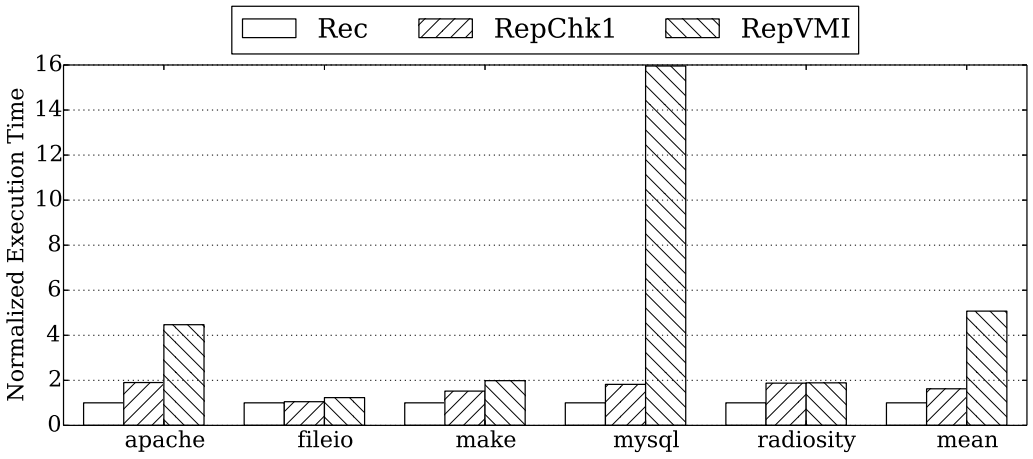


Figure 5.6: Execution time of alarm replay

at context switch boundary takes 4x longer than recording them. For mysql, it takes 16x. On the other hand, for filio, make, radiosity, with its modest kernel activity, it is around 2x.

CHAPTER 6

CONCLUSION

This thesis presents a detailed performance evaluation and analysis of virtual machine level record and replay framework and its related applications. Specifically, we evaluate 3 important modes in RnR, i.e. record, replay with checkpointing and replay with VMI analysis. The latter two modes are two representative usages of RnR.

According to our detailed analysis and quantitative results, record overhead mainly comes from the extra number of VMExits, which is caused by enforcing the guest virtual machine to trap into the hypervisor. It is necessary to assist logging synchronous non-deterministic inputs. Replay overhead is mostly caused by the complicated instruction counting logic, which is an essential component to enable replay of non-deterministic events at the exactly same system execution point as record. We find that accurate instruction counting algorithm suffers significant performance overhead from the imprecision of interrupt delivery mechanism in performance monitor unit. We show checkpointing overhead is closely related to the checkpointing frequency and the workload characteristics. Moreover, for replay with VMI analysis, the frequency and complexity are two major factors that affect overall performance.

We hope our results can be useful for both RnR developers and users. RnR developers can leverage our results to locate the major performance bottlenecks in the system and accordingly propose more efficient design. RnR users can have a more reasonable expectations of the system and customize RnR applications in a more performance efficient way, by setting appropriate checkpointing intervals or adjusting frequency of VMI analyses.

REFERENCES

[1] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 211–224, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844148

[2] A. Burtsev, "Deterministic systems analysis," Ph.D. dissertation, The University of Utah, 2013.

[3] R. Senthilkumaran and P. Kulkarni, "Insight: A framework for application diagnosis using virtual machine record and replay," 2014.

[4] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008, pp. 1–14.

[5] D. M. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," Ph.D. dissertation, University of California, Berkeley, 2006.

[6] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005, pp. 1–1.

[7] G. Lefebvre, B. Cully, C. Head, M. Spear, N. Hutchinson, M. Feeley, and A. Warfield, "Execution mining," in *ACM SIGPLAN Notices*, vol. 47, no. 7. ACM, 2012, pp. 145–158.

[8] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," ser. USENIX Ann. Tech. Conf., June 2004.

[9] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 80–107, 1996.

[10] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 30–39, 2010.

[11] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting Past and Present Intrusions Through Vulnerability-specific Predicates," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095820 pp. 91–104.

[12] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou, "Detecting covert timing channels with time-deterministic replay," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 541–554.

[13] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346321 pp. 308–318.

[14] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920313 pp. 347–356.

[15] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[16] 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," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[17] A. Velte and T. Velte, *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[18] "Qemu open source process emulator," http://qemu.org.

[19] T. Garfinkel, M. Rosenblum et al., "A virtual machine introspection based architecture for intrusion detection." in *NDSS*, vol. 3, 2003, pp. 191–206.

[20] B. Payne and M. Leinhos, "Libvmi," 2011.

[21] K. Nance, M. Bishop, and B. Hay, "Virtual machine introspection: Observation or interference?" *IEEE Security & Privacy*, no. 5, pp. 32–37, 2008.

[22] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. K. Iyer, "Reliability and security monitoring of virtual machines using hardware architectural invariants," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 13–24.

[23] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 233–247.

[24] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 477–487.

[25] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 1–14.

[26] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Vmm-based hidden process detection and identification using lycosid," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 91–100.

[27] G. Altekar and I. Stoica, "ODR: Output-Deterministic Replay for Multicore Debugging," ser. SOSP, October 2009.

[28] A. Basu, J. Bobba, and M. D. Hill, "Karma: Scalable Deterministic Record-Replay," ser. ICS, June 2011.

[29] T. Bressoud and F. Schneider, "Hypervisor-Based Fault-Tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, February 1996.

[30] Y. Chen, W. Hu, T. Chen, and R. Wu, "LReplay: A Pending Period Based Deterministic Replay Scheme," ser. ISCA, June 2010.

[31] J.-D. Choi and H. Srinivasan, "Deterministic Replay of Java Multithreaded Applications," ser. SPDT, August 1998.

[32] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling Dynamic Program Analysis from Execution in Virtual Environments," ser. USENIX ATC, June 2008.

[33] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution Replay of Multiprocessor Virtual Machines," ser. VEE, March 2008.

[34] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira, "Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism," ser. ASPLOS, March 2013.

[35] N. Honarmand and J. Torrellas, "RelaxReplay: Record and Replay for Relaxed-Consistency Multiprocessors," ser. ASPLOS, March 2014.

[36] D. R. Hower and M. D. Hill, "Rerun: Exploiting Episodes for Lightweight Memory Race Recording," ser. ISCA, June 2008.

[37] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging Operating Systems with Time-Traveling Virtual Machines," ser. USENIX Ann. Tech. Conf., April 2005.

[38] O. Laadan, N. Viennot, and J. Nieh, "Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems," ser. SIGMETRICS, June 2010.

[39] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Comp.*, April 1987.

[40] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism," ser. ASPLOS, March 2010.

[41] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," ser. ISCA, June 2008.

[42] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay," ser. ASPLOS, March 2009.

[43] S. Narayanasamy, C. Pereira, and B. Calder, "Recording Shared Memory Dependencies Using Strata," ser. ASPLOS, October 2006.

[44] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," ser. ISCA, June 2005.

[45] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: Probabilistic Replay with Execution Sketching on Multiprocessors," ser. SOSP, October 2009.

[46] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs," ser. CGO, April 2010.

[47] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas, "QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs," ser. ISCA, June 2013.

[48] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai, "Architecting a Chunk-Based Memory Race Recorder in Modern CMPs," ser. MICRO, December 2009.

[49] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, H. Jungwoo, and Y. Wu, "CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors," ser. MICRO, December 2011.

[50] X. Qian, H. Huang, B. Sahelices, and D. Qian, "Rainbow: Efficient Memory Dependence Recording with High Replay Parallelism for Relaxed Memory Model," ser. HPCA, February 2013.

[51] Y. Saito, "Jockey: A User-space Library for Record-replay Debugging," ser. AADEBUG, September 2005.

[52] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "DoublePlay: Parallelizing Sequential Logging and Replay," ser. ASPLOS, March 2011.

[53] G. Voskuilen, F. Ahmad, and T. N. Vijaykumar, "Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording," ser. ISCA, June 2010.

[54] M. Xu, R. Bodik, and M. D. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," ser. ISCA, June 2003.

[55] M. Xu, R. Bodik, and M. D. Hill, "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording," ser. ASPLOS, 2006.

[56] N. Honarmand and J. Torrellas, "Replay Debugging: Leveraging Record and Replay for Program Debugging," ser. ISCA, June 2014.

[57] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and V. Inc, "Retrace: Collecting Execution Trace with Virtual Machine Deterministic Replay," in *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, 2007.

[58] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh, "Pervasive Detection of Process Races in Deployed Systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043589 pp. 353–367.

[59] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685090 pp. 525–540.

[60] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, "Secloud: A Cloud-based Comprehensive and Lightweight Security Solution for Smartphones," *Comput. Secur.*, vol. 37, pp. 215–227, Sep. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2013.02.002

[61] Intel Corp., *Intel Virtualization Technology for Directed I/O*, October 2014, http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/vt-directed-io-spec.html.

[62] F. L. D. R. Gil Neiger, Amy Santoni and A. Rich UhligCharlesworth, "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization," *Intel Technology Journal*, vol. 10, no. 3, August 2006.

[63] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory," in *Recent Advances in Intrusion Detection*. Springer, 2010, pp. 178–197.