Hindawi
Scientific Programming
Volume 2018, Article ID 8939027, 11 pages
https://doi.org/10.1155/2018/8939027

Research Article

Debugging Nondeterministic Failures in Linux Programs through Replay Analysis

Shakaiba Majeed[1] **and Minsoo Ryu**[2]

[1]*Department of Computer and Software, Hanyang University, Seoul 04763, Republic of Korea*
[2]*Department of Computer Science and Engineering, Hanyang University, Seoul 04763, Republic of Korea*

Correspondence should be addressed to Minsoo Ryu; msryu@hanyang.ac.kr

Received 6 November 2017; Revised 22 February 2018; Accepted 8 March 2018; Published 12 April 2018

Academic Editor: Danilo Pianini

Reproducing a failure is the first and most important step in debugging because it enables us to understand the failure and track down its source. However, many programs are susceptible to nondeterministic failures that are hard to reproduce, which makes debugging extremely difficult. We first address the reproducibility problem by proposing an OS-level replay system for a uniprocessor environment that can capture and replay nondeterministic events needed to reproduce a failure in Linux interactive and event-based programs. We then present an analysis method, called replay analysis, based on the proposed record and replay system to diagnose concurrency bugs in such programs. The replay analysis method uses a combination of static analysis, dynamic tracing during replay, and delta debugging to identify failure-inducing memory access patterns that lead to concurrency failure. The experimental results show that the presented record and replay system has low-recording overhead and hence can be safely used in production systems to catch rarely occurring bugs. We also present few concurrency bug case studies from real-world applications to prove the effectiveness of the proposed bug diagnosis framework.

1. Introduction

Debugging is the hardest part of software development. Traditionally, the process of debugging begins by reproducing a failure, then locating its root cause, and finally fixing it. The ability to reproduce a failure is indispensable, as, in most cases, it is the only way to provide clues to developers in tracking down the sources of failure. However, in the case of some nondeterministic failures such as concurrency bugs, it is not always possible to reproduce the failure provided a given set of inputs and environmental configurations. Without the ability to reproduce, debugging becomes an inefficient and time-consuming process of trial and error. Consequently, some software practitioners report that it takes them weeks to diagnose such hard-to-reproduce failures [1].

To deal with nondeterministic failures, record and replay tools have been demonstrated to be a promising approach. Such tools record the interactions between a target program and its environment and later replay those interactions deterministically to reproduce a failing scenario. A number of record and replay systems have been proposed in recent years, but many of them incur high overheads [2–4], others lack supporting low-level events [5, 6], while others may require special hardware support [7, 8].

In this work, we first show that a computer program's nondeterministic behavior can be fully identified, captured, and reproduced with instruction-accurate fidelity at the operating system level by using existing hardware support in modern processors. From this vantage point, a developer can replay a nondeterministic failure of a program and effectively diagnose it using traditional cyclic debugging methods.

We then present an analysis method, called *replay analysis* that allows us to effectively locate the source of failures. Although many bugs can be diagnosed with the help of debuggers during replay, finding the root cause of certain elusive bugs such as concurrency bugs still remains a challenging task for developers. To help developers locate the root cause of such failures, we present a framework based on the proposed record and replay. The contributions of this paper are twofold:

(1) It presents an idea of OS-based record and replay system capable of intercepting the nondeterministic events occurring in interactive and event-driven programs. To substantiate this idea, we implemented it in an ARMv7 uniprocessor-based Linux system. The system incurs low overhead during recording. Therefore, it can be used in an always-on mode in testing phases or production systems to catch nondeterministic and rarely occurring bugs.

(2) It describes a replay analysis method for diagnosing concurrency bug failures based on the proposed record and replay system. The method specifically targets single-variable data races and atomicity violations. The method uses a combination of static analysis, dynamic tracing during replay, and delta debugging to identify failure-inducing memory access patterns that lead to a concurrency bug.

The remainder of this paper is organized as follows. In Section 2, we describe the record and replay system and its implementation details for the ARMv7-based Linux system. We present its evaluation results in the same section. In Section 3, we present the concurrency bug diagnosis framework based on the proposed record and replay system and show its effectiveness by presenting a few debug case studies. We present some related work in Section 4 and discuss the implications and limitations of our current work in Section 5. We finally conclude with Section 6.

2. OS-Level Record and Replay System

2.1. Overview. A program is considered to be deterministic if, when it starts from the same initial state and executes the same set of instructions, it then reaches the same final state every time. In modern computer systems, however, even sequential programs can show unpredictable behavior because of their interaction with the environment, such as I/O, file systems, other processes, and with humans through UIs. Moreover, the occurrence of interrupts and signals can result in varying control flow during successive runs of the same program.

For debugging, these subsequent runs of a failing program can be made deterministic by recording the nondeterministic factors in the original run and substituting their results during replay. For a user-level program, such factors generally include external inputs, system call return values, scheduling, and signals. There are indeed some other sources of nondeterminism that exist at the microarchitecture level, for example, cache or bus states, blocking of I/O operations, and memory access latencies. Such nondeterminism causes a timing variation which may affect when external data is delivered to a user program or when an asynchronous event-handler is invoked. To handle this type of nondeterminism, we use a logical notion of time by keeping track of the number of instructions executed by a process between two nondeterministic events. The logical time helps in maintaining the relative order of the nondeterministic events during replay and guarantees the replication of the functional behavior of the program. For a debugging usage model where the goal is to find errors in a user program, it is sufficient to reproduce the functional behavior of the program rather than its temporal behavior. Therefore, nondeterministic factors existing at the architecture or circuit level which may cause timing variations are out of scope of this work.

The nondeterministic events exposed to a user program can be captured at different abstraction levels, that is, library-level, OS-level, and hardware-level. In general, the higher the abstraction level is, the smaller the performance overhead is, but with less accuracy. In the current work, we implemented the record and replay framework at the OS-level because we believe that the operating system is the perfect place to intercept nondeterministic events with instruction-level accuracy before they are projected to a user-space program. Moreover, many modern computer architectures such as Intel x86, PowerPC, and ARM include rich hardware resources such as performance monitors, breakpoints, and watchpoints that can be exploited at the OS-level to support deterministic replay. Implementation at the OS-level also has the advantage that it does not require any modifications to the target program or the underlying architecture.

Figure 1 illustrates an overall idea of OS-level record and replay system. Using this system, a target program can be run in either a *record* or a *replay* mode. During the testing or production run, the program is executed in *record* mode wherein the system captures its nondeterministic events. Each event is stored in an event log created for the target program inside the kernel memory. As the program continues execution, events keep on adding to this log and are moved periodically to a log file in permanent storage.

When a program is set to run in *replay* mode, the prerecorded events from its log file are moved to the program's event log. Whenever the program tries to execute a nondeterministic event, for example, a system call, its results are extracted from the event log and sent to the target process. Thus, all the events are executed deterministically, and the program considers that those events are happening as they actually did during recording in view of both the data transferred and the timing of events. To perform cyclic debugging, it is possible to attach a standard debugger to the process being replayed, for example, GDB. The debugger can control the execution of the program normally through single step, continue, or breakpoint commands and present the results to the user as if they are generated live.

The record and replay system is implemented at the system call and signal interface in the operating system since together these interfaces represent most of the nondeterministic events found in sequential and event-based programs. These events include data from external devices and the file system, input from timers, interprocess communication, asynchronous event-notification, and interrupts. Therefore, the proposed record and replay system is suitable for debugging a large class of interactive programs including those that generally come with a Linux distribution and other high throughput programs such as Lynx [9], Lighttpd [10], and Nginx [11].

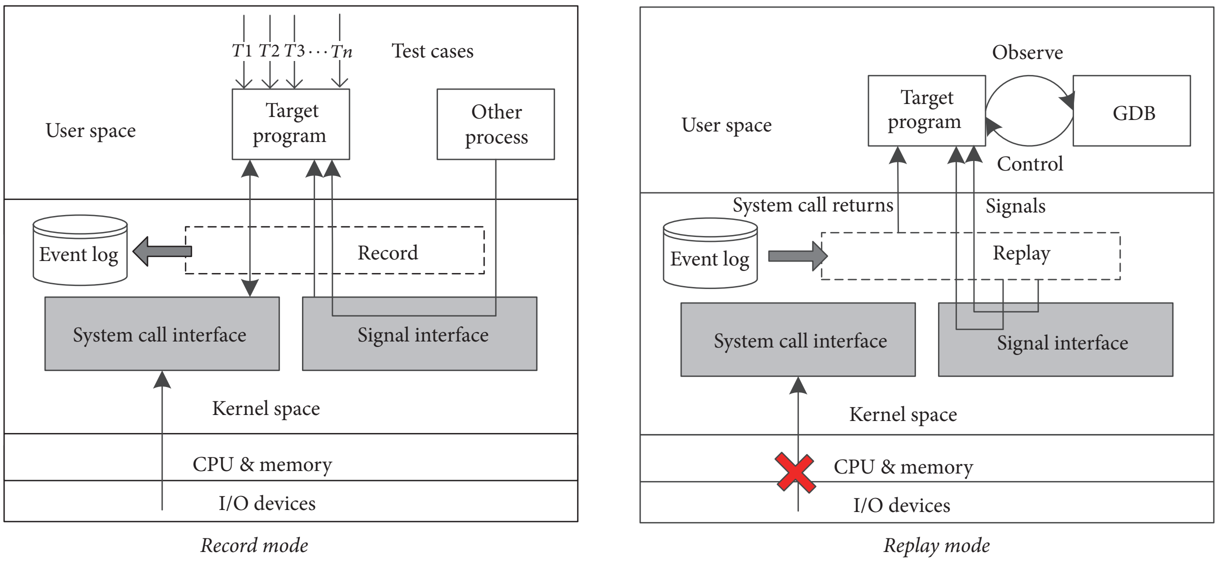

Figure 1: Record and replay system at OS-level.

In the following subsections, we discuss how we implemented the proposed record and replay framework for a Linux-based ARMv7 system. The entire record and replay system is transparent from the target program and is implemented in software by using support from the ARM's debug and performance management unit (PMU) architecture. Although the implementation details given here are specific to one architecture, the presence of hardware debug registers and performance monitoring counters in other architectures along with the existing support in commodity operating systems to use these resources makes our approach more generic.

2.2. System Calls. System calls in an operating system provide an interface for a user-level process to interact with its external environment by sending requests to the kernel. In a typical UNIX system, there are around 300 system calls. However, the effect of all the system calls may not be nondeterministic with respect to a process. We broadly categorize the system calls as follows:

(i) *I/O control*: read(), write(), socket(), sendmsg(), and so on.

(ii) *Interprocess communication (IPC)*: pipe(), mq_open(), mq_unlink(), and so on.

(iii) *Time related*: gettimeofday(), gettimer(), utimes(), and so on.

(iv) *Process control*: fork(), exec(), wait(), and so on.

(v) *Memory management*: mmap(), munmap(), mremap(), and so on.

The I/O system calls allow a process to interact with hardware devices, networks, and file systems whereas IPC system calls are used to interact with other processes. The results of these system calls cannot be predicted by the user application, so we consider these as nondeterministic system calls. Time-related system calls always return values local to a processor on which they are executed and are different every time.

The process control system calls manage a process status. Such calls only change or get the value of a process control block, so the associated events can be considered deterministic. The memory control system calls are used to handle memory allocation, heap management functions, and so on. Even if these system calls are recorded, we must re-execute them during replay to generate their side effects within the operating system kernel. Otherwise, if, for example, memory is not actually allocated during the replay phase, the kernel will throw an exception when the program tries to access the memory region it has supposedly allocated.

Recording of system calls belonging to the latter two categories is therefore redundant, and in the current work, we do not record them. By eliminating these system calls, we significantly reduce the recording overhead, making the entire record and replay system more efficient.

When a user program invokes a system call, the processor switches from user mode to kernel mode and begins executing the system call handler. If the program is in the recording mode, the system call execution is allowed normally, but before returning to user mode, we log the return value of the system call. In the ARM architecture, this value is returned in r_0 register. Some system calls also return nondeterministic data by updating special data structures in the kernel-space. For example, a read system call returns the data that has been read from a file descriptor on behalf of the user process making the system call. Such data is sent from kernel to user-space through copy_to_user or put_user functions. We log the data returned by system calls in these functions and add it to the corresponding system call log in the process's event log.

During the replay mode, when the program tries to execute a nondeterministic system call, its handler is replaced by a default function. This function reads the return value of the current system call from the recorded event log and sends it to the process. Any data associated with the system call which was saved during the recording is also returned to the program. Thus, we simulate the effect of a system call completely, rather than executing it during replay.

2.3. Signals. Signals are a type of software interrupt present in all modern UNIX variants. They are used for many non-trivial purposes, for example, interprocess communication, asynchronous I/O, and error-handling. Signals are sent to a process asynchronously and proactively by the kernel. Therefore, signal handling is a challenging task for any record and replay system. Many existing record and replay frameworks do not support signal replay. Some methods exist to support signal replay [12, 13], but they change a signal's semantics during the record phase by deferring its delivery until a certain synchronized point in the process execution, for example, invocation of a system call. Some applications are very sensitive to the occurrence of asynchronous events, and changing the semantics of signals can distort their behavior entirely; it is, therefore, important to guarantee the exact record and replay of signals.

Before describing our process of signal record and replay, we briefly discuss how signals are handled in Linux. When a signal is sent to a process, the process switches to kernel mode. If it is already in the kernel mode, then after carrying out the necessary tasks and before returning to the user mode, the kernel checks if there are any pending signals to be delivered to the process. If a pending signal is found, it is handled in the do_signal() routine, where the corresponding signal handler's address is placed into the program counter (PC), and the user mode stack is set up. When the process switches back to user mode, it executes the signal handler immediately.

During the recording mode, whenever a signal is delivered to a process being recorded, we log the signal number and the user process register context in the do_signal() routine before it modifies the current PC. The exact instruction in the process's address space "where" the signal arrived is indicated by the PC logged in the register context. However, recording only the instruction address is not sufficient because the same instruction may be executed multiple times during the entire program execution, for example, in a loop. We also need to log the exact number of instructions executed by the process to identify "when" the signal arrived. To count the number of instructions, we utilize ARM's performance monitor unit (PMU) architecture. The Cortex-A15 processor PMU provides six counters. Each counter can be configured to count the available performance events in the processor. We programmed one of the counters to count the number of instructions architecturally executed by a user process. During the recording phase when a nondeterministic event occurs, the current instruction count is also stored in that event's log. The instruction count is then reset, and it starts counting again until the process encounters another nondeterministic event to be logged. Thus, two sequentially occurring nondeterministic events, for example, a system call and a signal, are separated by the exact number of instructions executed between the two events by the process being recorded.

To replay the signals we make use of ARM's hardware breakpoint mechanism. During the replay phase, after processing an event, we always check what an upcoming event is in the process's recorded log. If the next event in the log is a signal, then a breakpoint is set at the instruction address of the user program recorded in the signal log. The instruction counter is reset to begin counting the instructions from the last replayed event. When the replayed program reaches the instruction where the breakpoint is set, a prefetch abort exception occurs, and the process switches to kernel mode. In the exception handler, we compare the current number of instructions executed to the instructions stored in the signal log. If they match, the signal is immediately delivered to the user process. If they are not, the breakpoint is maintained at the current PC using the ARM's breakpoint address match and mismatch events, and the process execution is allowed until both PC and instruction count match those of the recorded values in the signal log. In this way, the signal delivery to the target program with instruction-level accuracy is guaranteed during the replay phase. The replay process for a signal is illustrated in Figure 2.

2.4. Evaluation. To capture nondeterministic bug, recording is often required to be done in production systems. Therefore, it is very important that recording overhead be low enough and minimally intrusive to avoid any adverse effect on a production application's normal execution. We evaluated the performance of our record and replay system regarding recording overhead for various real applications. The experiments were performed on a Samsung Exynos Arndale 5250 board based on Cortex-A15 processor, with record and replay mechanism implemented in Linaro 13.09 server with a Linux 3.12.0-rc5 kernel.

We recorded and replayed some Linux applications, which are listed with their workload conditions in Table 1. These programs were run in their default configuration. For each of these applications, tests were repeated three times. We report here the average of the three results.

The performance overhead of recording the application workloads is shown in Figure 3. For each workload, we measured the performance as CPU time in seconds except for the *netperf* tests where it is measured as throughput in Mb/sec for the TCP stream test and completed transactions/sec for the request/response test.

The results are displayed normalized to native execution without recording. The overhead of recording was under 5% for all the experiments, except the TCP request/response test, which caused 15% overhead. The recording overhead is directly related to the size of the logs. The events generated during recording must be moved from the kernel memory to permanent storage, causing extra overhead. During the request/response test, a large number of requests are generated. Each request/response has the potential to generate enormous quantities of network data that must be logged, resulting in a relatively larger log size as compared to other tests and therefore causing more slowdown. The slowdown can be improved by compressing the logs or by

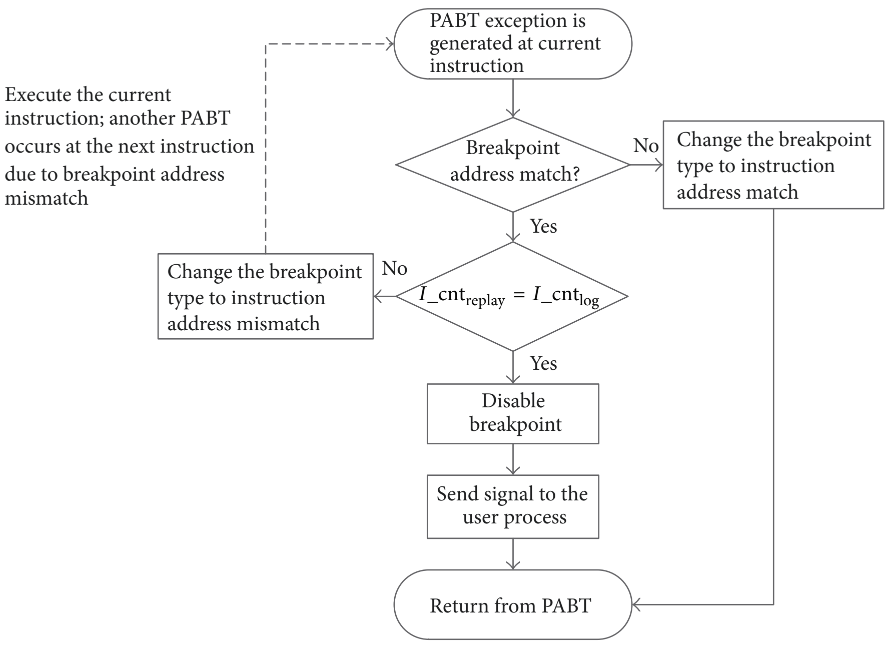
PABT exception is generated at current instruction
Breakpoint address match?
No
Change the breakpoint type to instruction address match
Yes
Execute the current instruction; another PABT occurs at the next instruction due to breakpoint address mismatch
Change the breakpoint type to instruction address mismatch
No
$I\_cnt_{replay} = I\_cnt_{log}$
Yes
Disable breakpoint
Send signal to the user process
Return from PABT

Figure 2: Signal replay process.

Table 1: Test application scenarios.

Program	Description	Workload
lame	A high-quality MPEG Audio Layer III (MP3) encoder	Encode a 3.5-minute (7751 frames) .wav file
GNU bc	An arbitrary precision numeric processing language calculator	Load the math library and process an input file containing mathematical operations
vim	Vim 7.3 text editor	Open an existing text file in vim, and append an eight-character string 10,000 times
bzip2	A high-quality data compression/decompression utility	Decompress Linux-3.0.1.tar.bz2 of size 73.1 MB
netperf-TCP_STREAM	Networking performance measuring benchmark	TCP_STREAM test between the local host and a remote host for 60 secs with default window size
netperf-TCP_RR	Networking performance measuring benchmark	TCP request/response test between the local host and a remote host for 10 secs

better scheduling of write operations to permanent memory. However, discussion of these optimization methods is out of the scope of current research.

3. Replay Analysis for Diagnosing Concurrency Bugs

Concurrency bugs are generally associated with multi-threaded programs. However, researchers have shown that they also exist in sequential [14], interrupt-driven [15], and event-based programs [16]. The execution of signal-handlers, interrupt-handlers, and other asynchronously invoked event-handlers interrupts the control flow of these programs and so introduces fine-grained concurrency. Data among event-handlers and the main code is shared through global variables. (In the rest of this paper, we shall use the term

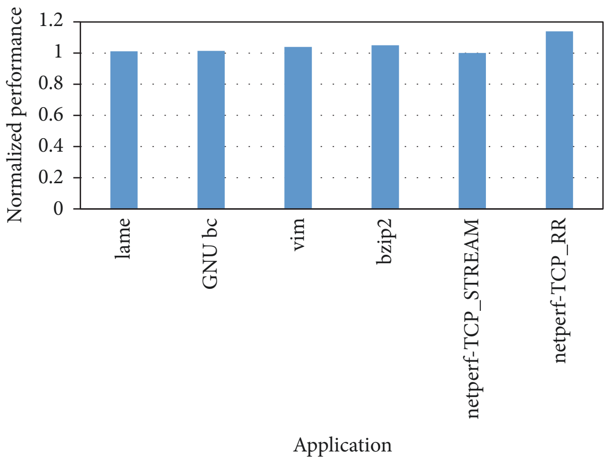

Figure 3: Recording overhead.

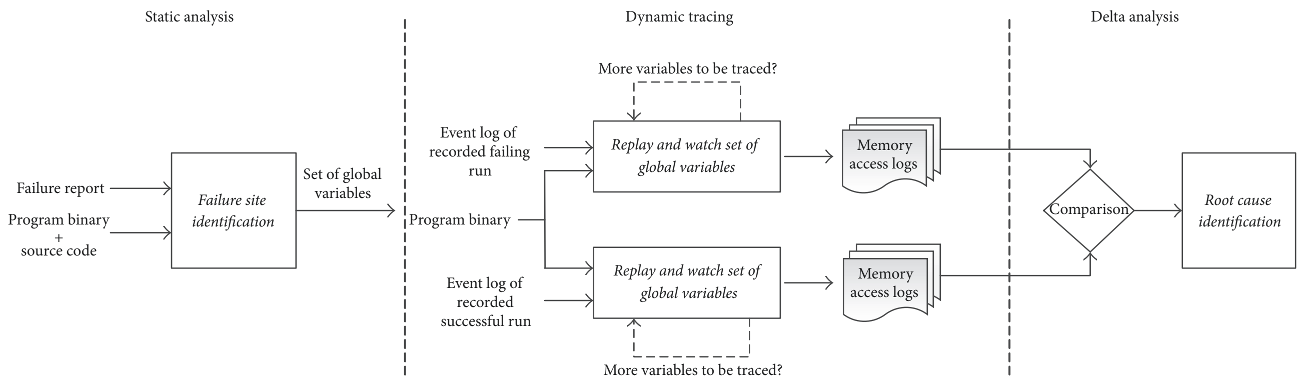
Static analysis
Dynamic tracing
Delta analysis
Failure report
Program binary + source code
Failure site identification
Set of global variables
Event log of recorded failing run
Program binary
Event log of recorded successful run
More variables to be traced?
Replay and watch set of global variables
Replay and watch set of global variables
More variables to be traced?
Memory access logs
Memory access logs
Comparison
Root cause identification

Figure 4: Replay analysis for diagnosing concurrency bugs.

event-handler for the signal handler, interrupt handler, and other callback functions that are asynchronously invoked in response to some events.) Such data may have an inconsistent state because it can be changed by both the main code and event-handler in a nondeterministic manner and can cause a program to fail unexpectedly. Our approach to diagnosing such failures is based on the record and replay system presented in Section 2, so the debug process can be started as soon as a buggy execution has been captured and its event log saved. Since the recording overhead is small enough to avoid any probe effect, it can be assumed that the recorded execution is identical to the original execution and will follow the same memory access order during replay. Therefore, we can dynamically track the program execution during replay to find out any memory access violations that have occurred during recording.

The proposed replay analysis works in three phases: static analysis, dynamic tracing, and delta analysis, as shown in Figure 4. In the first phase, we take as input a failure report, the program's source code, and the binary executable. Using these inputs, we identify the location in the program where the failure occurs. We then perform a static analysis of the program binary to extract the addresses of the global variables that are accessed within the identified scope. In the second phase, dynamic tracing phase, we replay the failing execution deterministically and insert hardware watchpoints on the addresses obtained from static analysis phase to log memory access to these locations. The same work is performed for a prerecorded successful execution of the same program. In the final phase, the delta analysis phase, we compare the memory access logs for each variable obtained from the failing execution to those in a successful execution in order to isolate the failure-inducing memory pattern.

3.1. Static Analysis. Many concurrency bug detection methods are proposed for shared memory parallel programs. These methods tend to trace memory access to all the shared memory locations in a program to detect possible concurrency bugs. Unlike concurrency bug detection, our proposed system aims to diagnose a concurrency bug that has caused a given program execution to fail. There may be a number of shared memory accesses in the program which are not related to a given failure. Therefore, it is redundant to track all the shared memory locations in a program while debugging. Hence, our first goal is to reduce the scope of shared memory access that might be involved in inducing a given failure. To do this, we make use of an important characteristic of bugs, observed by researchers [17–19], that concurrency bugs just like sequential bugs manifest themselves as common software failures, such as an incorrect output, an assertion violation, file corruption, and memory segmentation fault. Therefore, similar to sequential bugs, it is possible for developers to relate the concurrency bug failure to a specific section in the program source code, for example, a function. In the case of errors like incorrect output, assertion failure, or display of specific error messages (inserted by the developers or library), it is straightforward for the developers to locate the function in which the failure has appeared. However, in the case of other failures, such as a memory error, we make use of core dump. The core dump is typically generated as a by-product of a failed program execution. We load it in a debugger, for example, GDB, to obtain a call stack which helps us to identify the function in which a memory failure has been encountered.

Keeping in view the short propagation distance heuristic of a bug, we believe that the shared global variable involved in the concurrency bug must be accessed within that function. Therefore, we limit our scope of tracing memory access to only those global variables accessed in the identified function. We use the debugging symbols embedded into the program's binary to disassemble the target function. We extract the addresses of all global memory locations accessed within that function, eliminating all the memory access operations to the function's stack and also those referring to the read-only data section, as this access is not involved in concurrency bugs.

The output of the static analysis phase is a set of global variables that are accessed within the failure scope, and at least one of these is involved in the concurrency bug that we validate through dynamic tracing during replay.

3.2. Dynamic Tracing. To trace the access to shared memory locations, existing analysis methods typically rely on the use of heavyweight dynamic binary instrumentation tools such as Valgrind [20], DynamoRIO [21], and PIN [22]. Such tools

```
Input: Program executable ex, Set of global variables to trace SGV, Logs of a recorded execution log
Output: Memory access log: {op, val, PC} for each variable in SGV
(1)  while {SGV} ≠ $\phi$
(2)      $\{v_1, v_2, v_3, v_4\} \leftarrow$ Select(SGV)
(3)      $\{wp_1, wp_2, wp_3, wp_4\} \leftarrow$ SetHWwatchpoint($v_1, v_2, v_3, v_4$)
(4)      StartReplay(ex, log)
(5)      while (replay)
(6)          if wp $\in \{wp_1, wp_2, wp_3, wp_4\}$ is hit
(7)              addr ← ReadAddress(wp)
(8)              {op, val, PC} ← LogMemoryAccess(addr)
(9)          end if
(10)     end while
(11) end while
```

Algorithm 1: Dynamic tracing.

work by instrumenting every memory access instruction in the program binary at runtime. Since the instrumented code is executed at every memory access, it results in substantial slowdown [23]. Although the overhead during the replay should not be a big problem, the memory consumption by the instrumentation framework is not feasible for debugging programs on embedded platforms that have limited resources. The availability of the instrumentation tool for various platforms is also an issue.

Most importantly, since we aim to trace memory access for a limited subset of global variables obtained during the static analysis phase, we want to avoid the inherent expense of instrumenting memory access to every shared memory location as it is redundant to the bug diagnosis process. In the current work, rather than using dynamic binary instrumentation, we employ an alternate approach in which we use the processor's hardware watchpoint registers to monitor the access to the desired memory addresses. The watchpoint registers are used to stop a target program automatically and temporarily upon read/write operations to a specified memory address. These registers are often used in debuggers as data breakpoints. The main benefit of using hardware watchpoint registers is that they can be used to monitor the access to a memory location without any runtime overhead [24]. However, there are a restricted number of watchpoint registers available in a processor. In the ARM Cortex-A15 processor used in this research work, there are four hardware watchpoints available, which implies that we can track only four memory locations during replay analysis. In order to make use of the available number of watchpoint registers, we use a cyclic approach. We replay the recorded failed execution of the target program in a cyclic manner, and during each iteration, we trace the memory access of four global variables out of the entire list obtained from the static analysis phase. The replay process is repeated for four new variables until all the desired variables have been traced. Typically, there are hundreds of shared memory locations in a medium-sized program. By limiting the scope of the global variables to the failure site, a function, this number is typically reduced to a few tens. Hence, the overhead of repeating the replay cycle is comparable to the instrumentation slowdown, which can be as much as 20x for basic-block instrumentation without optimization [22].

To trace a given memory address, we program a watchpoint value (DBGWVR) and control (DBGWCR) register pair using Linux ptrace ability to read from/write to processor's registers. The DBGWVR holds the data address value used for watchpoint matching. The load/store access control field in DBGWCR enables the watchpoint matching conditional on the type of access being made [25]. Since we need to track every load and store operation to a given global variable, we set the watchpoint to be enabled for both types of access. When a watchpoint is hit, we log the memory access operation (read/write) on the variable performed by the current instruction, the updated value of the memory address, and the current program counter. The program counter is mapped to the statement that accesses the memory location using the symbolic debug information. This information is necessary to find out if the current memory access is performed by an event-handler or the main code.

The entire process of dynamic tracing is automatic and can be described by Algorithm 1.

3.3. Delta Analysis.

When a failure is encountered in a production run or during testing, the first step usually performed by the developers is to determine whether the root cause is simple re-execution. If the failure in the initial run is caused by a concurrency bug, then it is most likely to disappear when the test case is repeated. Thus, in the case of a concurrency bug, the developers have at least one passing execution of the same program with the same set of inputs.

In the domain of sequential errors, delta analysis is often used to compare the execution paths and variable values in two executions of a program to isolate faulty code regions and incorrect variable values. In the case of a concurrency bug, the failure is caused by conflicting memory access. Therefore, we can reach the root cause of the failure by comparing the memory access patterns of the global variables in failing and successful executions.

Two types of concurrency bugs are common in Linux programs considered in this research work: data races and atomicity violations. A data race occurs when a global variable is accessed by an event-handler and the main code in an unsynchronized way, and at least one of those access operations is a memory write operation. An atomicity violation is said to occur when the desired serializability among consecutive accesses of a shared memory location in the main code is violated by access to the same memory location in an asynchronously invoked event-handler. These bugs can be detected through special combinations of memory access operations that signify a data race or an atomicity violation. Specifically, we consider the standard data race patterns, that is, RW, WR, WW, and atomicity violation patterns, that is, RWR, RWW, WWR, and WRW found in multithreaded programs as conflicting memory access patterns for other types of programs that use event-handlers. Many such patterns may appear during the entire execution of a program and are not harmful to it; for example, it has been shown that only about 10% of real data races can result in software failures [26]. Reporting such data races usually just raises false alarms. Therefore, to identify the root cause of a given failure, we search the patterns mentioned above in the memory access logs of the failing run and match them to similar patterns in the successful run. Any conflicting memory access pattern that is present exclusively in the failing execution log is, therefore, the cause of a given failure.

3.4. Debug Case Studies. Concurrency bugs have been well-studied in the domain of thread-based programs, and therefore a number of bug databases are available to validate new algorithms. Unfortunately, no such bug database is available for sequential or event-based programs. We evaluated our proposed model of diagnosing concurrency bugs on a few real bugs caused by data races in concurrent signal-handlers reported in [14, 27] as well as some other programs. Here, we discuss only three case studies, but we believe that these results are representative for the large domain of concurrency bugs in Linux programs.

Bash 3.0. In Bash 3.0 when the terminal is closed, an event-handler is invoked to save the Bash history to a file. However, if the terminal is closed as soon as only one command is added to the history, it may not be saved into the file. The function used to save history is `maybe_append_history()`. A static analysis of this function reveals three shared global variables. These variables were traced during the replay of a failed and successful execution, and we found an atomicity violation pattern RRW in the variable `history_lines_this_session` in the failing execution. The situation arises when the first line from a new Bash session is added to the history, and the variable `history_lines_this_session` is used to keep the number of lines that Bash added to the current history session incremented from 0 to 1. This increment operation is assumed to be atomic, but it is actually not, and the event-handler interrupts this operation. Since the value of the variable is still zero, the event-handler assumes that there is nothing to be saved and does not write to the history file.

Lynx 2.8.7.2. In Lynx, the web link occasionally is not highlighted correctly in the text browser window. We statically analyzed the `LYhighlight()` function which returned addresses to 14 shared global variables. When tracing memory access to these variables during the replay of failing execution, an RWR atomicity violation pattern was found in the memory access log of the "`LYcols`" variable. The event-handler responsible was `size_change()`. We had to iterate the replay cycle four times to trace all the 14 shared memory variables, so the overhead of the replay cycle was 4x.

Ed 1.5. In Ed, when displaying a range of lines, the number of columns printed per line can be erroneous. The static analysis phase revealed three global variables accessed within the failure site `display_lines()`. Tracing these variables during the replay of the failed execution revealed an RW data race pattern in the "`window_columns_`" variable.

Bouncer 1.0. Bouncer is a small event-driven game [28] which uses two event-handlers, one to handle asynchronous user inputs from the terminal and the other to process a timer interrupt to control the speed of the animation. The program exhibits an assertion failure in the function `update_from_input()` if the number of speed increase/decrease requests from the user is not equal to the total timer adjustments performed. We recorded failing execution and traced the global variables accessed by the said function during the replay phase. We found that the failure was triggered because of an atomicity violation pattern RWR on the variable, `is_changed` shared between the timer event-handler and the function `update_from_input()`.

Concurrency bug failures are hard to reproduce and debug. In practice, the process of debugging such failures cannot be fully automated, and the involvement of developers remains essential in digging out the root cause. The case studies presented above support the same fact. If the program execution deviates from its intended behavior, we require the developers to identify that deviation and relate it to a specific code section that failed to meet the intended behavior. The rest of the process can be handled automatically and more efficiently by eliminating the debug effort which is redundant to a given failure scope.

We also observed that in the majority of the concurrency bugs, the failure occurs in the same function in which a memory access violation has taken place and hence it is possible to find the suspicious variable in the same function. For the remaining, we can find the suspicious variable by going upwards to the next level in the function call tree.

4. Related Work

4.1. Record/Replay Systems. Record/replay systems are implemented at different abstract levels of a computer system [29], for example, library-level, OS-level, virtual machine (VM) level, and architecture level. Library-level methods [5, 6] generate low-recording overheads, but they are not able to handle low-level asynchronous events correctly and also lack transparency. VM level methods such as [30] record and replay low-level events of a VM, but they incur huge overheads. Architecture level approaches [31, 32] also guarantee determinism at a low-level, but the implementation cost regarding design and verification is very high.

In the current work, we implemented record and replay at the OS-level. In doing so, we can faithfully reproduce the events at low-level while providing transparency to user-level programs. Although the recording overhead can be larger as compared to higher abstract level methods, it can be considered negligible as long as it does not perturb the natural execution of the application in the production system.

Flashback [33] also adopts an OS-level approach that records the target process through checkpoints. The checkpoints capture the in-memory execution state of a target process at a certain time using a shadow process. To enable debugging during the replay of an application, Flashback requires implementation of the checkpoint discard and replay primitives in a debugger such as GDB. Contrary to this, in our case, a standard debugger can be attached to an application being replayed without any modification to the application or debugger.

Scribe [13] also provides an execution replay mechanism at the OS-level. Our process of recording and replaying system calls is somewhat similar to the Scribe; however, during replay, Scribe re-executes the system calls. The purpose is to keep the application live during replay. This is helpful for fault-tolerant applications, where failing execution can be replaced by its replayed replica to continue the execution. Another critical difference is handling of asynchronous events like signals. In Scribe, the delivery of a signal to a process being recorded is deferred until a sync point so that its timing is deterministic. This approach can theoretically affect the program's correctness. For many interactive and time-sensitive programs, we cannot afford any perturbation of signal semantics during recording. We record the exact timing of signals through the instruction address in the program counter and the number of instructions accumulated in a PMU counter and inject it at the exact same instruction in the process during replay.

4.2. Diagnosing Concurrency Bugs. According to a study on the characteristics of bugs by Sahoo et al. [34], a large percentage of bugs found in software are of a deterministic nature (82%), occurring mostly because of incorrect inputs. The remaining bugs, which constitute a relatively small fraction, are nondeterministic such as concurrency bugs and are difficult to debug. Concurrency bugs are generally associated with multithreaded programs, and researchers have developed a variety of techniques to detect and diagnose such bugs for multithreaded programs. However, such bugs also exist in sequential, interrupt-driven, and event-based programs.

In [35], Regehr suggested random testing of interrupt-driven applications for exposing data races and other bugs in such programs. To randomly test an interrupt-driven application, a sequence of interrupts firing at specified times is generated. Next, the application is executed with interrupts arriving according to the sequence and observed for signs of malfunction.

Ronsse et al. [14] presented a method for detecting data races in sequential programs by adapting their existing data race detector for multithreaded programs. They use their dynamic instrumentation framework DIOTA to instrument and record all memory operations during runtime and to perform race detection during replayed execution. Their existing framework does not support recording of inputs from outside a process, and it is assumed that these inputs will be fixed. In comparison, our proposed record and replay system can provide feedback for all the inputs during the replayed execution which were intercepted through system calls and therefore guarantees the faithful re-execution of a program.

In [16], researchers present a dynamic race detection algorithm to be used with the causality model of event-driven web applications. On the other hand, Safi et al. [36] propose a static analysis method to detect race conditions in event-based systems to guarantee more code coverage and completeness. In the current work, we take advantage of both static analysis and dynamic analysis methods to carefully isolate the root cause of a concurrency failure in a specific execution.

5. Discussion

In this section, we discuss some implications and limitations of our work, along with some remaining open questions.

Applicability to Other Platforms. Our OS-based replay approach is based on the hypothesis that a computer program's nondeterministic behavior can be fully captured at the operating system level and reproduced with instruction-accurate fidelity by using some hardware support available on modern processors. To evaluate this hypothesis, we implemented a prototype in a commodity operating system, Linux, for an ARM platform. However, we expect that most of the features of our prototype can be easily ported to other commodity operating systems running on commodity hardware. For example, our approach of signal replay can also be achieved through the breakpoint registers (DR0–DR3) and the predefined performance event, "Instructions Retired," in the x86 architecture [37].

Implications of Memory Consistency Model. Consideration of memory consistency model is important for any multiprocessor deterministic replay system. ARMv7 has a weakly ordered memory model, meaning that the order of memory access operations performed by a CPU can be different from the order specified by the program. Our current prototype is uniprocessor, and we assume that the single CPU is aware of its own ordering and therefore can ensure that the data dependence is respected [38]. However, memory ordering will be another source of nondeterminism when extending our implementation to a multicore/multiprocessor environment. To handle this, we also need to record the order of shared memory operations among different cores.

Dynamically Allocated Variables. In the current concurrency bug diagnosis framework, the use of hardware watchpoints limits the system to track data races in globally defined variables only. However, data races are also possible on dynamically allocated variables, such as those defined on heap. In practice, it is not possible to trace the dynamically allocated variables without runtime instrumentation. Concerning this, the authors in [39] present an interesting approach in which an entire heap can be watched using software watchpoints inserted through dynamic instrumentation. This approach can be integrated with our replay analysis framework to increase the scope of target bugs.

Multivariable Access Violations. According to a study [40], single-variable concurrency bugs constitute two-thirds of the overall non-deadlock concurrency bugs. To target this large fraction, we therefore followed a rather simplistic but effective approach that considers only single-variable data race and atomicity violations. However, multivariable access violations also exist. For detecting such violations, we must be able to infer the relationships between multiple variables to complement their dynamic tracing. We leave this improvement for our future work.

6. Conclusion

Reproducing a nondeterministic failure for bug diagnosis is a key challenge. To address this challenge, we have presented a light-weight and transparent OS-level record and replay system, which can be deployed to production systems. It can faithfully reproduce both synchronous and asynchronous events occurring in programs such as system calls, message passing, nonblocking I/O, and signals. During the replay of a program, a standard debugger can be attached to it to enable cyclic debugging without any modifications to the program or debugger.

We also presented a method for diagnosing concurrency bug failures which is based on the proposed record and replay system. Given a failure to track down bugs, developers can collect memory access logs for a set of global variables during the replay of failing and passing execution and then compare them to identify any memory access violations causing the failure. Our experience with some real programs shows that this usage model can be very beneficial in locating the root causes of failures related to concurrency bugs.

Our current work is so far limited to debugging of sequential and event-based programs. In the future, we aim to extend it for multithreaded programs so that we can capture and reproduce the exact thread interleaving order in a failing execution and then identify the conflicting memory access operations that lead to program failure.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by the Ministry of Science and ICT (MSIT), Korea, under the programs [R0114-16-0046, Software Black Box for Highly Dependable Computing] and [2016-0-00023, National Program for Excellence in SW] supervised by the Institute for Information & Communications Technology Promotion (IITP), Korea.

References

[1] P. Godefroid and N. Nagappan, "Concurrency at Microsoft: an exploratory survey," in *Proceedings of the CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[2] K. Veeraraghavan, D. Lee, B. Wester et al., "Doubleplay: parallelizing sequential logging and replay," *ACM Transactions on Computer Systems*, vol. 30, no. 1, article 3, 2012.

[3] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay for multiprocessor virtual machines," in *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE '08*, pp. 121–130, March 2008.

[4] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th International Symposium on Code Generation and Optimization, CGO '10*, Toronto, Ontario, Canada, IEEE, April 2010.

[5] D. M. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," in *Proceedings of the USENIX Annual Technical Conference*, pp. 289–300, 2006.

[6] Z. Guo, X. Liu, W. Lin, and Z. Zhang, "Towards pragmatic library-based replay," Tech. Rep. MSR-TR-2008-02, Microsoft Research, 2008.

[7] D. R. Hower and M. D. Hill, "Rerun: exploiting episodes for lightweight memory race recording," in *Proceedings of the 35th International Symposium on Computer Architecture ISCA '08*, pp. 265–276, June 2008.

[8] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: recording and deterministically replaying shared-memory multiprocessor execution efficiently," in *Proceedings of the 35th International Symposium on Computer Architecture ISCA '08*, pp. 289–300, 2008.

[9] Lynx text web browser, 2017, http://lynx.invisible-island.net/.

[10] J. Kneschke, Lighttpd-Fly Light, 2017, https://www.lighttpd.net/.

[11] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, p. 2, 2008.

[12] Y. Saito, "Jockey: a user-space library for record-replay debugging," in *Proceedings of the 6th International Symposium on Automated and Analysis-Driven Debugging AADEBUG '05*, 2005.

[13] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," *Sigmetrics Performance Evaluation Review*, vol. 38, pp. 155–166, 2010.

[14] M. Ronsse, J. Maebe, and K. De Bosschere, "Detecting Data Races in Sequential Programs with DIOTA," in *Euro-Par 2004 Parallel Processing*, vol. 3149 of *Lecture Notes in Computer Science*, pp. 82–89, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[15] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue, "An effective method to control interrupt handler for data race detection," in *Proceedings of the 5th Workshop on Automation of Software Test, AST '10, in Conjunction with the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010*, May 2010.

[16] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 151–166, 2013.

[17] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," *Operating Systems Review (SIGOPS)*, pp. 235–248, 2005.

[18] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of Linux Kernel Behavior under Errors," in *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, pp. 459–468, June 2003.

[19] W. Zhang, J. Lim, R. Olichandran et al., "ConSeq: detecting concurrency bugs through sequential errors," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 251–264, 2012.

[20] N. Nethercote and J. Seward, "Valgrind: a program supervision framework," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 47–69, 2003.

[21] D. Bruening, *Efficient, transparent, and comprehensive runtime code manipulation [Ph.D. thesis]*, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[22] C.-K. Luk, R. Cohn, R. Muth et al., "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[23] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari, "Analyzing dynamic binary instrumentation overhead," in *Proceedings of the WBIA Workshop at ASPLOS*, 2006.

[24] P. Krishnan, "Hardware Breakpoint (or watchpoint) usage in Linux Kernel," in *Proceedings of the in Linux Symposium*, p. 149, 2009.

[25] "Cortex™-A15 MPCore™ Technical Reference Manual," 2017, http://infocenter.arm.com.

[26] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 22–31, 2007.

[27] T. Tahara, K. Gondow, and S. Ohsuga, "DRACULA: Detector of data races in signals handlers," in *Proceedings of the 15th Asia-Pacific Software Engineering Conference, APSEC '08*, pp. 17–24, 2008.

[28] Bouncer, An event-driven game, 2018, https://github.com/JLee80/An-event-driven-game.

[29] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen, "Deterministic replay: a survey," *ACM Computing Surveys*, vol. 48, no. 2, article 17, 2015.

[30] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation, OSDI '02*, pp. 211–224, December 2002.

[31] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: continuously recording program execution for deterministic replay debugging," in *Proceedings of the 32nd Interntional Symposium on Computer Architecture, ISCA '05*, pp. 284–295, June 2005.

[32] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 122–133, June 2003.

[33] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging," in *Proceedings of the in USENIX Annual Technical Conference, General Track*, pp. 29–44, 2004.

[34] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, vol. 1, pp. 485–494, Cape Town, South Africa, May 2010.

[35] J. Regehr, "Random testing of interrupt-driven software," in *Proceedings of the 5th ACM international conference on Embedded software*, 2005.

[36] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic, "Detecting event anomalies in event-based systems," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '15*, 2015.

[37] "Intel® 64 and IA-32 Architectures Developer's Manual," vol. 3b, 2018, https://software.intel.com.

[38] P. E. McKenney, "Memory ordering in modern microprocessors, part I," *Linux Journal*, vol. 2005, p. 2, 2005.

[39] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong, "How to do a million watchpoints: Efficient Debugging using dynamic instrumentation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 4959, pp. 147–162, 2008.

[40] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes - A comprehensive study on real world concurrency bug characteristics," *ACM SIGPLAN Notices*, vol. 43, no. 3, pp. 329–339, 2008.