IPSJ Journal Vol. 52 No. 8 2344–2354 (Aug. 2011)

Regular Paper

Simultaneous Virtual-Machine Logging and Replay

Shuichi Oikawa[†1] and Jin Kawasaki[†1,*1]

This paper proposes simultaneous virtual-machine logging and replay. It performs logging and replay simultaneously on the same machine through the use of two virtual machines, one for the primary execution and the other for the backup execution. While the primary execution produces the execution history, the backup execution consumes the history by replaying it. The size of an execution log can be limited to a certain size; thus, huge storage devices becomes unnecessary. We developed such a logging and replaying feature in a VMM. It can log and replay the execution of the Linux operating system. Our experiment results show the overhead of the primary execution is only fractional.

1. Introduction

This paper proposes simultaneous virtual-machine logging and replay. Virtual-machine logging and replay is a technique where a virtual machine monitor (VMM) [4),12)] logs the execution history of a virtual machine (VM) and the identical execution is replayed later. The technique can be used to analyze failures and possible intrusions [3),13)]. One of the shortcomings of virtual-machine logging and replay is the size of the log in which the execution history is saved; thus, it is not very realistic to apply the technique to commodity embedded systems since such huge storage devices are not available to them. These commodity embedded systems, such as mobile phones, car navigation systems, HD TV systems, and so on, become very complicated. Therefore, they can benefit from the logging and replay technique since it is almost impossible to make them free from defects.

The proposed system performs virtual-machine logging and replay simultaneously on the same machine [8),11)]. It employs two virtual machines, one for the primary execution and the other for the backup execution. These two virtual machines run on a VMM of a single system. While the primary execution produces its execution history, the backup execution consumes the history by replaying it. The size of an execution log can be limited to a certain size; thus, huge storage devices become unnecessary and the logging and replay technique can be applied more easily. The backup virtual machine can maintain the past state of the primary virtual machine along with the log to make the backup the same state as the primary. In case of the primary's failure, replaying the backup virtual machine from the saved state by following the saved log allows the execution path to the failure to be completely analyzed. Since the latencies from faults to failures are relatively short [5),7)], the cause of the failure can be captured and replayed for analysis. Therefore, simultaneous virtual-machine logging and replay can benefit from performing logging and replay.

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 proposed 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 provides an overview of the proposed 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

†1 Department of Computer Science, University of Tsukuba
*1 Presently with Information Technology R&D Center, Mitsubishi Electric Corporation

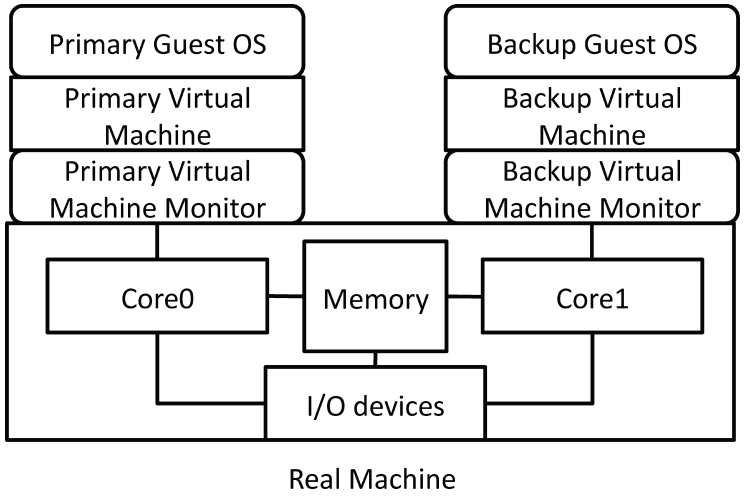


Fig. 1 Overall architecture of the proposed system.

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 like an ordinary guest OS that runs on a VMM. The only difference is that the primary VMM records the events that is described in the next section, so that its execution can be replayed on the backup.

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 to the shared memory, and the backup VMM reads the record. The mapping is created at 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 but not by Linux; thus, the existence of the shared memory region is not notified to Linux. The size of the shared memory is currently set to 4 MB. The size should be adjusted to an appropriate value depending on system usage.

The backup VMM reads the execution log from the shared memory, and provides the backup guest OS with the same events for replaying. The backup guest OS does not take any inputs from devices but only 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 for the design of our logging and replaying system and clarifies what needs to be logged. The basic idea of the logging and replaying of instruction execution is the treatment of factors external to the programs. These factors include inputs to programs and interrupts because they can change instruction execution streams and produced values. This paper describes the rationale specific to the IA-32 architecture [6].

3.1 Inputs

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. For example, the following code written in the IA-32 assembly language defines a function that adds 1 and 2, and returns the result, which is 3, in the %eax register:

```
1: add_noarg:
2:         movl    $1, %eax
3:         movl    $2, %ecx
4:         addl    %ecx, %eax
5:         ret
```

Because the data necessary to compute the result is included in the function definition, the returned result is always the same.

The following function defines a function that takes two arguments, adds them, and returns the result:

```
1: add_2args:
2:         movl    8(%esp), %eax
3:         addl    4(%esp), %eax
4:         ret
```

It does not contain the data necessary to compute the result, but obtains the data as the two arguments. Thus, the result depends on the arguments passed

from its caller. From the caller's point of view, however, the above function behaves always the same. It returns the same value if the data passed to the above function is the same; thus, the program behavior is always the same if the data passed to the function is included in the program as follows:

```
1:          movl    $2, 4(%esp)
2:          movl    $1, (%esp)
3:          call    add_2args
```

The following function is, however, totally different:

```
1: add_in_1arg:
2:          inl     $1, %eax
3:          addl    4(%esp), %eax
4:          ret
```

In this case, `add_in_1arg` takes one argument, copies the value from I/O port address 1, adds the argument and the value from the I/O port, and returns the result ⋆1. The value copied from the I/O port is not included in the program; thus, the result returned by the above function can be different because it depends on factors that are outside of the executed program.

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. If the values copied from I/O ports are different and those values are used for the conditions of branch instructions, the instruction execution stream of a program becomes different, and can produce and output different values. If the values copied from I/O ports are different and those values are copied to different I/O ports, a program outputs different values.

3.2 External Interupts

Interrupts change instruction execution streams. If only executed instruction addresses are examined, the inconsistent address changes caused by interrupts look the same as those by branches ⋆2. Software interrupts can, therefore, be treated in the same way as branches because they are caused by specific instructions. External interrupts are, however, caused by factors that are outside of the executed program. For example, when a key of the keyboard is pushed, an interrupt is asserted on a PC compatible system. The timing of a key stroke is not controlled by the program. Even when the program prompts a user to push a key, the user can push a key at a different time for each run of the program. Thus, the program cannot be expected to know the points where external interrupts are taken. 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 between 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 occur at the same points on the backup.

The problem is how to specify the 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.

The following program defines the function that calls function `do_something` a certain number of times as specified by the argument. After initializing registers, at line 4 it jumps to line 9 where the counter in the %ecx register is compared with the value given by the argument. If the counter is less than or equal to the argument value, it jumps to line 5, and calls `do_something` with the counter value as its argument. The loop is continued until the counter becomes greater than the argument value.

```
1: loop_calling:
2:          movl    $1, %ecx
3:          movl    4(%esp), %edx
4:          jmp     .L2
5: .L3:     pushl   %ecx
6:          call    do_something
7:          addl    $4, %esp
8:          incl    %ecx
```

⋆1 The use of I/O port address 1 is arbitrary and has no specific meaning.
⋆2 Because IA-32 instructions are variable length, near forward jumps may look consistent.

```
 9: .L2:    cmpl    %edx, %ecx
10:        jle     .L3
11:        ret
```

Let us consider an interrupt that was taken between lines 6 and 7. Just forcing the execution on the backup to call the interrupt handler after the first execution of line 6 may make many cases where the instruction execution streams on the backup are different from that of the primary. The interrupt could be taken after the second or later execution of line 6; thus, just recording the instruction address of line 6 as the point to call the interrupt handler is not enough. As this example illustrates, what we need to know is the number of times line 6 was executed. Counting that number, however, does not work because the arrival of an interrupt is unknown in advance; thus, we do not know for which line the number of iterations needs to be counted.

Instead, counting the number of branches works well even before an interrupt is coming in. For example, just before the first execution of line 7, after returning from `do_something`, the number of branches is 4, supposing there are no branches in `do_something`. It is because there are 2 jumps at lines 4 and 10, and 2 branches for calling and returning from `do_something`. Just before the second execution of line 7, the number of branches is 7. 3 is added because calling `do_something` causes 2 branches and 1 jump at line 10; thus, there are increments of 3 every time the execution reaches line 7.

Therefore, by recording the instruction address and the number of branches since the last interrupt, it is possible to specify the point where the interrupt is replayed.

The IA-32 architecture includes instructions that loops within a single instruction, and does not change the number of branches. The REP instruction repeats some types of load or store instructions for the number of times specified in the %ecx register. The following instruction repeats storing the value in the %eax register from the address specified by the %es:%edi register for the number of the times in the %ecx register:

```
1:         rep stosl
```

The repeating operation can be suspended by an interrupt, and be resumed again after the interrupt processing is finished. Therefore, the value of the %ecx register also needs to be recorded in order to cope with the cases when interrupts happen while the REP instruction is being executed.

3.3 Summary

In summary, the information needed to retrieve and record while logging on the primary is summarized as follows:

(1) the type of an event, an input or an interrupt,
(2) the input value or the IRQ#,
(3) the instruction address where the event happened,
(4) the number of branches since the last event, and
(5) the value of the %ecx register.

The information from (3) to (5) does not need to be recorded to replay an input event. If recorded, it can be used to make sure that an input event, which is going to be replayed, is the same as the recorded one.

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 the Intel VT-x [10] 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 logging, and then its recording. There are the two types of 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 determine which type of 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 control when the guest OS executes an IN instruction. It then executes the IN instruction with the same operand on behalf of the guest OS. Intel VT-x has a 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 can cause a VM exit. Bit 24 of the controls determines whether the executions of IN and OUT instructions cause VM exits. By setting this 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 the %eax register. The VMM copies the value to the guest %eax register, and it 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 the %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 the %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 in the shared memory. **Figure 2** shows the definition of the ring buffer entry structure in the C programming language. The fixed sized structure is used for both types of events. Structure members `index`, `address`, and `ecx` contain the event type, the instruction address where the event happened, and the value of the %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 bytes, which is well aligned with the IA-32 processor's cache line size of 64 bytes.

```
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.

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 a fixed value at 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 these indices are shared by both the logging side and the replaying side, they need to be shared.

4.2 Replaying

We describe the execution of the replaying that is performed by 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 perform. 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 control when the backup guest OS executes an IN instruction. The VMM on the backup sets up the 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 ensure the IN instructions are executed in the same order 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 anywhere 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 number of branches of the guest OS and the event record.
  (a) Proceed to the next step if they match. Go back to Step 1 above if they do not.
(3) Compare the values of the %ecx register of the guest OS and the event record.
  (a) Proceed to the next step if they match. Execute a single instruction and go back to Step 3 if they do not.
(4) Inject the interrupt into the guest OS.

At Step 1, the VMM needs to obtain 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 when executing 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 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 control after the single-step execution.

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 that was developed from scratch. The Linux operating system can boot and also be replayed on the VMM. All experiments described below were performed on a Dell Precision 490 system, which is equipped with a Xeon 5130 2.00 GHz 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 needed to boot Linux. Finally, we show the size of the log and a breakdown of the logged events.

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 necessary modifications to shadow paging, and shared memory. In order to support the logging and replaying mechanism, 1,694 SLOC were added. They include the code for logging, constructing the ring buffer, and 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 overhead of the logging and replaying, we first measured the boot time of Linux. The time measured is from `startup_32`, which is the beginning of the Linux kernel's boot sequence, up to the point where the init script invokes `sh`, which is the user level shell program. The measurements were

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

Table 2 Time to boot Linux with and without logging and replaying.

	Time to boot Linux [sec]	Ratio [%]
No logging	2.284	100.0
Primary with logging	2.286	100.1
Backup with replaying	2.319	101.5

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 (only 0.1% ⋆1), and the replaying imposes 1.5% of the overhead.

The logging and replaying impose 2 milliseconds and 35 milliseconds 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 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 such pointing more difficult because the same instruction addresses are executed multiple times; thus, the difference in the overheads between logging and 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 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 [9], but were modified to use the RDTSC instruction in order to measure the execution times. The measurements were performed 1,000 times, and their averages are reported.

Table 3 shows the results of the measurements. The results show that the overhead of the logging is 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 milliseconds 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 merely give insights into the analysis of the overheads.

Table 3 Benchmark results with and without logging and replaying.

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

Table 4 Breakdown of logged events.

Event	Count	Ratio [%]
Timer interrupts	207.5	8.00
Serial line interrupts	2	0.08
IN instruction	2,385	91.93
Total	2,594.5	100.01

The fork and exec system calls behave quite differently since they provide 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 using the copy-on-write technique. The exec system call, on the other hand, frees most of the 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 that typically uses REP instruction; thus, 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.

⋆1 It is actually slightly less since it has been rounded up to 0.1%.

⋆2 The total of the ratios is 100.01%, due to a rounding off error of 0.01%.

From the ratios of the breakdown, we can see that most of the logged events are IN instructions. The device driver of the serial line often uses IN instructions 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 instructions. 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 unmask the corresponding IRQ#.

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

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 overhead. We also discuss the current limitation of I/O and its possible improvements.

6.1 Reduction of Log Size

We discuss several ways to reduce the log size in order to make the deferred time longer. The purpose of simultaneous logging and replay is to enable the analysis of the execution path until failure; thus, making the deferred time longer increases the possibility for users to find causes of failures by replaying. Since the size of memory that can be used for logging is limited, the deferred time is also limited. Previous studies using fault injection techniques [5),7)], however, show that latencies from fault injections to system crash are relatively short; thus, the proposed system can very likely capture most causes of failures.

A straightforward way to reduce the log size is by reducing the size of each event. Currently, the fixed size of 32 bytes is used to record each event. 32 bytes was chosen because it is well aligned with the IA-32 processor's cache line size of 64 bytes.

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 consists 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 consists of 1 byte for the event type, 1 byte for IRQ#, 4 bytes for the instruction address, 5 bytes for the number of branches, and 4 bytes for the value of the %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 the %ecx register and the other without it.

By using record sizes of 2 bytes for an IN instruction event and 15 bytes for an interrupt event, the size of the log after booting Linux can be reduced to 7.91 KB. It is only 9.5% of the original log size; thus, a significant reduction of the log size is possible. By using these record sizes, 4 MB of the log buffer can defer the execution of the backup for 1,183.7 seconds (19 minutes and 43.7 seconds).

Another way to reduce the log size is to allow the VMM to emulate a serial device in a specific way. 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 the 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, reducing the number of events that need to be logged.

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 that 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. This is because the REP instruction is commonly used for data copying since it provides the most efficient way to copy data from one place to another. The REP instruction followed by the MOVS instruction copies the data pointed by the %esi register to the region specified by the %edi register using the %ecx register as its counter. It is a single instruction but repeats the execution of the MOVS instruction for the times specified by the

%ecx. An interrupt is serviced between the executions of the MOVS instruction within a single instruction pointer address. Thus, the only way to pinpoint the time to deliver an external interrupt within the REP instruction is to execute the instruction in single step mode until the value of the %ecx register becomes equal to the logged value. Execution in single step mode causes a VM exit after the execution of every MOVS instruction within the REP instruction. The VMM checks the value of the %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 the REP instruction becomes larger as its execution in single step mode is longer.

From the above reasoning, one straightforward way to reduce the replaying overhead is to avoid the use of the single step mode. It can be done by dividing the execution of the REP instruction into two parts, pre-injection and post-injection. The pre-injection and post-injection parts are executed before and after the injection of an external interrupt, respectively. By adjusting the value of the %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 the 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 the %ecx register is not used by the MOVS instruction within the 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 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.

6.3 Device I/O Modes

There are several modes to obtain inputs from I/O devices depending upon the processor architecture and the implementations of systems. Since the IA-32 architecture is equipped with the I/O port address space along with the memory address space, devices can be connected to either the I/O port address space or the memory address space. If a device is connected to the I/O port address space, it is accessed by separate instructions, IN and OUT. If a device is connected to the memory address space, it can be accessed by ordinary memory load and store instructions. This mode of I/O access is called memory-mapped I/O (MMIO). If a device is connected to the memory address space, it is possible to directly transfer between a device and memory without processor intervention. This mode of I/O access is called direct memory access (DMA).

The proposed system currently supports only devices connected to the I/O port address space. As described in Section 4, the result of the IN instruction is saved for the logging, and the saved data is used to replay an IN instruction event. Although the proposed system does not currently support MMIO devices, it can apply the same rationale also to the MMIO. If the VMM is notified when the guest OS is about to read data from MMIO devices, the VMM can log and replay accesses to MMIO devices. Such notifications can be enabled by setting up shadow page table entries appropriately. If the shadow page table entries that map MMIO devices are invalid, accessing those devices causes page faults. The VMM examines the faulted addresses and emulates accesses to them. At this time, it can save data for the logging or it can return the saved data for replaying. Therefore, MMIO devices can be supported easily.

The proposed system does not support DMA, and describing its support is outside the scope of this paper.

7. Related Work

There are several other studies on the logging and replaying mechanisms in VMMs. We took a similar approach to ReVirt [3], Takeuchi's Lightweight Virtual Machine Monitor [13], and Aftersight [14] in terms of the basic mechanisms. PMC is effectively used to count the number of events in their work as well as ours. They, however, store the log in storage devices, and perform the replay-

ing later. We propose a system where 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 neither uses the Intel VT-x nor emulates devices. Aftersight performs the replaying on the 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 the memory protection feature of an MMU.

Bressoud's fault tolerant system [1] uses a different mechanism from ours to enable the logging and replaying. 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 a network.

Remus [2] implements fault tolerance by checkpoint and restart mechanisms. A checkpoint mechanism saves the current state of an executing program, and a restart mechanism restores the saved state of an executing program and restarts its execution from the saved state. If a machine executing the program fails, its execution can be restarted on another backup machine from the saved state. Remus employs a VMM to save the state of an operating system and its applications and to restart the execution on a backup machine. Checkpoint and restart mechanisms may achieve better performance than the logging and replaying mechanism since it is possible for a checkpoint mechanism to save only the states changed from the previous checkpoint. Remus, however, does not redo the same execution on the backup machine; thus, it cannot provide execution paths towards failures.

8. Summary

We proposed a system that enables simultaneous virtual-machine logging and replay on the same system. It employs two virtual machines, one for the primary execution and the other for the backup execution. These two virtual machines run on a VMM of a single system. While the primary execution produces its execution history, the backup execution consumes the history by replaying it. The size of an execution log can be limited to a certain size; thus, huge storage devices becomes unnecessary and the logging and replay technique can be applied more easily.

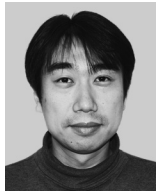
We developed such a logging and replaying feature in a VMM. The VMM is developed from scratch to run on an SMP PC compatible system. It can log and replay the execution of the Linux operating system. The experiments show that the overhead of the primary execution is only fractional, and the overhead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

References

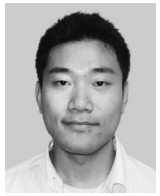
1) Bressoud, T. and Schneider, F.: Hypervisor-Based Fault Tolerance, *ACM Trans. Comput. Syst.*, Vol.14, No.1, pp.80–107 (1996).
2) Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N. and Warfield, A.: Remus: High Availability via Asynchronous Virtual Machine Replication, *Proc. 5th USENIX Symposium on Networked Systems Design and Implementation* (2008).
3) Dunlap, G., King, S., Basrai, M. and Chen, P.: ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay, *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, pp.211–224 (2002).
4) Goldberg, R.P.: Survey of Virtual Machine Research, *IEEE Computer*, pp.34–45 (June 1974).
5) Gu, W., Kalbarczyk, Z., Iyer, R.K. and Yang, Z.: Characterization of Linux Kernel Behavior under Errors, *Proc. 2003 International Conference on Dependable Systems and Networks*, pp.459–468 (2003).
6) Intel Corporation: IA-32 Intel Architecture Software Developer's Manual.
7) FERRARI: A Flexible Software-Based Fault and Error Injection System, *IEEE Trans. Comput.*, Vol.44, No.2, pp.248–260 (Feb. 1995).
8) Kawasaki, J. and Oikawa, S.: Co-Locating Virtual Machine Logging and Replay for Recording System Failures, *Proc. 10th IEEE International Conference on Computer and Information Technology*, pp.1352–1357 (2010).
9) McVoy, L. and Staelin, C.: LMbench: Portable Tools for Performance Analysis, *Proc. USENIX Annual Technical Conference*, pp.279–294 (Jan. 1996).
10) Neiger, G., Santoni, A., Leung, F., Rodgers, D. and Uhlig, R.: Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization, Technical Report, Intel Corporation (2006).
11) Oikawa, S. and Kawasaki, J.: Simultaneous Logging and Replay for Recording Evidences of System Failures, *Proc. 8th IFIP Workshop on Software Technologies*

for Future Embedded and Ubiquitous Systems (*SEUS 2010*), LNCS 6399, pp.143–154, Springer-Verlag (2010).
12) Rosenblum, M. and Garfinkel, T.: Virtual Machine Monitors: Current Technology and Future Trends, *IEEE Computer*, pp.39–47 (May 2005).
13) Takeuchi, S. and Sakamura, K.: Logging and Replay Method for OS Debugger Using Lightweight Virtual Machine Monitor, *IPSJ Journal*, Vol.50, No.1, pp.394–408 (Jan. 2009).
14) Chow, J., Garfinkel, T. and Chen, P.: Decoupling Dynamic Program Analysis from Execution in Virtual Environments, *Proc. USENIX 2008 Annual Technical Conference*, pp.1–14 (June 2008).

(Received November 29, 2010)
(Accepted May 14, 2011)
(Original version of this article can be found in the Journal of Information Processing Vol.19, pp.400–410.)



Shuichi Oikawa received his B.S., M.S., and Ph.D. degrees in Computer Science from Keio University in 1989, 1991, and 1996, respectively. He has been an Associate Professor of the Department of Computer Science at University of Tsukuba from 2004. Before joining University of Tsukuba, he worked at Keio University, Carnegie Mellon University, Intel Corporation, Sun Microsystems, and Waseda University. His research interest is systems software including operating systems and virtual machine monitors. He is a member of IPSJ, IEICE, and IEEE.



Jin Kawasaki graduated from Tokuyama College of Technology and received his B.E. from the National Institution for Academic Degrees and University Evaluation in 2008. He received his M.E. from University of Tsukuba in 2010. He is a researcher at the Information Technology R&D Center, Mitsubishi Electric Corporation from 2010. His major research interest is systems software. He is a member of IPSJ and IEICE.