TORNADO: A Novel Input Replay Tool*

Frank Cornelis Michiel Ronsse Koen De Bosschere
Department of Electronics and Information Systems
Ghent University
{fcorneli,ronsse,kdb}@elis.ugent.be

Abstract *This paper presents TORNADO, a unique input replay tool based on system call replay. It is capable of tracing program executions with an acceptable overhead of less than a factor 2. It is fully operational for the Linux operating system. The tool requires no instrumentation, recompilation or relinking of the applications. This paper also presents a new technique capable of tracing write operations performed by the OS kernel to user space.*

1 Introduction

Although a number of advanced programming environments, formal methods and design methodologies for developing reliable software are emerging, one notices that a big part of the development time is still spent while debugging and testing applications.

One of the reasons is that debugging and testing is still a human activity which is hard to automate. Most programmers still use classical debuggers which try to help with the debugging process by means of watchpoints and breakpoints. But these classical debugging aids become useless in the presence of modern software that is highly multi-threaded, distributed, and interactive (e.g., a web browser), since no two executions can be guaranteed to be identical.

The nondeterminism in modern software has a number of causes:

- the input used by a program (e.g., data read from disk, a network packet, mouse movement, ...) cannot always be guaranteed to reoccur during a re-execution.
- shared variables used by multi-threaded programs introduce so-called race conditions [4].

In order to enable programmers to use the classical cyclic debugging techniques for nondeterministic programs, so-called *record/replay* methods and tools have been developed. Such a tool *records* information about the nondeterminism that is encountered during a program execution to a *trace file*. During subsequent replayed executions the information from this file is consulted in order to guarantee a faithful *replay* of the previously recorded execution. As these re-executions are equivalent to the recorded execution, cyclic debugging is now possible again without the risk for probe effects [3].

The problem of detecting race conditions and replaying data races has been studied extensively in the past, although most research was just focusing on the detection of race conditions. Numerous solutions have been proposed, ranging from enforcing an order on all memory operations during replay [4] over synchronization replay [5] to logical thread scheduling replay [2].

This paper will focus solely on the nondeterminism caused by the non-repeatable input consumed by the program. This input is normally provided to an application by the operating system, e.g., an application uses a *system call* to ask the kernel for an I/O operation. Input obtained without using the kernel, e.g., using DMA directly to user memory or user memory mapped I/O will not be considered.

2 System Call Replay

The general idea behind system call replay is actually quite easy to grasp: during an execution we intercept all system calls executed, recording the user memory changes. We replay these system calls by re-applying these changes during a replayed execution. However, a number of problems arise with this simple approach. These problems, along with our solutions to them, will be discussed in the following sections.

*Available at: http://www.elis.ugent.be/~fcorneli/

2.1 Recording the changes in user memory

Most system calls write to well known memory regions, and it is easy to detect these regions by simply intercepting the system calls as most system calls use arguments that have a well defined purpose, e.g., the second argument of `read()` denotes a buffer. Things are far more complicated for system calls for which the used memory addresses are not easy to detect if one simply intercepts the system call. The most notable example is `ioctl()`, as used by a lot of device drivers. In our replay framework we developed a technique to trace every store operation performed by the kernel to user memory to be able to even trace system calls like `ioctl()`.

The Linux kernel forces code to use certain memory access primitives (e.g., `__put_user_asm`) for writing to user memory. Thus instrumenting these memory operations suffices to be able to trace every memory write operation from kernel to user memory. This only requires the recompilation of the kernel and kernel modules against the instrumented versions of these write primitives. No recompilation whatsoever is needed for the user space programs and libraries.

We instrumented the write primitives in such a way that they —besides performing the write operation— add an entry to a linked list for each store operation (see Figure 1). This linked list has been called the `memdiff`-list. Each entry contains the start address of the memory area written to and the number of bytes written.

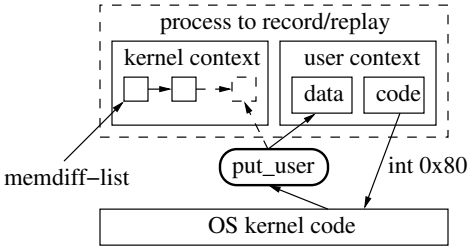

Figure 1: *Instrumentation of write operations.*

When we need to trace a system call that writes to difficult discoverable memory areas we reset the `memdiff`-list and activate the trace functionality of the kernel write primitives. During the system call the list gets stuffed with information on the primitive write operations. Afterwards the tracer can obtain this memory differences record from the kernel by means of a home-brew extension on the `ptrace()` system call.

2.2 Replaying the system calls

During the replay phase, the memory and return values are simply read from disk and fed to the application, with or without re-executing the actual system call.

It is not always desirable to replay all system calls because it might lead to inconsistencies within the system.

Not replaying any system call during a replay is of course no option either. Without system calls, the program no longer produces any output, making such a replay system rather useless for debugging purposes (black-hole effect).

Some system calls also need to be re-executed because of their side effects on the kernel context of the user program. If we neglect the re-execution of certain system calls the subset of the kernel context responsible for running the user program will become irreversibly inconsistent with the user context. This can lead to unrecoverable errors during the replay phase. Examples of such system calls are memory management calls like `mmap()` and `munmap()`.

To handle all the differences in system call replay behavior a fine-grained partial I/O subsystem has been implemented in our replay framework which can be tuned by means of XML configuration files.

A second problem concerns the so-called kernel context state of applications. The kernel context is used to store information about the application, e.g., the files it has opened. During a replayed execution, the kernel context will not necessarily be the same as during the original execution, possible leading to errors. To deal with this our replay system maps between the kernel context objects that are prone to this phenomenon (e.g., file descriptor mapping).

3 Experimental Results

To measure the I/O performance of OS primitives we did use (some of) the HBench-OS benchmark tests [1]. The results of these benchmark tests have been summarized in Table 1.

In this table the slowdown between a non-traced and a traced execution is shown. The 'Trace size/iteration' column denotes the average bytes of data needed to allow for a deterministic replay.

When studying these numbers one notices that the replay tool provokes the worst slowdowns on the low-level OS primitives (the `lat_syscall` micro-benchmarks). This is mainly caused by the costly

Micro-benchmark	Slowdown	Trace size/iteration (bytes)
`lat_syscall write`	8.36	6.08
`lat_syscall getpid`	8.60	5.03
`lat_mmap`	7.41	1048.58
`lat_mmap` (re-execution)	5.00	20.07
`lat_udp`	2.15	42.09
`lat_tcp`	1.97	39.01

Table 1: *HBench-OS 1.0 results for tracing operating system primitives.*

Application	Execution time (s)		Slow-	Trace size		System calls	
	Untraced	Traced	down	MiB	MiB/s	#	#/s
`gcc libreplay.c`	1.02	1.12	1.10	1.15	1.03	2162	1930.36
`gzip linux.tar`	20.38	24.55	1.20	170.10	6.93	7907	322.08
`gunzip linux.tgz`	3.66	8.46	2.31	38.28	4.52	6689	790.66
`ls -R /lib`	0.37	0.73	1.97	5.11	7.00	8533	11689.04
`ls -R /lib` (re-exec)	0.37	0.67	1.81	3.36	4.49	8533	12735.82

Table 2: *Benchmark results for tracing some real world (non-interactive) applications.*

context switches that are needed between the process and the tracer which live in separate memory spaces to reduce the probe effect. Of course no real world application exhibits such a weird behavior as these micro-benchmarks do (our worst case scenario).

The slowdown is of course also proportional to the trace data bandwidth required for a deterministic replay (e.g., the `mmap` benchmark with/without system call re-execution).

We also benchmarked some standard non-interactive applications which are closer to realistic (interactive) applications than the micro-benchmarks. The results of these tests have been summarized in Table 2. When taking a closer look at these numbers one notices that the slowdown for most benchmarks is lower than a factor of 2. This is rather good especially when looking at the size of the trace data that is required to allow for a deterministic replay.

4 Conclusions & Future Work

TORNADO is a unique system call replay tool that is fully operational for the Linux platform. It is capable of correctly replaying real world applications with nondeterministic input behavior. The overhead during the record phase is limited to less than a factor 2 for realistic programs.

Despite our effort more work should be done in this area to eliminate the drawbacks present record/replay tools are subject to, being overhead in both space and time. Recent tests with a new prototype replay tool using a complete in-kernel replay technique resulted in an overhead a factor 100 smaller than the current tool and thus look promising.

References

[1] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[2] J. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceeding of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, August 1998.

[3] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

[4] R. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, May 1993.

[5] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.