Simultaneous Logging and Replay for Recording Evidences of System Failures

Shuichi Oikawa, Jin Kawasaki

Simultaneous Logging and Replay for Recording Evidences of System Failures

Shuichi Oikawa Jin Kawasaki

Department of Computer Science, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan

Abstract. As embedded systems take more important roles at many places, it is more important for them to be able to show the evidences of system failures. Providing such evidences makes it easier to investigate the root causes of the failures and to prove the responsible parties. This paper proposes simultaneous logging and replaying of a system that enables recording evidences of system failures. The proposed system employs two virtual machines, one for the primary execution and the other for the backup execution. The backup virtual machine maintains the past state of the primary virtual machine along with the log to make the backup the same state as the primary. When a system failure occurs on the primary virtual machine, the VMM saves the backup state and the log. The saved backup state and the log can be used as an evidence. By replaying the backup virtual machine from the saved state following the saved log, the execution path to the failure can be completely analyzed. We developed such a logging and replaying feature in a VMM. It can log and replay the execution of the Linux operating system. The experiment results show the overhead of the primary execution is only fractional.

1 Introduction

Commodity embedded systems, such as mobile phones, car navigation systems, HD TV systems, and so on, became so complicated that it is almost impossible to make them free from defects. Since there are a large number of usage patters, manufactures are not able to perform tests that cover all possible usage patters in advance. Since users tend to operate those commodity embedded systems for a long time, users cannot remember exactly what they did when system failures occurred. In such situations, it is not clear which side is responsible for the system failures, and both sides can blame each other for the causes of the failures.

As those embedded systems take more important roles at many places for many people, there are more chances that their failures may directly connect to financial loss or physical damage. System failures do not mean only software failures, but they mean the whole other failures of the parts included in systems. When a software malfunction happens, it may affect mechanical parts and can damage something or somebody. System failures may not be considered to be failures at first. Because of the perceptions by customers and/or societies, they can suddenly become failures. Therefore, it is important to record and provide the evidences of system failures. The evidences make it easier to investigate the root causes of the failures and to prove the responsible parties.

We propose a system that enables the complete tracing of system failures and such execution traces are used as their evidences. We employ two virtual machines, one for the primary execution and the other for the backup execution. Those two virtual machines run on a virtual machine monitor (VMM) [5, 9] of a single system. The VMM records the primary execution, and generates an execution log in a buffer. Then, on the backup virtual machine, the VMM replay the primary execution following the log in the buffer. The backup execution is exactly the same as the primary execution as far as the executed instruction stream and the produced values are concerned. Maintaining the log at a certain size creates a time lag for the backup execution. In other words, the execution of the backup virtual machine follows the execution of the primary virtual machine, but the backup execution never catches up the primary execution by buffering the execution log. Thus, the backup virtual machine maintains the past state of the primary virtual machine along with the log to make the backup the same state as the primary. When a system failure occurs on the primary virtual machine, the VMM saves the backup state and the log. By replaying the backup virtual machine from the saved state following the saved log, the execution path to the failure can be completely analyzed and provides the concrete evidence of the system failure.

We developed such a logging and replaying feature in a VMM. The VMM is developed from scratch to run on an SMP PC compatible system. It can log and replay the execution of the Linux operating system. The experiments show that the overhead of the primary execution is only fractional, and the overhead of the replaying execution to boot up the backup is less than 2%. This paper presents the detailed design and implementation of the logging and replaying mechanisms.

The rest of this paper is organized as follows. Section 2 shows the overview of the propsed system architecture. Section 3 describes the rationale of the logging and replaying of the operating system execution. Section 4 describes the design and implementation of the logging and replaying mechanisms. Section 5 describes the current status and the experiment results, and Section 6 discusses the current issues and possible improvements. Section 7 describes the related work. Finally, Section 8 concludes the paper.

2 System Overview

This section describes the overview of the propsed system architecture. Figure 1 shows the overview of the architecture. The system runs on an SMP system with 2 processors. In the figure, there are 2 processor cores, Core 0 and 1, which share physical memory. The primary VMM runs on Core 0, and creates the primary virtual machine. The primary virtual machine executes the primary guest OS. Core 1 is used for the backup. The primary and backup VMMs are the same except that they run on different cores. The memory is divided into three parts, one for the primary, another for the backup, and the last one for the shared memory between the primary and the backup.

Users use the the primary guest OS. It interacts with devices through the primary VMM; thus, it is executed just as an ordinary guest OS that runs on a VMM. Only difference is that the primary VMM records the events described in the previous section, so that its execution can be replayed on the backup.

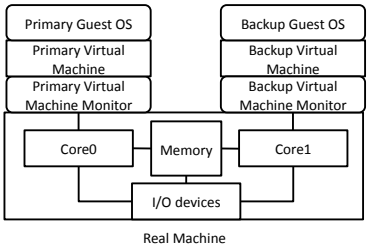

Fig. 1. Overall Architecture of the Proposed System

The execution log is transferred from the primary to the backup via shared memory. For every event that has to be logged, the primary VMM writes its record on the shared memory, and the backup VMM reads the record. The mapping is created at the boot time of the VMM. The shared memory is used only by the VMM since the log data is written and read by the VMM not by the Linux; thus, the existence of the shared memory region is not notified to the Linux. The size of the shared memory is currently set to 4 MB. The size should be adjusted to an appropriate value considering system usage.

The backup VMM reads the execution log from the shared memory, and provides the backup guest OS with the same events for the execution for the replaying. The backup guest OS does not take any inputs from devices but takes the replayed data from the backup VMM. The outputs produced by the backup guest OS are processed by the backup VMM. They can be output to a different unit of the same device or to an emulated device.

3 Rationale

This section describes the rationale to realize the logging and replaying of the operating system execution, and clarifies what needs to be logged. The basic idea of the logging and replaying of instruction execution is about the treatment of the factors outside of programs. Those factors include inputs to programs and interrupts, and both of them are events external to programs. This paper describes the rationale specific to the IA-32 architecture [6].

If a function contains the necessary data for its computation and does not interact with the outside of it, it always returns the same result regardless of the timing and the state of its execution. If a function accesses an I/O port and reads a value from it, the result can be different depending upon the value read from the I/O port. Therefore, in order to make the instruction execution streams and the produced values the same on the primary and the backup, the values copied from I/O ports must be recorded on the primary, and the same value must be provided from the corresponding port in the same order on the backup.

External interrupts change instruction execution streams, and external interrupts are caused by the factors that are outside of the executed program. Furthermore, external

interrupts are vectored requests. There are interrupt request numbers (IRQ#) that are associated with devices. Different IRQ# can be used in order to ease the differentiation of interrupt sources. Therefore, in order to make the instruction execution streams the same on the primary and the backup, the specific points where external interrupts are taken in the instruction execution stream and their IRQ# must be recorded on the primary, and the interrupts with the same IRQ# must be caused and taken at the same points on the backup.

The problem here is a way to specify a point where an external interrupt is taken. If there is no branch instruction in a program, an instruction address can be used to specify the point. There are, however, branches in most programs; thus, the number of branches needs to be counted. The IA-32 architecture includes instructions that loops within a single instruction, and does not change the number of branches. REP instruction repeats some types of a load or store instruction for the number of times specified in %ecx register. The repeating operation can be suspended by an interrupt, and be resumed again after the interrupt processing is finished. Therefore, the value of %ecx register also needs to be recorded. Therefore, by recording the instruction address, the number of branches since the last interrupt, and the value of %ecx register, it is possible to specify the point where the interrupt is replayed.

4 Logging and Replay

This section describes the design and implementation of the logging and replaying mechanism on the proposed system architecture. Our VMM is implemented on the IA-32 processor with Intel VT-x [8] feature. We take advantage of the capabilities available only to Intel VT-x for the efficient handling of the logging and replaying.

4.1 Logging

We first describe the retrieval of the information needed for the logging, and then its recording. There are the two types of the events that need to be logged, IN instructions and external interrupts. Event types can be captured from the VM exit reason. Intel VT-x implements the hardware mechanism that notifies the VMM about guest OS events. A VM exit is a transfer of control from the guest OS to the VMM, and it comes with the reason. By examining the VM exit reason, the VMM can distinguish which type of an event happened.

For an IN instruction event, the VMM needs to capture the result of the instruction execution. In order to do so, the VMM obtains the control when the guest OS executes an IN instruction, and executes the IN instruction with the same operand on behalf of the guest OS. Intel VT-x has the setting that causes a VM exit when an IN instruction is executed in the guest OS. The primary processor-based VM-execution controls define various reasons that cause a VM exit. Bit 24 of the controls determines whether the executions of IN and OUT instructions cause VM exits. By setting that bit, the VMM can obtain the control when the guest OS executes an IN instruction. The I/O port number that the IN instruction is about to access is provided in the VM exit qualification. The result of an IN instruction execution is returned in %eax register. The VMM copies the

```
typedef struct {
    unsigned long index;
    unsigned long address;
    unsigned long ecx;
    unsigned long reserved;
    unsigned long result_h;
    unsigned long result_l;
    unsigned long counter_h;
    unsigned long counter_l;
} RingBuffer;
```

Fig. 2. Definition of the Ring Buffer Entry

value to the guest %eax register, and also records the value in the log along with the event type.

For an external interrupt event, the VMM needs to capture the IRQ#, the instruction address where the interrupt happened, the number of branches since the last event, and the value of %ecx register. The IRQ# is available from the VM-Exit interruption-information field. The instruction address of the guest OS is defined in the guest register state. The number of branches is counted by the Performance Monitoring Counter (PMC). The IA-32 architecture has the PMC facilities in order to count a variety of event types for performance measurement. Those event types include branches. The value of %ecx register is saved by the VMM at a VM-exit.

The retrieved event record is saved in an entry of the ring buffer constructed on the shared memory. Figure 2 shows the definition of the ring buffer entry structure in the C programming language. The fixed size structure is used for the both types of the events. Structure members `index`, `addess`, and `ecx` contain the type of an event, the instruction address where the event happened, and the value of %ecx register, respectively. The result value associated with an event can hold up to 8 bytes (64 bits) in `result_h` and `result_l`. `counter_h` and `counter_l` contain the number of branches since the last event. A PMC register can be up to 8 bytes (64 bits) long, and it is 40 bits long for the processor we currently use for the implementation; thus, 2 unsigned long members are used to contain the number of branches. The size of the structure is 32 byte, which is well aligned with the IA-32 processor's cache line size of 64 byte.

The ring buffer on the shared memory is managed by 3 variables, `ringbuf`, `write_p`, and `read_p`. Variable `ringbuf` points to the virtual address of the shared memory. The 4 MB region for the shared memory is statically allocated by the configuration; thus, the variable is initialized to the fixed value at the boot time. Variable `write_p` points to the index variable that indicates where a new event is recorded. Variable `read_p` points to the index variable that indicates where an unread event is stored. Because those indices are shared by the logging side and the replaying side and they need to be shared, `write_p` and `read_p` and pointer vaiables.

4.2 Replaying

We describe the execution replaying following the recorded log. In order to replay an event, the VMM first reads the next event record from the shared memory. The read event record contains the event type information. The event type determines the next action for the VMM to do. As described above, there are two types of the events, an IN instruction event and an external interrupt event. We first describe the replaying of an IN instruction event, and then an external interrupt event.

In order to replay an IN instruction event, the VMM needs to obtain the control when the backup guest OS executes an IN instruction. The VMM on the backup sets up Intel VT-x in the same way as the primary to cause a VM exit when an IN instruction is executed in the guest OS. The record of an IN instruction event contains the data returned by the IN instruction on the primary. The VMM does not execute an IN instruction on behalf of the guest OS. The VMM simply returns the recorded data, instead, as the data read by the emulated IN instruction. We can make the order of the IN instructions execution the same on the backup as the primary as long as the results returned by the IN instructions are maintained the same.

The replaying of an external interrupt event is much harder. It is because an external interrupt can happen everywhere an interrupt is not disabled. In other words, there is no specific instruction that specifies where an external interrupt happens. Therefore, the VMM relies upon the information recorded on the primary and needs to point out the place and timing where an external interrupt event is injected into the guest OS.

The VMM determines the point to inject an external interrupt event by the following steps:

1. Execute the guest OS until the instruction address where the interrupt is supposed to be injected.
2. Compare the numbers of branches of the guest OS and the event record.
   (a) Proceed to the next step if they matches. Go back to Step 1 above if they don't.
3. Compare the values of %ecx register of the guest OS and the event record.
   (a) Proceed to the next step if they matches. Execute a single instruction and go back to Step 3 if they don't.
4. Inject the interrupt into the guest OS.

At Step 1, the VMM needs to obtain the control when the guest OS is about to execute the specific instruction address. The event record contains the instruction address. We use a debug register to cause a VM exit and for the VMM to obtain the control. The IA-32 architecture has hardware debug facilities that can cause a debug exception at the execution of the specific instruction address. The address is specified in a debug register. For Intel VT-x, the exception bitmap defines which exception causes a VM exit. By setting the exception bitmap appropriately, a debug exception causes a VM exit; thus, the VMM obtains the control.

At Step 3, we use the single-step execution mode, and do not use a debug register. It is a special mode for debugging. It executes only a single instruction and causes a debug exception; thus, by using this mode, the VMM obtains the control after the single-step execution.

Table 1. Source Line of Code to Support the Logging and Replaying Mechanism in VMM

Part	SLOC	Ratio [%]
VMM without logging and replay	9,099	65.4
SMP Support	3,123	22.4
Logging and replaying mechanism	1,694	12.2
Total	13,916	100

5 Current Status and Experiment Results

This section describes the current status and the experiment results. We implemented the proposed logging and replaying mechanism in our VMM. The Linux operating system can can boot and also be replayed on the VMM. All experiments described below were performed on the Dell Precision 490 system, which is equipped with Xeon 5130 2.00GHz CPU and 1 GB memory. The version of the Linux kernel is 2.6.23.

First, we show the implementation cost of the logging and replaying mechanism. Second, we show the overheads to boot the Linux. Finally, we show the size of the log and a breakdown of the logged events, and discuss a way to reduce the log size.

5.1 Implementation Cost

The logging and replaying mechanism described in this paper was implemented in our own VMM. Table 1 shows the source lines of code (SLOC) including empty lines and comments. In order to support an SMP environment and to run two Linux instances, one for the primary and another for the backup, 3,123 SLOC were added. They include the code for the initialization of the secondary core, SMP related devices, such as Local APIC and I/O APIC, the modifications to shadow paging, and the shared memory. In order to support the logging and replaying mechanism, 1694 SLOC were added. They include the code for the logging, the ring buffer, and the replaying. In total, SLOC increased 52.9 % in order to realize the proposed system architecture.

5.2 Boot Time Overheads

In order to evaluate the overheads of the logging and replaying, we first measured the boot time of the Linux. The time measured is from `startup_32`, which is the beginning of the the Linux kernel's boot sequence, and to the point, where the init script invokes `sh`, which is the user level shell program. The measurements were performed several times, and their averages are shown as the results.

Table 2 shows the results of the measurements. The table shows the actual boot times in seconds and also the ratio relative to the boot time of the original Linux. The logging imposes almost no overhead that is only 0.1 %[1], and the replaying imposes 1.5 % of the overhead.

[1] It is, actually, slightly less than 0.1 % but is rounded to it.

Table 2. Time to Boot Linux with and without Logging and Replaying

	Time to boot Linux [sec]	Ratio [%]
No logging	2.284	1
Primary with logging	2.286	100.1
Backup with replaying	2.319	101.5

Table 3. Benchmark Results with and without Logging and Replaying

	fork [mili sec]	fork+exec [mili sec]
No logging	1.336	2.087
Primary with logging	1.357	2.153
Backup with replaying	1.393	2.462

The logging and replaying impose 2 mili seconds and 35 mili seconds of the overhead, respectively. The logging requires VM exits everywhere events need to be recorded while the replaying also requires VM exits everywhere events may need to be replayed. The replaying imposes the more overhead than the logging because the replaying requires the VMM to point out the exact addresses and timings where events need to be replayed. As described in Section 3, loops and repeating instructions make that pointing more difficult because the same instruction addresses are executed multiple times; thus, the difference of the overheads of the logging and the replaying is caused by the cost to point out the exact addresses and timings where events need to be replayed.

5.3 Benchmark Overheads

We further performed the measurements using the two benchmark programs. One benchmark program measures the cost of the fork system call, and the other one measures the combined cost of the fork and exec system calls. They basically perform the same as those included in the LMbench benchmark suite [7], while they are modified to use the RDTSC instruction in order to measure times. The measurements were performed 1000 times, and their averages are shown as the results.

Table 3 show the results of the measurements. The results show that the overheads of the logging are small for both the fork and fork+excec system calls. The overhead of the replaying for the fork+exec system call is, however, higher. In order to further examine the source of the overhead, we calculate the costs of the exec system call from the measured values. The calculated costs of the exec system call from Table 3 is 0.751, 0.796, and 1.069 mili seconds for no logging, primary with logging, and backup with replaying, respectively. The ratios are 106%, 142%, and 134% for primary per no logging, backup per no logging, and backup per primary, respectively. Since the use of only the exec system call is not typical, those calculated overheads will directly impact overall system performance. They rather give the insights for the analysis of the overheads.

Table 4. Breakdown of Logged Events

Event	Count	Ratio [%]
Timer interrupts	207.5	8.00
Serial line interrupts	2	0.08
IN instruction	2385	91.93
Total	2594.5	100.01

The fork and exec system calls behave quite a lot differently as they provide the totally different functions. The fork system call creates a copy of the calling process. It needs to allocate an in-kernel data structure that represents a new process, while it can reuse the most of the other memory images for a new process by the copy-on-write technique. The exec system call, on the other hand, frees the most of memory images except for the in-kernel data structure of the calling process. It then allocates necessary memory regions in order to load an newly executing program. Such loading causes the numerous times of data copying, and external interrupt events during the data copying instructions are expensive to replay. It is very likely to be a reason why the overhead to replay the exec system call is high.

5.4 Size of Log and Breakdown of Logged Events

Table 4 shows a breakdown of the logged events while the Linux was booted.[2] From the ratios of a breakdown, we can see that the most of the logged events are IN instruction. The device driver of the serial line often uses IN instruction in order to examine if the device is ready to transmit data so that data can be written into it. The drivers of the other devices, such as PIC (Programmable Interrupt Controller) and PIT (Programmable Interval Timer), also use IN instruction. Especially, PIC is manipulated every time an interrupt handler is invoked in order to acknowledge an EOI (End of Interrupt handling) and also to mask and umask the corresponding IRQ#.

From the number of the total events, the size of the log after booting the Linux is calculated at 83.02 KB. 83.02 KB of the log is produced for 2.286 seconds of the boot time, meaning 36.32 KB of the log for every second; thus, by using 4MB of the log buffer, the execution of the backup can be deferred for 112.8 seconds if the same production rate of the log is assumed.

6 Discussion for Improvements

This section discusses possible improvements for the described logging and replaying mechanism. There are two obvious issues, one is the log size and the other is the overheads. We discuss them in the rest of this section.

[2] The total of the ratios is 100.01 %, which shows a round error of 0.01 %.

6.1 Reduction of Log Size

We discuss several ways to reduce the log size in order to make the deferred time longer. A straightforward way to do it is by reducing the size of each event. Currently, the fixed size of 32 byte is used to record each event. 32 byte was chosen because it is well aligned with the IA-32 processor's cache line size of 64 byte.

In order to reduce the record size, we can change the data size for the different event types. In other words, by employing the exact size for each event type, the log size can be reduced. The record size of an IN instruction event can be reduced to 2 bytes, which consist of 1 byte for the event type and 1 byte for input data. The record size of an interrupt event can be reduced to 15 bytes, which consist of 1 byte for the event type, 1 byte for IRQ#, 4 byte for the instruction address, 5 byte for the number of branches, and 4 byte for the value of %ecx register. If we can decode the instruction where an interrupt was injected at the time of the logging, an interrupt event can be divided into two types, one with the value of %ecx register and the other without it.

By using the record sizes of 2 byte for an IN instruction event and 15 byte for an interrupt event, the size of the log after booting the Linux can be reduced to 7.91 KB. It is only 9.5 % of the original log size; thus, the significant reduction of the log size is possible. By using these record sizes, 4MB of the log buffer can defer the execution of the backup for 1183.7 seconds (19 minutes and 43.7 seconds).

Another way is that the VMM emulates a serial device in a specific way to reduce the log size. The serial line device driver of the Linux kernel executes a loop to wait until the device becomes ready to transmit data by examining the status of the device issuing IN instruction. Instead, the VMM can always tell the Linux kernel that the device is ready to transmit data, and executes a loop to wait until the device becomes ready to transmit data. In this way, the Linux kernel does not need to issue a number of IN instructions, so that the events that need to be logged can be reduced.

6.2 Reduction of Overheads

We focus on reducing the replaying overhead since the replaying overhead is larger and the logging overhead is relatively small. The benchmark results described in Section 5.3 revealed replaying the exec system call is expensive, and the high overhead of replaying external interrupt events during the data copying instructions is a possible reason. It is so because REP instructions are commonly used for the data copying since it provides the most efficient way to copy data from one place to another. A REP instruction followed by the MOVS instruction copies the data pointed by %esi register to the region specified by %edi register using %ecx register as its counter. It is a single instruction but repeats the execution of the MOVS instruction for the times specified by %ecx. An interrupt is serviced between the executions of the MOVS instruction within the single instruction pointer address. Thus, the only way to pinpoint the timing to deliver an external interrupt within the REP instruction is to execute the instruction in the single step mode until the value of %ecx register becomes equal to the logged value. The execution in the single step mode causes a VM exit after the execution of every MOVS instruction within a REP instruction. The VMM checks the value of %ecx register. If the value is not equal to the logged one, it resumes in the single step mode. If the value becomes equal to

the logged one, the VMM cancels the single step mode and injects the logged external interrupt. Therefore, the overhead of replaying a REP instruction becomes larger as its execution in the single step mode is longer.

From the above reasoning, one straightforward way to reduce the replaying overhead is avoid the use of the single step mode. It can be done by dividing the execution of a REP instruction into the two parts, pre-injection and post-injection. The pre-injection and post-injection parts are the execution before and after the injection of an external interrupt, respectively. By adjusting the value of %ecx register and setting the breakpoint after the corresponding REP instruction, the execution of the pre-injection part finishes without a VM exit at every MOVS instruction within a REP instruction. The breakpoint is taken after the execution of the pre-injection part. Then, the VMM injects an external interrupt. After handling the injected external interrupt, the post-injection part is executed. It works because the value of %ecx register is not used by a MOVS instruction within a REP instruction. This way significantly reduces the number of VM exits; thus, we can expect the reduction of the replaying overhead by a certain amount.

Avoiding the use of the single step mode is also possible by a device on the primary at the time of logging. When an external interrupt is delivered, the VMM first takes it and examines the current instruction. If the current instruction is not REP, the VMM injects the interrupt. If the current instruction is REP, the VMM sets the breakpoint at the next instruction and resumes the execution. After the execution of the REP instruction, the VMM injects the interrupt. This way is much simpler, but the delivery of interrupts can be delayed.

7 Related Work

There are several other studies on the logging and relaying mechanisms in VMMs. We took a similar approach to ReVirt [4], Takeuchi's Lightweight Virtual Machine Monitor [10], and Aftersight [11] in terms of the basic mechanisms. PMC is effectively used to count the number of events in their work and ours. They, however, store the log in storage devices, and perform the replaying later. We propose the system that the both logging and replaying are executed simultaneously but with some time difference on the same machine. The details of the mechanism and implementation of our work are also different from them. ReVirt employs UMLinux as its VMM, which does not use Intel VT-x and emulates devices. Aftersight perform the replaying on QEMU system emulator but not a virtual machine on a VMM. Takeuchi's Lightweight Virtual Machine Monitor uses Intel VT-x but only supports a simple RTOS without a memory protection feature by MMU.

Bressoud's fault tolerant system [1] uses a different mechanism from ours to enable the logging and relaying. It performs the logging and the replaying periodically while our mechanism performs them on demand. The performance of Bressoud's system heavily depends of the timing parameter that defines the period for the logging and the replaying; thus, it does not perform well for interactive uses. It is different from ours also in terms of the system configuration that the primary and the backup are implemented on different machines connected with each other by network.

8 Summary

We proposed a system that enables the complete tracing of system failures and such execution traces are used as their evidences. We employ two virtual machines, one for the primary execution and the other for the backup execution. The backup virtual machine maintains the past state of the primary virtual machine along with the log to make the backup the same state as the primary. When a system failure occurs on the primary virtual machine, the VMM saves the backup state and the log. By replaying the backup virtual machine from the saved state following the saved log, the execution path to the failure can be completely analyzed and can provide a concrete evidence.

We developed such a logging and replaying feature in a VMM. The VMM is developed from scratch to run on an SMP PC compatible system. It can log and replay the execution of the Linux operating system. The experiments show that the overhead of the primary execution is only fractional, and the overhead of the replaying execution to boot up the backup is less than 2%.

References

1. T. Bressoud and F. Schneider. Hypervisor-Based Fault Tolerance. *ACM Transactions on Computer Systems*, Vol.14, No.1, pp.80-107, 1996.
2. 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. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pp. 164–177, October 2003.
3. B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.
4. G. Dunlap, S. King, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pp. 211-224, 2002.
5. R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pp. 34–45, June 1974.
6. Intel Corporation. IA-32 Intel Architecture Software Developer's Manual.
7. L. McVoy and C. Staelin. LMbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference*, pp. 279–294, January 1996.
8. G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Technical Report, Intel Corporation, 2006.
9. M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, pp. 39–47, May 2005.
10. S. Takeuchi and K. Sakamura. Logging and Replay Method for OS Debugger Using Lightweight Virtual Machine Monitor. *IPSJ Journal*, Vol. 50 No. 1, pp. 394–408, January 2009.
11. J. Chow , T. Garfinkel , P. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of the USENIX 2008 Annual Technical Conference*, pp. 1–14, June 2008.