Re-Animator: Versatile High-Fidelity System-Call Tracing and Replaying

Ibrahim Umit Akgun
Department of Computer Science
Stony Brook University

Research Proficiency Exam

Technical report FSL-19-02

May 17, 2019

Contents

1 Introduction **1**

2 Design **3**
2.1 Goals 3
2.1.1 Fidelity 3
2.1.2 Minimize Overhead 4
2.1.3 Scalable and Verifiable 4
2.1.4 Portability 4
2.1.5 Ease of Use and Extensibility 4
2.2 Fidelity 4
2.2.1 RA-Strace 5
2.2.2 RA-LTTng 6
2.3 Low-Overhead and Accurate 7
2.3.1 RA-Strace 8
2.3.2 RA-LTTng 8
2.4 Scalable and Verifiable 8
2.4.1 Re-Animator Replayer 9
2.4.2 Verifiable 9
2.4.3 Concurrent Lock-Free Design 9
2.4.4 Supporting Multiple Processes 9
2.4.5 Simulated System Calls 10
2.5 Portable 10
2.5.1 Source Code Size 11

3 Evaluation **12**
3.1 Testbed 12
3.2 Benchmarks 12
3.2.1 Micro-Benchmarks 13
3.2.2 Macro-Benchmarks 13
3.2.3 Replaying Benchmarks 13
3.3 FIO Micro-Benchmark 13
3.4 LevelDB Macro-Benchmark 17
3.5 MySQL Macro-Benchmark 20
3.6 Trace Statistics 20

4 Related Work **21**
4.1 Ptrace 21
4.2 Shared-Library Interposition 21
4.3 In-Kernel Techniques 21
4.4 Replayer Fidelity 22
4.5 Scalability 22
4.6 Portable Trace Format 23

5 Conclusion and Future Work **24**
5.1 Future Work 24

Abstract

Modern applications are complex and difficult to understand. One approach to investigating their characteristics is system-call tracing: captured traces can be analyzed to gain insight, or replayed to reproduce and debug behavior in various settings. However, existing system-call tracing tools have several deficiencies: (1) they often do not capture all the information—such as raw data buffers—needed for full analysis; (2) they impose high overheads on running applications; (3) they are proprietary or use non-portable trace formats; and (4) they often do not include versatile and scalable analysis and replay tools.

We have developed two prototype system-call trace capture tools: one based on `ptrace` and the other on modern Linux tracepoints. Both capture as much information as possible, including complete data buffers, and produce trace files in a standard DataSeries format. We also developed a prototype replayer that focuses on system calls related to file-system state. We evaluated our system on long-running server applications such as key-value stores and databases. We found that `ptrace`-based tracers can impose an order of magnitude in overhead, but require no superuser privileges or kernel changes. In contrast, our tracepoints-based tracer requires some kernel changes to capture buffers and optimize trace writing, but it has an overhead of only 1.8–2.3$\times$ for macro-benchmark applications. The replayer verifies that its actions are correct, and faithfully reproduces the on-disk state generated by the original application.

Chapter 1

Introduction

Modern applications are becoming ever more intricate, often using 3rd-party libraries that add further complexity [38]. Operating systems are adding multiple layers of virtualization [10, 73] and deep I/O stacks for both networks and storage devices [11, 40, 72]. In addition, current storage systems employ space-saving optimizations, including compression, deduplication, and bit-pattern elimination [13, 27, 51, 53, 61, 75–77, 84, 86, 90]. The result is that applications interact with the rest of the system in complex and unpredictable ways, making it extremely difficult to understand and analyze their behavior.

System-call tracing is a time-honored, convenient technique for tracking an application's interaction with the OS: `ptrace` can be used on a live application to start capturing system-call events, which can then be converted for human consumption using tools such as `strace` [89]. Such traces can be replayed to reproduce a system's state, reproduce an application's behavior without the need to recreate its input conditions and rerun the application, explore how an application may behave under different system configurations (e.g., performance analysis or debugging), and even stress-test other components (e.g., the OS or storage system) [1–3, 8, 20, 34, 43, 44, 50, 67, 68, 71, 83, 85, 93, 94]. Traces can also be analyzed offline (e.g., using statistical or machine-learning methods) looking for patterns or anomalies that may indicate performance bottlenecks, security vulnerabilities, etc. [41, 52, 69, 70]. Lastly, historical traces can help understand the evolution of computing and applications over longer time periods. Such long-term traces become more useful to evaluate the effects of I/Os on modern devices that can wear out quickly (SSDs) or have poorly understood internal behavior (e.g., garbage collection in shingled drives) [19, 23, 39, 47, 55, 91, 92].

Existing system-call tracing approaches, however, are deficient in several ways: (1) They often do not capture all the information that is needed to reproduce exact system and storage state. For example, buffers passed to the `read` and `write` system calls are often not captured, or are truncated. (2) Tracing significantly slows down the traced applications and even the surrounding system. These overheads can be prohibitive in production environments. As a result, tracing is often avoided in mission-critical settings, and traces of long-running applications are rare. (3) Past traces have often used their own custom format; documentation was lacking or non-existent; and sometimes no software or tools were released to process, analyze, or replay the traces. Some traces (e.g., those from the Sprite project [66]) have been preserved but can no longer be read due to a lack of tools. (4) Some tracing tools (e.g., `strace` [89]) have output formats that are intended for human consumption and are not conducive to automated parsing and replay [33, 42, 88].

In this paper we make the following six contributions:

1. We have designed and developed two prototype system-call tracing systems. The first, based on `ptrace` and `strace`, can be used without superuser privileges. The second uses Linux tracepoints [24] and LTTng [25, 59] and requires superuser privileges, but offers much better performance.
2. Both tracing systems capture as many system calls as possible, including all their data buffers and arguments. The information is captured in raw form, thus avoiding unnecessary processing to display them

for human consumption (i.e., “pretty-print”).

3. Both systems write the traces in DataSeries [5] format, SNIA’s official format for I/O and other traces. DataSeries is a compact, efficient, and self-describing binary trace format. This approach automatically allows users to use existing DataSeries tools to inspect trace files, convert them to plain text or spreadsheet formats, repack and compress trace files to save space, subset them, and extract statistical information.
4. In extensive evaluations, we found that `strace` imposes a high overhead on running applications, as much as an order of magnitude; yet on average, our DataSeries-enabled `strace` performs at least 30% better. Our tracepoints-based tracing system, while requiring Linux kernel changes to capture data buffers, adds an overhead of only 1.8–2.3$\times$.
5. We developed a DataSeries replayer that can replay most system calls, including all those that relate to file systems, storage, or persistent state. The replayer is designed to execute system calls as faithfully and efficiently as possible, preserving event ordering, reproducing identical process and thread states, handling multiple threads, and being able to replay large traces (hundreds of GB).
6. All our code and tools for both tracing systems and for the replayer are planned for open-source release. In addition, we have written an extensive document detailing the precise DataSeries format of our system-call trace files to ensure that this knowledge is never lost; this document will also be released and archived formally at SNIA.

Chapter 2

Design

Re-Animator is designed to: (1) maximize the fidelity of capture and replay, (2) minimize overhead, (3) be scalable and verifiable, (4) be portable, and (5) be extensible and easy to use. In this section, we first justify these goals, and then explain how we accomplish them.

2.1 Goals

2.1.1 Fidelity

State-of-the-art techniques for recording and replaying system calls have focused primarily on executing captured system calls accurately during replay [6, 15, 38, 48, 62, 88, 95]. In this work, we consider three dimensions of replay: (1) timing, (2) dependencies between processes and threads, and (3) on-disk state.

Of these, timing is probably the easiest to handle; the tracer needs to record accurate timestamps, and the replayer needs to reproduce the timing as precisely as possible [6]. However, many researchers have chosen a simpler—and entirely defensible—option, which is to simply replay calls as fast as possible (AFAP). That choice makes sense because it imposes maximum stress on the system being tested, which is often the preferred approach when replay is used to test new systems. For that reason, although we record accurate timestamps, we use AFAP replay. Replay that accounts for "think time" remains as future work.

Dependencies in parallel applications are more challenging; replaying dependencies incorrectly can lead to unreasonable conclusions, and in most cases will produce incorrect results. Previous researchers have used experimental [62] or heuristic [88] techniques to extract internal dependencies. For the current version of Re-Animator, we have chosen a conservative approach similar to *hfplayer* [36]: if two requests overlap (as measured by their beginning and ending times) we assume that they can be issued in any order; if there is no such overlap then we preserve the ordering recorded in the trace file. We plan to incorporate techniques from [88] in the future.

Finally, most prior tracing and replay tools have chosen to discard the actual data that was transferred. That decision makes sense, because collecting that data slows tracing and consumes significant amounts of disk space. However, modern storage systems use advanced techniques—such as deduplication [57, 80], compression [16, 53], repeated bit-pattern elimination [77], log-structured merge trees [64, 74], and B^{ϵ}-trees [46]—for which the performance depends on data content. For that reason, we believe that it is important to have the correct—and verified—state on the file system and disk after executing captured system calls. We have therefore designed Re-Animator to support the efficient capture and replay not only of system calls and their parameters, but also buffer contents, so that we can accurately reproduce the final state generated by the original application. Re-Animator's replayer supports verification of both return values and buffer contents (see Section 2.4). We discuss the details of all these features in Section 2.2.

2.1.2 Minimize Overhead

Since the goal of tracing is to record realistic behavior, anything that affects the performance of the traced application is undesirable. With the exception of tools that capture network packets by sniffing, all tracing methods necessarily add some overhead, which of course should be minimized. Overhead is introduced in several ways: (1) as each system call is made, a trace record must be created; (2) any data associated with the system call (e.g., a path-name or a complete write buffer) must be captured; and (3) the trace information must be written to a stable storage device. To keep overheads low, some tracing systems such as dtrace [17], ktrace [30], and SysDIG [14]—all three of which we tested ourselves—drop events when the rate of system calls exceeds a memory-consumption threshold; this is clearly undesirable if high fidelity is to be achieved.

Re-Animator offers two tracing tools: the first is based on `strace`, which uses the `ptrace` facility to intercept system calls, and the second is based on LTTng [25, 59], a relatively new Linux facility designed for tracing kernel events, using tracepoints [24]. The `strace` approach is simpler for the user but has higher overhead (see Section 2.3).

2.1.3 Scalable and Verifiable

Like any application, tracing tools should be reliable and should avoid arbitrary limitations. In particular, it should be possible to trace large applications for significant time periods; that implies that traces must be captured directly to stable storage (as opposed to fast but small in-memory buffers). Moreover, tracing a multi-process or multi-threaded application should avoid introducing additional synchronization points that would affect the application's behavior. For example, one application thread should not be suspended while waiting for another thread's trace information to be written to a common trace file.

In addition, it must be possible to verify that a replayer has replayed a trace correctly. We use three verification methods that can be disabled if desired: (1) when a system call is issued, we ensure that it received the same return code (including error codes) that was captured at trace time; (2) for calls that return information, such as `stat` and `read`, we validate the buffer contents; and (3) after a replay completes, we separately compare the on-disk state with that produced by the original application.

2.1.4 Portability

Tools are only effective if they actually get used; to be usable, a tool must run in the desired environment. To enhance portability, we use the DataSeries trace format [5] and developed a common library that standardizes trace capture. (We will release all of our software publicly after publication of this paper.)

2.1.5 Ease of Use and Extensibility

Ease of use is also critical to the effectiveness of a tool. That goal encompasses user-interface design, flexibility, and power. In the simplest case, the user need only run an application under our version of `strace`; the replayer is equally easy to use. The LTTng version is somewhat more challenging, since it requires a kernel modification, but once that has been done, the capture and replay processes are equally simple. We have also designed the code so that it is easy to add support for new system calls as necessary.

2.2 Fidelity

In this section we describe how Re-Animator accurately captures system calls, including the data buffers' contents. We describe our `strace` and LTTng modifications.

2.2.1 RA-Strace

Figure 2.1 shows the flow for tracing and capturing calls in RA-Strace; green components denote our modifications or additions. The `strace` tool is built upon the `ptrace` facility, which (among other things) hooks into system-call entry and exit points and notifies the tracer of each such occurrence. The tracer can then use `ptrace` to examine arguments and results as necessary. However, the tracer is a separate process context, and `ptrace` requires a system call each time the target process's registers or memory are accessed. Thus, multiple context switches are needed for `strace` to gather system-call arguments and results; doing so introduces significant overheads.

However, `ptrace` also has an advantage: since the traced process is halted for each system call, the tracer can delay the traced process until the event has been written to disk. Thus, the `strace` approach ensures that no events are ever lost.

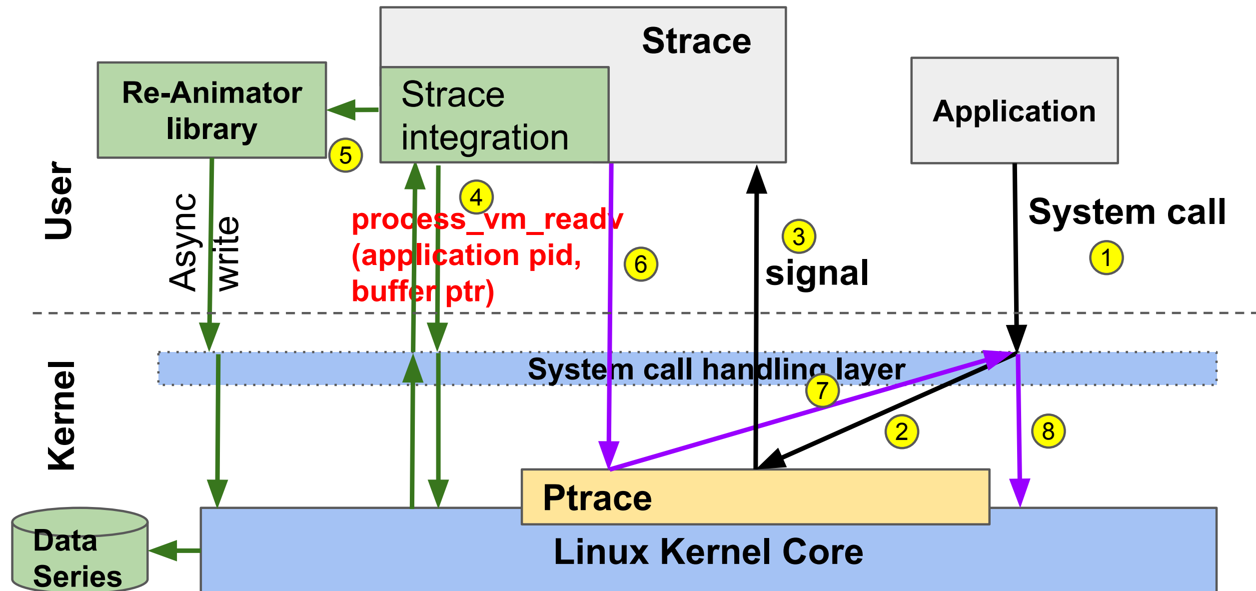


Figure 2.1: Strace architecture. Green boxes denote our additions or changes. After an application issues a system call (1), it is delivered to `ptrace` (2), which notifies `strace` (3). Strace uses `ptrace` calls (4) to collect information and write it using our library (5); it then uses `ptrace` (6) to free the process to run (7) and finally execute the system call (8).

For RA-Strace, we modified `strace` in two places: the entry and exit points for system calls. Ordering among threads is maintained by giving each system call a unique, monotonically increasing record ID. (In some exceptional situations, such as `readv`, `writev`, and `execve`, a single system call generates multiple trace records, in which case we assign the same ID to all; the DataSeries library disambiguates these cases.) Re-Animator collects common information for every system call, including the entry and exit times, return value, error code, process ID, and thread ID. Most calls are captured in the exit handler so that we can access the data returned by the call. However, non-returning calls such as `exit` and `execve` skip through the exit handler when they succeed, so Re-Animator records those calls at entry time (but still records failures when they occur).

Despite assigning unique IDs, there is a subtle problem with record ordering. In some cases, `clone` will yield to another, newly created thread *before* returning, so other `clone`d processes might be recorded before the first one returns, causing out-of-order trace records. We resolved this issue with an offline post-processing tool that reorders `clone` and `vfork` records in the DataSeries file as needed. This approach is

simpler and more robust than attempting on-the-fly reordering.

One of our most crucial design goals is to gather the data buffers passed to and from system calls. We do so for 38 system calls. Here, we consider `struct`s to be buffers: e.g., the full results of `getdents`, `stat`, and `ioctl` are captured. Although this maximalist approach is excellent from a scientific standpoint, it adds the challenge of efficient data collection. We capture the data using `process_vm_readv` and `ptrace`, plus some `strace` utility functions, as shown in Figure 2.1, step 4. After the trace information has been collected, RA-Strace uses our common library (RA-Lib, see Section 2.3) to efficiently write a record to the DataSeries file (Figure 2.1, step 5). We added a number of optimizations to our common library to ensure efficiency (see Section 2.3).

2.2.2 RA-LTTng

LTTng [59] is a extensible framework for Linux kernel tracing. Briefly, *tracepoints* [24] have been inserted in important functions, such as the system-call entry and exit handlers. When a tracepoint is activated, LTTng captures relevant information into a buffer that is shared with a user-level daemon, which then writes it to a file. To enable parallelism, the shared buffer is divided into *sub-buffers*, one per process, and the LTTng daemon uses user-space RCUs via `liburcu` [56] for lockless synchronization with the kernel. The data is written in the Common Trace Format (CTF) [58]; a tool called `babeltrace` can convert CTF to other formats (but it currently supports conversion only to human-readable text format).

Figure 2.2 shows the flow for tracing and capturing calls in LTTng; green components denote our modifications or additions. To make the system easier to use, we wrote a wrapper (Figure 2.2, step 1) that automates the tasks of starting the various LTTng components and the traced application.

The design of LTTng and CTF makes it challenging to capture large data buffers, such as when an application accesses megabytes in a single I/O, since the sub-buffers were designed to handle relatively small events. Instead, we capture data buffers directly to a secondary disk file using `vfs_write` (Figure 2.2, step 7). An advantage of using a separate file is that it can be placed on a different, larger storage medium if desired. We added a unique ID to each trace record and tag each captured buffer with the same ID, so that we can correlate them offline. To maximize parallelism, when a trace event is captured, we use a spinlock to assign a particular file offset to that event, and then write the data itself asynchronously. In the rest of this paper, we refer to this enhanced CTF format with secondary buffer-data files as *RA-CTF*. We describe the details of the asynchronous buffer writing process in Section 2.3.

We modified the pre-existing `babeltrace` tool to generate the DataSeries format [5], which allows us to group events on a per-thread basis, simplifying replay (CTF is purely sequential). The modified `babeltrace` tool also integrates the captured data buffers from the secondary file so that the DataSeries file produced is self-contained.

Because we write captured buffers asynchronously from multiple threads, they might be stored in a different order from that of events in the CTF file. Re-Animator tags the buffers with their size and unique record ID. When we convert CTF files to DataSeries, Re-Animator correlates the records with corresponding system-call events using unique IDs, and stores them in the correct order in the DataSeries file.

LTTng records the identity of traced process(es) in a hash table indexed by process ID; currently, it does not insert new PIDs or thread IDs into the table when `clone` is called. Rather than modifying `clone`, we changed the kernel to trace based on the Process Group ID (PGID), which does not change upon `clone`. Our wrapper sets the PGID before executing the target application and notifies the kernel. This technique worked for all of the applications we tested in this paper. (The LTTng developers are working on a proper solution to track which process should be traced.)

When we capture one of the 38 system calls that involve user data buffers, we first make a copy of that data (using `copy_from_user`) so that the user process can continue while we invoke `vfs_write`. Space for that copy is allocated using `kmalloc` or, if that fails due to the size of the allocation, `vmalloc` (in

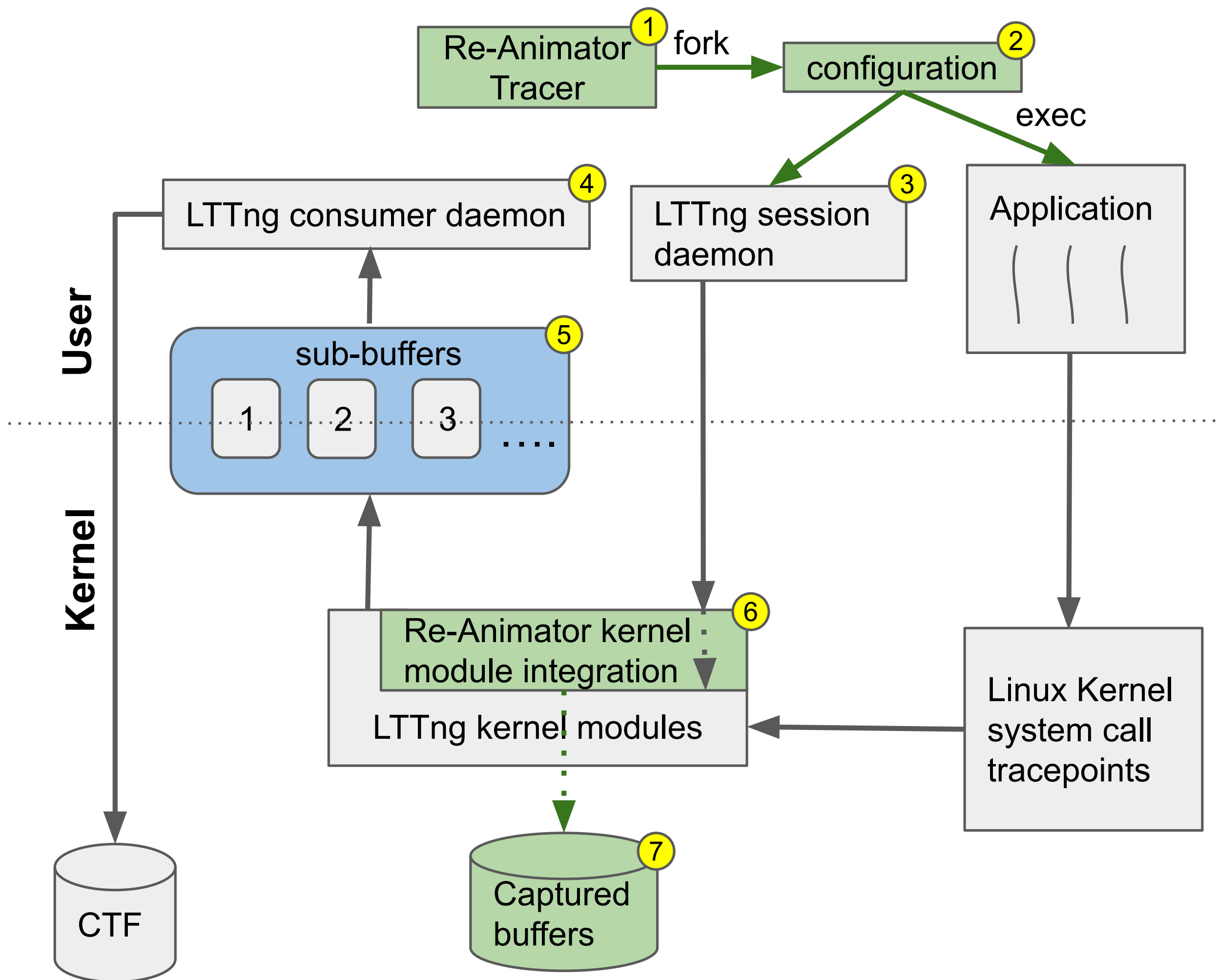


Figure 2.2: LTTng architecture using Linux kernel tracepoints. Green boxes denote our additions or changes. Our wrapper (1) launches the LTTng configurator (2), which invokes an LTTng session daemon (3) to control the operation and the consumer daemon (4) to collect events. LTTng tracepoints place events into sub-buffers (5) and invoke Re-Animator, which collects data buffers and writes them to a separate disk file (7).

our kernel context, using swappable kernel memory is permissible). Preferring `kmalloc` saves time in the common case.

LTTng signals the consumer daemon if any events were lost due to shortage of kernel buffers; we then increase the number of sub-buffers shared between the kernel and user-space and restart the traced application.

2.3 Low-Overhead and Accurate

One of the biggest drawbacks of system call tracing is that it adds overhead that can slow down an application—even to the point of failure. Adding overhead can also change execution patterns and timings. If we want to capture and replay server applications, this overhead can cause timeouts and dropped packets, and even failed queries. Re-Animator offers efficient and optimized versions of `strace` and LTTng, while maintaining high fidelity. RA-LTTng minimizes the overhead for capturing buffers and writes them efficiently.

2.3.1 RA-Strace

RA-Strace has two major components: (1) a Re-Animator library (RA-Lib) and (2) `strace` integration.

We designed RA-Lib as a common API for both RA-Strace and RA-LTTng. RA-Lib is an optimized library for writing system call information to DataSeries files. Thus it is useful for integration with other tracing systems such as DTrace [17] and Ktrace [30].

We described RA-Strace's integration in Section 2.2. We now describe how RA-Lib's modular design handles writing system call records efficiently. Integrating support for capturing a new system call is easy thanks to C++ abstractions, but the latter tend to add overheads. RA-Lib works as a router: it gets trace data and calls the corresponding handler to create an appropriate DataSeries record for writing. We designed RA-Lib independent of any capturing technique or the underlying OS; for example, we do not hard-code system call numbers, which change between OSes. RA-Lib has two main data structures to support this translation flexibility: (1) mapping system-call names to handler functions and DataSeries accessor objects that are responsible for writing system calls according to their types to DataSeries files, and (2) mapping system-call argument types to DataSeries types. These mapping mechanisms are heavily used; our initial implementation was $3\times$ times slower than the one reported in this paper. We profiled the code and changed many critical data structures to improve lookup and insert speeds (e.g., ordered hash maps permit enumeration in sorted order, but we had no need for such enumeration). We also added internal caching of frequently accessed DataSeries objects, to avoid having to re-retrieve them from the DataSeries file. We chose `TCMalloc` [32] over several other memory allocators as it provided the best performance for this project.

2.3.2 RA-LTTng

We detailed RA-LTTng's mechanisms for capturing buffers in Section 2.2. When RA-LTTng gets the captured buffer's content, it offloads the writing to a work queue (configurable, but we currently limit it to holding 32 items). Linux's work queue will then spawn at most 32 kernel worker threads to write one item each to persistent storage. This asynchrony allows the traced application to continue execution, interleaving with the kernel worker threads. When tracing an application that allocates a lot of memory and also runs CPU-intensive tasks, it is possible that the OS will not be able to schedule the trace-writing kthreads frequently enough to flush those trace records. To avoid losing any records, RA-LTTng blocks the traced application until the work queue drains, an approach that can further slow down applications but guarantees high fidelity. Slowdown due to a full work queue can be reduced by raising its maximum size above 32 items.

2.4 Scalable and Verifiable

In this section, we explain how Re-Animator provides a scalable and verifiable framework for both capturing and replaying. We first describe how our LTTng integration facilitates system-call tracing to make replaying verifiable. We then provide a detailed architectural design for Re-Animator's scalable and verifiable replayer, called RA-Replayer.

We have explained how Re-Animator captures buffers accurately and efficiently in Sections 2.2 and 2.3. Re-Animator leverages LTTng's architecture to collect as much data as it can without adding significant overhead. Capturing complete buffer data allows RA-Replayer to verify system calls on the fly and generate the same final disk state.

2.4.1 Re-Animator Replayer

We now detail the design of RA-Replayer. There are four major principles in our design: (1) verifiable, (2) concurrent lock-free design, (3) supporting multi-process applications with a user-space file-descriptor manager, and (4) simulating system calls that cannot be replayed.

2.4.2 Verifiable

During replay, Re-Animator checks that return values match those from the original run *and* that buffers contain the same content. Here, "buffers" refers to every single memory region that contains execution-related data, including results for calls like `stat`, `utimes`, `getdents`, `ioctl`, `fcntl`, etc. Since the trace file contains buffer contents for system calls that pass data to the kernel and change on-disk state, we can perform the same operation with the same data to produce the same on-disk state as the original execution. We have confirmed that Re-Animator generates the same content as the traced application by running several micro- and macro-benchmarks (see Section 3) and comparing the directory trees after the replay run. However, there are necessarily some limitations since a generated file's `atime`, `mtime`, and `ctime` will not be the same (absent a `utimes` call), and the results of reading `procfs` files like `/sys/block/sda/sda1/stat` might be different.

RA-Replayer links with complex DataSeries libraries among others (e.g., `libc`): it is therefore possible that some of these libraries will open, close, or otherwise manipulate file descriptors without our knowledge (e.g., the DataSeries library opens a descriptor for every extent (section) of a trace file so it can read from that extent's offsets while avoiding `lseeks`). Therefore, RA-Replayer also periodically scans its own file descriptors and validates that their expected state (open or closed) matches what our user-space file descriptor manager expects. The frequency of verifying FDs is configurable and can also be disabled.

2.4.3 Concurrent Lock-Free Design

Reading a trace file is often an I/O-bound task [5]. Fortunately, DataSeries stores traces efficiently, reducing I/O, and offers multi-threaded replay. We now describe the general architecture of RA-Replayer (Figure 2.3) and explain why it is scalable and low-overhead.

To coordinate replay without introducing extra system calls, we use lock-free data structures provided by Intel's Threading Building Blocks [45]. RA-Replayer is divided into two major components: First, a single thread reads the DataSeries file and fills a priority queue (PQ) for each execution thread; there is one thread for each process or thread being replayed. A DataSeries file is naturally divided into *extents* (see Section 2.5), one for each type of system call. The reader thread retrieves several records from these extents and inserts them into the PQs; at run time it measures the rate that the execution threads drain their PQs and dynamically adjusts the amount of data it next retrieves from the extents. Second, execution threads drain their PQs and execute system calls with appropriate arguments. RA-Replayer supports 67 system calls (including those that we can only simulate). Each call's implementation provides functions to execute it, verify its results, and optionally log results.

In both the reader and the execution threads, we use `TCMalloc` [32] for efficient, concurrent memory allocation.

2.4.4 Supporting Multiple Processes

RA-Replayer can replay multi-process applications by spawning threads inside the replayer. Since the replayer is a single process, we have a user-space file-descriptor manager that keeps track of file descriptor allocations for each emulated process. The manager keeps track of FD changes caused by operations such as `dup`, `pipe`, and `execve`, including supporting unusual cases such as the `O_CLOEXEC` flag. This was

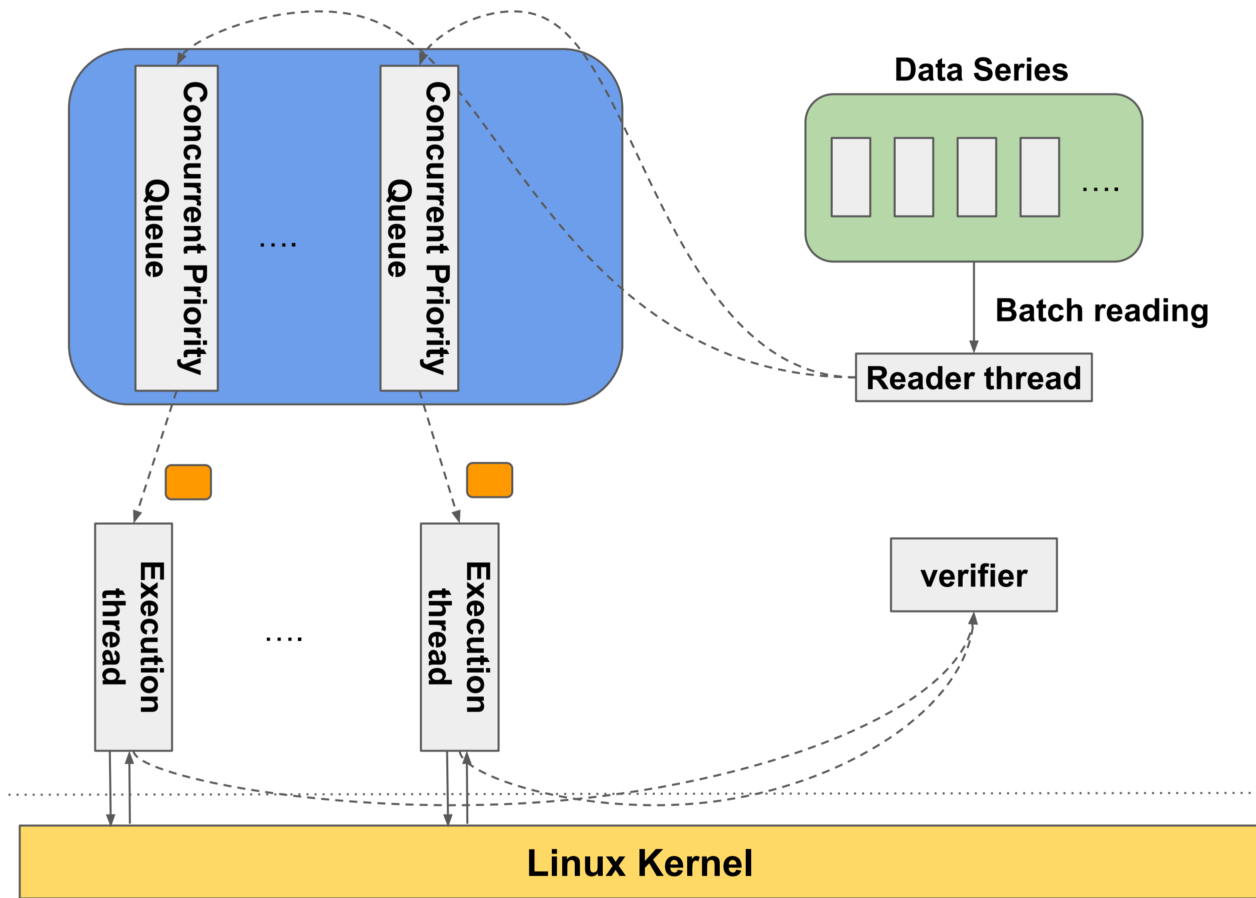


Figure 2.3: RA-Replayer architecture.

particularly important for correctly replaying two processes that exchange data over a `pipe`. Our earlier prototype attempted to replay all traced processes inside a single replayer process, but it proved too complex to manage the per-process FD states and to avoid deadlocking on reading a pipe if the pipe-writer event had not yet happened (and cannot be replayed while the single replayer thread is blocked). The lesson we learned is that any high-fidelity replayer should be multi-threaded to better emulate the original traced environment.

2.4.5 Simulated System Calls

We designed Re-Animator to capture as much data as possible, which means that it sometimes captures system calls, parameters, and buffers that cannot be replayed (such as operations on sockets, which would require complex connections to network resources that might not behave in a reproducible fashion). Instead, we simulate these calls using data recorded in the DataSeries file, by discarding the call and assuming it succeeded or failed as before, and filling in any buffers such as those returned from a socket. Capturing data for simulated system calls not only enables future research opportunities but also helps us keep our FD mappings accurate.

2.5 Portable

To allow our tools to be used as widely as possible, we store trace output in the DataSeries format [5], and to design our replayer to read that same format. DataSeries is a compact, flexible, and fast format first

developed at HP Labs; a C++ library and associated tools provide easy access to the format.

A DataSeries file is organized into a number of *extents*, each of which has a schema defined in the file header. We are using an updated version of the SNIA schema for system-call traces [78], which we plan to submit to SNIA when it is complete. In our design each extent stores records of the same system call type. One exception is multi-record system calls such as `writev`: we store an initial record with the common information (file descriptor, offset, and iovcnt), followed by N records, one for each iovec. Unlike prior tools, which tended to capture only the information of interest to a particular researcher, we have opted for a maximalist approach, recording as much data as possible. Doing so has two advantages: (1) it enables fully accurate replay, and (2) it ensures that a future researcher—even one doing work we did not envision—will not be limited by a lack of information. (E.g., Ou *et al.* [65] concluded that `fcntl` calls were never used in the LASR traces, when in fact those calls were simply not captured.)

In particular, in addition to *all* system call parameters, we record the precise time the call began and ended, plus the PID, thread ID, parent PID, and process group ID of the issuing process. If the call failed, we also record the `errno` returned. By default we also record the data buffers for reads and writes. Lastly, each record of a trace is assigned a unique, monotonically increasing record ID.

When replaying, we reproduce nearly all calls precisely—even failed ones. The original success or failure status of a call is verified to ensure that the replay has been accurate, and we compare all returned information (e.g., `stat` results and data returned by `read`) to the original values.

However, there are certain practical exceptions to our "replicate everything" philosophy: for example, if it were followed slavishly, replaying network activity would require that all remote computers be placed into a state identical to how they were at the time of capture. Given the complexities of the Internet and systems such as DNS, such precise reproduction is impossible. Instead, we simulate the network: sockets are created but not connected, and I/O calls on socket file descriptors are simply discarded.

2.5.1 Source Code Size

Over a period of three years, we wrote nearly 20,000 lines of C/C++ code (LoC). We added or modified 1,783 LoC in `strace`, 3,928 LoC for the tracer-integration library with DataSeries, 7,760 for the replayer and another 1,005 for the record-sorter tool. We added or modified 1,135 LoC in LTTng's kernel module, 1,624 LoC for the LTTng user-level tools, and finally 2,347 LoC for the babeltrace2ds converter.

Chapter 3

Evaluation

Our Re-Animator evaluation goals were to measure its overheads, show replayer performance numbers, and get a taste for other practical uses of the portable trace files we have collected (e.g., useful statistics).

3.1 Testbed

Our testbed includes four identical Dell R-710 servers, each with two Intel Xeon quad-core 2.4GHz CPUs and configured to boot with 4GB RAM. Each server ran CentOS Linux v7.6.1810, but we installed and ran our own 4.19.19 kernel with RA-LTTng code changes. Each server had three drives to minimize I/O interference: (1) A Seagate ST9146852SS 148GB SAS as a boot drive. (2) An Intel SSDSC2BA200G3 200GB SSD ("test drive") for the benchmark's test data (e.g., where MySQL would write its database). We used an SSD since they are becoming popular on servers due to their superior random-access performance. (3) A separate Seagate ST9500430SS 500GB SAS HDD ("trace-capture drive") for recording the captured traces, also used for reading traces back during replay onto the test drive. Since our traces are written sequentially, using a non-SSD drive here did not impact trace-writing or trace-reading performance, as SAS drives offer good sequential performance.

Although our four servers had the same hardware and software, we verified that identical, repeated experiments on them yielded results that did not deviate by more than 1–2% across servers.

3.2 Benchmarks

We ran a large number of micro- and macro-benchmarks. Micro-benchmarks are useful to highlight the worst-case behavior of a system by focusing on specific operations. Macro-benchmarks show the realistic, real-world performance of applications with mixed workloads. For brevity, we describe only a subset of those tests in this paper, focusing on the most interesting trends, including worst-case scenarios. All benchmarks were run at least five times; standard deviations were less than 5% of the mean unless otherwise reported. Each benchmark was repeated under four different conditions: (1) an unmodified program (called "Vanilla") without any tracing to serve as the baseline; (2) the program as traced using an unmodified `strace` tool ("Strace") where all (text) output was saved to a file on the trace-capture drive to simulate capturing actual trace records; (3) the program as traced using our modified `strace`, which directly writes a DataSeries file onto the trace-capture drive ("RA-Strace"); and (4) the program traced using our modified `LTTng`, which directly records binary in RA-CTF format ("RA-LTTng") (see Section 2.2.2).

3.2.1 Micro-Benchmarks

To capture traces, we first ran the FIO micro-benchmark [29], which tests read and write performance for both random and sequential patterns; each FIO test ran with 1, 2, 4, and 8 threads. We configured FIO with an 8GB dataset size to ensure it exceeded our 4GB server RAM size and thus exercised sufficient I/Os. (We also ran several micro-benchmarks using Filebench [4] but omit the results since they did not differ much from FIO's.)

3.2.2 Macro-Benchmarks

We ran two realistic macro-benchmarks: (1) LevelDB [54], a key-value (KV) store with its own `dbbench` exerciser. We asked LevelDB to run 8 different pre-configured I/O-intensive phases: fillseq, fillsync, fillrandom, overwrite, readrandom1, readrandom2, readseq, and readreverse. We configured database sizes of 1GB, 2GB, 4GB, and 8GB (by asking `dbbench` to generate 10, 20, 40, and 80 million KV pairs, respectively); and for each DB size we ran LevelDB with 1, 2, 4, and 8 threads. Finally, because RA-LTTng currently does not capture `mmap` events, we configured LevelDB to use regular `read` and `write` system calls. (2) MySQL [81] with an 8GB database size. We configured `sysbench` [82] to run 4 threads that issue queries to MySQL for a fixed one-hour period.

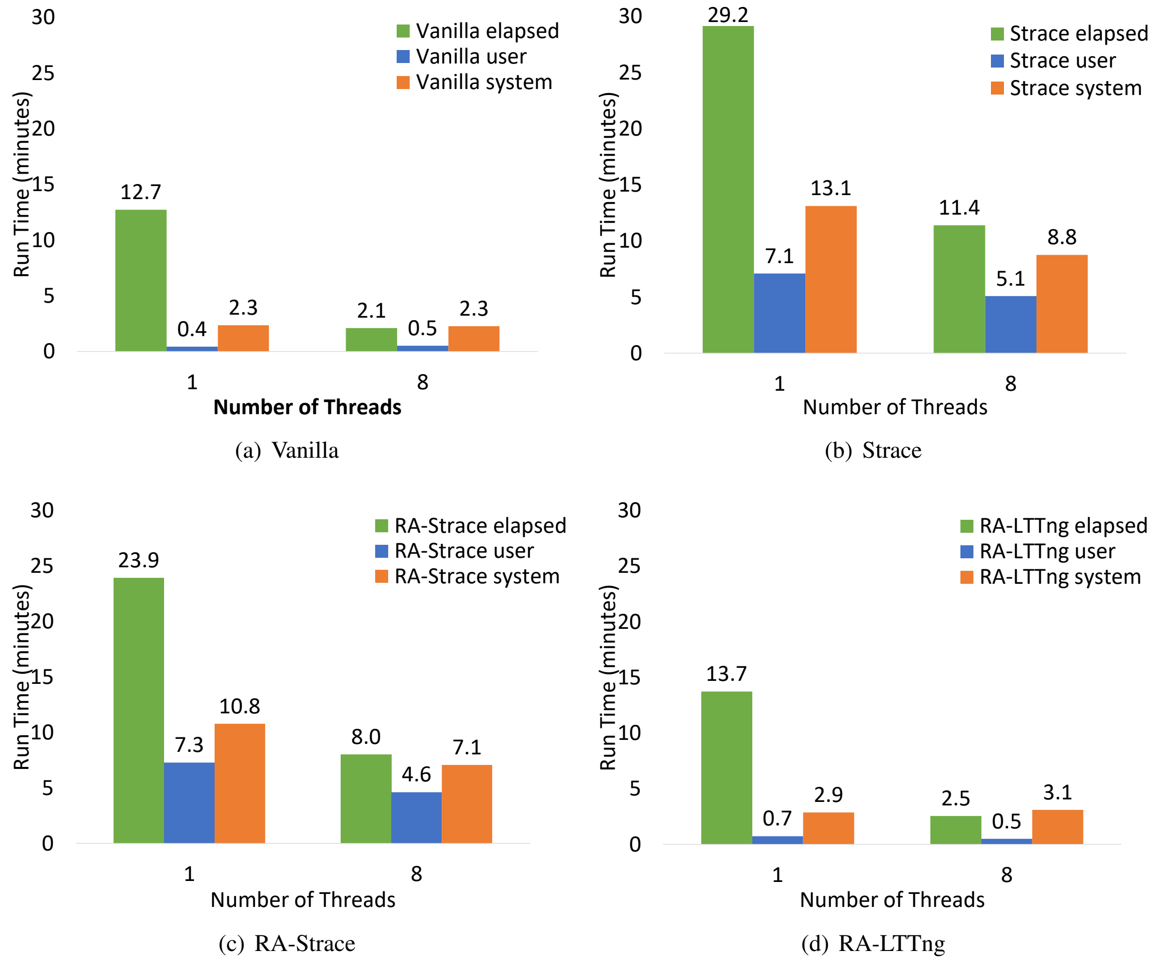
3.2.3 Replaying Benchmarks

We report the times to replay some of the larger DataSeries trace files we captured. As described in Section 2.4, our replayer runs as-fast-as-possible (AFAP), verifies all system call return values and buffers at runtime, ensures an accurate on-disk state replay, and preserves non-overlapping events' timings. We replayed every trace captured in this project and manually verified (e.g., using `diff -r`) the on-disk state after replay compared with the vanilla program; no anomalies were found. Our replay timings do not include the time to run `diff`, since such verification would not normally be necessary in a "production" environment.

Recall that RA-LTTng stores traces using the enhanced RA-CTF format (Linux's CTF format for system-call records, modified to include record IDs, plus separate indexed binary files to store system-call buffers); therefore we used the offline `babeltrace2ds` tool we developed to convert RA-CTF traces to DataSeries format before replaying the latter. `Babeltrace` can consume a lot of I/O and CPU processing cycles to convert between formats, in part because it has to build global mapping tables for objects between the two formats; nevertheless, this conversion need only be done once and can be performed offline without affecting any application's run. In a large experiment we conducted, `babeltrace2ds` took 13 hours to convert a 255GB RA-CTF file (from a LevelDB experiment) to a 214GB DataSeries file; the latter file size is smaller because the DataSeries binary format is more compact than RA-CTF's. The conversion was accomplished on a VM configured with 128GB RAM. At its peak, `babeltrace2ds`'s resident memory size exceeded 60GB. These figures justify our choice to perform this conversion offline, rather than attempt to integrate a complex and large DataSeries library, all written in C++, into the C-based Linux kernel. Optimizing `babeltrace2ds`—currently single-threaded—was not an express goal of this project and is left to future work.

3.3 FIO Micro-Benchmark

We report the time (in minutes) to run FIO with 1 or 8 threads. (The results with 2 and 4 threads were in between the reported values, but we do not have enough data to establish a trend curve based on the number



(a) Vanilla

(b) Strace

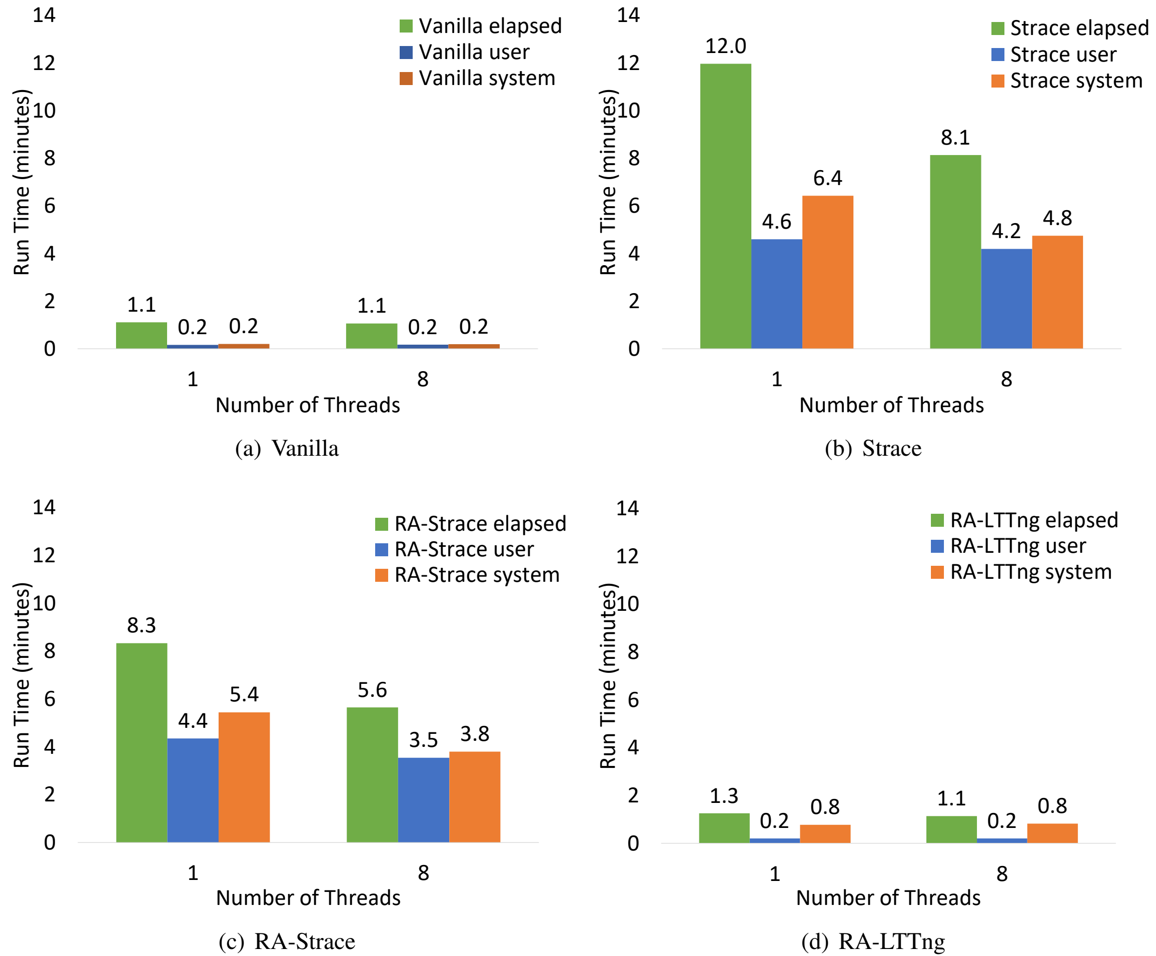
(c) RA-Strace

(d) RA-LTTng

Figure 3.1: FIO random read times in minutes (elapsed, user, and system).

of threads.) We report elapsed, user, and system times separately. We include all dirty-data flushing as well as trace records persisting in our measurements.

Figures 3.1, 3.2, 3.3, and 3.4 show FIO's random-read, sequential-read, random-write, and sequential-write times, respectively. Several trends seen in this data were the same for FIO's sequential reads and both random and sequential writes. These trends (some of which are unsurprising) are: (1) Compared to Vanilla, all tracing takes longer. (2) Strace was the slowest, followed by RA-Strace, with RA-LTTng being the most efficient tracer. (3) Running FIO with 8 threads instead of one reduces overall times thanks to better I/O and CPU interleaving. Our servers have 8 cores each, and their SSDs are inherently parallel devices that can process multiple I/Os concurrently [18, 26, 49]. We focus on the one-thread results below. (4) Strace adds significant user time to allocate and copy buffers from the traced process, and to format them for human consumption. Strace also adds significant system (kernel) time due to context switches and `ptrace` handling. Together with the additional I/O needed to write its human-formatted output, Strace takes 2.3–11× longer than Vanilla (an untraced FIO). (5) RA-Strace improves on Strace primarily due to writing an efficient binary-formatted DataSeries file, but still incurs significant user- and system-time overheads. Compared to Vanilla, RA-Strace's elapsed times are 1.9–7.9× slower; RA-Strace is still 22–45% better than Strace. (6) RA-LTTng further improves overheads thanks to its efficient, in-kernel, asynchronous tracing and logging. Compared to Vanilla, RA-LTTng's elapsed times are only 8–33% slower; RA-LTTng is 1.7–7.2×



(a) Vanilla

(b) Strace

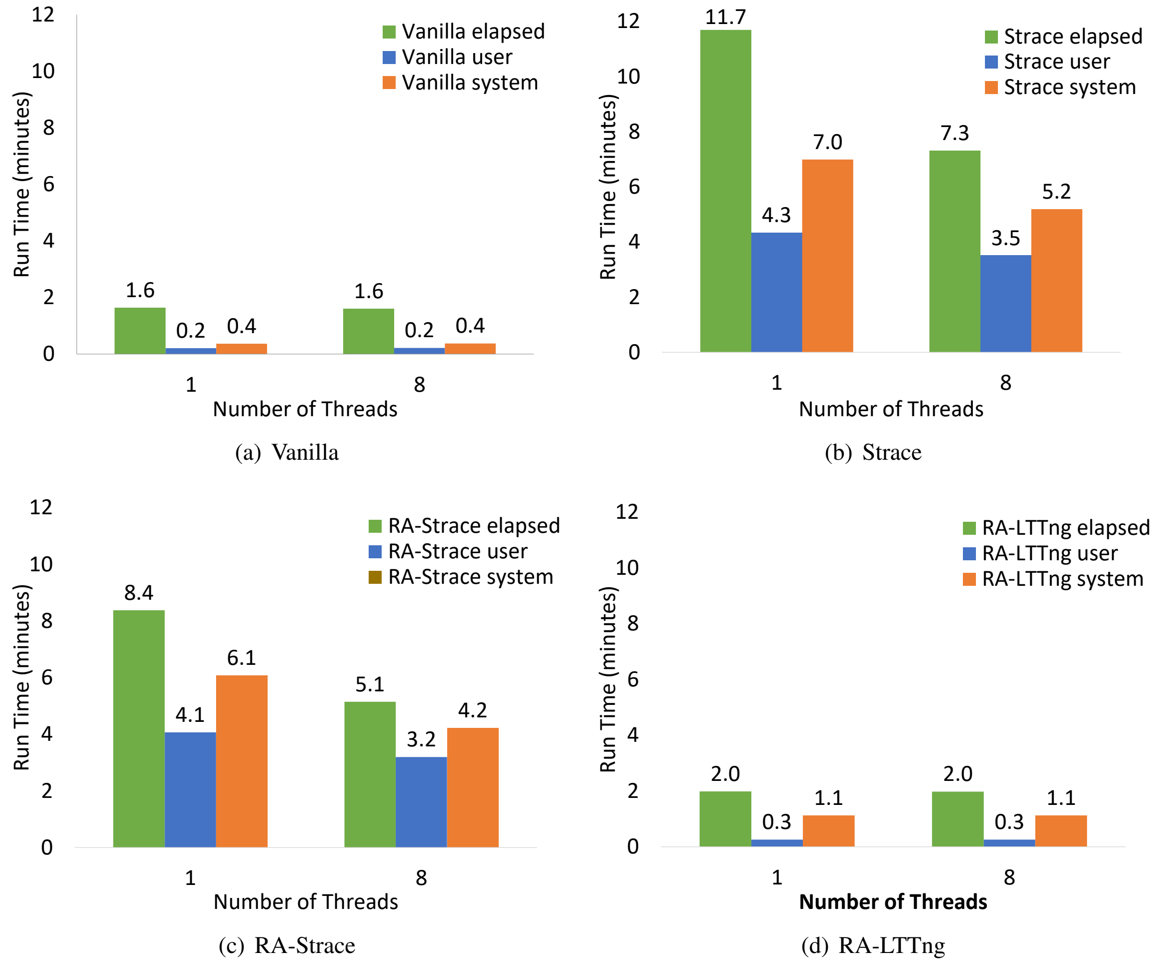
(c) RA-Strace

(d) RA-LTTng

Figure 3.2: FIO sequential read times in minutes (elapsed, user, and system).

faster than RA-Strace. (7) RA-LTTng adds much less user and system time overhead than Strace and RA-Strace, because it performs most its actions inside the kernel and we use asynchronous threads that permit better interleaving of I/O and CPU activities. (8) The FIO random-read test is the most challenging: unlike writes, which can be processed asynchronously, uncached reads are synchronous. Sequential reads are easier to handle than random reads thanks to read-ahead, which is why even the Vanilla elapsed time for random-read (Figure 3.1(a)) takes about 10× longer the other three FIO runs. This makes all elapsed times (for Strace, RA-Strace, and RA-LTTng) in Figure 3.1 longer than their counterparts in other FIO runs. Because the system is more frequently blocked on I/Os in FIO's random-read benchmark, the overheads imposed by tracing, relative to Vanilla, are lower: 2.3× for Strace, 1.9× for RA-Strace, and 1.1× for RA-LTTng.

Figure 3.5 shows the elapsed time needed to replay the FIO traces captured by RA-LTTng; the traces captured with RA-Strace were functionally identical although event timings differed, but executing them as-fast-as-possible (AFAP) was nearly identical to the results in this figure. (We remind the reader than an optimized replayer was not an express goal of the work reported in this paper.) We observed the following trends: (1) Overall, RA-Replayer has to read the trace file from disk and process it before it can be replayed. Thus, RA-Replayer adds significant user time with one thread. (2) Whereas the FIO application has to perform CPU computations (e.g., compute random offsets to write), RA-Replayer does not need to do so (or user times would have been even longer). (3) Because RA-Replayer performs CPU processing and

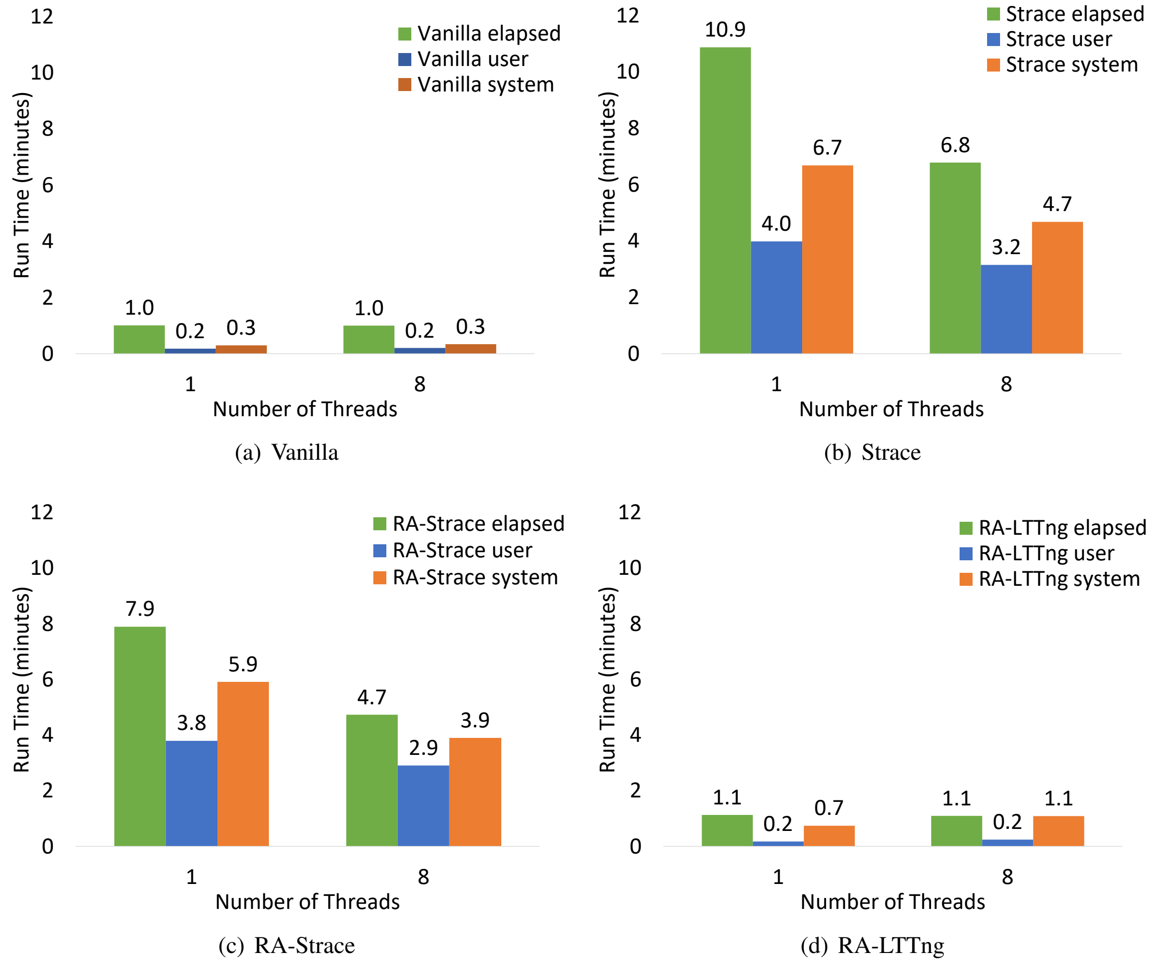


(a) Vanilla (b) Strace (c) RA-Strace (d) RA-LTTng

Figure 3.3: FIO random write times in minutes (elapsed, user, and system).

memory operations, it can interleave those with I/O activity of replaying the actual trace. Thus, overall elapsed time overheads are more moderate than might be expected (0–72% slower). (4) One exception to these observations is the random-read FIO benchmark. As explained above for trace-capture results, this benchmark has the worst I/O behavior. Because I/O is much slower here, RA-Replayer can catch up faster when replaying AFAP events. And because RA-Replayer does not have the same "think time" that FIO requires, overall RA-Replayer performs *faster* than the original FIO benchmark: 39% faster for one thread and 10% faster for 8 threads.

We noted that with 8 threads, system time overheads can reach 37× and user times reach as high as 46×. We investigated this overhead and found it due to a large number `sched_yield` system calls that the Intel's Threading Building Blocks (TBB) [45] executes in order to coordinate multiple threads: 1–8 replayer threads, one master reader thread to coordinate reading from extents, and multiple threads that RA-Replayer spawns to read from different extents; although we spawn one thread per extent type, which corresponds to a specific system call, most of these threads are idle because we found that many applications execute a small number of system calls most of the time (see Section 3.6). For example, the FIO sequential-read experiment needs to executes 2.1M system calls recorded in the trace file. The TBB library adds another 1.1M `sched_yield` calls but these consume a negligible amount of time (see Figure 3.5(b)). The number of `sched_yields` grows to 1.5M for 2 threads, 10.7M calls for 4 threads, and then jumps up to 422M



(a) Vanilla

(b) Strace

(c) RA-Strace

(d) RA-LTTng

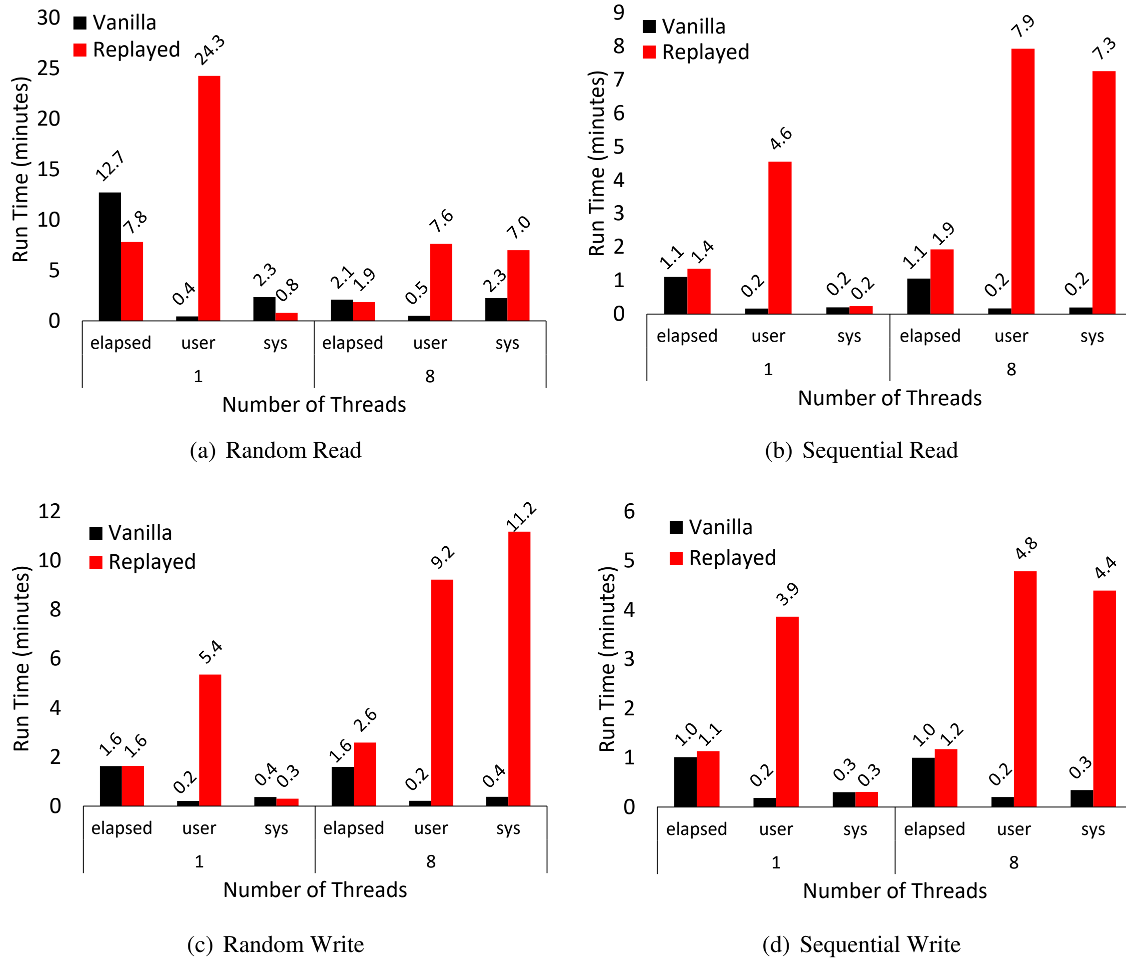
Figure 3.4: FIO sequential write times in minutes (elapsed, user, and system).

`sched_yield` calls for 8 threads. At that point the contention for our 8-core servers becomes unreasonable: 8 replayer threads, one master reader thread, and several more active threads that DataSeries spawns to read from different extents concurrently. That contention causes the system time to grow considerably in the 8-thread case. We plan to investigate ways to reduce this contention in future research.

3.4 LevelDB Macro-Benchmark

Figure 3.6 reports the total run time for LevelDB on a 1GB database, using 4 threads and the default sequence of phases described in Section 3.2. Note that the 1GB DB is smaller than our 4GB system memory; this is actually a worst-case benchmark compared to larger DB sizes because more system calls can execute without blocking on slow I/Os, while Re-Animator still need to persistently record every system call and its buffers to a dedicated trace-capture drive. Thus, the overhead of Re-Animator is higher in this case. We can see that Strace was more than 10× slower than the vanilla program. RA-Strace improved this overhead to 7.8×, thanks to our more efficient binary trace format. RA-LTTng improved this overhead even further, to 2.3×, due to its in-kernel asynchronous tracing infrastructure.

Figure 3.7 shows LevelDB's random-read performance (in ms/operation) for different-sized databases. We chose to report detailed results for LevelDB because the random-read phase showed the most interesting



(a) Random Read (b) Sequential Read

(c) Random Write (d) Sequential Write

Figure 3.5: Elapsed time for vanilla execution vs. replayed trace (minutes). Note the Y-axis ranges differ in each bar graph.

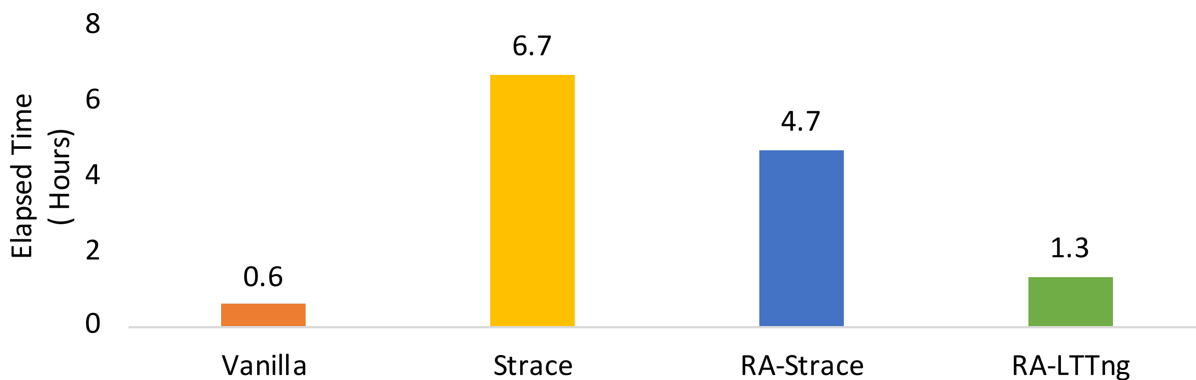


Figure 3.6: LevelDB elapsed times (hours).

patterns and also exercises both the I/O subsystem and the OS intensely. We omit results for `strace` alone because it ran fairly slowly: on the 1GB DB, `strace` was more than 7× slower than RA-Strace.

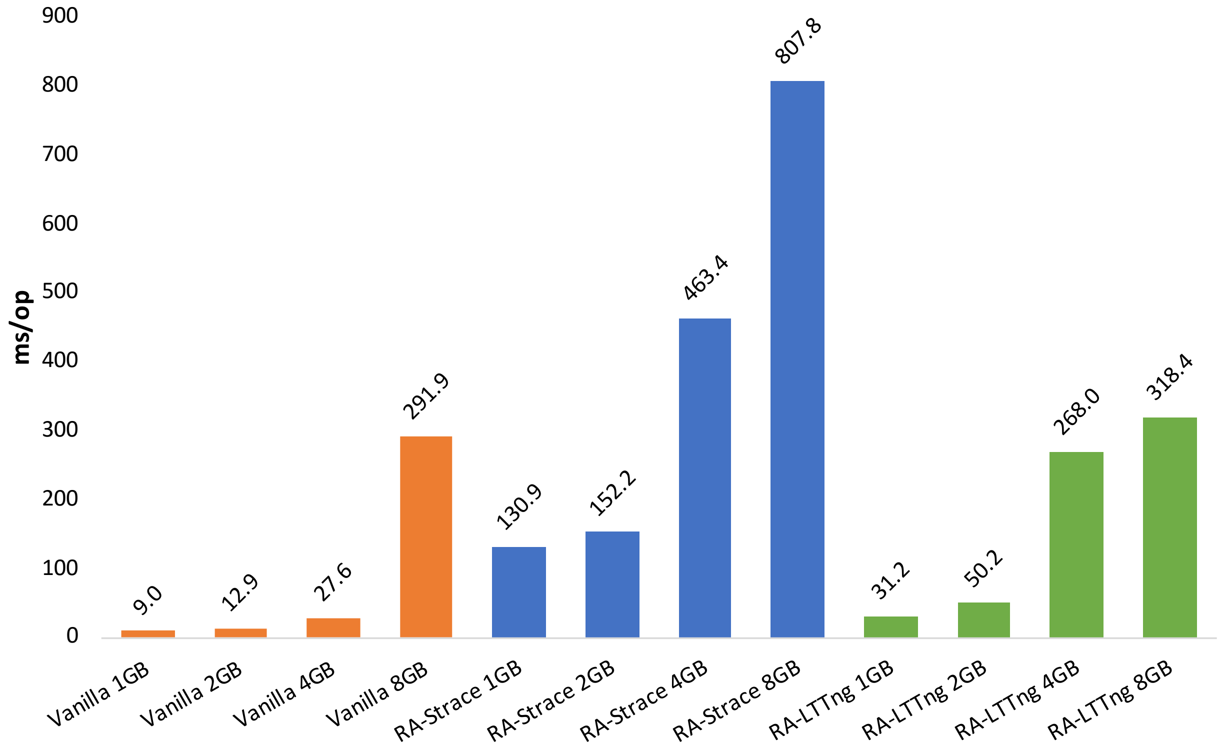


Figure 3.7: LevelDB read-random latency for different-sized databases (in milliseconds per operation).

As expected, latency grows as the DB size grows. Once the DB size grows to 8GB—double the 4GB RAM of our test servers—significant swapping and paging activity takes place; even for vanilla instances, the latency for 8GB is more than 10× larger than for the 4GB DB.

On the whole, we see that RA-Strace is slower than vanilla on every DB size, commensurate with our micro-benchmarks results (Section 3.3). When the DB size fits in memory (1GB), RA-Strace is 14.5× slower; when the DB size is large enough to cause more I/O activity (8GB), this relative overhead drops to 2.8× because system calls become more I/O-bound and run longer.

Relative to Vanilla, when the DB fits in memory (1GB), RA-LTTng is 3.5× slower; when the DB size is large enough to cause more I/O activity (8GB), this relative overhead drops to only 9% slower, thanks to RA-LTTng's superior scalability. Overall, RA-LTTng performs better than RA-Strace in all four DB sizes: 1.7–4.2× better.

Both RA-Strace and RA-LTTng show a jump in latency when going from 2GB to 4GB DB sizes—an increase not seen in the vanilla benchmark (`dbbench`). The reason is that the 4GB DB mostly fits in memory under Vanilla, and hence incurs few paging I/Os, especially because dbbench generates its data on the fly (in memory). Tracing, however, requires additional I/Os to write the trace itself, and these I/Os compete for page-cache space (and shared I/O busses) with the benchmark itself.

We captured a small trace of LevelDB running on a 250MB database, using one thread, with the default sequence of phases described in Section 3.2: it took 81 seconds elapsed time. The DataSeries trace file for this experiment was 25GB in size. Replaying it took 300 seconds, or 3.7× longer. We verified the on-disk state after replaying this trace and it was identical to the original LevelDB run. Improving RA-Replayer's speed is part of our future work.

3.5 MySQL Macro-Benchmark

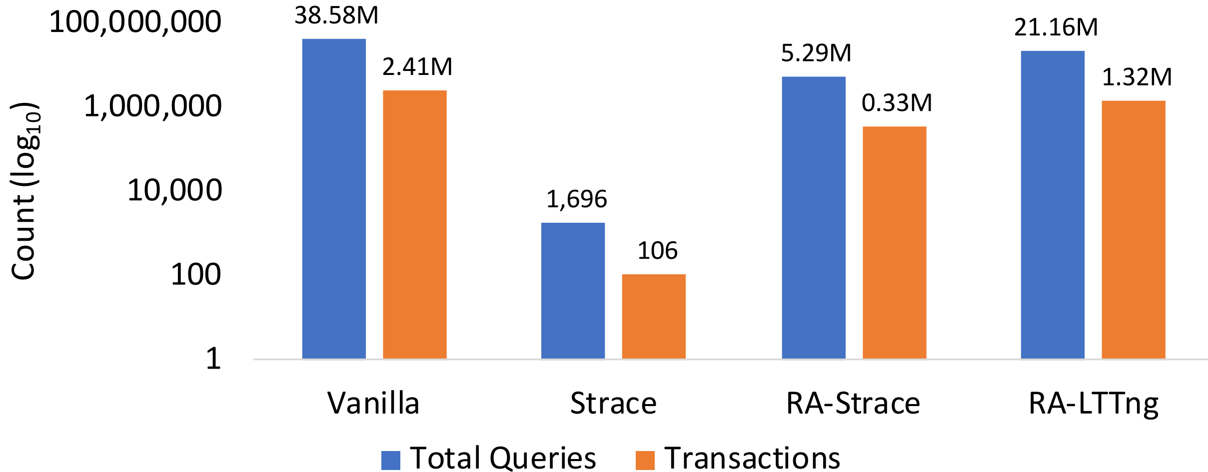


Figure 3.8: Counts ($\log_{10}$) of MySQL queries and transactions completed within a one-hour period.

Figure 3.8 shows the counts of total queries and transactions completed within one hour by `sysbench` issuing requests to MySQL (on a $\log_{10}$ scale). One or more queries were sent as a single transaction, hence the number of transactions is lower than the total number of queries. In one hour, Vanilla completed 38.5M queries. Relative to Vanilla, Strace performed extremely poorly—more than four orders of magnitude worse; the overheads added by `strace` were just high enough to cause nearly all the MySQL queries to fail due to timeouts. Because our RA-Strace is more efficient than `strace`, it was able to complete 5.3M queries (about 14% of Vanilla's performance); and because our RA-LTTng is even more efficient than RA-Strace, it completed 21.2M queries (about 55% of Vanilla, or $4\times$ more than RA-Strace). These measurements explain why users are leery of tracing live "production" applications and justify why system call tracing must have overheads that are as low possible.

3.6 Trace Statistics

DataSeries comes with a tool called `dsstatgroupby`, which can extract useful statistics from DataSeries trace files. Although a detailed analysis of such statistics is beyond the scope of this paper, to demonstrate the usefulness of using traces beyond just replaying them, we highlight a few useful metrics that we extracted.

For example, the LevelDB experiment executed a total of 6,378,938 system calls (23 unique calls). 99.87% of all those calls were to `write` and `pread`. The distribution of buffer sizes passed to `write` ranged from 20B to 64KB, with many odd and sub-optimal sizes just above 4KB. We noted that over 3M `write` calls used a specific—and highly inefficient–buffer size of 138B. We hypothesize that the odd-sized writes are related to atomic transactions in this KV store, suggesting that there may be significant room for improving LevelDB's performance by using an alternate data structure.

Similarly, the MySQL experiment executed a total of 8,763,035 system calls (37 unique). Four dominating calls—`pwrite`, `pread`, `fsync`, and `write`—accounted for 99.95% of the calls. We found that most `pread` calls were exactly 16KB in size and thus highly efficient. We also observed 2.5M `fsync` calls (e.g., to flush transaction logs). We further explored the latency quantiles of `fsync`: about 20% of all calls took less than 1ms but 0.01% of all calls (about 250) took over 100ms to complete (exhibiting tail latencies observed by other researchers [23, 39, 47, 55]).

Chapter 4

Related Work

There are several approaches to tracing system calls: based on `ptrace`, interposing shared libraries, and in-kernel methods.

4.1 Ptrace

Because `ptrace` [35] has been part of the Unix API for decades, it is an easy way to track process behavior. `strace` [89], released for SunOS in 1991, was one of the earliest tools to build upon `ptrace`; a Linux implementation soon followed, and most other Unix variants offer similar programs such as `truss` [28] and `tusc` [12]. On Microsoft Windows, StraceNT [31] offers a similar facility.

All of these approaches share a similar drawback: because the trace is collected by a separate process that uses system calls to access information in the target application, overheads are unusually high (as much as an order of magnitude). In most cases, the CPU cost of collecting information overwhelms the I/O cost of writing trace records. In theory, the cost could be reduced by modifying the `ptrace` interface, e.g., by arranging to have system-call parameters collected and reported in a single `ptrace` operation. To our knowledge, however, there have been no efforts along these lines.

4.2 Shared-Library Interposition

A faster alternative to `ptrace` that still requires no kernel changes is to interpose a shared library that replaces all system calls with a trace-collecting version [22, 60]. Since the shared library runs in the same process context, data can be captured much more efficiently. However, there are also a few drawbacks: (1) the technique does not capture early-in-process activity (such as loading the shared libraries themselves; (2) interposition may be difficult in `chroot`ed environments where the special library might not be available; (3) trace collection in a multithreaded process may require additional synchronization; and (4) if the user wishes to interpose more than one library, integration may be challenging. However, nothing in Re-Animator's design would preclude our techniques from being used in an interposition context.

4.3 In-Kernel Techniques

The lowest-overhead approach to capturing program activity is to do so directly in the kernel, where all system calls are interceptable and all parameters are directly available. Several BSD variants, including Mac OS X, offer ktrace [30], which uses kernel hooks to capture system-call information. Solaris supports Dtrace [17] and Windows offers Event Tracing for Windows (ETW) [63]. All of these approaches capture

events into an in-kernel buffer that is later emptied by a separate thread or process. Since kernel memory is a precious resource, all of these in-kernel tools limit how much memory they use to store traced events, and they drop events if not enough memory is available. We have verified this event-drop experimentally for both dtrace and ktrace; ETW further limits any single captured event to 64KB in size.

The Linux Kprobes facility [21] has been used to collect read and write operations [79], but the approach was complex and incomplete. A more thorough implementation is FlexTrace [87], which allows users to make fine-grained choices about what to trace; FlexTrace also offers a blocking option so that no events are lost. However, it does not capture data buffers.

Linux's LTTng allows the user to allocate ample kernel buffers to record system calls, limited only by the system's RAM capacity; however, as we noted earlier, vanilla LTTng does not capture data buffers. Our RA-LTTng captures those buffers directly to a separate file for later post-processing (and blocks the application if the buffers are not flushed fast enough, ensuring high fidelity).

Many modern applications use `mmap` to more efficiently read and write files, but `ptrace`-based systems cannot capture `mmap`ed events (e.g., page faults and dirty page flushes). In-kernel tracers can do so (e.g., TraceFS [7]). RA-LTTng currently does not track those events but we plan to add the capability as part of our future work.

Finally, unlike `strace` and RA-LTTng, which have custom code to capture every `ioctl` type, neither Ktrace nor DTrace can capture buffers unless their length is easily known (e.g., the 3rd argument to `read`), and thus neither captures `ioctl` buffers at all. Moreover, Ktrace flushes its records synchronously: in one experiment we conducted (FIO 8GB random-read using one thread), Ktrace imposed higher overheads than our RA-LTTng, consuming at least 70% more system time and at least 50% more elapsed time.

4.4 Replayer Fidelity

To the best of our knowledge, no system-call replayer exists that can replay the buffers' data (e.g., to `write`). ROOT [88], which is based on `strace`, concentrates on solving the problem of correctly ordering multi-threaded traces. It does not capture or replay actual system-call buffers. //TRACE [62] also concentrates on parallel replay but does not reproduce the data passed from `read` and to `write`. We attempted to compare ROOT and //TRACE to RA-Replayer but were unable to get them to run, even with the help of their original authors.

RA-Replayer has options to verify that each replayed system call returned the same status (or error if traced as such), as well as to verify each buffer (e.g., after a `read`). If any deviation is detected, we support options to log a warning and then either continue or abort the replay. We are not aware of any other system-call replayer with such run-time verification capabilities.

Thus, RA-Replayer faithfully reproduces on-disk state: file names and namespaces, file contents, and most inode metadata (e.g., inode type, size, and UID and GID if replayed by a superuser). Because replaying happens after the original capture, one limitation we have is that we do not reproduce inode access, change, and modification times accurately—but the relative ordering of these timestamps is preserved.

Like hf-player [36, 37], we use heuristics to determine how to replay events across multiple threads: any calls whose start-to-end times did *not* overlap are replayed in that order.

4.5 Scalability

All system-call tracers can capture long-running programs, but using a binary trace format (e.g., as all in-kernel tracers do) enables such tools to reduce I/O bottlenecks and the chance of running out of storage space. That is why we modified `strace` to capture raw data and buffers and store them in the DataSeries format, rather than verbosely displaying the traced calls for human consumption.

ROOT [88] parses traces from several formats and then produces a C program that, when compiled and run, will replay the original system calls. We believe this compiler-based approach is limited: whereas RA-Replayer can replay massive traces (we replayed traces that were hundreds of GB in size), compiling and running such huge programs may be challenging if not impossible on most systems.

4.6 Portable Trace Format

Dtrace [17], ktrace [30], and ETW [63] use their own binary trace formats. `Strace` does not have a binary format; its human-readable output is hard to parse to reproduce the original binary system call data [33, 42, 88]. (In fact, one of the reasons we could not get ROOT to run, despite seeking assistance from its authors, is that the text output format of `strace` has changed in a fashion that is almost imperceptible to humans but incompatible with ROOT's current code.) Only LTTng uses a binary format, CTF [58], that is intended for long-term use. However, CTF is relatively new and it remains to be seen whether it will be widely adopted; in addition, because it is a purely sequential format, it is difficult to use with a multithreaded replayer. Non-portable, non-standard, and poorly documented formats have hampered researchers interested in system call analysis and replay (including us) for decades. Thus, we chose DataSeries [5], a portable, well documented, open-source, SNIA-supported standard trace format. Our RA-Strace writes DataSeries files directly. DataSeries comes with many useful tools to repack and compress trace files, extract statistics from them, and convert them to other useful formats (e.g., such as plain text and CSV). The SNIA Trace Repository [9] offers approximately 4TB of traces in this format. We left LTTng's CTF format in place, so as not to require massive code changes or complex integration of C++ into the kernel; instead, we wrote a standalone tool that can convert CTF files to DataSeries ones offline.

Chapter 5

Conclusion and Future Work

Tracing and trace replay are essential tools for debugging systems and analyzing their performance. We have built Re-Animator, which captures system-call traces in a portable format and replays them accurately. We found that regular `strace` can impose overheads as high as an order of magnitude for applications such as LevelDB and MySQL. We provide two capture methods, one based on `strace`, which operates entirely at the user level but with overheads of 7.3–7.8×, and a second one based on LTTng, which requires small kernel modifications but runs with lower overheads of only 1.8–2.3× compared to an untraced application. Unlike previous systems, we capture *all* information, including data buffers for system calls such as `read` and `write`, needed to reproduce the original application exactly.

Our replayer is designed for precise fidelity. Since it has access to the original data, it correctly reproduces behavior even on modern systems that employ data-dependent techniques such as compression and deduplication. The replayer verifies its actions as it performs them, ensuring that the final on-disk state matches the original. We have traced and replayed a number of popular applications and servers, comparing outputs to ensure that they are correct.

5.1 Future Work

(1) In the future, we plan to add support for tracing applications that use `mmap` to read and write files (which is possible only when tracing at the kernel level). (2) We will also investigate adding our user-level modifications to similar tools that run on `DTrace`- and `ktrace`-based systems. (3) The kernel-based tracing systems we investigated can lose CTF events under heavy loads when insufficient kernel memory buffers are available (but not system call buffers). We plan to investigate techniques to throttle applications that generate many events, to allow traces to be written persistently without loss of any event. (4) Finally, using our tools, we plan to collect long-term traces that will enable new areas of research, such as cyber-security and machine learning.

Bibliography

[1] Cristina L. Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H. Campbell. Metadata traces and workload models for evaluating big storage systems. In *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, pages 125–132. IEEE, 2012.

[2] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Towards realistic file-system benchmarks with CodeMRI. *ACM Performance Evaluation Review*, 36(2):52–57, September 2008.

[3] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic *impressions* for file-system benchmarking. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 125–138, San Francisco, CA, February 2009. USENIX Association.

[4] George Amvrosiadis and Vasily Tarasov. Filebench github repository, 2016. https://github.com/filebench/filebench/wiki.

[5] Eric Anderson, Martin F. Arlitt, Charles B. Morrey III, and Alistair Veitch. DataSeries: An efficient, flexible, data format for structured serial data. *ACM SIGOPS Operating Systems Review*, 43(1), January 2009.

[6] Eric Anderson, Mahseh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 45–58, San Francisco, CA, March/April 2004. USENIX Association.

[7] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: A file system to trace them all. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 129–143, San Francisco, CA, March/April 2004. USENIX Association.

[8] Maen M. Al Assaf, Mohammed I. Alghamdi, Xunfei Jiang, Ji Zhang, and Xiao Qin. A pipelining approach to informed prefetching in distributed multi-level storage systems. In *Proceedings of the 11th IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, August 2012. IEEE.

[9] Storage Networking Industry Association. IOTTA trace repository. http://iotta.snia.org, February 2007. Cited December 12, 2011.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, Bolton Landing, NY, October 2003. ACM SIGOPS.

[11] Muli Ben-Yehuda, Michel Factor, Eran Rom, Ashivay Traeger, Eran Borovik, and Ben-Ami Yassour. Adding advanced storage controller functionality via low-overhead virtualization. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[12] Chris Berlin. *tusc - trace unix system calls*. Hewlett-Packard Development Company, L.P., 2011. *http://hpux.connect.org.uk/hppd/hpux/Sysadmin/tusc-8.1/man.html*.

[13] Jeff Bonwick. ZFS deduplication, November 2009. *https://blogs.oracle.com/bonwick/zfs-deduplication-v2*, Retreived April 17, 2019.

[14] Gianluca Borello. System and application monitoring and troubleshooting with sysdig. In *Proceedings of the 2015 USENIX Systems Administration Conference (LISA '15)*. USENIX Association, November 2015. *https://sysdig.com*.

[15] Alan D. Brunelle. Blktrace user guide, February 2007.

[16] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–9, Boston, MA, October 1992. ACM Press.

[17] Brian M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 15–28, 2004.

[18] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, HPCA '11, pages 266–277, 2011.

[19] William Chen. You must unlearn what you have learned ... about SSDs, February 2014. Available from *http://storage.toshiba.com/corporateblog*.

[20] Youmin Chen, Jiwu Sh, Jiaxin Ou, and Youyou Lu. HiNFS: A persistent memory file system with both buffering and direct-access. *ACM Transactions on Storage*, 14(1):4:1–4:30, April 2018.

[21] Will Cohen. Gaining insight into the Linux kernel with kprobes. *RedHat Magazine*, March 2005.

[22] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *USENIX Conference Proceedings*, pages 267–278, Boston, MA, June 1994. USENIX. *https://www.usenix.org/legacy/publications/library/proceedings/bos94/curry.html*.

[23] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[24] Mathieu Desnoyers. Using the Linux kernel tracepoints, 2016. *https://www.kernel.org/doc/Documentation/trace/tracepoints.txt*.

[25] Mathieu Desnoyers and Michel R. Dagenais. Lockless multi-core high-throughput buffering scheme for kernel tracing. *Operating Systems Review*, 46(3):65–81, 2012.

[26] Cagdas Dirik and Bruce Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 279–289, New York, NY, USA, 2009. ACM.

[27] Laura DuBois, Marshall Amaldas, and E. Sheppard. Key considerations as deduplication evolves into primary storage. White Paper 223310, NetApp, Inc., March 2011.

[28] Sean Eric Fagan. *truss - trace system calls*. FreeBSD Foundation, July 24 2017. https://www.freebsd.org/cgi/man.cgi?truss.

[29] fio—flexible I/O tester. http://freshmeat.net/projects/fio/.

[30] FreeBSD Foundation. *ktrace -- enable kernel process tracing*, July 24 2017. https://www.freebsd.org/cgi/man.cgi?query=ktrace&manpath=FreeBSD+12.0-RELEASE+and+Ports.

[31] Pankaj Garg. StraceNT - strace for Windows. https://github.com/l0n3c0d3r/stracent. Visited April 23, 2019.

[32] Sanjay Ghemawat. TCMalloc: Thread-caching malloc. https://gperftools.github.io/gperftools/tcmalloc.html, April 2019.

[33] Roberto Gioiosa, Robert W. Wisniewski, Ravi Murty, and Todd Inglett. Analyzing system calls in multi-OS hierarchical environments. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2015.

[34] Dinan Srilal Gunawardena, Richard Harper, and Eno Thereska. Data store including a file location attribute. United States Patent 8,656,454, December 1 2010.

[35] M. Haardt and M. Coleman. *ptrace(2)*. Linux Programmer's Manual, Section 2, November 1999.

[36] Alireza Haghdoost, Weiping He, Jerry Fredin, and David H.C. Du. hfplayer: Scalable replay for intensive block I/O workloads. *ACM Transactions on Storage (TOS)*, 13(4):39, 2017.

[37] Alireza Haghdoost, Weiping He, Jerry Fredin, and David H.C. Du. On the accuracy and scalability of intensive I/O workload replay. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 315–328, 2017.

[38] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP '11)*, Cascais, Portugal, October 2011. ACM Press.

[39] Jun He, Duy Nguyen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Reducing file system tail latencies with Chopper. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 119–133, Berkeley, CA, USA, 2015. USENIX Association.

[40] Dean Hildebrand, Anna Povzner, Renu Tewari, and Vasily Tarasov. Revisiting the storage stack in virtualized NAS environments. In *Proceedings of the Workshop on I/O Virtualization (WIOV)*, 2011.

[41] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, November 1999.

[42] Jiří Horkỳ and Roberto Santinelli. From detailed analysis of IO pattern of the HEP applications to benchmark of new storage solutions. *Journal of Physics: Conference Series*, 331, 2011. http://inspirehep.net/record/1111456/files/jpconf11_331_052008.pdf.

[43] H. Howie Huang, Nan Zhang, Wei Wang, Gautam Das, and Alexander S. Szalay. Just-in-time analytics on large file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2011. USENIX Association.

[44] Zhisheng Huo, Limin Xiao, Qiaoling Zhong, Shupan Li, Ang Li, Li Ruan, Shouxin Wang, and Lihong Fu. MBFS: a parallel metadata search method based on Bloomfilters using MapReduce for large-scale file systems. *Journal of Supercomputing*, 72(8):3006–3032, August 2016.

[45] Intel Corporation. Threading building blocks(tbb). https://www.threadingbuildingblocks.org, April 2019.

[46] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, Santa Clara, CA, February 2015. USENIX Association.

[47] Nikolai Joukov, Ashivay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating system profiling via latency analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[48] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and efficient replaying of file system traces. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 337–350, San Francisco, CA, December 2005. USENIX Association.

[49] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *HotStorage '14: Proceedings of the 6th USENIX Workshop on Hot Topics in Storage*, Philadelphia, PA, June 2014. USENIX.

[50] Yangwook Kang, Jingpei Yang, and Ethan L. Miller. Efficient storage management for object-based flash memory. In *Proceedings of the 18th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 407–409, Miami Beach, FL, August 2010. IEEE.

[51] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *Trans. Storage*, 6(3):13:1–13:26, September 2010.

[52] Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14(5):24–42, 1997.

[53] Rachita Kothiyal, Vasily Tarasov, Priya Sehgal, and Erez Zadok. Energy and performance evaluation of lossless file data compression on server systems. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM.

[54] LevelDB, January 2012. http://code.google.com/p/leveldb.

[55] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[56] liburcu. Userspace RCU. https://liburcu.org, April 2019.

[57] Xing Lin, Fred Douglis, Jim Li, Xudong Li, Robert Ricci, Stephen Smaldone, and Grant Wallace. Metadata considered harmful…to deduplication. In *HotStorage'15*, 2015.

[58] Linux Foundation. The common trace format. https://diamon.org/ctf/, April 2019.

[59] LTTng. LTTng: an open source tracing framework for Linux. https://lttng.org, April 2019.

[60] Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. A multi-level approach for understanding I/O activity in HPC applications. In *IEEE International Conference on Cluster Computing (CLUSTER)*, September 2013.

[61] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Pilip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2016. USENIX Association.

[62] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Julio Lopez, James Hendricks, Gregory R. Ganger, and David O'Hallaron. //TRACE: Parallel trace replay with approximate causal events. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 153–167, San Jose, CA, February 2007. USENIX Association.

[63] Microsoft Corporation. *Event Tracing*. https://docs.microsoft.com/en-us/windows/desktop/etw/event-tracing-portal. Visited April 23, 2019.

[64] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[65] Jiaxin Ou, Kiwu Shu, and Youou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 12:1–12:16, London, UK, April 2016. ACM.

[66] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[67] Thiago Emmanuel Pereira, Livia Sampaio, and Francisco Vilar Brasileiro. On the accuracy of trace replay methods for file system evaluation. In *Proceedings of the 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, San Francisco, CA, February 2014.

[68] Thiago Emmanuel Pereira, Jonhnny Weslley Silva, Alexandro Soares, and Francisco Brasileiro. BeeFS: A cheaper and naturally scalable distributed file system for corporate environments. In *Proceedings of the 28th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*, Gramado, Brazil, May 2010.

[69] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, Washington, DC, August 2003.

[70] Amit Purohit, Charles P. Wright, Joseph Spadavecchia, and Erez Zadok. Cosy: Develop in user-land, run in kernel-mode. In *Proceedings of the 2003 ACM Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 109–114, Lihue, Hawaii, May 2003. USENIX Association.

[71] Jian-Ping Qiu, Guang-Yan Zhang, and Ji-Wu Shu. DMStone: A tool for evaluating hierarchical storage management systems. *Journal of Software*, 23:987–995, April 2012.

[72] Himanshu Raj and Karsten Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2007.

[73] Jose Renato Santos, Yoshio Turner, G.(John) Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, June 2008. USENIX Association.

[74] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2013. USENIX Association.

[75] Mark A. Smith, Jan Pieper, Daniel Gruhl, and Lucas Vill Real. IZO: Applications of large-window compression to virtual machine management. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 2008.

[76] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[77] Vaughn Stewart. Pure storage 101: Adaptive data reduction. *https://blog.purestorage.com/pure-storage-101-adaptive-data-reduction*, March 2014.

[78] Storage Networking Industry Association. Posix system-call trace common semantics. *https://members.snia.org/wg/iottatwg/document/8806*, September 2008. Accessible only to SNIA IOTTATWG members.

[79] Jian Sun, Zhan-huai Li, Xiao Zhang, Qin-lu He, and Huifeng Wang. The study of data collecting based on kprobe. In *2011 Fourth International Symposium on Computational Intelligence and Design*, volume 2, pages 35–38. IEEE, 2011.

[80] Zhen "Jason" Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, and Erez Zadok. Cluster and single-node analysis of long-term deduplication patterns. *ACM Transactions on Storage (TOS)*, 14(2), May 2018.

[81] Sun Microsystems. MySQL. *www.mysql.com*, Dec 2008.

[82] sysbench. Scriptable database and system performance benchmark. *https://github.com/akopytov/sysbench*, April 2019.

[83] Rukma Talwadker and Kaladhar Voruganti. ParaSwift: File i/o trace modeling for the future. In *Proceedings of the 28th USENIX Large Installation Systems Administration Conference (LISA)*, pages 119–132, Seattle, WA, November 2014. USENIX Association.

[84] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *Proceedings of the Linux Symposium*, pages 83–95, Ottawa, Canada, July 2014.

[85] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80, May 2008.

[86] Yoshihiro Tsuchiya and Takashi Watanabe. DBLK: Deduplication for primary block storage. In *Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, 2011.

[87] Marc-André Vef, Vasily Tarasov, Dean Hildebrand, and André Brinkmann. Challenges and solutions for tracing storage systems: A case study with spectrum scale. *ACM Transactions on Storage (TOS)*, 14(2):18, 2018.

[88] Zev Weiss, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ROOT: Replaying multithreaded traces with resource-oriented ordering. In *Proceedings of the 24th ACM Symposium on Operating System Principles (SOSP '13)*, pages 373–387, Farmington, PA, November 2013. ACM Press.

[89] Wikimedia Foundation. strace. https://en.wikipedia.org/wiki/Strace. Visited April 22, 2019.

[90] Yair Wiseman, Karsten Schwan, and Patrick M. Widener. Efficient end to end data exchange using configurable compression. *ACM SIGOPS Operating Systems Review*, 39(3):4–23, 2005.

[91] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 329–342, Berkeley, CA, USA, 2013. USENIX Association.

[92] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 139–150, Helsinki, Finland, 2012.

[93] Quan Zhang, Dan Feng, Fang Wang, and Sen Wu. Mlock: building delegable metadata service for the parallel file systems. *Science China Information Systems*, 58(3):1–14, March 2015.

[94] Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Zettabyte reliability with flexible end-to-end data integrity. In *Proceedings of the 29th IEEE Symposium on Mass Storage Systems and Technologies*, Long Beach, CA, May 2013.

[95] Ningning Zhu, Jiawu Chen, and Tzi-Cker Chiueh. TBBT: Scalable and accurate trace replay for file server evaluation. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 323–336, San Francisco, CA, December 2005. USENIX Association.