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

Ibrahim Umit Akgun
Stony Brook University
iakgun@cs.stonybrook.edu

Geoff Kuenning
Harvey Mudd College
geoff@cs.hmc.edu

Erez Zadok
Stony Brook University
ezk@cs.stonybrook.edu

ABSTRACT

Modern applications use storage systems in complex and often surprising ways. Tracing system calls is a common approach to understanding applications' behavior, allowing offline analysis and enabling replay in other environments. But current system-call tracing tools have drawbacks: (1) they often omit some information—such as raw data buffers—needed for full analysis; (2) they have high overheads; (3) they often use non-portable trace formats; and (4) they may not offer useful and scalable analysis and replay tools.

We have developed Re-Animator, a powerful system-call tracing tool that focuses on storage-related calls and collects maximal information, capturing complete data buffers and writing all traces in the standard DataSeries format. We also created a prototype replayer that focuses on calls related to file-system state. We evaluated our system on long-running server applications such as key-value stores and databases. Our tracer has an average overhead of only 1.8–2.3×, but the overhead can be as low as 5% for I/O-bound applications. Our replayer verifies that its actions are correct, and faithfully reproduces the logical file system state generated by the original application.

CCS CONCEPTS

• **Software and its engineering** → *Operating systems*; • **General and reference** → *Performance.*

KEYWORDS

System-call Tracing and Replaying, Benchmarking, Storage Systems

SYSTOR '20, June 2–4, 2020, Haifa, Israel

ACM ISBN 978-1-4503-7588-7/20/06…$15.00
https://doi.org/10.1145/3383669.3398276

ACM Reference Format:
Ibrahim Umit Akgun, Geoff Kuenning, and Erez Zadok. 2020. Re-Animator: Versatile High-Fidelity Storage-System Tracing and Replaying. In *The 13th ACM International Systems and Storage Conference (SYSTOR '20), June 2–4, 2020, Haifa, Israel.* ACM, New York, NY, USA, 14 pages. *https://doi.org/10.1145/3383669.3398276*

1 INTRODUCTION

Modern applications are becoming ever more intricate, often using 3rd-party libraries that add further complexity [40]. Operating systems have multiple layers of abstraction [9, 79] and deep network and storage stacks [10, 42, 78]. In addition, storage systems employ techniques like compression, deduplication, and bit-pattern elimination [13, 29, 55, 57, 66, 80, 82, 83, 91, 94, 98]. The result is that applications interact with the rest of the system in complex, unpredictable ways, making it difficult to understand and analyze their behavior.

System-call tracing is a time-honored, convenient way to study an application's interaction with the OS; for example, tools such as `strace`[97] can record events for human analysis. Such traces can be replayed [96] to reproduce behavior without needing to recreate input conditions and rerun the application, exploring its behavior in different situations (e.g., performance tuning or analysis [15, 39, 44, 48, 49, 60, 69, 73, 90, 92, 103]), or to stress-test other components (e.g., the OS or storage system) [1–3, 8, 21, 36, 45, 46, 51, 73, 74, 77, 89, 93, 101, 102]. Traces can also be analyzed offline (e.g., using statistical or machine-learning methods) to find performance bottlenecks, security vulnerabilities, etc. [43, 75, 76], or identify malicious behavior [32, 54]. Historical traces can help understand the evolution of computing and applications over long intervals. Such long-term traces are useful in evaluating the effects of I/O on devices that wear out quickly (SSDs) or have complex internal behavior (e.g., garbage collection in shingled drives) [20, 24, 41, 47, 61, 99, 100].

However, existing system-call tracing approaches have drawbacks: (1) They often do not capture all the information needed to reproduce the exact system and storage state, such as the full data passed to `read` and `write` system calls. (2) Tracing significantly slows traced applications and even the surrounding system, which can be prohibitive in production environments. Thus, tracing is often avoided in mission-critical settings, and traces of long-running applications are

rare. (3) Traces often use custom formats; documentation can be lacking or non-existent, and sometimes no software or tools are released to process, analyze, or replay the traces. Some traces (e.g., those from the Sprite project [72]) have been preserved but can no longer be read due to a lack of tools. (4) Some tools (e.g., `strace` [97]) produce output intended for human consumption and are not conducive to automated parsing and replay [35, 44, 96].

In this paper, we make the following six contributions: **(1)** We have designed and developed Re-Animator, a system-call tracing package that uses Linux tracepoints [25] and LTTng [26, 64] to capture traces with low overhead. **(2)** Our tracing system captures as much information as possible, including all data buffers and arguments. **(3)** We write the traces in DataSeries [5], the format suggested by the Storage Networking Industry Association (SNIA) for I/O and other traces. DataSeries is compact, efficient, and self-describing. Researchers can use existing DataSeries tools to inspect trace files, convert them to plain text or spreadsheet formats, repack and compress them, subset them, and extract statistical information. **(4)** Our system adds an average overhead of only 1.8–2.3× to traced applications (in the best case, only 5%). **(5)** We developed a prototype replayer that supports 70 selected system calls, including all that relate to file systems, storage, or persistent state. The replayer executes the calls as faithfully and efficiently as possible and can replay traces as large as hundreds of GB. **(6)** All our code and tools for both tracing and replaying 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 by SNIA.

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.

Any tracing tool can capture sensitive information such as file names, inter-file relations, and even file contents. Re-Animator is intended for environments where such capture and processing of such traces is acceptable to all parties. When privacy is a concern, anonymization may be required [7, 58], but privacy is outside this paper's scope.

2.1 Goals

Fidelity. State-of-the-art techniques for recording and replaying system calls have focused primarily on timing accuracy [6, 16, 40, 48, 67, 96, 103]. Our work considers three replay dimensions: (1) timing, (2) process–thread interdependencies, and (3) the logical POSIX file-system state. Because correct replay requires accurately captured data, this paper focuses primarily on trace capture; our prototype replayer demonstrates this accuracy but does not seek optimality.

Of the three dimensions above, timing is probably the easiest to handle; the tracer must record accurate timestamps, and the replayer should reproduce them as precisely as possible [6]. However, many researchers have chosen a simpler—and entirely defensible—option: replay calls as fast as possible (AFAP), imposing maximum stress on the system under test, which is often the preferred approach when evaluating new systems. For that reason, although we capture precise timestamps, our prototype uses AFAP replay.

Dependencies in parallel applications are more challenging; replaying them incorrectly can lead to unreasonable conclusions or even incorrect results. Previous researchers have used experimental [67] or heuristic [96] techniques to extract internal dependencies. The current version of Re-Animator uses a conservative heuristic similar to *hfplayer* [38]: if two requests overlap in time, we assume that they can be issued in any order; if there is no overlap then we preserve the ordering recorded in the trace file.

Finally, most prior tracing and replay tools discard the transferred data to speed tracing and reduce trace sizes. However, modern storage systems use advanced techniques—such as deduplication [62, 87], compression [17, 57], repeated bit-pattern elimination [83], etc.—whose performance depends on data content. We thus designed Re-Animator to optionally support efficient capture and replay of *full* buffer contents so as to accurately reproduce the original application's results.

Since capturing data buffers can generate large trace files, Re-Animator can optionally replace the data with summary hashes. However, full data capture can enable future research into areas such as (1) space-saving storage options (e.g., compression, deduplication); (2) copy-on-write and snapshot features; (3) complex program behaviors; and (4) security. We discuss the details of our features in Section 2.2.

Minimize overhead. Since our goal is to record realistic behavior, anything that affects the traced application's performance is undesirable. Tracing necessarily adds overhead in several ways: (1) as each system call is made, a record must be created; (2) any data associated with the call (e.g., a pathname or a complete write buffer) must be captured; and (3) the information must be written to stable storage. To reduce overhead, some tracing systems, such as DTrace [18], ktrace [33], and SysDIG [14]—all of which we tested—drop events under heavy load; this is clearly harmful to high fidelity. Some tools can be configured to block the application instead of losing events, which is also undesirable since it can

affect the application's timing. Re-Animator's primary tracing tool, RA-LTTng, is based on LTTng [26, 64], an efficient Linux tracing facility [25]. However, LTTng does not capture buffer contents, so we had to add that feature. RA-LTTng uses a combination of blocking, asynchronous, and lockless mechanisms to ensure we capture all events, including data buffers, while keeping overhead low.

Scalable and verifiable. Tracing tools should always avoid arbitrary limitations. It should be possible to trace large applications for long periods, so traces must be captured directly to stable storage (as opposed to fast but small in-memory buffers). In addition, it must be possible to verify that replay has been done correctly. We use three verification methods: (1) when a system call is issued, we ensure that it received the same return code (including error codes) as was captured; (2) for calls that return information, such as `stat` and `read`, we validate the returned data; and (3) after replay completes, we separately compare the logical POSIX file system state with that produced by the original application.

Portability. Tools are only effective if they are usable in the desired environment. To enhance portability, we chose the DataSeries trace format [5] and developed a common library that standardizes trace capture.

Ease of use and extensibility. User-interface design, flexibility, and power are all critical to a tool's effectiveness. Our framework requires a kernel patch, but capture and replay use simple command-line tools. It is easy to add support for new system calls as necessary.

2.2 Fidelity

Re-Animator is based on LTTng [64], an extensible Linux kernel tracing framework. LTTng inserts *tracepoints* [25] in functions such as the system-call entry and exit handlers. When a tracepoint is hit, information is captured into a buffer shared with a user-level daemon, which then writes it to a file. For parallelism, the shared buffer is divided into *sub-buffers*, one per traced process; the LTTng daemon uses user-space RCUs [27] for lockless synchronization with the kernel. The data is written in Linux's Common Trace Format (CTF) [63], which the `babeltrace` tool converts to human-readable text.

Figure 1 shows LTTng's flow for tracing and capturing calls; green components denote our changes. For ease of use, a wrapper (Figure 1, step 1) automates the tasks of starting the LTTng components and the traced application.

Since the sub-buffers were designed for small records, it is hard to capture large data buffers, such as when a single I/O writes megabytes. Instead, we capture those directly to a secondary file (Figure 1, step 7) in a compact format that contains a cross-reference to the CTF file, the data, and its length. An advantage of the separate file is that it can be

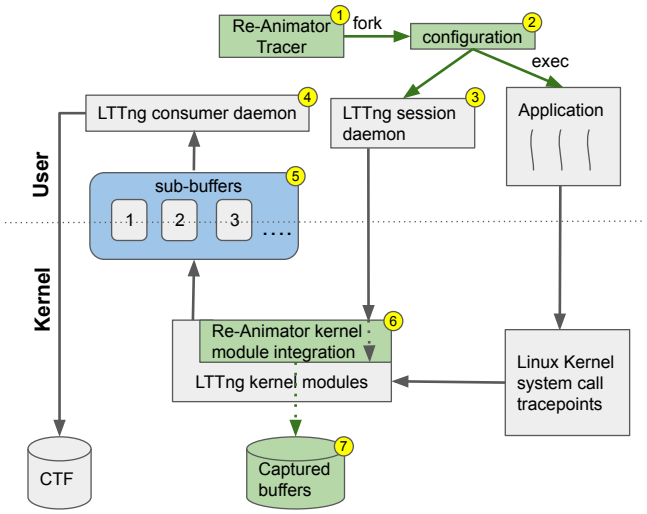


Figure 1: LTTng architecture using Linux kernel tracepoints. Green boxes denote our additions or changes. Our wrapper (1) launches the LTTng configurator (2), which invokes an LTTng session daemon (3) to control the operation and the consumer daemon (4) to collect events. LTTng tracepoints place events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

placed on a different, larger or faster storage device. For parallelism, when we capture one of the 38 system calls that involve data, we copy the user buffer, choose a file offset under a spinlock, and then write the data asynchronously. In the rest of this paper, we refer to this enhanced CTF format with secondary buffer-data files as *RA-CTF*.

We modified `babeltrace` to generate the DataSeries format [5], which can group events on a per-thread basis and includes the captured data, simplifying replay.

To correctly capture system calls in multi-threaded and multi-process applications, we modified LTTng to follow forked processes. (LTTng's developers are working on a more complete solution to the problem of process tracking.)

Memory-mapped files. Many modern applications use `mmap` to access for efficient file access. Unlike user-level system-call tracers [7], RA-LTTng can capture and replay `mmap`ed operations. To do so, we integrated two new kernel tracepoints into LTTng's framework. When an application reads an `mmap`ed file for the first time, a page fault fetches its data. RA-LTTng tracks every insertion into the page cache; if it is to an `mmap`ed file, we add it to a map of inode numbers to page lists and capture the page contents. We also capture page cache insertions caused by readahead operations.

When an application writes to an `mmap`ed file, the cached page is marked dirty, to be later flushed. RA-LTTng monitors

all cache write-backs for mmaped files and writes a copy of the page's contents to the secondary trace file; this is performed asynchronously along with regular write and related system calls as described in Section 2.3. We avoid duplicate writes; for example, if a write causes a page cache write-back operation, we record only the write event and its data.

RA-Replayer fully supports tracing the entire mmap API by replaying reads and writes captured from mmaped file accesses. We use pseudo-system-call records, mmap_pread and mmap_pwrite, for these operations. RA-Replayer can replay mmap_pread and mmap_pwrite "natively" by accessing related pages and causing page faults accordingly, or it can emulate the mmap system calls' actions by calling pread and pwrite. RA-Replayer manages the virtual memory layout for each replayed process; it keeps track of replayed virtual memory areas and where they map to traced processes' virtual address spaces. RA-Replayer can also replay supporting functions for mmap such as msync, madvise, and mprotect.

2.3 Low-Overhead and Accurate

One of the biggest drawbacks of tracing is that it slows the application, changing execution patterns and timings. Server applications can experience timeouts, dropped packets, and even failed queries. Re-Animator minimizes overhead while maintaining high fidelity.

We detailed RA-LTTng's mechanisms for capturing system calls and their data into two separate files in Section 2.2. LTTng allocates a fixed amount of sub-buffer memory; it was designed to cap overheads even if events are dropped (it counts and reports the number of drops). By default, LTTng allocates 4MB for the sub-buffers. We verified that by expanding them to just 64MB—negligible in today's computers—none of our experiments lost a single event.

In addition, we implemented a pseudo-blocking mode in RA-LTTng to ensure that it never loses events. When the sub-buffers fill, RA-LTTng throttles applications that produce too much trace data. First, we try to switch contexts to the user-level process that drains sub-buffers (lttng-consumerd) using the Linux kernel's yield_to API. However, yielding to a specific task inside the kernel succeeds only if the target is in the READY or RUNNING queues. If we are unable to activate the consumer daemon, we sleep for 1ms and then yield to the scheduler. This gives the consumer time to be (re-)scheduled and drain the sub-buffers. We detail the overhead of blocking mode in Section 3.3.

We took a different approach to capturing data buffers, which are much larger than LTTng's event records. When RA-LTTng gets the buffer's content, it offloads writing to a Linux *workqueue* (currently configured to 32 entries). Linux spawns up to one kernel thread per workqueue entry to write the data to disk. This asynchrony allows the traced application to continue in parallel. When tracing an application that generates events at an unusually high rate, it is possible that the OS will not be able to schedule the trace-writing kthreads frequently enough to flush those records. To avoid losing any data, RA-LTTng configures the work queues to block (throttle) the traced application until the queue drains, which can slow the application but guarantees high fidelity. The overhead can be further reduced by increasing the maximum size of the work queue (at the cost of more kernel memory and CPU cycles).

In the future we plan to integrate LTTng's capture mechanism with our data-buffer workqueues while maintaining the goal of capturing all events.

2.4 Verifiable

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 logical disk state.

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 any region that contains execution-related data, including results from calls like stat, getdents, ioctl, fcntl, etc. Since the trace file tracks all calls that pass data to the kernel and change the logical POSIX file system state, we can perform the same operations with the same data to produce the same logical state as the original execution. Furthermore, RA-Replayer verifies that each call produces the same results (including failures and error codes). 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. Note that bit-for-bit identity cannot be achieved, since a generated file's timestamps will not be the same (absent a utimes call), and the results of reading procfs files like /proc/meminfo might be different. Thus, when we use the term "logical state" we are referring to those parts of the state that can reasonably be recreated in a POSIX environment, and RA-Replayer's verification checks only those fields. Both Re-Animator and RA-Replayer are configurable. For example, if Re-Animator is run with data-buffer capture disabled, RA-Replayer allows the user to replay writes using either random bytes or repeated patterns. In that case, RA-Replayer automatically adapts to the captured trace (e.g., it does not try to verify buffer contents that were never captured). RA-Replayer also supports logging with multiple warning levels, and logs can be redirected to a file. Lastly, the aforementioned checks to

verify buffer contents and return values can be enabled (displaying warnings or optionally aborting on any mismatch).

Our current work has focused primarily on accurate trace capture, so RA-Replayer is only a prototype. Nevertheless, we have ensured that its design will support future enhancements to minimize overhead, reproduce inter-thread dependencies, and maximize accuracy and flexibility. These features remain as future work.

2.5 Portable

To allow our tools to be used as widely as possible, we capture and replay in DataSeries [5], a compact, flexible, and fast format developed at HP Labs; a C++ library and associated tools provide easy access. A DataSeries file contains a number of *extents*, each with a schema defined in the file header. We developed an updated version of the SNIA schema for system-call traces [85], which SNIA plans to adopt. Each extent stores records of one system-call type. Unlike prior tools, which often captured only the information of interest to a particular researcher, we have chosen 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.

In particular, in addition to *all* system call parameters, we record the precise time the call began and ended, the PID, thread ID, parent PID, process group ID, and `errno`. By default we also record the data buffers for reads and writes.

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 the 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.

Source code size. Over a period of 3.5 years, we wrote nearly 20,000 lines of C/C++ code (LoC). We added 3,957 LoC for the library that integrates the tracer with DataSeries, 8,422 for the replayer and another 1,005 for the record-sorter tool. We added or modified 2,124 LoC in LTTng's kernel module, 1,696 LoC for the LTTng user-level tools, and finally 2,565 LoC for the `babeltrace2ds` converter.

3 EVALUATION

Our Re-Animator evaluation goals were to measure the overhead of tracing, demonstrate that accurate replay is possible, and get a taste for other practical uses of the portable trace files we have collected (e.g., useful statistics).

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 a deliberately small 4GB of RAM. Each server ran CentOS Linux v7.6.1810, but we installed and ran our own 4.19.19 kernel with the 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. Our traces are written (appended) sequentially in two different file streams (CTF and RA-CTF). But from the disk's perspective, we are generating random accesses because a single HDD head has to seek constantly between those two streams, to append to two different files. This seeking is compounded by the fact that CTF records are small and written frequently whereas RA-CTF records are comparatively large but produced less often. For that reason, we also experimented with writing the trace files to a faster device (Samsung MZ1LV960HCJH-000MU 960GB M.2 NVMe).

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

3.1 Benchmarks

We ran a large number of micro- and macro-benchmarks. Micro-benchmarks can 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 our 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 two different conditions: (1) an unmodified program (called "Vanilla") without any tracing, to serve as a baseline; and (2) the program traced using our modified LTTng, which directly records results in RA-CTF format ("RA-LTTng") (see Section 2.2).

Micro-benchmarks. To capture traces, we first ran the FIO micro-benchmark [31], 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 server's 4GB of RAM 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.)

Macro-benchmarks. We ran two realistic macro-benchmarks: (1) LevelDB [59], a key-value (KV) store with its own `dbbench` exerciser. We ran 8 different pre-configured I/O-intensive phases: fillseq, fillsync, fillrandom, overwrite, readrandom1, readrandom2, readseq, and readreverse. We selected DB sizes of 1GB, 2GB, 4GB, and 8GB by asking `dbbench` to generate 10, 20, 40, and 80 million KV pairs, respectively; and for each size we ran LevelDB with 1, 2, 4, and 8 threads. Finally, although LevelDB uses `mmap` by default, we also configured it to use regular `read` and `write` calls, which exposed different behavior that we captured and analyzed. (2) MySQL [71] with an 8GB database size. We configured `sysbench` [88] to run 4 threads that issued MySQL queries for a fixed one-hour period.

Format conversion. Recall that RA-LTTng stores traces using our enhanced RA-CTF format (Linux's CTF for system calls, slightly enhanced, plus separate binary files to store system-call buffers); therefore we wrote `babeltrace2ds`, which converts RA-CTF traces to DataSeries format before replaying the latter. `Babeltrace2ds` can consume a lot of I/O and CPU cycles, but the conversion is done only once and can be performed offline without affecting an application's behavior. In one large experiment, `babeltrace2ds` took 13 hours to convert a 255GB RA-CTF file (from a LevelDB experiment) to a 214GB DataSeries file; the latter is smaller because the DataSeries format is more compact than RA-CTF's. The conversion was done 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 attempting to integrate the large and complex DataSeries library, all written in C++, into the C-based Linux kernel. Optimizing `babeltrace2ds`—currently single-threaded—was not a goal of this project.

Because RA-Replayer is a prototype, we omit results for it here, as they would not be indicative of the performance of a production version.

3.2 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 between the reported values, but we do not have enough data to establish a curve based on the number of threads.) We report elapsed, user, and system times separately. The results include the time for flushing all dirty data and persisting trace records.

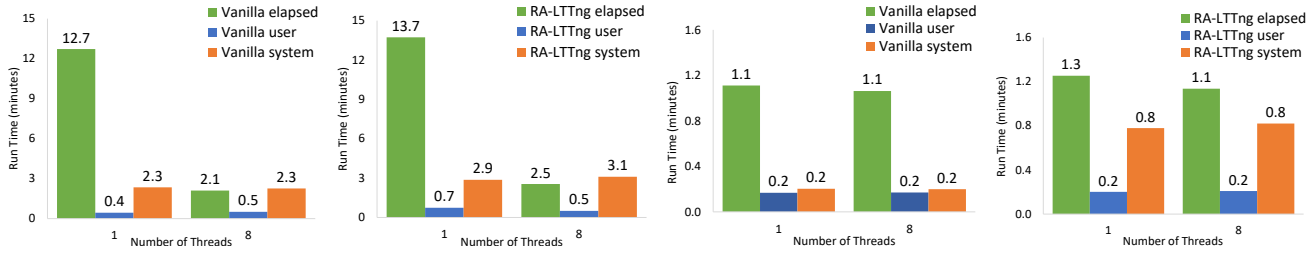
Figures 2 and 3 show FIO's read and write times, respectively. Several trends in this data were the same for sequential reads and both random and sequential writes. These trends (some of which are unsurprising) are: (1) Compared to Vanilla, all tracing takes longer. (2) Running FIO with 8 threads instead of one reduced 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 [19, 28, 50]. We focus on the single-threaded results below. (3) RA-LTTng is low-overhead thanks to its efficient, in-kernel, asynchronous tracing and logging. Compared to Vanilla, RA-LTTng's elapsed times are only 8–33% slower. This is because RA-LTTng performs most of its actions in the kernel, and we use asynchronous threads to permit interleaved I/O and CPU activities. (4) 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 ones thanks to read-ahead, which is why even the Vanilla elapsed time for random reads (Figure 2(a)) was about 10× longer than the other three FIO runs. This made all elapsed times in Figures 2(a) and 2(b) longer than their counterparts in other figures. Because the system was more frequently blocked on I/Os in FIO's random-read benchmark, the overheads imposed by tracing, relative to Vanilla, were lower: only 1.1×.

3.3 LevelDB Macro-Benchmark

We ran LevelDB on a 1GB database, using 4 threads and the default sequence of phases described in Section 3.1. The LevelDB benchmarks took 36 minutes without tracing and 78 minutes with RA-LTTng tracing enabled (2.2× longer). 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 needs to persistently record every system call, including its buffers, to a dedicated trace-capture drive. Thus, the relative overhead of Re-Animator is higher in this case. Nevertheless, RA-LTTng had an overhead of only 2.2×, thanks to its asynchronous in-kernel tracing infrastructure.

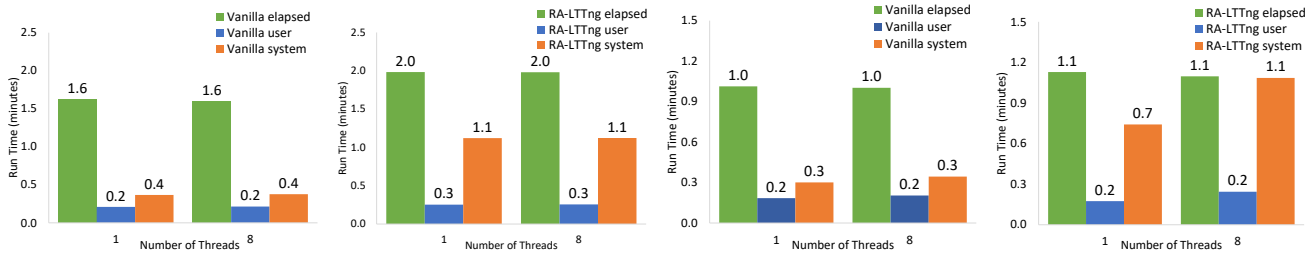
Figure 4 shows LevelDB's random-read performance (in ms/operation) for different-sized databases. We chose to report detailed results for LevelDB because the random-read phase showed the most interesting patterns and also exercised both the I/O subsystem and the OS intensely.

As expected, latency grew with the DB size. Once the DB size reached 8GB—double the 4GB RAM of our test servers—significant swapping and paging activity took place; even



(a) FIO Vanilla read random (b) FIO RA-LTTng read random (c) FIO Vanilla read sequential (d) FIO RA-LTTng read sequential

Figure 2: FIO random and sequential read times in minutes (elapsed, user, and system). Vanilla denotes FIO without tracing; RA-LTTng denotes FIO with full tracing enabled. Note that the Y-axis scales differ between the random and sequential pairs of figures.



(a) FIO Vanilla write random (b) FIO RA-LTTng write random (c) FIO Vanilla write sequential (d) FIO RA-LTTng write sequential

Figure 3: FIO random and sequential write times in minutes (elapsed, user, and system). Vanilla denotes FIO without tracing; RA-LTTng denotes FIO with full tracing enabled. Note that the Y-axis scales differ between the random and sequential pairs of figures.

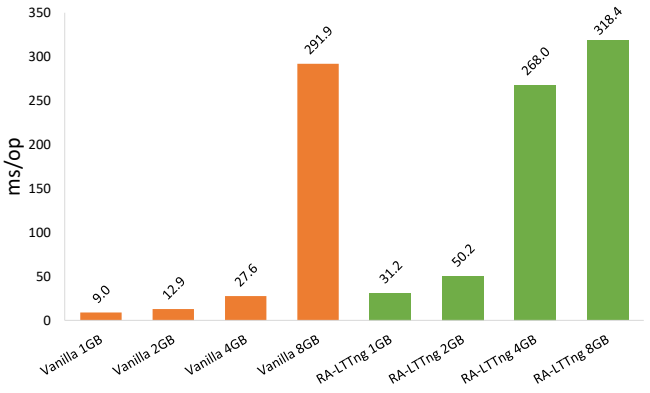


Figure 4: LevelDB read-random latency for different-sized databases (in milliseconds per operation).

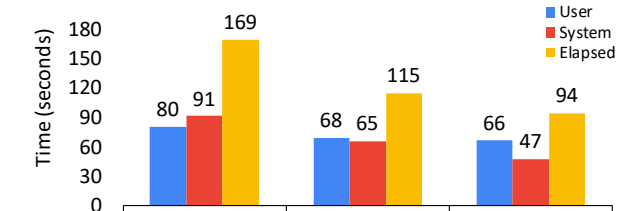
for vanilla instances, the latency for 8GB was more than 10× larger than for the 4GB DB.

Relative to Vanilla, when the DB fit in memory (1GB), RA-LTTng was 3.5× slower; when the DB was large enough to cause more I/O activity (8GB), this overhead dropped to only 9% slower, thanks to RA-LTTng's scalability.

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

We captured a small trace of LevelDB running on a 250MB database, using one thread, with the default sequence of phases described in Section 3.1; the elapsed time was 81 seconds. The DataSeries file for this experiment was 25GB. We verified the logical POSIX file system state after replaying this trace; it was identical to the original LevelDB run.

We also captured a LevelDB workload with 80M KV pairs, running on a server with 24GB RAM (instead of 4GB). We found that runtime did not change substantially, because LevelDB was surprisingly CPU-bound: 30–32% of the cycles were to compress and decompress its own data, and 25–26% were to `memcpy` buffers before decompression and then look up keys.



	1MB	2MB	4MB
Overhead(X)	1.72	1.22	1.00
#Yields	26788	3794	53

Figure 5: RA-LTTng blocking-mode overheads for tracing LevelDB db_bench with its default database configuration. We used 3 different RA-LTTng sub-buffer sizes (4MB, 2MB, and 1MB); we show user, system, and elapsed times for each experiment. The table below also shows the elapsed time overheads relative to the 4MB baseline and the number of yield calls executed.

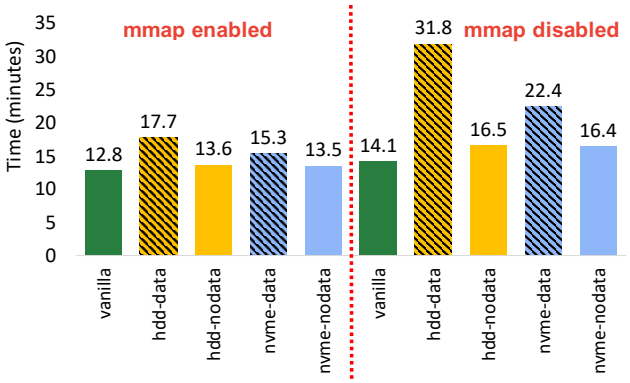


Figure 6: Results of HDD vs. NVMe devices used to record traces, along three dimensions: (i) capturing traces on HDD vs. NVMe (yellow vs. blue bars); (ii) capturing with data buffers vs. only system-call metadata (hatched pattern vs. clear); and (iii) capturing LevelDB using `mmap` vs. `read`/`write` (separated by vertical dotted line).

LevelDB using `mmap` *vs.* `read`/`write`. LevelDB uses `mmap` by default, but it can also use regular `read` and `write` system calls. To briefly demonstrate RA-LTTng's utility in investigating application behavior, we traced LevelDB using both modes. We ran LevelDB benchmarks with a 10M KV database configuration. In `mmap` mode, RA-LTTng captured around 100,000 4KB-sized page-read operations; in `read`/`write` mode, we captured nearly 9.8 million operations, mostly small `pread` requests, around 2.2–2.3KB in size (LevelDB writes small KV pairs). Many of these `pread`s read the same page-cached data repeatedly. As anticipated, we also observed that the captured DataSeries file in `read`/`write` mode was 56× larger than in `mmap` mode. These figures demonstrate RA-LTTng's usefulness: a LevelDB application developer may want to investigate running it with `mmap` (seeing only unique reads and writes) or without (seeing all repeated, cached reads, with their original sizes)—and then optimize the program.

Replaying LevelDB on different file systems. We also ran a short experiment to test RA-Replayer's utility for evaluating other file systems. We captured a trace of LevelDB with 5M KV pairs running on Ext4 and replayed it on different file systems. On XFS the system time for this workload was 18% lower, whereas on Btrfs the system time was 31% higher. Btrfs supports data compression options; when replaying with the base (LZO) compression, no space was saved on disk, because LevelDB by default *already* compresses its data, but Btrfs's `compress-force=zlib` option was able to save 17% further space—but the system time was 10% higher than without compression.

RA-LTTng blocking mode. Section 2.3 explained how we integrated blocking mode into RA-LTTng. We now show how much overhead blocking mode can introduce with different sizes of sub-buffers, as seen in Figure 5. We traced `db_bench` with its default configuration (100MB database size). We found out that without any blocking functionality, the minimum amount of sub-buffer memory we needed to ensure that RA-LTTng will not lose any events is 4MB. Considering the 4MB sub-buffer configuration as a baseline, we calculated overheads based on the elapsed time for tracing under the default event-dropping mode and our enhanced blocking mode (which does not lose any events). If we put memory pressure on RA-LTTng in blocking mode, RA-LTTng throttles the application, and we then see higher system and elapsed times. The additional system time is explained by correlation with the number of `yields` that RA-LTTng performed. Note that the reason there were (just) 53 `yields` in the 4MB configuration is that RA-LTTng starts throttling just before the sub-buffers get 100% full. The threshold for the throttling system is also configurable (we used 80%).

Tracing LevelDB on NVMe vs. HDD.. Initially our experiments used a large, inexpensive, SAS HDD to store traces. But, as we found the HDD to be too slow for large-scale tracing, we wanted to show how much performance can be improved by using a faster tracing device such as an NVMe SSD (Figure 6). Although Re-Animator was designed for high fidelity and thus to capture full data buffers, for many users it is enough to capture just system-call meta-data events (and perhaps small buffers). To explore these options, we also compared cases where we captured data buffers, captured only system-call metadata, and captured with and without `mmap` support. We configured `db_bench` for a 1GB database

and ran ten different experiments. We repeated each experiment five times and ensured that the standard deviation was below 5% of the mean. Figure 6 shows that tracing with data buffers added anywhere from 20% overhead (best case of `mmap` enabled & NVMe) to 126% overhead (worst case of `mmap` disabled & HDD) The higher overheads were because when LevelDB is configured to not use `mmap`, it issues a `(p)read` or `f(p)` write for each access and therefore invokes a large number of system calls. That causes Re-Animator to add 22–79% more overhead compared to the `mmap`-enabled versions of the same configurations. If a user wants to reduce overheads and does not need to capture data buffer contents, Re-Animator offers trace capturing with only 5–17% overhead. We observed that switching the trace-capture device from HDD to NVMe reduced the tracing overhead by 14-30% when data-buffer capturing was enabled. However, if we only captured system-call metadata (no data buffers), there was a negligible overhead difference between using HDD and NVMe as trace-capture device; that is because the HDD is fast enough when the amount of data written to it sequentially is smaller.

During these experiments, we also discovered an example of interactions between tracing performance and the behavior of the traced program. The LevelDB benchmark uses a foreground thread to exercise the database; a background thread compacts data and pushes it down the LSM tree. The foreground thread monitors the progress of compaction and uses two heuristics if compaction falls behind: (1) when a threshold of uncompacted data is reached, the foreground thread sleeps for 1ms to let the background thread run, and (2) if a higher threshold is reached, the foreground thread blocks until compaction has made significant progress. In addition, these compactions might create an avalanche of multi-level compactions [53]. By their nature, these two threads issue different numbers of system calls, which in turn means they are delayed by different amounts when we trace them. In the benchmark, the compaction thread issued more calls. When we ran the test with `mmap` disabled, we observed different behavior depending on the device used to store traces. Storing traces on the HDD slowed the compaction thread significantly more, and hence the 1ms sleep happened more frequently. In contrast, storing traces to the faster NVMe device had less impact, so that the compaction did not fall so far behind. However, while this case did complete faster than when tracing to the HDD, we also observed around 20 million more `pread` calls when storing traces on the NVMe. Interestingly, when we increased LevelDB's in-memory buffers from the default 4MB to 100MB, the behavioral difference between recording traces on HDD and NVMe disappeared. While we are still investigating the exact reasons for these effects, we believe they are due to the number of compactions: with less memory, LevelDB has to perform compaction more often, which moves more data across its layers. Kannan *et al.* [53] have also reported that increasing the in-memory buffer sizes can reduce compaction frequency. We believe that LevelDB should not be hard-coding its buffer sizes but rather should adapt them to the workload (or at least permit users to configure that amount at run time). Nevertheless, this somewhat counter-intuitive finding highlights the usefulness of tracing applications to understand their behavior.

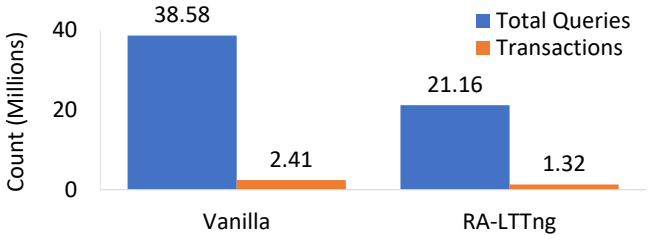


Figure 7: Counts (Millions) of MySQL queries and transactions completed within a one-hour period.

3.4 MySQL Macro-Benchmark

Figure 7 shows the counts of total queries and transactions completed within one hour by `sysbench` issuing requests to MySQL. 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. On the same test, RA-LTTng completed 21.2M queries (about 55% of Vanilla) in one hour.

3.5 Trace Statistics

DataSeries comes with a tool called `dsstatgroupby`, which can calculate useful statistics from trace files. As an example of its utility, we highlight a few helpful metrics that we extracted in our experiments. For example, the LevelDB experiment executed a total of 6,378,938 system calls (23 unique calls). 99.87% of those 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 inefficient—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 with an alternate data structure.

Similarly, the MySQL experiment executed 8,763,035 system calls (37 unique) in total. Four dominating calls—`pwrite`, `pread`, `fsync`, and `write`—accounted for 99.95% of the operations. Most `pread`s were exactly 16KB in size and thus highly efficient. There were 2.5M `fsync`s (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% (about 250) took over 100ms to complete, exhibiting tail latencies also observed by other researchers [24, 41, 47, 61].

4 RELATED WORK

System calls can be traced by using `ptrace`, by interposing shared libraries, or with in-kernel methods.

Ptrace. Because `ptrace` [37] has been part of the Unix API for decades, it is an easy way to track process behavior. `strace` [97], 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` [30] and `tusc` [11]. On Microsoft Windows, StraceNT [34] 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, the overhead is unusually high (as much as an order of magnitude).[1] 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.

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 (e.g., TraceFS [7]) can do so. RA-LTTng also captures and replays `mmap`ed events with a minimal Linux kernel modification; see Section 2.2.

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 [23, 65]. Since the shared library runs in the same process context as the application, 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 can be difficult in `chroot`ed environments where the special library might not be available; (3) trace collection in a multi-threaded process might require additional synchronization; and (4) interposing other libraries as well can be challenging.

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 [33], which uses kernel hooks to capture system-call information. Solaris supports DTrace [18] and Windows offers Event Tracing for Windows (ETW) [68]. All of these approaches capture into an in-kernel buffer that is later emptied by a separate thread or process. Since kernel memory is precious, all of these tools limit the memory they use to store traced events, and drop events if not enough memory is available. We have verified this event-drop phenomenon experimentally for both DTrace and ktrace. ETW further limits any single captured event to 64KB.

The Linux Kprobes facility [22] has been used to collect read and write operations [86], but the approach was complex and incomplete. A more thorough implementation is FlexTrace [95], 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, and the fine-grained tracing can be a disadvantage if the traces are later used for a different purpose, since desired data might not have been captured.

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 in Section 2.3, vanilla LTTng does not capture data buffers. 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).

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 RA-LTTng, consuming at least 70% more system time and at least 50% more elapsed time.

Replayer fidelity. To the best of our knowledge, no system-call replayer exists that can replay the buffers' data (e.g., to `write`). ROOT [96], 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 [67] 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), and to verify each buffer (e.g., after a `read`). If any deviation is detected, we can log a warning and 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 the logical POSIX file system state: file names and namespaces, file contents, and most inode metadata (e.g., inode type, size, permissions,

[1] We initially considered using `strace` for this project, but our evaluations showed that its overhead was at least 5–15× and often exceeded two orders of magnitude. We therefore did not consider `strace` a viable alternative for our purposes, as the overhead was too high to be considered practical.

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 hfplayer [38, 39], 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.

We have also investigated other types of recording and replaying frameworks, such as Mozilla RR [70]. Mozilla RR is designed for deterministic recording and debugging; it replays a traced execution *alongside* an actual binary: for any system call in the binary, Mozilla RR emulates it from the traced data and skips executing the actual call. RA-LTTng is different because (1) we do not require having a binary to replay, and (2) we actually want to re-execute the original system calls so as to reproduce OS and file system behavior as faithfully as possible.

Scalability. All system-call tracers can capture long-running programs, but using a binary trace format (e.g., as all in-kernel tracers do) allows such tools to reduce I/O bottlenecks and the chance of running out of storage space.

ROOT [96] 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.

Portable trace format. DTrace [18], ktrace [33], and ETW [68] 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 [35, 44, 96]. (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 [63], 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 multi-threaded 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. DataSeries comes with many useful tools to repack and compress trace files, extract statistics from them, and convert them to other formats (e.g., such as plain text and CSV). The SNIA Trace Repository [84] 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 converts CTF files to DataSeries ones offline.

Low-overhead networked storage tracing. Another approach to tracing storage systems is network monitoring [12, 52]. However, it is limited only to network file-systems and is limited to capturing information transmitted between nodes: system calls intercepted by client-side caches do not produce network activity and hence are not caught. Re-Animator, however, offers richer traces, such as capturing `mmap`-related reads and writes. Conversely, collecting traces by passive network monitoring can have low overheads.

5 CONCLUSIONS AND FUTURE WORK

Tracing and trace replay are essential tools for understanding systems and analyzing their performance. We have built Re-Animator, which captures system-call traces in a portable format and replays them accurately. Our capture method, based on LTTng, requires small kernel modifications but has and average overhead of only 1.8–2.3× compared to an untraced application; the lowest overhead was just 5%. 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 systems that employ data-dependent techniques such as compression and deduplication. The replayer verifies its actions as it performs them, ensuring that the final logical POSIX file system state matches the original. We have traced and replayed a number of popular applications and servers, comparing outputs to ensure that they are correct.

Future work. The work described in this paper concentrated on building a powerful tool, and we have provided a few examples of RA-LTTng's research usefulness. Re-Animator can be applied to research such as application performance analysis, cyber-security, and machine learning, among others, and we now plan to use it to collect and analyze long-term application traces. We also plan to integrate Linux *workqueues* to RA-LTTng for capturing CTF records and unifying the trace capturing system.

6 ACKNOWLEDGMENTS

We thank the anonymous Systor reviewers for their valuable comments. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; and NSF awards CCF-1918225, CNS-1900706, CNS-1729939, and CNS-1730726.

REFERENCES

[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] 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.

[10] 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.

[11] 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*.

[12] Matt Blaze. NFS tracing by passive network monitoring. In *Proceedings of the USENIX Winter Conference*, San Francisco, CA, January 1992. USENIX Association.

[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] Edwin F. Boza, Cesar San-Lucas, Cristina L. Abad, and Jose A. Viteri. Benchmarking key-value stores via trace replay. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 183–189. IEEE, 2017.

[16] Alan D. Brunelle. Blktrace user guide, February 2007. Available at *https://docplayer.net/20129773-Blktrace-user-guide-blktrace-jens-axboe-jens-axboe-oracle-com-user-guide-alan-d-brunelle-alan-brunelle-hp-com-18-february-2007.html*. Visited May 15, 2020.

[17] 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.

[18] 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.

[19] 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.

[20] William Chen. You must unlearn what you have learned... about SSDs, February 2014. *https://storage.toshiba.com/corporateblog/post/2014/04/07/You-Must-Unlearn-What-You-Have-Learnede280a6About-SSDs*.

[21] 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.

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

[23] 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*.

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

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

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

[27] Mathieu Desnoyers and Paul E. McKenney. Userspace RCU. *https://liburcu.org*, April 2019.

[28] 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.

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

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

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

[32] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE, 1996.

[33] 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*.

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

[35] 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.

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

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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*.

[45] 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.

[46] 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.

[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] Brijender Kahanwal and Tejinder Pal Singh. Towards the framework of the file systems performance evaluation techniques and the taxonomy of replay traces. *arXiv preprint*, arXiv:1312.1822, 2013.

[50] 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.

[51] 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.

[52] Ardalan Kangarlou, Sandip Shete, and John D. Strunk. Chronicle: Capture and analysis of NFS workloads at line rate. In *FAST*, pages 345–358, Santa Clara, CA, February 2015. USENIX Association.

[53] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, 2018.

[54] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, pages 351–366, 2009.

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

[56] 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.

[57] Geoff Kuenning and Ethan L. Miller. Anonymization techniques for URLs and filenames. Technical report UCSC-CRL-03-05, Storage Systems Research Center, Jack Baskin School of Engineering, University of California, Santa Cruz, Santa Cruz, California, September 2003.

[58] LevelDB, September 2019. *https://github.com/google/leveldb*.

[59] Bingzhe Li, Farnaz Toussi, Clark Anderson, David J. Lilja, and David H.C. Du. TraceRAR: An I/O performance evaluation tool for replaying, analyzing, and regenerating traces. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–10. IEEE, 2017.

[60] 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.

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

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

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

[64] 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.

[65] 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.

[66] 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.

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

[68] Robert H.B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 1–11, New York, NY, USA, 1993. ACM Press.

[69] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability.

In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 377–389, 2017.

[70] Oracle Corporation. MySQL. *http://www.mysql.com*, May 2020.

[71] 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.

[72] Thiago Emmanuel Pereira, Livia Sampaio, and Francisco Vilar Brasileiro. On the accuracy of trace replay methods for file system evaluation. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 380–383, San Francisco, CA, February 2013. IEEE.

[73] 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.

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

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

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

[82] Storage Networking Industry Association. IOTTA trace repository. *http://iotta.snia.org*, February 2007.

[83] 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.

[84] 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.

[85] 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.

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

[87] 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.

[88] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[89] 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.

[90] A. Traeger, I. Deras, and E. Zadok. DARC: Dynamic analysis of root causes of latency distributions. In *Proceedings of the 2008 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2008)*, pages 277–288, Annapolis, MD, June 2008. ACM.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

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

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] 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.