소프트웨어 블랙박스:
운영체제 기반 전시스템 재현 방법
(Software Black Box: OS-based Full System Replay)

윤 현 민† (Hyunmin Yoon)         유 민 수†† (Minsoo Ryu)

요 약 결정적 재현 방법은 컴퓨터 시스템의 실행을 결정적으로 재현함으로써 디버깅, 고장감내, 보안 및 사후분석을 빠르고 편리하게 여러 영역에서 효과적으로 이용될 수 있다. 기존에 제안된 전시스템 재현 방법들은 전시스템 재현을 위해 특수한 하드웨어의 구현 또는 가상머신 기술의 지원을 요구한다는 제약이 있다. 본 논문에서는 특수한 하드웨어의 구현 또는 가상머신 기술의 지원을 요구하지 않는 소프트웨어 기반 전시스템 재현 기술인 소프트웨어 블랙박스(SBB)를 제안한다. 본 논문에서 제안하는 SBB는 과거 실행된 응용 프로그램과 운영체제의 실행을 결정적으로 재현한다. ARMv7 Cortex-A9 시스템 상에 구현된 SBB의 프로토타입은 응용 프로그램과 운영체제에서 발생할 수 있는 경쟁상태와 교착상태를 성공적으로 재현하였으며 낮은 성능 오버헤드를 보였다.

키워드: 전시스템 재현, 디버깅, 고장감내, 신뢰성, 검증

Abstract Deterministic replay mechanisms have proved to be useful in many areas including debugging, fault tolerance, security, and postmortem analysis because they can deterministically reproduce a computer system's execution. However, proposed full-system replay mechanisms have limited applicability because of their reliance on special hardware instrumentation or virtual machine (VM) technology. In this paper, we present a purely software-based approach to full-system replay, a software black box (SBB) that does not require either special hardware instrumentation or virtual machine technology. Our proposed SBB can deterministically replay a full software system, including both applications and the OS itself. We have implemented a prototype of SBB in an embedded RTOS on top of ARMv7 Cortex-A9 and have carried out experiments to evaluate our approach. Our experiments demonstrate that SBB can successfully reproduce subtle concurrency bugs, such as races and deadlocks that may occur both in applications and in the OS kernel. We also show that the event and data logging of SBB incurs such small performance overhead that it can be enabled permanently in the OS kernel.

Keywords: full-system replay, debugging, fault-tolerance, reliability, verification
1. Introduction

Deterministic replay is a technology that records and replays a computer system’s execution. Since deterministic replay can reproduce identical behavior during re-execution, it has shown very effective in many areas including debugging [1,2,3], fault tolerance [4,5], security [6] and post-mortem analysis [7].

Previous work on deterministic replay mostly centered around replaying a single process [8], an application [2], or a specific aspect of nondeterministic behavior [9] rather than full-system level behavior. While those approaches provide useful concepts and techniques for deterministic replay, their applicability is limited since debugging or analysis activity often deals with system-wide execution behavior to identify and locate complex bugs or malfunctions. More importantly, some bugs or malfunctions are often closely related to the underlying operating system’s behavior or I/O operations [10], thus requiring a full-system replay mechanism.

Some researchers proposed full-system replay mechanisms that rely on special hardware instrumentation or virtual machine (VM) technology. Xu et al. proposed a hardware technique for full-system replay of multiprocessor execution, called FDR (flight data recorder), that is capable of capturing architecture-level non-deterministic events and data associated with I/O, interrupts, and DMA [11]. Dunlap et al. proposed a VM-based approach, called ReVirt, that encapsulates the target system inside a virtual machine and provides an OS-on-OS style replay [6]. Oliveira et al. also proposed a VM-based approach, ExecRecorder, that is very similar to ReVirt in their structure and function [12]. However, these approaches have limited applicability because of their reliance on special hardware instrumentation or virtual machine (VM) technology.

In this paper, we present an effective software-based approach to full-system replay, software black box (SBB), that does not require any special hardware instrumentation or virtual machine technology. Our proposed SBB, as an OS instrumentation that is embedded within the target operating system, aims to record and replay a full software system including both the operating system and applications.

To the best of our knowledge, this is the first work that demonstrates the feasibility of a pure OS-based approach to full-system replay. Previously, this problem has been considered to be a very challenging task as many researchers regarded operating system behavior as nondeterministic [6,13], resulting in alternative approaches like VM-based solutions. Furthermore, the problem is made worse by the probe effect, i.e., perturbations on system behavior caused by the monitoring probes in the target system. Unfortunately, the probe effect has been less well addressed in previous work and most researchers simply attempted to avoid the probe effect by placing their monitoring probes outside of the target system [8,11].

This work makes three significant contributions. First, we attempt to show that the operating system behavior can be deterministically replayed in most practical cases. This attempt has mainly been motivated by the piecewise deterministic (PWD) assumption [14,15] and interrupt-based deterministic replay mechanisms [16,17], which led us to build a proof-of-concept system. Based on our experience, we present the detailed design and implementation of SBB for uniprocessor embedded system. Second, we discuss the probe effect caused by OS instrumentation and describe how to deal with the probe effect effectively. Last but not least, we also propose to capture performance related information in the record phase and make it available during the replay phase. We believe that this approach would be very useful for performance debugging and profiling since it can provide the programmer with correct performance data.

The remainder of the paper is organized as follows: Section 2 and 3 describe our design and implementation of SBB, respectively. Section 4 shows the experimental evaluation of SBB. Section 5 provides an overview of related work. Section 6 concludes the paper with our summary and future research.

2. Design of Software Black Box

Our approach to deterministic replay builds on the piecewise deterministic (PWD) assumption [14,15], which postulates that process execution consists of deterministic intervals and external inputs are the only sources of nondeterminism. This assumption
allows us to focus on a small set of external inputs, thereby making deterministic replay very simple and efficient.

We observe that most software program’s behavior is piecewise deterministic since most microprocessors execute the program’s instructions deterministically at the programmer-visible level. Thus, most operating systems can also be viewed as piecewise deterministic. This observation leads to the conclusion that full-system replay can be achieved simply by logging and replaying nondeterministic hardware inputs, i.e., interrupts and I/O data. Deterministic replay now becomes straightforward: (1) remember the initial state; (2) log interrupts and I/O data during execution; and (3) replay the logged interrupts and I/O data at the same execution points in the replay phase as those in the record phase.

Deterministic replay can be made easier by using hardware performance counters. Hardware performance counters are a set of hardware registers that can be configured to count performance related events such as instruction retirements and cache misses. In this paper, we use hardware performance counters for two specific purposes. The first one is to identify the correct execution points of asynchronous interrupts. The other one is to provide performance related information during the replay phase. Note that hardware performance counters are available on most modern microprocessors like Pentium 4 and ARM Cortex-A9 [18].

We now introduce our software black box (SBB). It consists of five major components, event recorder, performance recorder, performance log and event replayer, as shown in Figure 1. During the record phase, the event monitor records interrupts and I/O data into a special data structure, called event log. When the kernel terminates, the event recorder stores the event log in safe non-volatile storage. The performance monitor records hardware performance counter values and stores them in the performance log. During the replay phase, the event replayer emulates the interrupts stored in the event log.

2.1 Event Recorder

In order to efficiently record interrupts and I/O data, the event recorder is implemented within OS interrupt handlers and I/O device drivers. A fundamental question here is what and how much information we need to record. By taking advantage of hardware performance counters, it suffices to record the following four pieces of information for deterministic replay.

- Event type and source (who).
- Data (what).
- Program counter value or I/O address (where).
- Instruction count (when).

The first element represents the type and source of event. It is used to distinguish between interrupts and I/O data access. It is also used to identify the source of interrupt or I/O address. The second element represents the data obtained by interrupt handling or I/O data access. The third element may represent the program counter (PC) address where an interrupt occurred or the I/O address from which an I/O operation read the data. The last element represents an instruction count. We use instruction counts to determine when interrupts occurred. The following example shows why instruction counts are required.

As shown in Figure 2, suppose an interrupt occurs when the program counter address is 0x3000100C. This PC address can be revisited and another
interrupt may happen to occur at the same PC address. In that case, two different interrupts will be associated with the same PC address. Instruction counts can then be used to distinguish between the two interrupts in such a case.

We now formally define the event record format as a six-tuple <record number, event type and source, data, program counter value, instruction count, fingerprint>. Here, the record number has been added to maintain the order of event occurrence. The fingerprint has been added to compensate the possible error in instruction count. Note that hardware performance counters may contain some error and its measurement can also contribute to the error [19]. To overcome this problem, we introduce a fingerprint technique. We define a set of processor register values as a fingerprint. We can use fingerprint to distinguish between two interrupts when they are associated with the same PC address and the difference between their instruction counts are too small.

2.2 Performance Recorder

The performance recorder captures various types of performance counter values including cycle counts, instruction counts and cache miss counts. Note that the replay activity during re-execution impacts the performance of replayed system and also affects the performance counter values. This prevents the programmer from reconstructing the original performance data. Thus, to make correct performance data available for the programmer during the replay phase, we propose to record performance counter values during the record phase.

Performance counter recording takes place under the following four circumstances: (1) when an asynchronous interrupt occurs; (2) when an I/O device driver reads data from hardware; (3) when the operating system makes a scheduling decision; and (4) when an application attempts to read a certain’s performance counter. The first two circumstances are the same as those under the event recorder performs event recording. The third circumstance has been included to facilitate providing process- or thread-specific performance data. For the last circumstance, we assume that user applications are allowed to access performance counters only through OS-provided APIs. By modifying those APIs to record their readings, we can provide the recorded performance data in the replay phase. Otherwise, user applications will see different readings on different runs, which prevents deterministic replay.

2.3 Probe Effect and Deterministic Replay

Instrumentation code embedded in the target system causes the probe effect, which may manifest itself in diverse forms of side effects both on the operating system and processor. In general, there are two possible approaches to dealing with the probe effect. One approach is to attempt to eliminate the probe effect by placing monitoring probes outside of the target system. Previous work on deterministic replay mostly adopted this approach. For example, application-only replay mechanisms such as SCRIBE [13] and Flashback [8] implement the probes in the operating system that sits beneath the target application. Full–system replay mechanisms, hardware techniques [3,11] and VM-based schemes [6,12], implement the monitoring probes in hardware or within the host kernel, not in the target OS kernel.

The other approach is to leave the monitoring probes permanently enabled in the target system. Since the monitoring probes are viewed as part of the system, the “always-on” approach seems to allow us to ignore the probe effect. In general, this approach could be a viable solution if we can make the side effects of monitoring probes sufficiently small.

For our problem, we may rely on the always-on approach since we preclude the use of any external support like hardware instrumentation. However, the always-on approach is only applicable to the event monitor and performance monitor, but not applicable to the event replayer since it cannot be enabled in the record phase. As shown in Figure 3 and Figure 4, the event recorder and performance recorder can be enabled both in the record and replay phase, while the event replayer can be enabled only in the replay phase.

Note that the event replayer may affect the operating system’s execution such as process scheduling, memory management and various types of I/O operations, which may prevent correct deterministic replay. It can also impact the processor’s states such as registers, cache and pipeline, which directly influences the software execution. For example, the event replayer may cause an unintended page fault, which will change
the OS-level memory mapping and processor’s TLB (translation lookaside buffer) state. This suggests that the event replayer must run in non-pageable kernel memory.

In this paper, we choose to instrument an exception handler for the event replayer. We believe that the exception handler is the best place for the event replayer to run. First, it runs in non-pageable kernel memory, thus not affecting the operating system’s memory mapping. Second, it does not affect the operating system’s scheduling decisions since it has no thread of control. Third, we can invoke it at any desired instant of time by appropriately arranging an exception such as a breakpoint exception or performance counter overflow exception. This allows us to successfully replay asynchronous interrupts at desired execution points.

### 2.4 Event Replayer

The event replayer performs full-system replay by appropriately handling the recorded events and data guided by the event log. In order to avoid any external disturbance during the replay phase, the event replay mechanism keeps all hardware interrupts disabled and prohibits I/O drivers from accessing real hardware. In stead, the event replay mechanism emulates hardware interrupts and supplies requested I/O data consulting the event log.

For interrupt emulation, the event replayer makes use of hardware debug facilities, i.e., breakpoints and single stepping. The event replayer uses a breakpoint exception to initiate interrupt emulation at the same relative execution points in the replay phase as those in the record phase. When a breakpoint exception occurs, the event replayer first checks if this moment is eligible for interrupt emulation by comparing the current instruction count and the recorded one in the event log. Due to some measurement errors and possible inaccuracy of hardware performance counters, we use a margin of error $\Delta IC$ and fingerprints to determine correct execution points.

We now describe the process of interrupt emulation following the flow chart in Figure 5. Let $IC_{\text{replay}}$ be the current instruction count and $IC_{\text{record}}$ be the saved instruction count during the record phase. When a breakpoint exception occurs at some recorded program counter address $PC_{\text{record}}$, the event replayer first compares $IC_{\text{replay}}$ and $IC_{\text{record}}$:

$$\left| \frac{IC_{\text{replay}} - IC_{\text{record}}}{IC_{\text{record}}} \right| > \Delta IC$$

When Eq. (1), the event replayer resumes the original software execution without initiating interrupt emu-
In doing so, the event replayer uses a single stepping exception to resume the interrupted execution.

\[
\left| \frac{IC_{\text{replay}} - IC_{\text{record}}}{IC_{\text{record}}} \right| \leq \Delta IC
\]  

(2)

When Eq. (2), the event replayer again checks if the fingerprint matches. If so, the event replayer initiates interrupt emulation. Specifically, the event replayer first examines the event log to see what type of interrupt should be emulated and then emulates interrupt handling as if the same type of interrupt has occurred.

For I/O emulation, the event replayer maps all I/O addresses into a special memory region, called I/O emulation buffer, and copies the I/O data stored in the event log into that region. During replay, the event replayer keeps the I/O emulation buffer always consistent with the event log and I/O driver operations are redirected to I/O emulation buffer for every I/O read access.

3. Implementation

We have implemented a prototype of SBB in an embedded RTOS, called QURIX [16], on top of 1GHz ARMv7 Cortex-A9 Samsung Exynos S4210 microprocessor. The development board includes a multiprocessor SOC, however QURIX with SBB only supports a single core. The SBB provides four major functions for event and performance recording, `record_irq()`, `save_log()`, `record_io()` and `record_performance()`.

`record_irq()`: This is invoked in IRQ handlers. It is responsible for recording five pieces of information: record number, event type and source, program counter value, instruction count, and fingerprint. Here, care should be taken for program counter value and fingerprint. These two entries must be taken from the context of interrupted code that were running when the interrupt occurred, not from the current IRQ context. For example, program counter value should be the address of the last executed instruction, which can be calculated by subtracting 8 bytes from the link register value in standard ARM processors. The fingerprint entry should also be created from the context of interrupted code. When an interrupt occurs, some register values are saved in stack. Those saved register contents are used for the interrupt handler to return to the interrupted code. We should also use the saved register contents to form a fingerprint. Note that we do not attempt to calculate the exact value of instruction count since the difference would be minor when compared to other types of of errors.

`save_log()`: This is used to save the event log in non-volatile storage when the kernel terminates. There are two cases when `save_log()` is executed in the kernel, normal termination and abnormal termination. Accordingly, `save_log()` is inserted in both the normal termination code and exception handlers that deal with abnormal termination.

`record_data()`: This is invoked in device driver routines. We inserted `record_data()` into two QURIX device driver routines, `io_read_timer()` and `io_read_serial()`. This function stores all data read from timer and serial devices into the event log, and also carries out an optimization to reduce the amount of log data. The main purpose of the optimization is to minimize the log data generated from busy-wait polling. Note that an I/O operation often needs to check a status flag of I/O device in a busy waiting loop. In this case, the event log would be filled with a lot of consecutive event records that contain identical contents. To alleviate this problem, we use the run-length encoding technique [20] to cluster those identical event records into a single event record.

`record_performance()`: As stated previously, this is invoked in four cases: when an asynchronous interrupt occurs; when an I/O device driver reads data from hardware; when the operating system makes a scheduling decision; and when an application attempts to read a certain performance counter. We inserted `record_performance()` into `record_irq()` and `record_data()`. We also inserted `record_performance()` into the OS scheduler function `schedule()`. QURIX provides a system call `read_hpc()` for performance counter access. We also modified `read_hpc()` to invoke `record_performance()`.

The SBB provides three functions for event replay, `replay_irq()`, `io_read_timer()` and `io_read_serial()`.

`replay_irq()`: This is invoked in the Data Abort exception handler that is responsible for handling breakpoint exceptions in ARM [21,22]. Its main function
is to examine instruction counts and fingerprints and makes a decision. If the decision is no, it causes a single stepping exception to resume the interrupted execution. If the decision is yes, it invokes an appropriate IRQ handler. We used $\Delta IC = 5\%$ as a margin of error, which has been empirically determined through our experiments. Note that it is very difficult to find an accurate margin of error or compensate the instruction count errors [23]. Also, there have been some studies to improve the accuracy of fingerprint techniques [24,25]. These are out of scope of this paper, and thus we do not endeavor to discuss these issues.

io_read_timer() and io_read_serial(): These two functions replace their original versions in QURIX and record all I/O data read from I/O devices. These new functions are different from the original ones in that they read I/O data from the event log, not from real hardware.

4. Evaluation

In this section we evaluate SBB through case studies and performance measurements. In summary, we could successfully reproduce subtle concurrency bugs, races and deadlocks, both in user-level programs and the operating system kernel. Our measurements show that the performance overhead caused by the event recording of SBB is also moderate, i.e., 0.28% and 1.78%.

4.1 Concurrency Bug Reproduction

We wrote a simple concurrent program consisting of multiple threads, each of which runs a loop for 300,000 iterations. On every loop iteration, each thread modifies the same shared variable without using a lock and stores it in a shared buffer guarded by a lock. The first thread decrements the variable by one at each access; the second thread increments the variable by two; and the last thread increments the variable by three. The threads were scheduled by a round robin scheduler with 10ms time quantum.

With the shared variable initially set to 0, the final value of shared variable must be $1,200,000$, i.e., $(-1+2+3) \times 300,000 = 1,200,000$, when these threads are properly synchronized. Thus, we can easily detect race conditions in this example by simply checking the final value of the shared variable. We can also determine when the race condition occurred by looking at the sequence of values stored in the shared buffer. We ran this example program over 240 times and verified that race conditions can be reproduced at the same execution points in replay phase as those in record phase.

The second example demonstrates the reproduction of deadlocks. We wrote a dining philosophers program [26] in which five philosopher threads are given five chopsticks. Each thread executes a loop thinking and eating. The time durations of thinking and eating were chosen using random numbers, for which cycle counts were taken as seed values. In this example, a deadlock may occur when every philosopher has picked up a chopstick simultaneously, and waits for another chopstick without releasing her chopstick. With this example, we were also able to reproduce the same deadlock situation during replay.

The last example demonstrates that SBB is able to reproduce race conditions even in the OS kernel. We again used the dining philosophers program in which five threads executes printf() concurrently. In the original QURIX kernel, there is a kernel buffer that stores the character string received through printf(). To avoid race conditions, this kernel buffer is protected by a lock. For our experiments, we removed the lock that protects the kernel buffer and then ran the dining philosophers program. In this experiment, we could also successfully reproduce race conditions that may occur even in the OS kernel.

4.2 Performance Measurement

We ran the above producer-consumer program to evaluate the performance of SBB. In doing so, we enabled two I/O devices, timer and serial devices, and used the round robin scheduling with 10ms time quantum.

We first examined the performance overhead incurred by the event recorder and performance recorder. The results are presented in Figure 6 and Figure 7, where NRC represents the execution of target system without SBB, ERC represents the execution of target system with both event recorder and performance recorder. Figure 6 and Figure 7 show that the overhead of event recorder and performance recorder is moderate. The performance overhead caused by the event recorder (ERC) ranges from 0.28% to 1.78% and its average is 1.16%. We can also observe that the performance overhead does
not increase with the number of threads. Figure 7 also shows that the combined overhead of event recorder and performance recorder (ERC+PRC) is low. The additional overhead ranges from 0.60% to 1.41% and its average is 0.60%. Our results are comparable to the previous results reported in [8,16,27]. Note that the overhead of RT-Replay’s recording operation is 0.74% and the overhead of logging synchronization operations is 2.1% of CPU utilization in [27]. Here, we can also see that the combined overhead of ERC+PRC is slightly less than the overhead of ERC. This anomaly may be attributed to measurement errors or the inaccuracy of performance counter.

The next set of experiments shows the size of event log. Figure 8 presents the results, where the event log grows at around 12.26KB/sec. This is also comparable to previous results [16,27]. The storage bandwidth needed by RT-Replayer and Recplay is 1.3KB/sec and 7KB/sec to 4KB/sec, respectively.

Finally we also examined the performance of event replayer. Figure 9 presents the results showing that the overhead of event replayer is high. The overhead ranges 1162.92% to 1184.81%. The replay time is heavily influenced by the event rate of the program and the search time of the interrupt replay point. In the evaluation shown in Figure 9, the target application repeatedly executes the execution section requiring a short execution time. Such a program causes break-point exception handling to be performed in a short period to detect the point at which the interrupt event is reproduced, and then the reproduction time becomes long. However, the performance of event replayer is not as much important as the performance of event recorder and performance recorder, since the event replayer runs off-line.

5. Related Work

Deterministic replay has been one of the biggest research fields due to its wide range of application areas over 40 years [28]. Early replay systems focused on single processor replay [6,9,29-31]. Recently, multiprocessor replay has received much attention due to the growing use of multicore processors. For
multiprocessor replay, many researchers focused on the problem of logging and replaying shared memory accesses [2,32]. Other well-known results include hardware mechanisms [3,5,11], software-based replayers to debug multithreaded applications [2,5,8,27,33], and checkpoint-based approaches [4]. In the following, we examine previous work from a full-system replay point of view.

Previous work on deterministic replay centered around replaying a single process, an application, or a specific aspect of non-deterministic behavior rather than full-system level nondeterministic behavior. Leblanc and Mellor-Crummey focused on the nondeterministic behavior produced by interactions between processes. They modeled process interactions as operations on shared objects and presented Instant Replay that can record and replay a partial order of concurrent access to shared objects [2]. Choi and Srinivasan examined synchronization operations among concurrent Java threads. They introduced the notion of logical thread schedule and proposed DejaVu that can capture and replay the same order of synchronization operations [9]. Srinivasan et al. proposed Flashback to record and replay interactions between a program and the underlying operating system, i.e., system call invocations, signals and shared memory operations [8].

Xu et al. proposed a hardware-based approach, called FDR (flight data recorder), for full-system replay of multiprocessor execution [11]. FDR was designed to record architecture-level non-deterministic events and data associated with I/O, interrupts, and DMA. FDR is a hardware technique that should be implemented as additional hardware blocks within a processor architecture, and thus is not applicable to typical computer hardware. Moreover, FDR generates log data more than 20 MB/second, which is too much to be practical [3,11].

Dunlap et al. proposed a software-based approach, called ReVirt, for full-system replay based on virtual machine (VM) logging and replaying [6]. ReVirt encapsulates the target system inside a virtual machine and logs external input beneath the virtual machine. In their VM-based system consisting of a host operating system, VMM (virtual machine monitor), and guest operating system, timer interrupts and I/O interrupts are delivered to the guest operating system in the form of signal (SIGALRM, SIGIO, and SIGSEGV). I/O input data from external entities are delivered by system calls recv, read, and gettimeofday. ReVirt logs and plays back these two types of nondeterministic events to deterministically replay the target system (both operating system and applications). Oliveira et al. also proposed a VM-based approach, ExecRecorder, that is very similar to ReVirt in their structure and function [12]. However, both approaches rely on the virtual machine mechanism, thus are not applicable to common direct-on-host systems.

6. Conclusion

In this paper, we presented a software-based full-system replay mechanism, software black box (SBB), that is able to record and replay a full software system including both the operating system and applications. We implemented a proof-of-concept system and showed that SBB can reproduce subtle concurrency bugs, races and deadlocks, while incurring sufficiently low performance overhead ranging from 0.28% to 1.78%. There still remain a lot of work for future research. First, we plan to apply the proposed SBB to a full-fledged operating system like Linux. This seems to be very challenging because Linux kernel has higher complexity than the research kernel QURIX and supports devices that generate high I/O stresses such as network and storage. Second, we are also looking at diverse types of benchmarks for further evaluation. Once our SBB is ported to Linux, then lots of benchmarks would be available for our experiments. Finally, we will explore the problem of deterministic replay for multiprocessors. We believe that some ideas and techniques provided by SBB could also be very helpful for multiprocessor deterministic replay.

References


[28] D. F. Bacon and S. C. Goldstein, "Hardware


