XenLR: Xen-based Logging for Deterministic Replay*

Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
hjin@mail.hust.edu.cn

***Abstract*—*Virtual machine* (VM) based logging-and-replay technology has attracted much attention for system security, fault tolerance, and debugging. However, so far there is not a replay system designed on virtual machine monitor Xen. In this paper, XenLR presents the design and implementation of a logging tool for full system replay on Xen. To reduce the design complexity, XenLR is achieved on a lightweight VM mini OS. Our preliminary work dedicates to identify architectural non-deterministic events and record them compactly. XenLR does not modify the guest OS and only need to record the external inputs (keyboard input and time update on mini OS). The log data are saved in the privileged domain's file system through transmission channels. The experiments indicate that XenLR results in low time and space overhead. Overhead due to logging is imperceptible for interactive use. The log files grow at 1.4MByte/day, so it causes litter space consumption for today's large capacity disk storage.**

***Keywords* - Xen; log; replay; non-deterministic events; branch counter**

I. INTRODUCTION

Full system logging and replay technology is heavily studied for system reliability and security recently. In conjunction with a checkpoint of the system state, execution logging and replay gives the ability to reconstruct the past entire state at any point in time over the replay interval. Such ability makes this technology widely used to perform analysis of malicious code attacks, system recovery from a fault or misused state, and system debugging.

A logging-and-replay system must incur low space and performance overhead, log full system running state, provide full system replay capabilities, and be resilient against attacks. Previous approaches fail to meet these requirements. Most of them log and replay only a single process or application [1, 2]. Some require changes in the host and guest OS [3], or do not have yet a fully implemented replay component [4].

A simple way to apply logging and replay to a wide range of software is to implement it with virtual machines. Running software in a virtual machine allows a user to take advantage of execution replay without modifying software. It also has the advantage of being able to use execution replay on an operating system kernel.

VM-based logging and replay technology brings great benefit for system security. Although both UNIX and Windows OS can log various system events, the audit logs provided by current systems fall in two ways of what are needed: integrity and completeness. Current system loggers lack integrity because they assume the operating system kernel is trustworthy; but when the operating system is attacked they may become ineffective. Current system loggers lack completeness because they do not log sufficient information to recreate or understand all attacks. With virtualization technology, these two problems could be solved easily.

The *virtual machine monitor* (VMM) [5] makes a much better trusted computing base than the guest operating system due to its narrow interface and small size. The narrow VMM interface restricts the actions of an attacker and this smaller code size makes it easier to verify the VMM. By logging the events in the narrow interface of the VMM, any exception information of one domain can be saved easily onto other domains, thus provide much more secure for the system audit logs.

There are several projects focus on virtual machine based system logging and replay, yet each of these systems has some shortcomings: the VMM they chose has detrimental consequences for performance; some need to modify the guest OS; some have too large size of log file; some use the system time of guest OS. These issues limited the scope of application scenarios.

This paper studies the full system logging issues based on virtual machine technology. We present a novel log system (XenLR) to handle aforementioned problems. XenLR is ported on Xen [6] that takes para-virtualization approach. XenLR does not need to modify the guest OS. Our log-and-replay prototype is implemented by logging and replaying non-determinism events on x86 uniprocessor with support of performance counter. In order to reduce the amount of log information that needs to be used for replay, we only record the non-determinism events, which affect the process's computation state. Logging should identify non-deterministic events and encode them compactly. The log files are saved beyond the logged domain and organized to different structures with the different events.

The rest of this paper is organized as follows. Section II introduces the related work. The design and implementation details are described in Section III. Section IV presents the

* The paper is supported by National 973 Basic Research Program of China under grant No.2007CB310900.

978-0-7695-3540-1/08 $25.00 © 2008 IEEE
DOI 10.1109/FCST.2008.31

performance evaluation and demonstrates the effectiveness of our system. We conclude this work in Section V.

II. Related Work

BugNet [4] is based on the insight that recording the register file contents at any point in time, and then recording the load values that occur after that point can enable deterministic replaying of a program's execution. BugNet obviates the need for tracking program I/O, interrupts and DMA transfers, which would have otherwise required more complex hardware support. BugNet focuses on being able to replay the application's execution and the libraries it uses, but not the whole operating system.

Flashback [7] is a lightweight OS extension for software debugging that provides replay capabilities for an application. The main idea is to use shadow processes to capture the in-memory state of a process at a specific execution point and log the process' interactions with the system. In Flashback, the system calls invoked by a process during its execution is intercepted. Flashback replaces the default handler for each system call.

ReVirt [3] is a logging and replay system for analyzing intrusions that runs integrated with a VM and performs the logging in the host OS. It logs all non-deterministic events that can affect the execution of the virtual machine process, asynchronous virtual interrupts and all input from external entities. ReVirt uses UMLinux [9] which the guest OS runs on as a process. ReVirt is implemented as a set of modifications in the host kernel. The logging can be started only when the guest OS starts. XenLR needs not to modify the host kernel and can be started at any time.

ExecRecorder [10] can replay the execution of an entire system (not only a process or a distributed application in isolation) by checkpointing the complete system state (virtual memory and CPU registers, virtual hard disk and memory of all virtual external devices) and logging all architectural nondeterministic events. Its log file is structured as following: 1 byte of event type; 4 bytes of tick difference indicates the number of instructions executed between the two non-deterministic events; 11 bytes of specific input events; 1 byte of IRQ number. However, ExecRecorder' implementation is based on Bochs [11], which is a hardware simulator and its performance is so bad that makes ExecRecorder hard to put into practice.

ReTrace [12] is a trace collection tool using deterministic replay technology based on VMware hypervisor [13, 14]. ReTrace operates in two stages: capturing and expansion. ReTrace capturing accumulates the minimal amount of information necessary to later recreate a more detailed execution trace. It captures (records) only non-deterministic events resulting in low time and space overheads on supported platforms. ReTrace expansion uses the information collected by the capturing stage to generate a complete and accurate execution trace without any data loss or distortion. The trace file contains the following fields: processor CPL; instruction pointer (IP); exceptions, faults and interrupts; registers: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, EFLAGS; segments: ES, CS, SS, DS, FS, GS; control registers: CR0, CR2, CR3, CR4. On average, ReTrace generates 4.8 bytes log file per thousand instructions. It seems a litter too large comparing with our implementation. With the different design strategy on Xen, the content of log in XenLR is very different form ReTrace. We do not log the content of CPU registers when an interrupt happens. What we record is the interrupts and the pointer at which it happens in the instruction stream. The size of our log file is far less than ReTrace.

III. System Design And Implementation

XenLR is a log system for full system replay based on Xen. As a preliminary implementation, XenLR only logs the non-deterministic events on mini OS, which is a lightweight guest OS ported on Xen. The log files are organized with different structures for different events. This scheme makes our design much more scalable than previous approaches. By adding corresponding interception and recording components, the system can be extended to log different type of non-deterministic events.

A. Xen and XenLR structure overview

The implementation of our prototype is based on Xen 3.10, which is an open source virtualization software using para-virtualization technology [16].

A running instance of guest virtual machine on Xen is named as a user domain. The most important domain is the privileged domain, also known as domain 0. Domain 0 is a modified Linux OS and automatically started when Xen boots. Only domain 0 can access the control interface of the VMM, through which other VMs can be created, destroyed, and managed. It runs daemon software *Xend* that manages other domains, which can access the physical resources provided by domain 0's control and management interface in Xen. This structure allows Xen hypervisor to remain a thin code layer between the guests and the hardware, rather than becoming a large and complex piece of machinery like full kernel.

Xen guests use hypercalls to perform privileged operations. These operations include memory mapping, setting trap tables and interrupt gates, accessing debugging hardware, switching stack pointers and segment registers, and so on.

Xen uses the shared info page to deliver certain kinds of information between the guest OS and Xen. Some information, such as the speed and load of the CPU, the memory access models, the hardware functionality, and the system time, requires privileges to access directly from the hardware, and they may be read frequently by the guest OS. There is also a shared info structure that contains information related to each virtual CPU.

Virtual interrupts in Xen are called *events*. Each domain can apply for 1024 event channels at most. Each event channel is assigned to a specific virtual CPU. Events, either from hardware-based virtual devices (such as virtual time interrupt) or from virtual devices in other domains, are sent through the hypervisor. An event channel is marked as pending in a bit array in the shared info page. There is also a bit array for masking event channels, and a per-virtual-CPU event pending and event mask flag. These are used to

implement once-only virtual interrupt delivery, as well as interrupt disabling.

Now we introduce the architecture of XenLR shown in Figure 1. We choose mini OS as the guest domain, which is a lightweight kernel that can be run by the Xen hypervisor. It is in the Xen source file. We develop interception and transport module for each type of non-deterministic events (external input and time update interrupts in mini OS). In our prototype, the external input is mainly got via the peripheral device keyboard. Its actions are captured and delivered to the Xen buffer ring. A daemon reads the data in Linux kernel space and finally saves them in the file system. With the supports of performance counter of x86 processors, each non-deterministic event can be marked by the branch instruction count, which is saved combining with each item of log data.

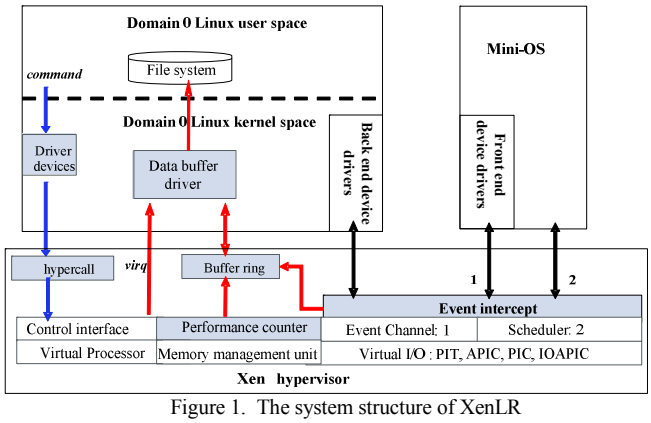

Figure 1. The system structure of XenLR

B. Non-deterministic events

A non-deterministic event is one that causes a state transition but is not fully determined by the previous state, i.e., the event could not be predicted with certainty only from the knowledge of the previous state(s). In a computer system with uniprocessor, events are time and external input. Time refers to the exact point in the execution stream at which an event takes place. For example, to replay an interrupt, we must log the instruction at which the interrupt occurred. External input refers to data received from a non-logged entity, such as a human user or another computer. External input enters the processor via a peripheral device, such as a keyboard, mouse, or network card.

The hardware interface of Xen has few surprises, in regards to non-determinism. The results of hypercalls are deterministic; these results do not need to be logged and replayed. The timing of virtual interrupt deliveries must be logged as an asynchronous event. Updates to the shared info page are visible to the guest as well. These must be logged and replayed as asynchronous events.

Because the mini OS does not own a file system and does not support the peripherals such as mouse and network card, the non-deterministic events are just keyboard input and time update.

1) Keyboard input: The mini OS captures the keyboard input through a structure named *xencons_interface*. This structure is shared among domains and its address is initialized when mini OS boots. The key values should be delivered to the privileged domain and are saved in the file system. *Xenconsoled* (a device driver in domain 0 kernel space) and *Xenconsole* (a process in domain 0 user space) are collaborated for transporting the key value between domain 0 and mini OS. Domain 0 deals with the key value delivering to mini OS by the following steps:

(1) When mini OS is created, *Xenconsoled* apperceives its existence, opens */dev/ptmx* to get a new *pseudo-tty*, sets the master device as O_NONBLOCK status, writes its name of *tty slave* to *Xenstore* and put the new *pty master* device in its *select* loop. At this point, *Xenconsoled* keeps characteristics coming from the mini OS in its own buffer. As the *pty slave* is closed, and *select* does not deem that the *pty master is* ready for writing (or reading).

(2) *Xenconsole* starts, and opens the *pty slave* device.

(3) As the data already reach *Xenconsoled*'s private buffer and *select* returns that the *pty master* has gotten ready for writing, *Xenconsoled* begins to write the data to the *pty master* device immediately. At this time the *pty master* goes back to ECHO state. The *pty master* has gotten ready for reading at the next *select* loop. *Xenconsoled* will read back the data which it has just written, and send them back to the kernel of mini OS.

Data transportation between domain 0 and mini OS is achieved by putting the data to a shared structure *xencons_interface* and delivering a virtual interrupt through event channel. Thus, we can get the key value from *xencons_interface* in domain 0's user space and log it before the data is sent to mini OS.

2) Time update: There are two kinds of data structures related to the time [17]. One is wall clock time and another is time values of VCPU. Wall clock time is specified as an offset to be added to the current real time and it can be modified by control software in the user space. Wall clock time can be read from the structure *shared_info*. Guest OS can get the wall clock time value by system call *gettimeofday*(). Current system time and CPU frequency can be gotten from the structure *vcpu_time_info*, where the time values of VCPU store. Once *vpuc_time_info* is updated, Xen sends virtual time interrupt to notify guest OS for VCPU time updating.

For mini OS, we only need to consider the time values for VCPU. The structure *vcpu_time_info* has these members:

- *version* (32 bits): It is used to ensure the guest gets consistent time updates.
- *tsc_timestamp* (64 bits): Its value means *time stamp counter* (TSC) at last update of time interval.
- *system_time* (64 bits): It expresses time (in nanoseconds) passed since the machine boots.
- *tsc_to_system_mul*: It is the multiplier factor (nanoseconds) which expresses the corresponding relation between TSC and the system time.
- *tsc_shift*: The variant saves shifting of cycle counter in nanoseconds which is used to calculate current system time.

We can calculate current system time with the following formula: *system_time* + ((*tsc* - *tsc_timestamp*) << *tsc_shift*) *

tsc_to_system_mul. The CPU frequency can be calculated from the formula: ((10^9 << 32) / *tsc_to_system_mul*) >> *tsc_shift*.

Unlike the keystroke values, these members of *vcpu_time_info* are updated by Xen hypervisor periodically. We need to *malloc* a buffer ring to save the time values and transfer the data to domain 0's file system.

C. Performance counter

In order to replay an event at the same time in an instruction stream, we need a way to identify individual instructions in the instruction stream. In practice, this might be an actual count of instructions. In order to label un-instrumented code, we need the assistance of the hardware. Most modern x86 processors define a number of performance counters, which count various events on the system. We use these registers in our abstract instruction count.

We use the branch counter *retired_branch_type* of Intel Pentium processors [17] to get branch counts. The properly configured performance counters can be activated at any time when the processor is executing in rings 1-3. We only need to record the counter value on the time when the processor switches to mini OS. A global variable in hypervisor is used to save and restore the performance counters value when the context switches. When the VCPU of mini OS is scheduled in, the value saved in the variable is written back to branch counter, and the counter starts at the value. When the VCPU is scheduled out, we read current counter value and save it in the variable. The counter value would be uploaded to domain 0 and saved in the log file. Because the hypervisor is running on the ring 0, it would not add additional overhead to the branch count.

D. Transmission channels

We need channels to transfer control commands and other data. Different external inputs are handled with different mechanism. All the external inputs should be saved in domain 0's file system. In XenLR, we design three kinds of channels: control command transmission channel is used to transfer control data; counter transmission channel is used to transfer the performance counter's value; time value transmission channel is used to transfer the updating of VCPU's time. These channels are mainly composed of two parts: a device driver in domain 0 and a hypercall in Xen hypervisor. Their relationship is shown in Figure 2.

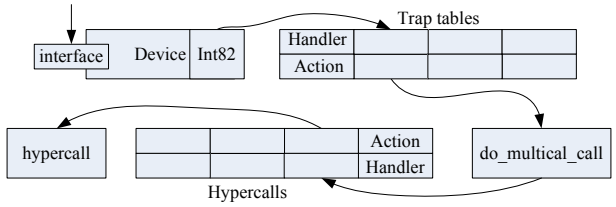

Figure 2. The relationship between device and hypercall

1) Control command transmission channel: This channel is used to deliver control commands, which include the information of the domain ID to be logged and the time to start or stop logging. The channel is implemented by adding a control device in domain 0 and a hypercall in Xen hypervisor. To enter domain 0 kernel space, the *ioctl* function in control device need to be invoked. In kernel space, we use a hypercall to deliver control commands. Different commands can be transferred by setting different flags in the hypercall.

2) Counter transmission channel: This channel is used to deliver the counter value from Xen hypervisor to domain 0's user space. When a branch instruction occurs, the counter value should be fetched and transferred to user space. Figure 3 shows the process of transmission. The function *ioctl* is the interface between the user space and kernel driver. The hypercall *Hypervisor_perfc_op* is used to access the Xen hypervisor. The counter value returns to the user space along the same invoking path.

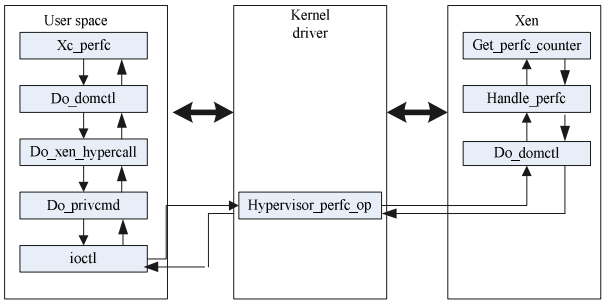

Figure 3. The transmission channel of performance counters

3) Time value transmission channel: The design and implementation of this channel should be given careful consideration. The VCPU's time is updated periodically within short interval, so large size of data will be generated quickly. The time value is delivered from the hypervisor, passing the domain 0's kernel space and finally reaches the user space. To improve the efficiency of data reading/writing and also to avoid data loss while it is being transmitted, we design three data buffer areas in each space as shown in Figure 4. The data is finally saved in the domain 0's file system.

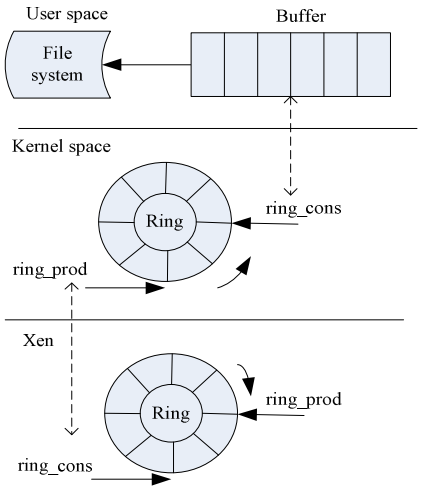

Figure 4. The structure of time value transmission channel

There are two buffer rings in the transmission channel. The buffer rings are managed in producer consumer model. The hypervisor produces the updating time value and the kernel space device consumes it; while for the process in user space, the driver of kernel space buffer ring is the producer. The data finally are transferred to the file system.

We use select model [18] in the user space to consume the data in buffer device. A poll function is used to transfer time value form Xen hypervisor to the buffer device. We set a threshold for the hypervisor buffer to trigger data delivering operation. Once the residual space reaches the fixed value, the hypervisor would send a virtual interrupt to device driver, and then a new buffer ring is allocated to pull the data stored in the hypervisor buffer. We save the data to the hard disk using cached I/O file operation finally.

E. Structure of log file

We log different kinds of external inputs with different structures in different log files. For keyboard inputs, we use 1 byte to save the key value and 8 bytes to save the branch marks (counter value). Additional other 3 bytes are used for data alignment. For time updating, we have observed that the *version* difference between each time of *vcpu_time_info* updating does not exceed 64, so we log just 1 byte as the *version* information. This approach decreases the bytes required to record the *vcpu_time_info*'s *version* item from 4 bytes to 1 byte. Thus, for the structure of *vcpu_time_info*, we only need 22 bytes to save its members and 2 bytes for data alignment. Finally, to log each time updating, we use 24 bytes for the structure of *vcpu_time_info* and 8 bytes for the branch mark. We do not log the type of virtual interrupts in the log file because different log files mean the different virtual interrupts. Each item of record in the log file means a virtual interrupt happening at the same point. The approach we organize the log file reduces the storage space consumption and makes the records more compact and easy to handle.

IV. PERFORMANCE EVALUATIONS

In our experiments, we first analyze the size of the log files generated by XenLR and then measure the log growth rate. We have also analyzed the performance overhead in domain 0 incurred by the logging component.

The experiments are executed on a computer with a 2GHz Intel Celeron CPU and 512MB RAM. The version of Xen is 3.1.0 and the kernel of domain 0 is Linux-2.6.18. The mini OS is configured with one VCPU and 32MB memory.

A. The size of log files

We need 12 bytes to log a keystroke event and 32 bytes to log the time updating of VCPU. The data is written to log file when the data size accumulates to 4096 bytes in the buffer.

In our experiments we find the VCPU's time is updated once per two seconds. It means the file size for time logging increases 4K bytes every 256s. That is to say the log growth rate for time updating is 1.2MB/day, so XenLR brings small space consumption on mini OS.

B. The time overhead of logging

We are concerned about the overhead that arise form logging the keystroke event on mini OS. The overhead mainly comes from the process of getting the value of performance counter. We measure the execution time of the function as follows: we first test the execution procedure for 100 times and compute the average time consumption, then test another 100 times and compute the average value of the total 200 times execution, continue this test process with the same way until the average value gets a stable value. Figure 5 shows the result of test process. We can find that the additional process cost 12us while logging the keystroke event on mini OS. Using the same approach, we measure the time cost to handle a keystroke event without logging. The average time is 0.02s. The overhead caused by logging the keystroke value can be calculated like this: (12/(0.02*1000000))*100%=0.06%, which is really negligible for the end user.

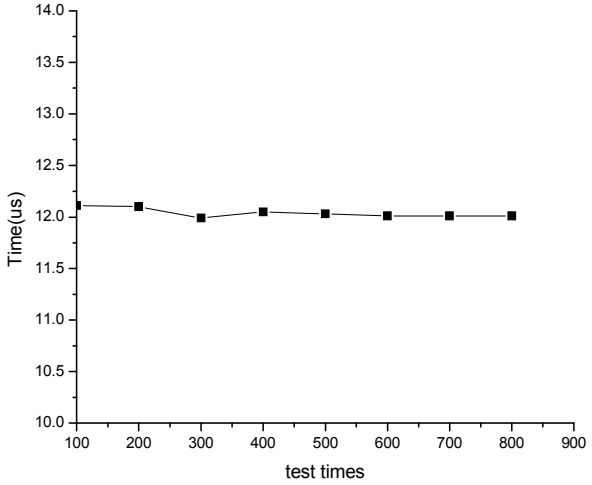

Figure 5. Time overhead caused by logging the key value

Next we quantify the overhead of time updating on mini OS. We first test the time cost to transfer different size of data from hypervisor to user space for 100 times. The average result shows that only 17us is cost for each time updating (once in 2second) and only 209.4us to dump 4096 bytes data from hypervisor buffer to use file system, while such size of data is generated in 256s, so the data transferring is amortized over a long period of time. The overhead can be calculated like this: (17*(256/2)+209.4)/(256*1000000)= 0.0009%.

The experiments illustrate that XenLR causes a little space and time overhead to log the keystroke events and time updating on mini OS. To make a comparison with other log systems, Table I shows the space and time overhead caused by the aforementioned system (the data comes form the cited papers). Our prototype shows great improvement comparing with previous systems because the VM we logged is a lightweight OS and do not support complicated applications, the information need to record is comparative less.

TABLE I. THE SPACE AND PERFORMANCE OVERHEAD

Log system	VMM	Logged VM	Over-head	Log growth rate
ReVirt	UMLinux	Linux	0~8%	0.2GB/day
ExecRec order	Bochs	Linux or Windows	4%	5.4GB/hour
Retrace	VMware	Linux or Windows	5.09%	4.8 Byte/kilo -instructions
XenLR	Xen	Mini-OS	<0.1%	1.4MB/day

V. CONCLUSIONS

This paper describes how XenLR logs non-deterministic events on mini OS (a lightweight OS) for replay. XenLR is constructed on Xen which uses para-vitualization technology. There are only two kinds of external inputs for mini OS, the keyboard input and the time update. The key value can be captured in domain 0's user space. The time value is updated by the hypervisor. We configure the performance counter and record the branch count to mark those non-deterministic events. We design three kinds of channels to transfer control commands and data. The log data are saved in the privileged domain's file system through transmission channels. In order to reduce the log file size, we identify non-deterministic events and encode them compactly with different structure.

XenLR has some prominent traits comparing with other VM-based log system. The features are mainly in the following aspects:

- Mark of non-deterministic event: XenLR records instruction branch count but not the tick time in the guest OS when the non-deterministic event occurs.
- Integrity of guest OS: XenLR does not modify the guest OS.
- Inter-domain storage: XenLR saves the log files of mini OS on the domain 0's file system.
- High performance: our experiments illustrate that XenLR results in low time and space overhead.

To validate the correctness and rationality of our logging-and-replay model, mini OS is chosen as the logged VM for simplify. In the future, we attempt to log and replay current commercial OS on Xen. There are some complex issues that should be considered further. We would like to employ this paper to solicit feedback for our further design and implementation.

REFERENCES

[1] S. Joshi and A. Orso, "SCARPE: A Technique and Tool for Selective Record and Replay of Program Executions", *Proc. IEEE ICSM'07*, Oct. 2007.

[2] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating Bugs as Allergies - A Safe Method to Survive Software Failures", *Proc. SOSP'05*, Oct. 2005.

[3] 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", *Proc. OSDI'02,* Dec. 2002.

[4] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging", *Proc. ISCA'05*, June 2005.

[5] R. P. Goldberg, "Survey of Virtual Machine Research", *IEEE Computer*, pp.34-45, June 1974.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A. Warfield, "Xen and the art of virtualization", *Proc. SOSP'03*, Oct. 2003.

[7] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging", *Proc. USENIX'04*, June 2004.

[8] K. Buchacker and V. Sieh, "Framework for Testing the Fault-tolerance of Systems including OS and Network Aspects", *Proc.HASE'01*, pp.95-105, Oct. 2001.

[9] J. Dike, "A User-Mode Port of the Linux Kernel", *Proceedings of the 2000 Linux Showcase and Conference*, pp.63-72. Oct. 2000.

[10] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong, "ExecRecorder: VM-based full-system replay for attack analysis and system recovery", *Proc. ASID'06*, pp.66-71, Oct. 2006.

[11] Bochs: the Open Source IA-32 Emulation Project. http://bochs.sourceforge.net.

[12] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman, "ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay", *Proc. MoBS'07*, June 2007.

[13] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization", *Proc. ASPLOS'06*, pp.2-13, Aug. 2006.

[14] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", *Proc. USENIX'02*, pp.1–14, June 2001.

[15] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing Performance Overheads in the Xen Virtual Machine Environment", *Proc. VEE'05*, pp.13-23, June 2005.

[16] Xen: the Open Source virtual machine monitor for X86 Project interface manual. http;//www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html

[17] Intel Corporation, *The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, Tech. Rep. 2001.

[18] W. R. Stevens and S. A. Rago, *Advanced Programming in the UNIX Environment*, 2nd ed., Addison Wesley Professional, June 17, 2005.