國立交通大學

網路工程研究所

碩 士 論 文

基 於 虛 擬 機 器 做 整 體 系 統 狀 態 回 復

VM-based Instruction Level Whole-system Replaying

研 究 生：許家維

指導教授：謝續平　教授

中 華 民 國 九 十 八 年 六 月

基於虛擬機器做整體系統狀態回復

VM-based Instruction Level Whole-system Replaying

研　究　生：許家維　　　　Student：Chia-Wei Hsu

指導教授：謝續平　博士　　Advisor：Dr. Shiuhpyng Shieh

國 立 交 通 大 學

網 路 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年八月

基於虛擬機器做整體系統狀態回復

研 究 生：許家維　　　　　　　　　　指導教授：謝續平 博士

國 立 交 通 大 學

網 路 工 程 研 究 所

摘要

在除錯系統跟惡意軟體分析領域之中，系統執行狀態回復是一項很重要的議題。系統回復不僅僅可以讓使用者清楚地知道當時出錯的狀況，也可以用於事後的還原。然而，現今的系統日誌和系統回復的技術並無法被廣泛的被利用。原因就在於系統回復的功能很難達成，且無法完全地、清楚地讓使用者詳細的知道系統狀況。系統回復實做的難處在於下列幾點：1)大多數的狀態回復，只能針對單一程序。2）由於要獲得回復的資訊，可能要修改現有的作業系統或是軟體。3）硬體中斷和程序排程的資訊很難從軟體層獲得。4）在系統回復的同時，要確保不會影響到執行的結果。基於以上四點原因，我們利用虛擬機器實做出一個具有回復性、準確性的回復系統。此系統對於軟體除錯、惡意程式分析和系統還原都具有極大的貢獻。由於本系統只考慮紀錄不可變因素，已達到有效的減低系統回復的運算量與其記錄的空間量。根據這些有效的記錄資訊，我們可以精確的回復系統執行狀態，甚至確保指令執行的順序不被更變，而達到更正確的分析結果。

VM-based Instruction Level Whole-system Replaying

Student: Chia-Wei Hsu Advisor: Dr. Shiuhpyng Shieh

Department of Network Engineering

National Chiao-Tung University

Abstract

Replaying of execution sequence and state transition of a system is very useful for software testing, malware analysis and post-attack recovery. However, existing system logging and replaying techniques have restricted abilities and hence cannot be applied widely. Most of them are unable to perform a general whole-system analysis for the following reasons: 1) It can only replay a single process's running. 2) Modification needs to be done in OS kernel 3) Non-deterministic events such as interrupts and context switches cannot be replayed. 4) An intrusive analysis might influence the replaying result. This paper proposed a general whole-system VM-based logging and replaying mechanism. To record efficiently, our scheme only takes non-deterministic information into account such as most hardware interrupts and non-deterministic data from external I/O devices. Based on the recorded data, the accuracy of the replaying is assured. The state transition of the whole-system can be perfectly replayed; even the execution sequence of all instructions is preserved.

誌　　謝

終於，到了寫致謝的日子。在這兩年內，很感謝身邊朋友的幫忙。首先，是我的指導老師謝續平教授。在剛進來實驗室的時候，發現自己對研究並沒有想像中的熱衷，是老師一而再，再而三的教導，讓我對研究產生了熱誠與興趣。再來是感謝剛進來的上一屆的學長姐，陪伴了我碩一的懵懂無知。尤其是繼偉學長，給了我很多啟發與經驗，當然討論出很多驚奇的夢想。當然還要感謝我媽，辛苦的賺錢養育我到研究所。雖然，我很少回家孝順老媽，但我想也可以回家看看我媽了。謝謝實驗室的助理們，明華無微不至的照顧與提醒，讓我很難把事情搞砸。麥可當初幫我負擔了很多計畫的事情，讓我專心的做研究。順瑩帶給實驗室的歡笑。還有，碩二的大家，好多好多事情發生在這兩年，真是有趣的一群人。感謝在柏克萊的學長Heng Yin，謝謝他在美國指導我。最後，特別感謝一個特別的好朋友---Norah同學。在美國研究的大半時間，有她陪著我一起度過難關，很感謝她的幫助與陪伴。如果少了她，這篇論文很難順利的產生，也因為她，我的論文有苦澀也有快樂。她的存在，賦予我的題目有更深的含意。一個人的力量真的很小，我所有的成功都要歸於其他人。

Table of Content

List of Figures

List of Tables

1. Introduction

Replaying of virtual machine (VM) execution[1-8] is very useful to be applied to software testing[3, 4, 9, 10] and malware analysis. Many aspect of software testing make it become a complicated work because following reasons. First, non-deterministic event always occur in process execution such as hardware interrupts and software exceptions. Sometimes those non-deterministic events would not be recorded in system logs or debug messages. What is worse that it is hard to reproduce these events in a short time, such as race conditions and deadlocks. In some cases, the event cannot be reproduced at all because it is timing-related. To review the information, replaying the whole-system is the only option we have. Second, intrusive analysis would affect its result. No matter how the analysis is performed, instruction must be injected, which would always affect timing, even scheduling. Virtual machines could help us analyze the target process non-intrusively.

Malware analysis is another application of replaying execution. For example, some malwares would only be activated at a specific instant. After that, they may delete themselves to erase evidence of existence. Mostly, people analyze the malware after attacks. With replaying ability, programmer can analyze the attack process of malware on the fly. Some powerful analyzers use emulation, which require a large amount of computation. The overhead makes the analysis environment different from the real world. Consequently, we need a transparent replaying tool for malware analysis.

Replaying has been drawing much attention due to its applicability. Numerous works has been done already. However, we believed that a successful replaying tool should provide following functionalities.

1.1. Whole-system recording & replaying

Conventional execution replaying techniques are defective on account of whole-system replaying. Some works record system calls history for replaying and ignore hardware interrupts. Thus they are heavily operating system dependent. It is a cumbersome work to maintain different recorder for different versions of system calls. There were also works which focus on replaying a single process by recording reads from IPC pipes, files, and all system calls. However, in modern software design a process may interact with numerous processes, libraries, or modules to accomplish its task. Without whole-system replaying mechanism, we can only acquire limited information. It may perform whole-system replaying by replaying all of them simultaneously; however the interaction and synchronization would be cumbersome.

Another reason for taking hardware interrupts into consideration is correctness. Without hardware interrupts recorded, the dependency of threads would be impossible to be replayed perfectly. Besides, some of instructions can return non-deterministic or timing-dependent values directly without invoking system calls such as ***rdtsc*** (Read timestamp counter). Neither system call trapping nor single process replaying can replay such execution.

1.2. Non-intrusive recording

Recording of execution sequence causes reasonable time and space overhead. Hence, intrusive system recording would impact on results of execution dramatically[11]. To avoid unnecessary influence, we need to place recording component outside the system. VM-based recording could achieve this goal with little performance overhead. Besides, a virtual machine gives us full control over the whole-system; therefore, timing inaccuracy caused by recorder can be eliminated. Another advantage of using a virtual machine is that we need only record interrupts and external I/O of VM, since does not all internal events are deterministic.

1.3. Replaying correctness

To perform a deterministic replay with correct sequence of execution, all inputs from external devices and unpredictable hardware interrupts must be recorded since they are non-deterministic factors. All non-deterministic events and their timing should be carefully logged, so we can replay them at the exact instant as in previous execution. By doing this, there is no non-deterministic factor in the replaying process, so that replaying correctness could be assured.

1.4. Transparent replaying

Most replay system claim they are fine-grain replay for analysis. Generally, the transparency depends on their analysis. For example, a database debugger tool always only care about the data dependency, a single process analyzer usually do not take the thread scheduling into account, a system developer just depends on log of system calls to imagine what taken place at system crash. But there are still many hidden information can make software analysis more clear such as content switch and hardware activity. That hidden information cannot be captured directly by software, so we need to use virtual machine to emulate above hardware activity. Our goal is that ensuring the instruction dependency is identical with previous execution, even the multi-thread process. In other words, we replay total order of instruction by our replay system.

In this paper, we proposed a whole-system, transparent and deterministic replaying system, which satisfy requirements listed above. It can replay the whole-system states instruction-by-instruction. Besides, we have a deterministic replay because of recording non-deterministic event in our logs. We implemented our system in QEMU, a faster dynamically binary translator, and can be used to integrate analysis tool for many purposes.

We can replay the transition of whole-system states with low space and compute overhead. Simultaneously, it is a transparent replay without any non-deterministic events.

The rest of paper is organized as follow. Section 2 gives introduction to related works. In section 3 we formalize the replaying problem definition. Architecture of our system and implementation is given in Section 4. Some possible future works are shown in section 5. Finally, we conclude our work in section 6.

2. Related Work

Rollback system recovery and VM-based log analyzer are very popular in recent years. With zero day attack raise, computer security faces an unprecedented challenge. Virtual machine can provide an isolated environment for analyzing malware. It is also used for standalone system debugging as honeypot. Following are related works on replay system state:

2.1. Jockey

Jockey[12] is a record and replay tool for single process debugging in Linux. It do not need to change the target binary and any programming language or API. It can record the non-deterministic data by pre-loading as module. Jockey segregates resource of target program for avoiding being compromised. But it cannot record hardware level non-deterministic event such as memory access races, thread scheduling interrupt and interrupts. Jockey only focus on one simply process debugging but cumbersome whole-system debugging.

2.2. Flashback

Flashback[13] provides rollback and replay ability for software debugging. It forks a new process as shadow processes for checkpoint. Then it captures the memory state at specific execution point and interaction between processes. Nevertheless, Flashback cannot replay thread dependency correctly because it is hard to trap the interrupt of thread scheduling. Without the correctly context switching, the dependency of thread would be change at re-execution. Thus, even Flashback is a lightweight replay for software debugging; it cannot debug some problems with hardware interrupts.

2.3. Revirt

Revirt[7] logs enough information for replay even in long-term system. Because of based on virtual-machine monitor (VMM), it needs to modify the kernels of guest OS. Revirt consist of two parties to monitor, one is guest user, and another is guest kernel. Both of them are building on host system as processes. By delivering signal SIGUSR1, the guest kernel can trap the system called by guest user. Additionally, it records non-deterministic events to follow a set pattern by using SIGIO and SIGSEGV. Revirt also replace some instruction can return non-deterministic results. Specifically, the ***rdtsc*** (read timestamp counter) and ***rdpmc*** (read performance monitoring counter) get CPU's information directly. It replaces that functionality by using other time-related system call. Thus the environment of whole-system would be some differences from real. Moreover, Revirt only replays specified guest system and have some restricts for guest OS. This feature makes Revirt losing generality for many applications. A useful replay system must achieve a transparent and general replaying for general purpose.

2.4. XenLR

XenLR[8] is achieved on a lightweight VM (Mini OS) replay. It causes a little time and space overhead to log the keystroke and time updating on Mini OS. XenLR do not think about file system and process interaction because of Xen, a vitalization VM so that many non-deterministic events cannot be capture. But in real system, the file system and threads are usually an essential part of the system.

2.5. BugNet

BugNet[14] focuses on replaying application's execution and sequence of memory access, not a whole-system replay. It collects the execution information of program before crash. It uses the First-Load Log (FLL) to record the load instruction return value. They also record synchronization information by Memory Race Logs (MRL) so that it can replay race condition of memory.

Even BugNet can replay at least tens of millions of instructions with low overhead, but it only cares about memory access. There are still many events it cannot replay such as hardware interrupts and I/O from device. Replay interval of BugNet can be terminated by interrupts and system calls, so it is still not enough for analysis if we wonder know what happened before several minutes.

2.6. FDR

FDR[15] can replay multi-processor with low overhead by using hardware recorder. To achieve faithful replay, all of external inputs in cache need to be recorded on each processor. It can provide approximately 1 second replay because of the size limitation of record buffer. However, FDR is hard to be applied widely because its short replay interval of system and hardware support. In our system, we use VM to make replay system more flexible to have long period replay.

2.7. ExecRecorder

Based on Bochs, ExecRecorder[5] perform hardware interrupts and whole-system replay. It can replay the executions of entire system by checkpoints and logs of non-deterministic events. A checkpoint is a duplicate of Bochs VM process via the fork system call. When replaying the system, ExecRecorder invokes the suspended child process by SIGUSR1. Same as our system, the implementation of ExecRecorder does not address DMA and multiprocessors. But Bochs has heavy computation overhead in emulation so that it is hard to be applied in large-scale analysis.

2.8. Summary

All of above replay system cannot guarantee that the instruction order is absolutely identical with previous run. Ignoring the instruction dependency does not impact the analysis result of execution because they only care about race condition between multi-threads. Mostly, there are many system issues in virtue of instruction dependency such as cause of program crash and activity of malware. By only synchronizing resource of processes, which erases much information for instruction level analysis. In our system, we focus on instruction dependency and reproduce the same execution order to ensure the faithful analysis result.

Name	**Record Hardware Interrupts**	**Record External I/O**	**Determinism**	**Hardware Support**
Application Level Replay				
Flashback	No	No	Single process, memory	No
Jockey	No	Partial	Single process, memory	No
BugNet	No	Partial	Single process, memory	Yes
System Level Replay				
Revirt	Yes	Yes	Multi-thread dependency	No
Our Scheme	Yes	Yes	Instruction dependency	No
XenLR	Yes	Partial	Keyboard input, Time	No
ExecRecorder	Yes	Yes	Multi-thread dependency	No
FDR	Yes	Yes	Multi-processor	Yes

Table 2.1 Related work comparison

3. Problem Definition

A machine state $\boldsymbol{s}$ is a set of states of register bank $\boldsymbol{r}$, memory region $\boldsymbol{m}$, and hardware disk content $\boldsymbol{hd}$. Let us denote $\boldsymbol{s} = \{\boldsymbol{r}, \boldsymbol{m}, \boldsymbol{hd}\}$. We state that $\boldsymbol{s} = \boldsymbol{s'}$ if and only if $\boldsymbol{s}$ and $\boldsymbol{s'}$ have same values in their register bank, memory and hard disk. Using the above definition of $\boldsymbol{s}$, the state of a virtual machine $\boldsymbol{M}$ at the time $\boldsymbol{i}$ could be denoted as $\boldsymbol{s_{Mi}}$. A special state $\boldsymbol{s_{M0}}$ stands for the initial system state. After executing $\boldsymbol{n}$ serious instructions, the trace of the state transition can be symbolized as $\boldsymbol{S_M = \{s_{M0}, s_{M1}, \ldots, s_{Mn}\}}$. Now, we can formally define the replaying problem:

Given $\boldsymbol{M}$'s trace of the state transition $\boldsymbol{S_M = \{s_{M0}, s_{M1}, \ldots, s_{Mn}\}}$, we say that machine $\boldsymbol{N}$ **replays** $\boldsymbol{M}$'s execution $\boldsymbol{S_M}$ if and only if $\boldsymbol{s_{Mn} = s_{Nn}}$. Note that above definition does not require $\boldsymbol{s_{Mi} = s_{Ni}}$ for all $1 \leqq \boldsymbol{i} \leqq \boldsymbol{n}$. However, we believed that $\boldsymbol{s_{Mn} = s_{Nn}}$ is sufficient to imply that in most cases.

Obviously, $\boldsymbol{s_0}$ is the only information required to replay a system $\boldsymbol{M}$ on $\boldsymbol{N}$ without any non-determinism. Unfortunately, it is not the common case. A useful computational system always interacts with external data and commands. To perfectly replay such a system with non-deterministic inputs, we need this information, too. Let us denote all non-deterministic inputs happened during $\boldsymbol{M}$'s execution as $\boldsymbol{I_M}$.

In this work, we tried to implement a system which can:

(1) Record $\boldsymbol{I_M}$ during $\boldsymbol{M}$'s execution.

(2) Replay $\boldsymbol{M}$'s execution $\boldsymbol{S_M}$ on $\boldsymbol{N}$ by feeding $\boldsymbol{N}$ with $\boldsymbol{s_{M0}}$ and $\boldsymbol{I_M}$.

4. System Overview

Our system captures non-deterministic events by using virtual machine to achieve whole-system replay. We can roughly divide the replay procedures into three works. Firstly, virtual machine plays an important role of intercepting all the information of system. Secondly, we extract the non-deterministic events which are necessary information from record. Finally, the system state can be correct replayed with our recorded data. We would describe them in following.

4.1. Virtual machine

Virtual machine emulates hardware activity faithfully by software programming. In generally, the execution would be very similar to the physical computer. Because all the information of execution can be tracked, it is powerful to be applied for whole-system debugging. For example, we can access any location of memory or monitor all the executed instruction by CPU. Thus we implement our replay system on virtual machine.

QEMU is a full system emulator in which unmodified operating system by using dynamic translator. The dynamic translator performs a runtime conversion of the target CPU instruction into the host instruction. Because of dynamic translator can be modified to add user functionality, every instruction would be monitored and controlled. QEMU also emulates interrupts and exception for correct execution. Hardware interrupts can be trapped and recorded for replay. Besides, QEMU enable external inputs from users such as mouse and keyboard and network. In other words, all the necessary replay information is included in serial states of QEMU.

4.2. Tracking Non-deterministic Value

By monitoring virtual machine, replay of system is taken as matter of course. With ability of hardware computation rising, it is impossible to record all the state of system execution. However, a practical replay system should be low overhead. In our system, we implement an efficient re-generated whole-system replay of execution. To find the factor of system state transition is significant for decreasing complexity of recording. The source of factors can be called non-deterministic events because they cannot be predicted the timing and the value. With recorded non-deterministic events, replaying system can be simplified by using them to re-generate serious execution. For example, when we type a character from keyboard, the hardware interrupt is send before load keyboard value from port I/O. There are two non-deterministic cannot be re-generated in next execution. One is timing of interrupt and interrupt number, another is timing of load instruction and return value. If we replay those non-deterministic events exactly, the result of execution must be completely replayed with the same result of execution.

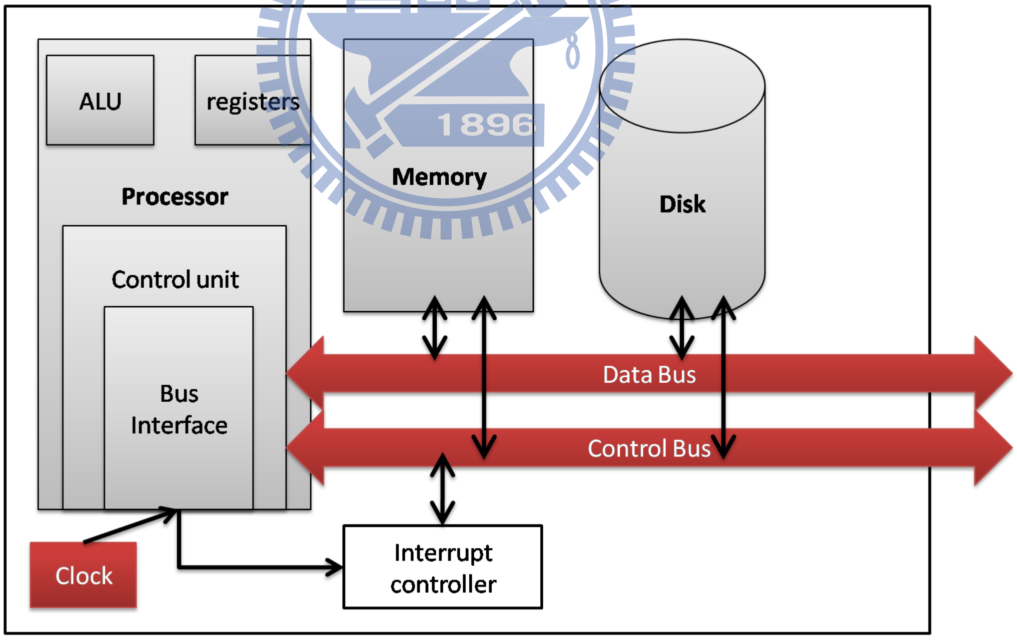

Figure 4.1 Replaying scope of virtual machine

4.3. Record & Replay

Efficient recording and fine-grain replaying is a trade-off. In recording phase, we hope the less computation the better. By doing this, the recording overhead is decreased but it might cause the replay information ambiguous. For example, if the packets from the Internet do not be recorded, it is hard to make the execution result correctly because content of packets are difficult to be re-generated. Our system takes many factors which can impact the execution into account such as user and hardware activity. We considerate those factors as non-deterministic events for replying.

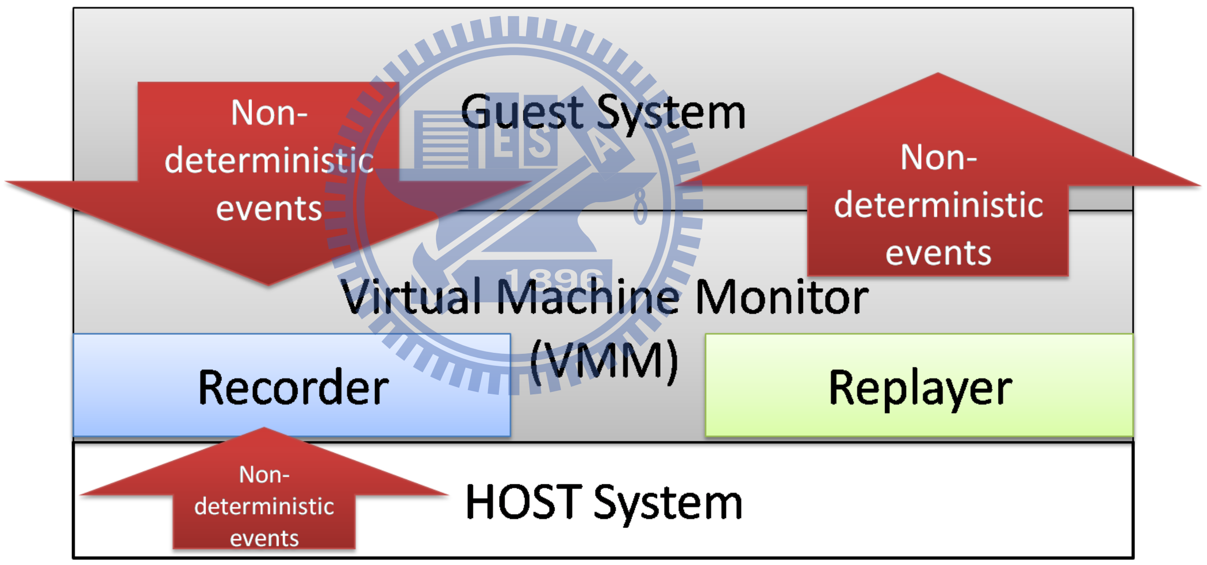

Figure 4.2. Our system overview

5. Non-determinism

A non-deterministic event is one of the reasons for transiting system states. In our system, we considerate all the deterministic event, or you can call them arithmetic, are in the black box instead of non-deterministic event. The states of external devices, such as hardware disk, network card, keyboard and mouse, can be ignore because they are too complicated to synchronize their states as previous execution. However, it is also impossible to re-generate a real hardware signal during replay. Thus we record the data in VM hardware simulator, and then reply the return data at exact time. Our system only cares about hardware interrupt, port I/O, memory-mapped I/O and DMA (direct memory access). We discuss in details following.

5.1. Hardware Interrupt

Computers need interrupts to communicate with each external device. Numerous tasks are accomplished by them. For example, context-switching, disk I/O, DMA data transfer, etc. Interrupting is a way to disturb CPU execution, and hence becomes a source which generates non-deterministic events. However, not all of them are non-deterministic. For example, exceptions and software interrupts could never be non-deterministic. Exceptions are generated by deterministic execution of instructions existing in memory which remains constant before any hardware interrupt. A software interrupt can only be generated by instructions existing in memory, which is internal data storage. In other words, we can ignore all the exceptions and software interrupts and reduce time and space usage when recording.

QEMU translates the binary to translate blocks for execution. It will not be interrupted at anytime, so it checks whether interrupts are peddling or not before execution function. We modify this function for recording our interrupt, including instruction counter, which represents how many instructions are executed, interrupt number, the number indicate what should the PIC (programmable interrupt controller) handled.

5.2. External Input

A non-deterministic event is one of the reasons for system state transitions. In our system, we only take account of the difference of execution between runs and record factors of change. However, system transition is arithmetic event so that it cannot change result with identical inputs. Thus we do not considerate all the outputs of every device. Also, we only care about the state of VM but other external device. The states of devices outside VM, such as hardware disk, network card, keyboard and mouse, can be ignore because they are too complicated to synchronize when replaying. For example, we can simply generate a keyboard interrupt in VM without synchronizing the state of real keyboard. It has no influence if we do not synchronize their states. Therefore, we record all non-deterministic events during original execution, and then reply them at exact time. Our system only takes hardware interrupts, port I/O, memory-mapped I/O and DMA (direct memory access) into account to record and replay efficiently.

Port I/O and memory-mapped IO are the same concept of the system inputs. Almost non-deterministic inputs of the system are from user or other computer. For example, a user types commands as input or receives a packet from other computer. Programs cannot decide when those data coming, this is a reason that external inputs are non-deterministic.

5.3. Time Related Instruction & Clock

Some instruction, ***rdtsc*** (Read Timestamp Counter) and ***rdpmc*** (read performance counter), can access data of CPU directly. It is possible to return different value in re-execution time as non-deterministic events. Because they cannot be trapped by hooking system calls or monitoring hardware interrupts, those instructions need to be handled to achieve record and replay ability. However, we want to ensure the time of replay system is identical with original run, so the return value must be replayed correctly with same value and same related time.

All of hardware devices need clocks to work correctly. QEMU emulates clock interrupt which is used for context-switch by host system timer. When host timer sends a signal to QEMU, it will check which task is expired and selects next task from waiting queue. Replaying clocks becomes a difficult mission that we need to trigger the clock interrupts with correct value and correct timing. There are tens of millions instructions per seconds are computed on VM, and we need to decide which instruction should be corresponding to clock interrupts for context-switch. Without the same timing of content-switch, the order of instructions cannot be replayed because of different task dependency.

5.4. DMA and Multi-processor

DMA is another non-deterministic event in our system. DMA allows devices within the computer to access system memory independently of the CPU. We have to handle with this, because it would affect the integrity of data for read and write. This event is very difficult to synchronize between original executions and replay of recorded. When DMA happening, QEMU would read host disk for simulating the guest system's disk. In the real world, it is unlikely for accessing disk in the same time. However, we plan to propose a transparent replay system, and ensure all the sequence of execution is the same as previous. Synchronizing the state of memory and device transition is tricky, so we do not address this problem into our implementation. Therefore, all of DMA events are blocked during recording. By doing this, there are no any factors that can impact the correctness of replay system.

Multi-processor is current trend of operating system. To synchronize the recourse of processor makes system debugging more complicate than before. In our system, we let the instruction order of re-execution is completely the same as recorded. Therefore, replaying transparent multi-processor execution is so tricky that we do not solve this problem in our system. We disable the multi-processor ability on QEMU for correctness of replay.

6. Implementation

In our system design, there are three components compose system replay ability record and replay. Firstly, we need an initial checkpoint, which contains all the machine states. Secondly, non-deterministic events must be recorded by recorder. Finally, replayer uses the record to recover system execution. Following are our descriptions in detail.

6.1. Checkpoint

Checkpoint saves the virtual machine states, including states of device, memory, CPU register and the content of all the writable disks. The program execution is divided into multiple checkpoints. We can replay the system by using interval of checkpoints. Mostly, the early checkpoint is used to be the beginning of replay, and the later one is used to check the correctness of replay.

We combine QEMU's snapshot with following information into our checkpoint header:

Checkpoint Identifier (CID): Checkpoint identifier is used to decide the interval of replay. There would be many checkpoints on virtual machine. For flexible replaying, we can choose two of all the checkpoints to be replay interval.

Interval Instruction Number: The interval instruction number help replay system to know when should stop replay. If the checkpoint is not last one, the interval instruction number would be the number of executed instructions to next one.

Virtual Machine Snapshot Identifier: To record the states of virtual machine have been implemented by QEMU's snapshot. We integrate the snapshot into our checkpoint information. The ID of snapshot is associated with the recorded VM state.

Log File Offset: When the VM is replaying, file offset can let replay system to know where the data begin. Because all the recorded data are in the same file, we can seek the record by using log file offset.

6.2. Recorder

The record component logs the entire non-deterministic event into files. The component is made up many modification of QEMU. This part would record every non-deterministic event with ordering of instruction execution. Recorder is composed of many modifications to record emulated I/O and interrupts. We check the state of VM is recording or not when I/O operation or interrupts are taking place. If the state of VM is recording, we will record current data from I/O port or number of interrupt by our record function. We separate our implementation into several parts and describe them in detail as following.

6.2.1. Instruction Counter

To replay instruction dependency correctly, instruction counter plays an important role for instruction-related sequence. It counts how many instructions is virtual CPU computed. Each recorded event will be corresponding with one instruction counter. Instruction counter reveals not only the order of recorded events but also exact timing of replay. Additionally, it can help us to debug replay system when the sequence of re-execution events do not matched with recorded data.

6.2.2. External Inputs

Inputs can change execution result and influence execution state of process. For an operating system, transition of execution state can be varied by many kinds of external inputs, e. g. network packets, keyboard typing and mouse clicking. We consider user behavior and peripheral activity which are mentioned above to be non-deterministic events because of their unpredictability. Consequently, recording non-deterministic events into files makes them to be predictable during re-execution run. Besides, we also record additional information with data type for decreasing log size. For example, the IA-32 allows transmission of data type can be a byte, word or a double word between peripheral. We compress additional information into simply integers to reduce the space overhead.

6.2.3. Interrupts

Interrupts are used to accomplish communication not only hardware but also software. System execution always accompanies interrupts when I/O request is issued or context-switch in multiprocessing system. QEMU invokes interrupts during executing instructions of guest system and handles them periodically. We record the interrupts in QEMU's handler function with corresponding instruction counter.

6.2.4. Clock

All of real world devices need clock to synchronize their operation, and QEMU, a machine emulator has no exception. QEMU emulates clock interrupts by using signal (SIGALARM, SIGIO), and those virtual interrupts are triggered when the specified signal are arriving. However, VM cannot predict host system activity, so it always emulates interrupts handling after emulating amount of code execution. To faithfully replay thread scheduling, the timing and the value of clocks must be recorded with corresponding instruction counter.

Otherwise, it stands for the related timestamp. It is very significant for some of time dependent event such as rdtsc, rdpmc. We combine all above information for recording, a very simple format for decreasing record space.

6.3. Replayer

We implemented transparency and deterministic replay in our system. All of the recorded data are corresponding to instruction counter for ensuring time dependency. Transition of the system state will be the same as previous recorded, including thread scheduling, interactions of process, hardware interrupts from external device and even data of packets from network. However, we do not take the fine-grain replaying of external devices states into account. One of the reasons, synchronizing state of external devices is more complicated than only replaying their output. For example, you can regenerate a packet from a website, but change the website status to send the packet like previously. In other words, states of external devices are not in our concern so that their states would not be transparent.

Non-deterministic events are the determinants for achieving deterministic replay. A non-deterministic event causes a state transition without corresponding to previous states. By checking instruction counter, system states can be replayed predictably because of removing the entire non-deterministic factor. However, we let all the events be deterministic; in addition to we cannot handle such as DMA. During replay of system, all system events are deterministic.

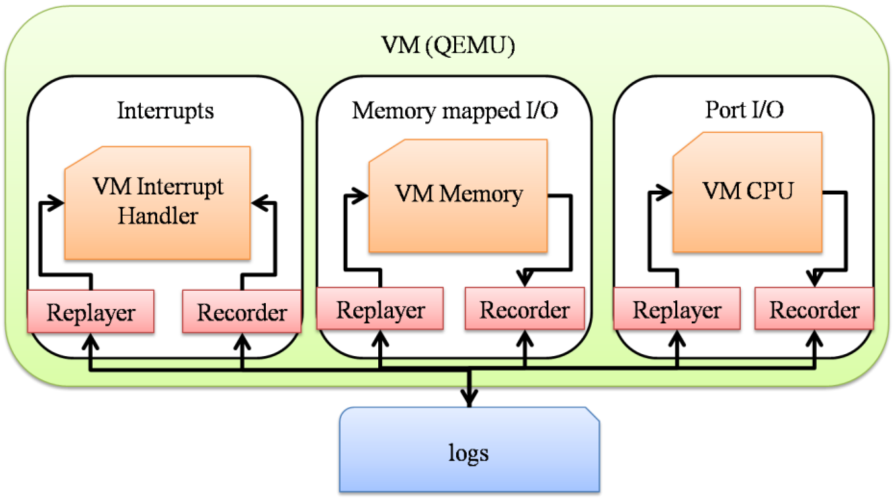

Figure 6.1 Implementation of recording and replaying

7. Evaluation

Our replay system can be evaluated by two aspects; one is correctness, another is performance. First of all, we describe our experiment environment and design in detail. Secondly, correctness of system replaying can be verified by multi-threads scheduling. Finally, we evaluate the space overhead of recorded data.

7.1. Evaluation Method

To evaluate our system, we load a VM image of Ubuntu 9.04 which is one of popular Linux distributions for our experiment environment. It provides completely functionality for normal users, so our experiment result can be very close to real world system replaying. Besides, Ubuntu is convenient to evaluate replaying correctness, and replaying common operating system is more convinced than only replaying specified or small linux system. Our replay system only connects keyboard and mouse. The host system runs on Intel 2.5 GHz Dual core and 8GB of RAM.

To evaluate computation correctness, we design experiment to confirm that instruction dependency can be replayed. We write a testing program and the outputs will be delivered to host system. After replaying, guest system will create another copy in host system. By doing this, we can compare the content of two logs to ensure correctness of replaying. Simultaneously, we monitor the increase of log size and set host timer to calculate how much time does the experiment take. We evaluate correctness, space-overhead and computation –overhead in detail following.

7.2. Correctness Evaluation

For general purpose, our replay system can replay many Linux command such as ***ps*** (lists all the active processes state), ***ls*** (lists directory content) and ***date*** (shows the system time). To replaying the foregoing commands, we must replay correct interrupts, user behavior, external inputs and time-related event (***rdtsc***). However, those events are instruction-dependency senseless that cannot highlight the accuracy of our replay system. Thus we design further complicated evaluation for instruction-level replaying.

Instruction dependency can be reduced into multi-threads dependency. Because of thread is the smallest computing unit in most of operating system. Our system faithfully replays the timing of context-switch, and feeds recorded input value for correct condition branch. Therefore, we will have the same instruction dependency as recording runs.

We implemented two experiment of multi-thread competition. Firstly, printing thread identity is a simply way to show the multi-threads ordering. Here is the sample code that prints the thread number. Every created thread is locked until the variable ***go*** is set. Because it is hard to predict which thread will be selected, the sequence of thread number is non-deterministic in real system. The possibility of re-generating the sequence is 1/***n!***, in other words, it is impossible to create two identical sequence when ***n*** is a large number. In our system, all of non-deterministic events are recorded including context-switch, thus we can replay the scheduling problem even with 100 threads.

```
void *hello(void *arg)
{
   while( go == 0 );
   parm *p=(parm *)arg;
   printf("[%2d] \n", p->id);
}
int main(int argc, char* argv[]) {
   Initialize();
   for (i=0; i<n; i++)
      my_thread_create( hello , i );
   go = 1;
   OtherCode();
}
```

Figure 7.1 Example of printing thread number (exp1)

Another experiment is thread race-condition. The different instruction dependency will cause different result of access competition. Figure 4 is producer-consumer problem example code. Producers increase the share instance, and consumers decrease it. All of created threads modify the same variable ***item***. Because it cannot be guaranteed that access variable ***item*** is atomic, some modification failures are due to other threads overriding. This experiment is repeated many times with 40 threads and save those output into log. Then we confirm whether the output is matched during replaying.

```
void *producer( void *arg){
   while( go == 0 );
   item++;
}
void* consumer(void * arg){
   while( go == 0 );
   item--;
}
int main()
{
   Initialize();
   for (i=0; i<n; i++){
      my_thread_create( producer, i);
      my_thread_create( consumer, i);
   }
   OtherCode();
}
```

Figure 7.2 Example of producer-consumer problem (exp2)

7.3. Performance & Storage

Our replay system takes a few space-overhead and reasonable computation-overhead. Space-overhead of log are very small because of selective recording. Our replay system does not record all the executed instruction and all the I/O. We only take non-deterministic events into account for efficient recording. The log size average 350KB per minute; on the other hand, the recorded data increases 5.3 bytes per thousand of instructions.

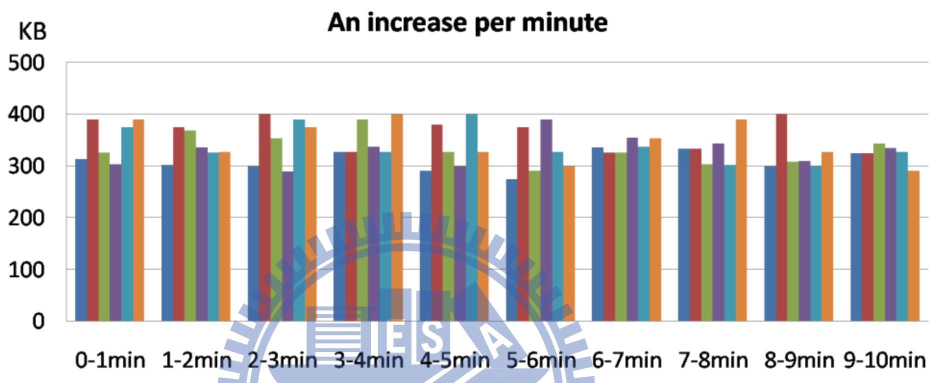

Figure 7.3 Increase of log size per minute with random number of threads printing

The computation-overhead depends on the complexity of guest OS. We randomly execute linux command on small linux guest system, and execution time will cost about double times than native VM. Additionally, we also replay user behavior on Windows XP guest system, but the performance is not as good as small linux. The reason of computation slow down is that our replay system executes many condition branch and instructions. To count computed instructions, every instruction on guest system need to append an addition to instruction counter. However, the overhead of replaying is more than recording, because the recorded non-deterministic events must decide whether replays them or not each instruction. Otherwise, the number of non-deterministic events will affect the replaying performance. The more data is recorded, the slower replaying is.

Computation overhead related to native emulated VM	
Small Linux	2.2x ~ 2.5x
Windows keystroke	2.5x ~ 3.5x
Ubuntu	2.5x ~ 3.5x

Table 7.1 Computation-overhead of replaying

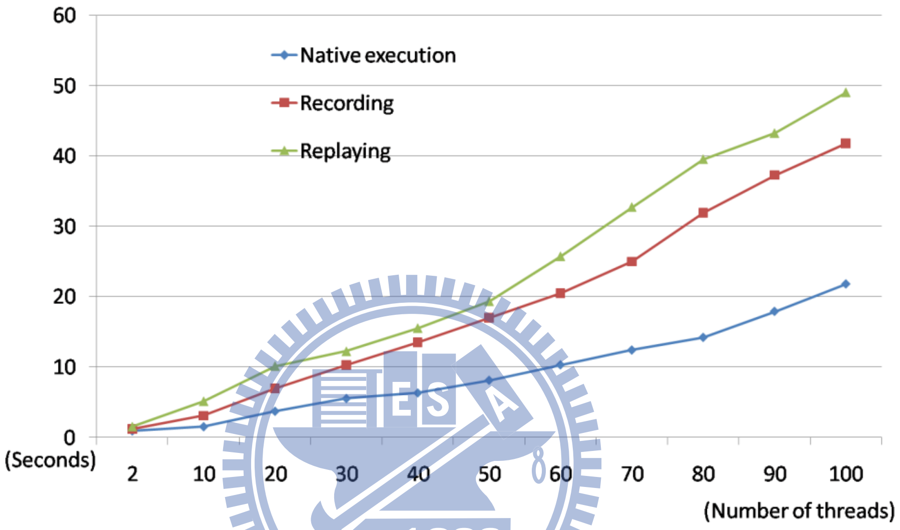

Figure 7.4 Execution time of thread printing (exp1)

8. Future Work

Our system is extendable for replay of DMA and malware analysis. Even replay of DMA is a difficult problem; because of DMA is asynchronous IO operation. When guest system launches a DMA transmission, the virtual CPU will wait for virtual DMA interrupts for ensuring job finished. The arriving time of virtual DMA interrupts depend on host system IO operation. Unfortunately, we cannot predict the time consumption of host system that it is hard to replay the virtual DMA interrupts correctly. But it still has a solution for replaying. We can halt the VM for waiting the host system data access finish. By doing this, guest system can access the data with correctly timing and keeps system states in a good shape.

Additionally, we plan to integrate a security analyzer in our system for security analysis tool such as dynamically taint analysis or intrusion detection system. It would be helpful for any kind of expensive compute consumption analysis.

9. Conclusion

Replaying execution sequence and recording the state transition of a system are very useful for software testing, malware analysis and post-attack recovery. Unfortunately, current approaches cannot provide complete replaying functionalities therefore they cannot be applied widely. Whole-system replay can recover the complete situation of previous execution time. Non-intrusive recording can prevent the result from being interfered. Replaying correctly can guarantee that the sequence of executions is identical with that of the previous execution.

We implement a system based on QEMU for transparent and deterministic whole-system replay. Moreover, we can save the space of logs by only caring about the impacts of non-deterministic. The sequence of system replay must be deterministic if every factor of states transition is predictable. Our system is extendable by integrating other security analysis tool, thus enables its wide application in the future.

10. Reference

[1] P. Montesinos, L. Ceze, and J. Torrellas, “DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Ef?ciently,” in Proceedings of the 35th International Symposium on Computer Architecture, 2008.

[2] P. Montesinos, M. Hicks, S. T. King *et al.*, “Capo: a software-hardware interface for practical deterministic multiprocessor replay,” in Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, Washington, DC, USA, 2009.

[3] 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, Anaheim, CA, 2005.

[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, Boston, Massachusetts, 2008.

[5] D. A. S. d. Oliveira, J. R. Crandall, G. Wassermann *et al.*, “ExecRecorder: VM-based full-system replay for attack analysis and system recovery,” in Proceedings of the 1st workshop on Architectural and system support for improving software dependability, San Jose, California, 2006.

[6] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman *et al.*, “Execution replay of multiprocessor virtual machines,” in Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Seattle, WA, USA, 2008.

[7] G. W. Dunlap, S. T. King, S. Cinar et al., “ReVirt: enabling intrusion analysis through virtual-machine logging and replay,” SIGOPS Oper. Syst. Rev., vol. 36, no. SI, pp. 211--224, 2002.

[8] H. Liu, H. Jin, X. Liao et al., “XenLR: Xen-based Logging for Deterministic Replay,” in Proceedings of the 2008 Japan-China Joint Workshop on Frontier of Computer Science and Technology, 2008.

[9] F. Qin, J. Tucek, Y. Zhou et al., “Rx: Treating bugs as allergies---a safe method to survive software failures,” ACM Trans. Comput. Syst., vol. 25, no. 3, pp. 7, 2007.

[10] J. Tucek, S. Lu, C. Huang et al., “Triage: diagnosing production run failures at the user's site,” in Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, Stevenson, Washington, USA, 2007.

[11] K. Nance, M. Bishop, and B. Hay, “Virtual Machine Introspection: Observation or Interference?,” IEEE Security and Privacy, vol. 6, no. 5, pp. 32--37, 2008.

[12] S. Yasushi, “Jockey: a user-space library for record-replay debugging,” in Proceedings

of the sixth international symposium on Automated analysis-driven debugging, Monterey, California, USA, 2005.

[13] S. M. Srinivasan, S. Kandula, C. R. Andrews et al., “Flashback: a lightweight extension for rollback and deterministic replay for software debugging,” in Proceedings of the annual conference on USENIX Annual Technical Conference, Boston, MA, 2004.

[14] S. Narayanasamy, G. Pokam, and B. Calder, “BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging,” in Proceedings of the 32nd annual international symposium on Computer Architecture, 2005.

[15] M. Xu, R. Bodik, and M. D. Hill, “A "flight data recorder" for enabling full-system multiprocessor deterministic replay,” in Proceedings of the 30th annual international symposium on Computer architecture, San Diego, California, 2003.