Using Machine Learning to Improve Operating Systems' I/O Subsystems

A Dissertation Proposal presented

by

Ibrahim Umit Akgun

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-22-02

June 2022

Stony Brook University
The Graduate School

Ibrahim Umit Akgun

We, the Dissertation proposal committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this Dissertation proposal

Erez Zadok - Dissertation Advisor
Professor, Computer Science Department

Anshul Gandhi - Chairperson of Dissertation Proposal
Professor, Computer Science Department

Aruna Balasubramanian
Professor, Computer Science Department

Geoff Kuenning
Professor, Computer Science Department, Harvey Mudd College

Michael Mesnier
Principal Engineer, Intel Labs

This Dissertation proposal is accepted by the Graduate School

Celia Marshik
Interim Dean of the Graduate School

Abstract of the Dissertation Proposal

Using Machine Learning to Improve Operating Systems' I/O Subsystems

by

Ibrahim Umit Akgun

Doctor of Philosophy

in

Computer Science

Stony Brook University

2022

Despite the ever-changing nature of computing systems, operating systems and storage systems are still following the architectures, algorithms, and structures built decades ago. Modern software stacks generate complicated and dynamic workloads which are running on statically configured storage stacks. To provide the best performance for various dynamic workloads, we need self-adaptive, dynamically configured storage systems. However, considering the current design principles of storage and operating systems, there is no support system to achieve self-adaptability.

One of the possible solutions to fulfill the self-adaptability needed in storage and operating systems is approaching operating system problems with machine learning assistance. Researchers have tried using machine learning to solve operating system problems; however, existing solutions are either not practical or not versatile enough. Therefore, we propose a complete pipeline to build machine learning models to improve operating system components, especially I/O subsystems and their performance. First, we provide a low-overhead and high-fidelity data-collection framework to trace and collect data from inside operating systems. We then develop a lightweight and efficient machine learning (ML) framework that can run at the kernel level and tune kernel parameters to improve I/O performance.

We have applied our machine learning framework, called KML, to tune disk readahead sizes according to workload-type predictions. We used RocksDB as our benchmarking platform. We can improve I/O performance for RocksDB's benchmark workloads, including realistic ones (*e.g.*, Facebook's mixgraph), by up to 2.3×. We also include another use case: NFS rsize. We observed as much as 15× performance improvements for the NFS rsize use-case.

It is our thesis that operating systems have many heuristics built largely by hand over many years, and yet operating systems cannot easily adapt to changing environment and workload conditions; therefore, we believe that compact and efficient machine learning engines should become a first-class citizen inside operating systems and be used to improve I/O subsystems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer hardware, software, storage, and workloads are constantly changing. Storage performance heavily depends on workloads and the precise system configuration [32, 157]. Storage systems and OSs include many parameters that can affect overall performance [33, 31, 198]. Yet, users often do not have the time or expertise to tune these parameters. Worse, the storage and OS communities are fairly conservative and resist making significant changes to systems to prevent instability or data loss. Thus, many techniques currently used were historically developed with human intuition after studying a few workloads; but such techniques cannot easily adapt to ever-changing workloads and system diversities.

For example, readahead values, while tunable, are often fixed and left at their defaults. Correctly setting them is important and difficult when workloads change: too little readahead wastes potential throughput and too much pollutes caches—both hurting performance. Some OSs let users pass hints (*e.g.*, `fadvise`, `madvise`) to help recognize files that will be used sequentially or randomly, but these often fail to find optimal values for complex, mixed, or changing workloads. We experimented with a variety of modern workloads and many different values of readahead: in our prior work, we confirmed that no single readahead value is optimal for all workloads [8]. Another example of tunable parameters in the network storage settings is the default read-size (`rsize`) parameter in NFS: if set too small or large, performance suffers. In addition to storage and file system layers, network and block device layers also have a lot of important knobs to tune.

Machine Learning (ML) techniques can address this complex relationship between workloads and tunable parameters by observing actual behavior and adapting on-the-fly, and hence may be more promising than fixed heuristics. ML techniques were recently used to predict index structures in KV stores [109, 48], for

database query optimization [108], improved caching [170], cache eviction policies [187], I/O scheduling [80], and more.

In this thesis proposal, we describe our ML approach to improve storage performance by dynamically adapting to changing I/O workloads. We designed and developed a versatile, low-overhead, light-weight system called *KML*, for conducting ML training and prediction for I/O subsystems. KML defines generic ML APIs that can be used for a variety of subsystems; we currently support several deep neural networks and decision-tree models. We designed KML to be embeddable inside an OS or the critical path of the storage system: KML imposes low CPU and memory overheads. KML can run synchronously or asynchronously, giving users the ability to trade-off prediction accuracy vs. overhead.

Developing and tuning ML-based applications can be its own challenge. Therefore, we designed KML to run identically in user- or kernel-level. Users can develop and debug ML solutions easily in the user level, then upload the same model to run identically in the kernel. To start developing ML models with KML, users who are OS developers should gather data from target I/O subsystem.

One of the most crucial parts of the ML pipeline is generating input data. Collecting input data from the target subsystem without introducing overheads in the operating system is critical. Because when the tracing systems impose overheads on the operating system, input data for the ML applications contains noise. Therefore noisy input data may lead to building unstable and even incorrect ML models. To address this problem, we designed a low-overhead, high-fidelity, data-collection framework called Re-Animator, to gather input data for ML pipeline.

Re-Animator is designed to trace operating system internals, and system calls with the capability of capturing buffer contents. Context-aware tracing enables operating system developers to study how workloads interact with operating systems and design workload-specific optimizations. Re-Animator also includes a system call replayer to replay system call traces in an as-is manner; this helps operating system developers to reproduce the same scenarios for debugging and testing purposes. Re-Animator's system call replayer can also be used as a benchmarking tool to mimic realistic workloads.

We propose that machine learning algorithms can be a viable solution for building self-adaptive I/O subsystems and improving I/O performance. To support our theory, we demonstrate KML's usefulness with two case studies: (i) adapting readahead values dynamically and (ii) setting NFS `rsize` values automatically. In both cases, we aim to adapt these values within one second, yet under changing and even mixed workloads. We also plan to build ML models to improve fairness and performance for TCP congestion control algorithms and I/O schedulers. This

way we will provide more proofs to demonstrate KML's generalization capability and versatility.

The rest of this thesis proposal is organized as follows. In Chapter 2, we outline our thesis statement; we also provide background information about machine learning frameworks and how the KML machine learning pipeline works. In Chapter 3, we discuss related works. In Chapter 4, we explain the details of Re-Animator design and provide performance analysis and evaluation of Re-Animator. In Chapter 5, we describe the KML machine learning framework and two use cases. We then go over our proposed and future work in Chapter 6 and conclude this thesis proposal in Chapter 7.

Chapter 2

Motivation

In this chapter, we discuss our vision (Section 2.1). In Section 2.2, we explain why we built a new ML framework from scratch instead of using off-the-shelf ones. We then describe what type of operating system components can be a good candidates for in-kernel ML applications (Section 2.3).

2.1 Thesis Statement

KML is designed to replace OS-level I/O subsystem heuristics and system parameter tuning. Thus, a KML application first observes the target component by collecting data from probes that are placed in the kernel. The data collected is used to train an ML model using KML's functionality APIs. The KML application then uses the model to predict and tune system parameters. Our use cases follow the *observe-and-tune* paradigm to reduce overhead introduced by the ML models. Therefore, we do not impose extra overheads on I/O and storage components that require low latency and predictable performance. The two KML use-cases that we detail in this thesis proposal are ML models developed (1) to tune readahead sizes on a per-disk and per-file basis and (2) to tune NFS `rsize` value. We chose these two examples because: (i) their storage components can significantly benefit from fine-tuned parameters, (ii) they require adaptation to a variety of different workloads, which is crucial in providing optimal performance, and (iii) it proves that adding asynchronous ML computation (see Section 5.3.3) will *not* negatively impact critical I/O paths.

2.2 Machine Learning Framework

Developing new solutions and optimizations for storage and OSs requires a highly efficient design that carefully considers the needs of the OS. Modern ML libraries are designed for building general-purpose ML approaches, and tend to rely on many third-party libraries (*e.g.*, in C++ or Python) to handle core ML components. This is why porting an existing ML framework to run in the kernel requires redesigning the entire ML core. Instead of porting a relatively large and complicated existing ML framework, we designed and implemented KML from scratch, enabling a low-overhead, light-weight, and offering versatility for OSs and storage systems.

We considered three different modes of operation for KML and implemented two of those. KML supports either user-space or kernel-space training—both performing kernel-space inference. One can consider moving the inference to user-space as well. However, user/kernel co-operated machine learning frameworks can become unnecessary complicated. For example, if user-space library-training threads cannot get scheduled in time, the user-space library can start losing critical training data. While this problem may still occur in our modes of operation, operating in kernel space gives us with more control over thread scheduling. Considering the high data sampling rate needed in our use cases, placing data processing and normalization in user space results in greater loss of data compared to inside the kernel. In the case of inferencing in user space, losing observation data can also cause performance problems. Because losing observation data can lead to inaccurate predictions, which creates misconfigurations for target components, these misconfigurations often end up causing performance degradation. However, KML's in-kernel architecture supports both synchronous and asynchronous inferencing with negligible latency. We still believe that a user-kernel co-operated design might be beneficial for some cases that do not require high data sampling. However user-kernel co-operated mode is not a critical feature for building functioning ML models for operating systems. That is why we are leaving the implementation of a user-kernel co-operated design to future work.

2.3 Machine Learning Applications

Operating system developers design low-latency and efficient algorithms to keep up with the desired performance expectations. That is why it is hard to for ML models to run in the critical path or decision-making parts of core OS algorithms.

Therefore, we chose I/O subsystems as an early adopter and *observe-and-tune* as our paradigm. We can utilize the I/O waiting time to collect data and run ML inferences. In this way, we can integrate ML models to I/O subsystems without adding too much overhead. To this end, we chose our proposed work related to other I/O subsystems such as networking and block device layer. We discuss use cases that are already implemented in Section 5.2 and outline our proposed work in Chapter 6.

Chapter 3

Related Work

In this chapter, we survey related works about data collection frameworks and ML approaches for operating systems.

3.1 Data Collection Framework

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

Ptrace. Because `ptrace` [75] has been part of the Unix API for decades, it is an easy way to track process behavior. `strace` [190], 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` [65] and `tusc` [21]. On Microsoft Windows, StraceNT [71] 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

[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 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 [16]) can do so. RA-LTTng also captures and replays `mmap`ed events with a minimal Linux kernel modification; see Section 4.2.2.

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 [47, 128]. 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 [69], which uses kernel hooks to capture system-call information. Solaris supports DTrace [29] and Windows offers Event Tracing for Windows (ETW) [133]. 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 [45] has been used to collect read and write operations [171], but the approach was complex and incomplete. A more thorough implementation is FlexTrace [186], 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 4.2.3, 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 [188], 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 [132] 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, 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 [76, 77], 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 [138]. 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 [188] 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 [29], ktrace [69], and ETW [133] 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 [72, 188, 86]. (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 [126], 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 [14], 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 [168] 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 [22, 101]. 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.

3.2 Machine Learning Framework

Machine learning in systems and storage In follow-up work to Mittos [79], a custom neural network was built that makes inferences inside the OS's I/O scheduler queue. The neural network decides synchronously whether to submit requests to the device using binary classification [80]. There are notable differences between that system and our KML. That system was trained offline using TensorFlow and exclusively trained in user space. Additionally, each of their two layers were custom built. Conversely, KML provides a more flexible architecture. KML training, retraining, normalization, repeated inference—all are possible and accomplished with ease in any combination of online, offline, synchronous, or asynchronous settings. Lastly, KML easily supports an arbitrary number of generalizable neural network layers; our experiments demonstrate more expressive classification abilities on a more diverse set of devices.

Laga *et al.* [112] improved readahead performance in the Linux Kernel with Markov chain models, netting a 50% I/O performance improvement in TPC-H [181] queries on SATA-SSDs. In contrast, our experiments ran on a wider selection of storage media (NVMe-SSD and SATA-SSD) and workloads. In TPC-H, we show improvements up to 39% despite TPC-H being a completely new workload for our readahead model. Moreover, our results illustrate that our readahead model can improve I/O throughput by as much as 2.4×—all while keeping memory consumption under 4KB, in comparison to Laga *et al.*'s much larger 94MB Markov chain model.

Parameter tuning for storage and operating systems has been a challenge and researchers approached this problem using control theory [161] and data distribution analysis for storage clusters [4]. Some research has attempted to apply ML techniques to OS task scheduling [136, 39], with small reported performance improvements (0.1–6%). Nevertheless, it is becoming increasingly popular to apply ML techniques to storage and OS problems including: tuning SSD

configurations [117], memory allocation [129], TCP congestion [61], building smart NICs [160], predicting index structures in key-value stores [109, 48], offline black-box storage parameter optimization [34], reconfigurable kernel data-paths [148], local and distributed caching [187, 170], database query optimization [108], and cloud resource management [46, 51, 52, 164].

Machine learning libraries for resource-constraint systems A myriad of ML libraries exist—some general purpose and others more specialized. Popular general-purpose ML libraries include Tensorflow [2], PyTorch [141], and CNTK [44]. Conversely, libraries like ELL [63], Tensorflow Lite [177], SOD [163], and Dlib [58] specialize to run on resource-constrained or on-device environments, KML differentiates itself by targeting OS-level applications and designed for OS and storage systems specifically. Inside the OS, resources are *highly* constrained, prediction accuracy is vital, and even small data-path overheads are unacceptable.

Adapting readahead and prefetching Readahead and prefetching methods are both well-studied problems [158, 159, 56, 110] and see use in distributed systems [114, 180, 38, 59, 121, 135, 120, 42]. Many have attempted to build statistical models to optimize and tune systems [158, 159, 68]. However, the main limitation of statistical models is their inability to adapt to novel new workloads and devices. We have shown that our model can adapt to *never-before-seen* workloads and devices. Another way to improve a readahead system is to predict individual I/O requests and file accesses by observing workload patterns [56, 110, 194, 183, 87, 11, 189, 196]. Predicting file accesses using hand-crafted algorithms is a reasonable first approach. However, such manual labor simply cannot keep up with the diversity and complexity of ever-changing modern workloads. Conversely, as long as we have training data, ML models can adapt, retrained as needed, and optimize much faster. Simulations are also viable solutions for readahead and prefetching problems [70, 36, 151, 195, 203]. However, simulations are computationally expensive and are limited to the datasets that the models are trained and tested with. Additionally, the models produced in simulations are not designed for resource-constrained environments, making it non-trivial to migrate such models to the kernel. It is possible to use a user-space library to intercept file accesses [193] or to require application-level changes [199]. In contrast, KML requires no application changes and is capable of intercepting `mmap`-based file accesses.

Finally, while techniques exist to improve NFS performance, we are unaware

of automated ones that use ML [96].

Chapter 4

Re-Animator: Data collection framework

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.

4.1 Introduction

Modern applications are becoming ever more intricate, often using 3rd-party libraries that add further complexity [81]. Operating systems have multiple layers of abstraction [18, 156] and deep network and storage stacks [149, 19, 83]. In ad-

dition, storage systems employ techniques like compression, deduplication, and bit-pattern elimination [192, 107, 162, 130, 182, 23, 62, 106, 166, 176, 167]. 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` [190] can record events for human analysis. Such traces can be replayed [188] 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 [178, 25, 95, 204, 175, 77, 86, 98, 116, 137, 143]), or to stress-test other components (e.g., the OS or storage system) [1, 6, 7, 17, 41, 73, 88, 90, 100, 144, 143, 147, 174, 179, 202, 201]. Traces can also be analyzed offline (e.g., using statistical or machine-learning methods) to find performance bottlenecks, security vulnerabilities, etc. [146, 85, 145], or identify malicious behavior [67, 105]. 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) [40, 50, 118, 82, 94, 197, 200].

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 nonexistent, and sometimes no software or tools are released to process, analyze, or replay the traces. Some traces (e.g., those from the Sprite project [140]) have been preserved but can no longer be read due to a lack of tools. (4) Some tools (e.g., `strace` [190]) produce output intended for human consumption and are not conducive to automated parsing and replay [72, 188, 86].

In this chapter, we make the following six contributions: **(1)** We have designed and developed Re-Animator, a system-call tracing package that uses Linux tracepoints [53] and LTTng [127, 54] 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 [14], 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 for-

mats, 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.

4.2 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 [111, 16], but privacy is outside this chapter's scope.

4.2.1 Goals

Fidelity. State-of-the-art techniques for recording and replaying system calls have focused primarily on timing accuracy [15, 27, 204, 95, 132, 81, 188]. 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 chapter 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 [15]. 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 [132] or heuristic [188] techniques to extract internal dependencies. The current version of Re-Animator uses a conservative heuristic similar to *hfplayer* [76]: 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 [172, 124], compression [107, 28], repeated bit-pattern elimination [167], 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 4.2.2.

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 [29], ktrace [69], and SysDIG [24]—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 [127, 54], an efficient Linux tracing facility [53]. 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 [14] 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.

4.2.2 Fidelity

Re-Animator is based on LTTng [127], an extensible Linux kernel tracing framework. LTTng inserts *tracepoints* [53] 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 [55] for lockless synchronization with the kernel. The data is written in Linux's Common Trace Format (CTF) [126], which the `babeltrace` tool converts to human-readable text.

Figure 4.1 shows LTTng's flow for tracing and capturing calls; green components denote our changes. For ease of use, a wrapper (Figure 4.1, 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 4.1, 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 placed on a different, larger or faster storage

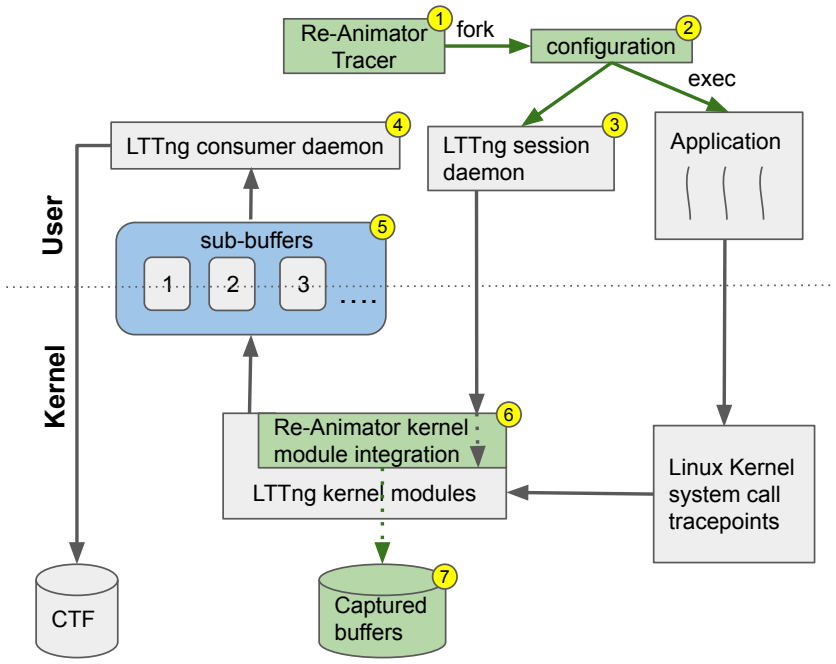


Figure 4.1: LTTng architecture using Linux kernel tracepoints. Green boxes denote our additions or changes. Our wrapper (1) launches the LTTng configurator (2), which invokes an LTTng session daemon (3) to control the operation and the consumer daemon (4) to collect events. LTTng tracepoints place events into sub-buffers (5) and invoke Re-Animator, which collects data buffers and writes them to a separate disk file (7).

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 chapter, we refer to this enhanced CTF format with secondary buffer-data files as *RA-CTF*.

We modified `babeltrace` to generate the DataSeries format [14], 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 [16], 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 `mmap`ed 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 4.2.3. 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 `mmap`ed 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`.

4.2.3 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 4.2.2. 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 appli-

cations 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 4.3.3.

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.

4.2.4 Verifiable

We have explained how Re-Animator captures buffers accurately and efficiently in Sections 4.2.2 and 4.2.3. 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 sev-

eral micro- and macro-benchmarks (see Section 4.3) 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 `write`s 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.

4.2.5 Portable

To allow our tools to be used as widely as possible, we capture and replay in DataSeries [14], 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 [169], 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.

4.3 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.

4.3.1 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 chapter, 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 4.2.2).

Micro-benchmarks. To capture traces, we first ran the FIO micro-benchmark [66], 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 [13] but omit the results since they did not differ much from FIO's.)

Macro-benchmarks. We ran two realistic macro-benchmarks: (1) LevelDB [115], a key-value (KV) store with its own `dbbench` exerciser. We ran 8 different preconfigured 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 [139] with an 8GB database size. We configured `sysbench` [173] to run 4 threads that issued MySQL queries for a fixed one-hour period.

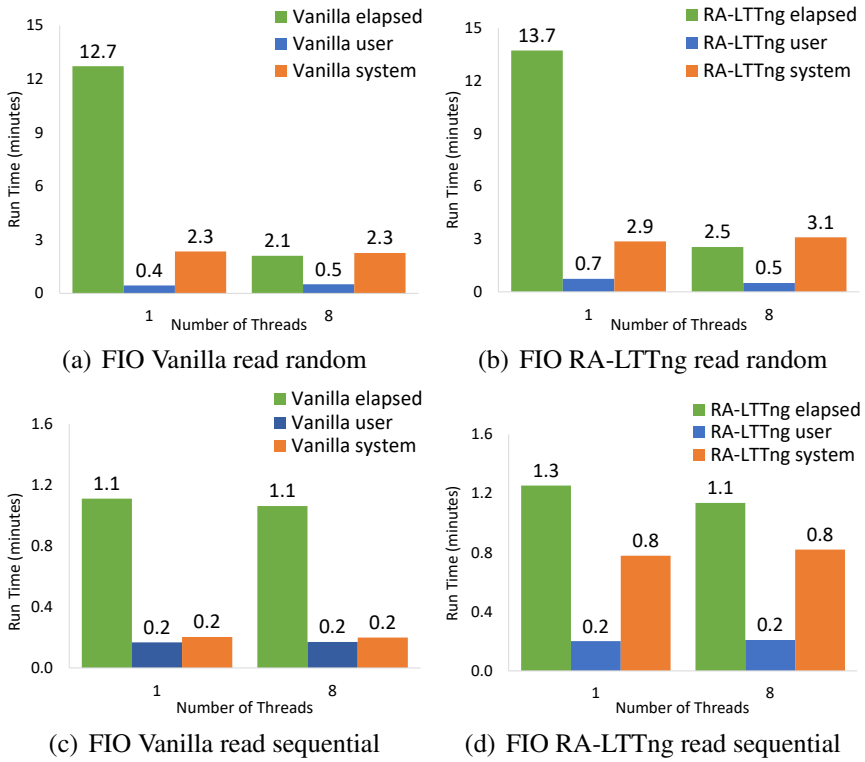
Format conversion. Recall that RA-LTTng stores traces using our enhanced RA-CTF format (Linux's CTF for system calls, slightly enhanced, plus separate binary files to store system-call buffers); therefore we wrote `babeltrace2ds`, which converts RA-CTF traces to DataSeries format before replaying the 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.

4.3.2 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 4.2 and 4.3 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 [99, 57, 37]. We focus on the single-threaded results



(a) FIO Vanilla read random

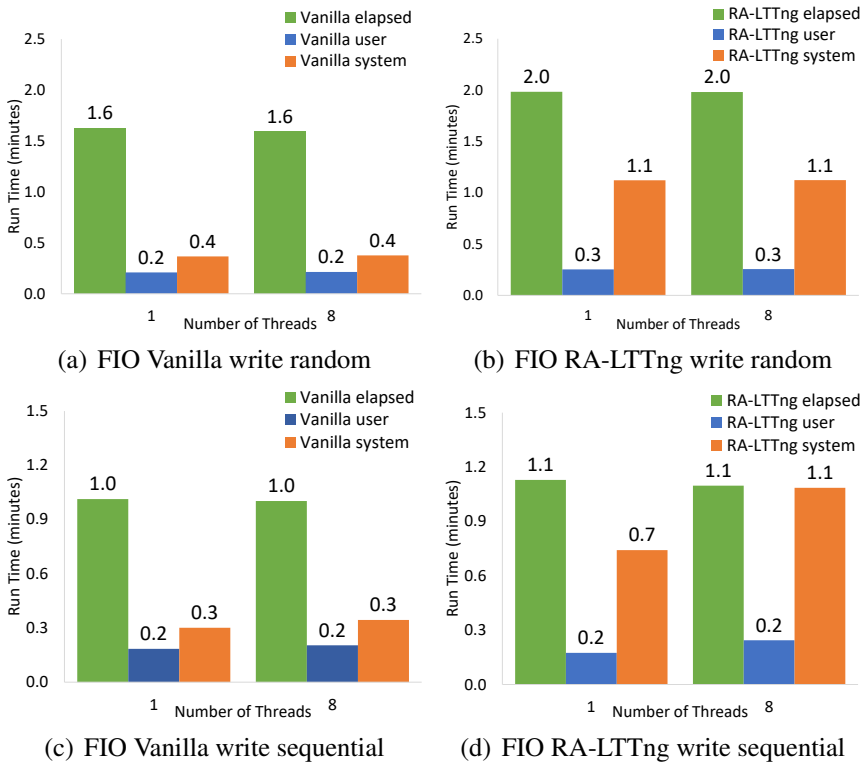
(b) FIO RA-LTTng read random

(c) FIO Vanilla read sequential

(d) FIO RA-LTTng read sequential

Figure 4.2: 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.

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 4.2(a)) was about 10× longer than the other three FIO runs. This made all elapsed times in Figures 4.2(a) and 4.2(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×.



(a) FIO Vanilla write random

(b) FIO RA-LTTng write random

(c) FIO Vanilla write sequential

(d) FIO RA-LTTng write sequential

Figure 4.3: 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.

4.3.3 LevelDB Macro-Benchmark

We ran LevelDB on a 1GB database, using 4 threads and the default sequence of phases described in Section 4.3.1. 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.

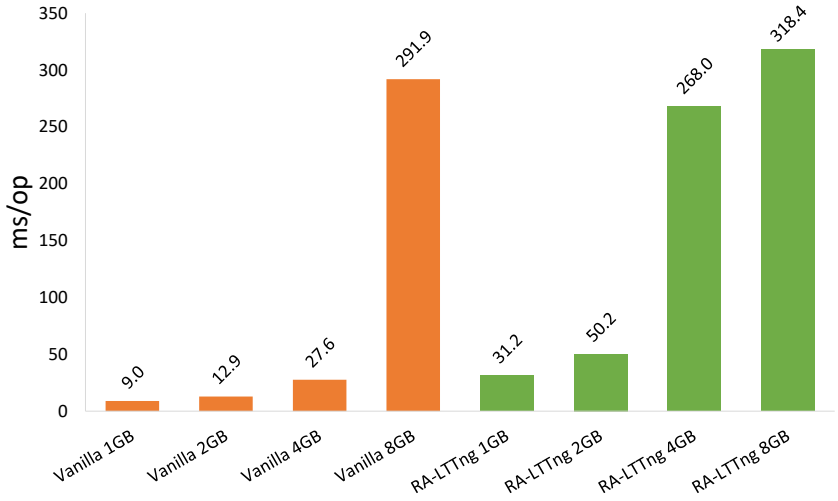


Figure 4.4: LevelDB read-random latency for different-sized databases (in milliseconds per operation).

Figure 4.4 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 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 4.3.1; the elapsed time was 81 seconds. The DataSeries file for this experiment was 25GB.

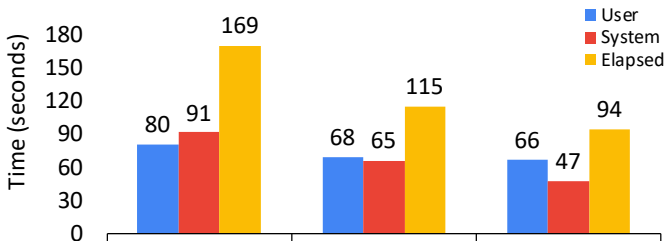
We verified the logical POSIX file system state after replaying this trace; it was identical to the original LevelDB run.

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.

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 4.2.3 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 4.5. We traced `db_bench` with its default configuration (100MB database size). We found out



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

Figure 4.5: 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.

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 4.6). 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

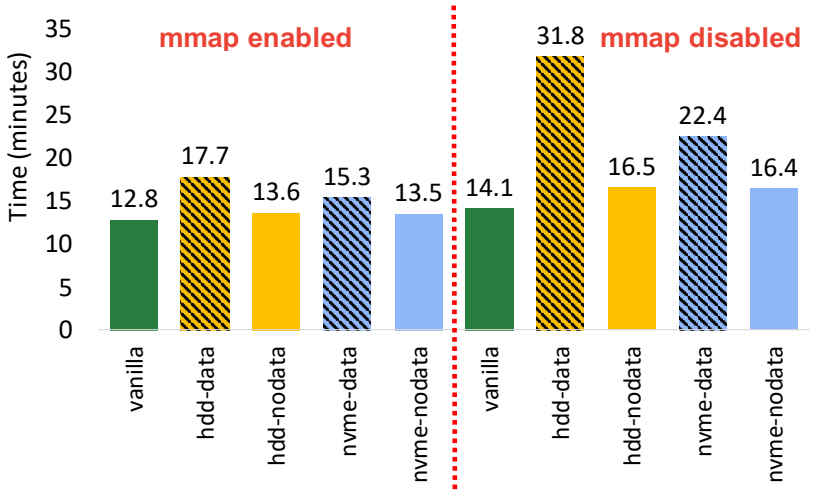


Figure 4.6: Results of HDD vs. NVMe devices used to record traces, along three dimensions: (i) capturing traces on HDD vs. NVMe (yellow vs. blue bars); (ii) capturing with data buffers vs. only system-call metadata (hatched pattern vs. clear); and (iii) capturing LevelDB using `mmap` vs. `read`/`write` (separated by vertical dotted line).

and ran ten different experiments. We repeated each experiment five times and ensured that the standard deviation was below 5% of the mean. Figure 4.6 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 Lev-

elDB 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 [102]. 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.* [102] 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.

4.3.4 MySQL Macro-Benchmark

Figure 4.7 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.

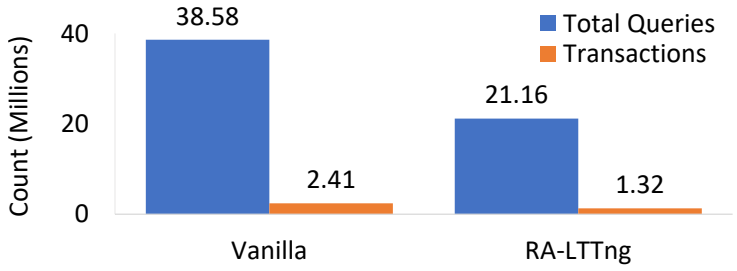


Figure 4.7: Counts (Millions) of MySQL queries and transactions completed within a one-hour period.

4.3.5 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 [50, 118, 82, 94].

4.4 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 over-

head 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 chapter 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.

Chapter 5

KML: ML Framework for Operating Systems

Operating systems include many heuristic algorithms designed to improve overall storage performance and throughput. Because such heuristics cannot work well for all conditions and workloads, system designers resorted to exposing numerous tunable parameters to users—thus burdening users with continually optimizing their own storage systems and applications. Storage systems are usually responsible for most latency in I/O-heavy applications, so even a small latency improvement can be significant. Machine learning (ML) techniques promise to learn patterns, generalize from them, and enable optimal solutions that adapt to changing workloads. We propose that ML solutions become a first-class component in OSs and replace manual heuristics to optimize storage systems dynamically. In this chapter, we describe our proposed ML architecture, called KML. We developed a prototype KML architecture and applied it to two case studies: optimizing read-ahead and NFS read-size values. Our experiments show that KML consumes less than 4KB of dynamic kernel memory, has a CPU overhead smaller than 0.2%, and yet can learn patterns and improve I/O throughput by as much as 2.3× and 15× for two case studies—even for complex, never-seen-before, concurrently running mixed workloads on different storage devices.

This chapter makes five contributions:

1. We show that lightweight ML can indeed become a first-class citizen inside storage systems and OSs;

2. We offer flexibility through synchronous or asynchronous training and the ability to offload training to the user-level;

3. We introduce the idea of generic ML APIs that can be expanded to support additional and future ML techniques;

4. We apply KML to two important optimization problems (readahead and NFS `rsize` values); and

5. We evaluate our solutions using multiple, complex, and even mixed workloads, as well as two different storage devices. We demonstrate throughput improvements up to 2.3× for readhead and up to 15× for `rsize`. We show that ML models trained on a few workloads can generalize and optimize throughput for never-before-seen workloads or devices. And finally, we show that KML has small CPU overheads ($< 0.2\%$) and dynamic memory footprint (4KB), well worth the overall I/O improvements.

Next, Section 5.1 describes KML's design. Section 5.2 describes our two use cases (readahead and NFS `rsize`). Detailed evaluation of KML and two use cases are in Section 5.3. We describe possible future works in Section 5.4.

5.1 KML's Architecture

Modern ML libraries are often general-purpose, rely on many large third-party libraries (*e.g.*, in C++ or Python), and designed to process lots of data using massive processing power (*e.g.*, GPU clusters). Porting such ML systems to an OS kernel would be impractical, because an OS is a highly constrained and unforgiving environment. Thus, we chose to develop an ML framework from scratch—designed for low-overhead, light-weight, and highly tailored to OSs and storage systems and OS developers.

KML high-level design choices Figure 5.1 demonstrates two different operating modes that we built. KML supports (a) in-kernel training and inference and (b) user space offline training and in-kernel inference. Once a model is built in user space, it can be loaded into the kernel as is. KML has a highly modular design: the core ML code base is shared by both user and kernel space. Operation mode (a) is designed for performance and accuracy, especially under high-I/O rates, because collecting and copying lots of I/O event data out of the kernel imposes high overheads. Operation mode (b) is designed to simplify ML model development for OS/storage developers. Users can develop and test an ML model

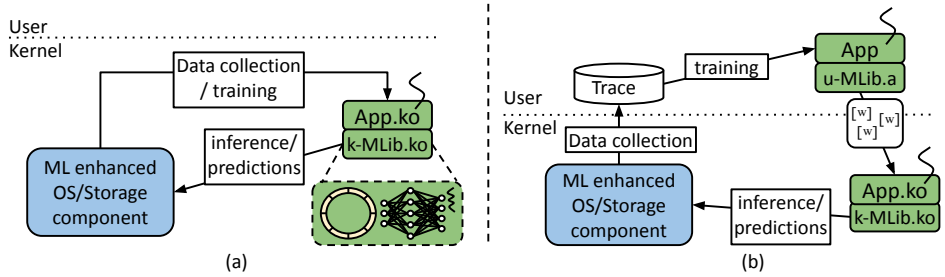


Figure 5.1: Two different operational modes that we built to achieve a high efficiency ML framework for tuning OS-level storage systems: (a) kernel space training and inference and (b) offline user space training and kernel space inference.

design more easily in user space, testing different features, ML architectures, and hyper-parameters to reach a stable and accurate model.

5.1.1 Design Overview

Easy to develop and extend In Figure 5.1(b), KML is compiled and linked with an application for both kernel and user space. `u-MLib.a` and `k-MLib.ko` are built using the same KML source code. We developed a wrapper layer for the KML development API: KML's core code is uniform across both user and kernel APIs. This identical abstraction speeds up development, eases debugging, and facilitates extensibility (see Section 5.1.3). Nevertheless, we recognize that while we aim to make ML-based solutions easier to use, developers still require a good understanding of OS and storage system internals.

Low overhead To make ML approaches practical for storage systems, they must have low computational and memory overheads. ML solutions have three phases that consume much memory/CPU resources: (i) inference (*i.e.*, prediction), (ii) training, and (iii) data processing & normalization. We support asynchronous training and inference capabilities to reduce interference on the data path; KML also uses efficient communications between the data collection and model training & inference components, to help scalability and stability of ML-based designs. To reduce the data collection overheads, developers can facilitate subsampling techniques that are provided in KML. We detail our design choices to reduce these overheads in Section 5.1.4.

5.1.2 Fundamentals of Core ML library

KML provides primitives for building and extending ML models. This involves building algorithms for training ML models (*e.g.*, back-propagation, decision-tree induction) and building the mathematical functions needed to implement them. The library design allows for seamless extensibility of library functionality. Additionally, our ML functionality is easily debugged in user space as it uses identical code and APIs in kernel space.

Mathematical and matrix operations Most ML algorithms rely heavily on basic mathematical functions and matrix algebra. For example, a neural network classifier uses functions such as matrix multiplication/addition, `softmax`, and exponentiation. Hence, we implemented kernel versions of such common ML functions using well known approximation algorithms.

Layer and loss-function implementations One can think of a neural network as a composition of layers and one or more loss functions. Many of these building blocks are used across many different neural network architectures. Layers like a fully connected layer, ReLU [134], or sigmoid are essential building blocks of many neural networks; loss functions are also fairly common across many applications. Both layers and loss functions implement two main functionalities, one during the inference (forward) phase and another during the back-propagation (training) phase. We implemented these common components and their forward and back-propagation functionality from scratch in KML: layer/loss functions, data structures related to the layer/loss, etc.

Inference and training When stacked together, the elements of a conventional neural network can form a DAG. Thus, a neural network inference means traversing the DAG starting from the initial node(s) (where the inputs are provided), toward the resulting nodes (where the neural network output is produced). KML implements a standard training method used in neural networks—back-propagation [153]. KML also includes Stochastic Gradient Descent (SGD) which uses the gradients computed using back-propagation to optimize the neural network weights.

5.1.3 KML's Modular Design

We now elaborate on KML's operation modes: (i) in-kernel training and inference (see Figure 5.1(a)), and (ii) user space training and in-kernel inference (see

Figure 5.1(b)).

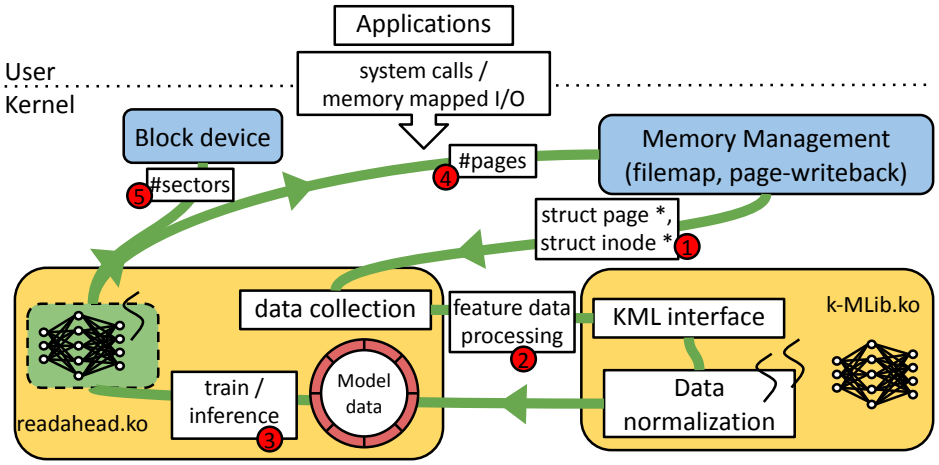


Figure 5.2: KML kernel space training and inferencing architecture.

Training in kernel space We use the readahead use case to describe how KML works in kernel training and inference mode. Figure 5.2 shows KML's framework (`k-MLib.ko`), a KML application (`readahead.ko`), and target storage components (Block device and Memory Management subsystems). The yellow background denotes KML related components. The blue background depicts the target storage components, which are specific to the readahead case. The green line represents execution and data flow. Numbered boxes refer to transitions happening between the components.

As we mentioned in Section 4.1, we designed our use cases based on a continuous *observe-and-tune* principle. In its first stage, the readahead module observes and collects data. Since our target component is the memory management (*e.g.*, page cache) system, the readahead module starts collecting data from this component (Figure 5.2 ①). The readahead module then extracts features and transfers them to the KML framework to be normalized (Figure 5.2 ②). After the data processing and normalization stage is done, if the readahead module is operating in training mode, it trains on the normalized data, and the execution flow ends here. However, if the readahead module is operating in inference mode, it feeds the normalized data to the readahead neural network model and tunes the target components based on the model's prediction (Figure 5.2 ③).

How a KML application optimizes a target component depends on the problem and its solution. Here, the readahead module updates readahead sizes on a per-file basis (Figure 5.2 ④) or a per-device basis (Figure 5.2 ⑤). When the readahead module is inferencing, execution flow forms a *closed-circuit*. After the readahead module changes readahead sizes, OS memory state changes; thereafter, new inputs go to the readahead neural network model, leading to updated predictions. Therefore, ML is particularly suitable to solve problems that require an ongoing cycle of observing and tuning.

In the ML ecosystem, data collection is a crucial part. One reason we offer kernel training is to train on data collected with a high sampling rate. Tracing OSs and storage systems with high accuracy and sampling rates is challenging [9]. Nevertheless, tracing tools like LTTng [127] can bring overhead down to as little as 5%. Additionally, traces may still be inaccurate due to data loss. LTTng collects trace data in shared user/kernel lockless circular buffers; under heavy sampling loads, some trace events can be dropped if LTTng's user-level processing threads do not consume the samples fast enough. However, operating in kernel space gives KML more control over thread scheduling to reduce loss of sampled events. Since our use cases may require high sampling rates for I/O events, placing data processing and normalization in user space would lose too much valuable data than in the kernel. Still, we believe a user-kernel co-operated design may be beneficial in some cases (part of our future work).

Training in user space Building ML solutions is an iterative process. To find the essential features and build accurate models, we need to run multiple data analyses, train, then test an ML model with different architectures and hyper-parameters. To speed up model development and debugging, KML offers offline user-space training and kernel inferencing mode (see Figure 5.1(b)). As KML's user- and kernel-space libraries use the same APIs and code base, models trained in user space can be loaded into the kernel as is.

Figure 5.3 shows how the readahead model works in operation mode. Components highlighted in yellow represent KML-specific implementations. The red arrows denote the offline data collection and training paths.

We started by collecting training data using in-kernel tracing of the target storage components [9]. Next was feature-extraction; this is where user-space training was useful, because we could run various analyses, test different features, and implement many data-normalization techniques without re-running I/O experiments. After we finalized the feature selection, we trained and tested the readahead ML

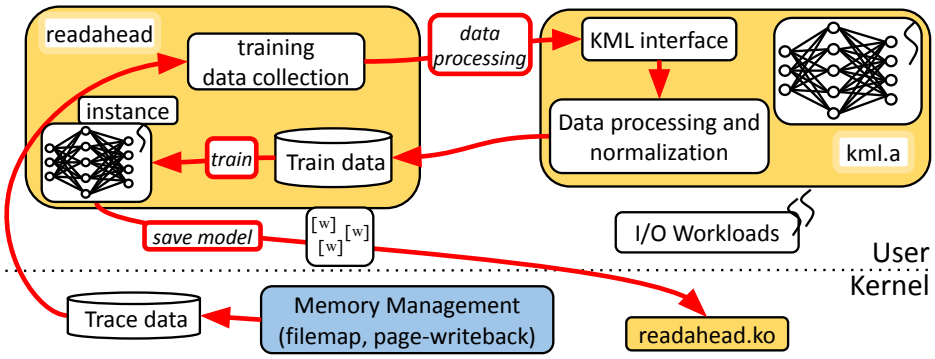


Figure 5.3: KML user-space training & kernel-space inference architecture.

model in user space, varying several hyper-parameters; we used Tune [122] to optimize our hyper-parameters. When the readahead ML model was ready for real-time testing, the only remaining step was to save the trained model to a KML-specific file and load it into the readahead kernel module. KML APIs facilitate all the functionality necessary for building, training, saving, and deploying ML models in-kernel.

To ensure identical kernel and user APIs, we use wrappers to abstract external functionality. KML's development API provides 30 functions that fall into five categories: (i) memory management, (ii) threading, (iii) logging, (iv) atomic operations, and (v) file operations. For example, we have a simple wrapper called `kml_malloc` that calls `malloc` in user-level and `kmalloc` in kernel space. For brevity, full API details and prototypes are omitted, but are included as part of our released code (see Section 5.1.6); Table 5.1 presentes a few examples of the KML API.

```
loss *build_loss(void *internal, loss_type type);
void add_layer(layers *existing_layers, layer *new_layer);
void create_async_thread(model_multithreading *multithreading,
                         model_data *data, kml_thread_func func, void *param);
sgd_optimizer *build_sgd_optimizer(float learning_rate,
                                   float momentum, layers *layer_list, loss *loss);
```

Table 5.1: KML API examples

5.1.4 Computational & Memory Overheads

OSs and storage systems are susceptible to performance degradation and increased latency if computational and memory resources are not carefully managed. Therefore, we designed KML with efficient CPU and memory usage in mind. There is often a positive correlation between the computational and memory footprint of an ML model and its training and inference accuracy. Hence, KML is highly configurable, letting users trade-off overheads vs. prediction accuracy to best suit the problem at hand.

Reducing computational overheads Matrix manipulation is a computationally intensive ML building block that relies on floating-point (FP) operations. OSs often disable the floating-point unit (FPU) in the kernel to reduce context-switching overheads. To address this, we considered three approaches: (1) quantization, (2) fixed-point representations, and (3) temporarily enabling the FPU unit in kernel space. Quantization provides compact representation, allows developers to compute matrix manipulation operations, and does not require an FPU [43, 74, 154, 49, 80]. Quantization can help reduce computational and memory overheads, but it reduces accuracy [89]. Fixed-point representation computes FP operations using integer registers. Since all FP operations are emulated, integration of fixed-point representation is fairly easy and even faster in certain cases [39, 123]. However, fixed-point representation works within fixed ranges which can result in numerical instability [113]. Since both accuracy and stability are vital KML design goals, we chose a third alternative: KML temporarily enables the FPU in the Linux kernel using `kernel_fpu_begin` and `kernel_fpu_end`. To avoid context-switch overheads, we minimize the number of code blocks that use FPs and keep these blocks small.

Reducing memory overheads Three factors affect KML's dynamic memory consumption: (1) ML model-specific data, (2) KML's internal memory allocations at training and inference, and (3) data collection for both training and inference. ML model-specific data and KML's internal memory usage depends on the number of layers, layer sizes, and layer types. KML uses dynamic memory allocation for all internal usage purposes (*e.g.*, layer gradients); this helps reduce interference and memory pressure. KML gathers input data in a lock-free circular buffer; then, an *asynchronous training thread* trains on gathered data. When collecting data with a high sampling rate, the size of the lock-free circular buffer

is important to the ML model's performance and accuracy. Users need to configure the size of the circular buffer to account for the data sampling rate such that the asynchronous training thread can catch up with processing. If the size of the circular buffer is misconfigured, KML may lose useful training data, which can reduce the resulting ML model's accuracy.

Operating under resource-constrained conditions KML exposes a memory allocation and *reservation* API for ML internals. The primary motivation behind KML's memory reservation capabilities is to ensure predictable performance and accuracy, even under memory pressure. This allows KML to operate without worry of memory allocation lagging or failing, which would hurt performance and accuracy.

Data processing & asynchronous training To make ML solutions generalizable, data normalization is often utilized. KML supports data normalization functionalities such as moving average, standard deviation, and Z-score calculation. However, data normalization often requires heavy FP computation. Thus, KML supports offloading training, inference, and data normalization to a separate *asynchronous thread*—away from the data path itself. This thread communicates with other KML components (*e.g.*, data collection) using a lock-free circular buffer. By default, we let Linux schedule this kthread as needed; KML also supports pinning the kthread to a CPU core, to ensure it gets higher scheduling priority when high sampling rates are required.

Subsampling is another viable solution to reduce data collection overheads, which KML supports. However, subsampling can reduce prediction accuracy, so care is needed to select a suitable sampling rate. In Section 5.3.3 we evaluate the impact of subsampling windows on overheads, prediction accuracy, and overall I/O performance.

5.1.5 Stability & Explainability

Both the training and inference phases for ML solutions can be computationally intensive. Except for model initialization and saving models to files, KML APIs involve no other I/Os. KML's impact on the stability of storage performance is limited to memory-allocation latency and concurrency. Memory allocations in both user and kernel space can use locking mechanisms, which could incur unexpected latencies. To minimize these problems, KML allocates memory only

in the asynchronous training thread. KML uses a lock-free circular buffer for data communication and reserves 512 bytes of additional memory to further ensure stability under memory-pressure conditions. Lastly, we applied standard k-fold cross validation techniques to ensure the stability of our ML solutions.

ML solutions can suffer from unexpected behavior and are harder to explain. Conversely, traditional heuristics have well-defined behaviors often expressed as closed-form formulas. An ML algorithm may behave erratically when used in new, unforeseen settings, which could hurt system performance where ML is deployed. This type of issue is difficult to troubleshoot due to the long-standing explainability problems that affect ML models [5]. KML currently supports two ML models: neural networks and decision trees. Decision tree predictions are more explainable because they are represented as a tree of successive IF-THEN statements, bisecting the range of the features considered. Deep neural networks, however, are more challenging to explain and verify. Nevertheless, recent work focuses on explainability in ML [93, 155, 150, 5]. While we plan to improve KML model stability using feedback-based control algorithms in the future, we currently focus on demonstrating that ML *can* tune storage system parameters *better* than existing heuristics.

5.1.6 Implementation

KML contains 12,213 lines of C/C++ code (LoC). KML's core ML part has 5,539 LoC, which can be compiled in both user and kernel space. Our readahead neural network model code is nearly 1K LoC long: 486 LoC for collecting data, initializing the model, creating an inference thread, and changing block-level and file-level readahead sizes; and another 351 LoC for model definition, data processing, and normalization. Our NFS neural network model also includes nearly 1K LoC: 435 LoC for data collection, model initialization, and running inference to predict workload type; and 338 LoC for creating the model and manipulating data.

All of our code has been released on GitHub (https://github.com/sbu-fsl/kernel-ml), which includes examples, sample data, models, and full API documentation (all 30 methods).

5.2 Use Cases

We now detail our two use cases: (1) readahead neural network and decision tree models and (2) NFS neural network model. We describe the following for each: (i) problem definition, (ii) data collection for training, (iii) data preprocessing and feature extraction, and (iv) building the ML model.

5.2.1 Use Case: Readahead

Problem definition Readahead is a technique to prefetch an additional amount of storage data into the OS caches in anticipation of its use in the near term. Determining how much to read ahead has always been challenging: too little readahead necessitates more disk reads later and too much readahead pollutes caches with useless data—both hurt performance. The readahead value is a typical example of a storage system parameter: while tunable, it is often fixed and left at its default. Some OSs let users pass hints via `fadvise` and `madvise` to help the OS recognize files that will be used purely sequentially or randomly, but these often fail to find optimal values for varied, mixed, or changing workloads. Next, we detail our readahead neural network design (following Figure 5.3). Our goal is to predict optimal readahead sizes while running under dynamic I/O workloads.

Studying the problem We experimented with running 4 different RocksDB [64] benchmarks, each with 20 different readahead sizes (8–1024), and attempted to determine the readahead sizes that yield the best performance (in ops/sec) for each workload. This became our training data, which can help predict readahead values for *other* workloads and environments. This investigation revealed that each workload has a unique behavior and requires a different readahead size to reach optimal performance. We further investigated the correlations between file access patterns, RocksDB workload labels, and performance. This helped us determine the information and features needed to build a good model, as described below.

Data collection We used LTTng [127] to collect trace data, which we then used for finding useful features for the readahead problem. We captured most page cache tracepoints [53] (*e.g.*, `add_to_page_cache`, `writeback_dirty_page`). We collected and processed over 20GB of traces by running multiple 10-minute RocksDB benchmarks on an NVMe-SSD device. Ten minutes was sufficient for RocksDB to reach a steady state. After examining these traces, we selected a set

of candidate features based on our domain expertise. We then picked the features of interest and decided where to call hook functions which are responsible for gathering necessary information (*e.g.*, `struct page`) for inference. Our hook functions provide three important raw values: (1) time difference from the beginning of execution, (2) `inode` number, and (3) page offsets of the files that were accessed in locations where the hooks were called.

Data preprocessing & normalization We summarize the input data at one-second intervals to ensure we can quickly adapt to changing I/O workloads while ensuring stability under short-term activity spikes. Based on our domain expertise, and through model experimentation, we selected the following five features for our model: the number of transactions taking place each second, the calculated cumulative moving mean and the cumulative moving standard deviation of page offsets, the mean absolute page offset differences for successive transactions, and the `inode` number (to ensure we process only RocksDB file accesses). Before we fed these features to our readahead neural network, we applied Z-score normalization to each feature.

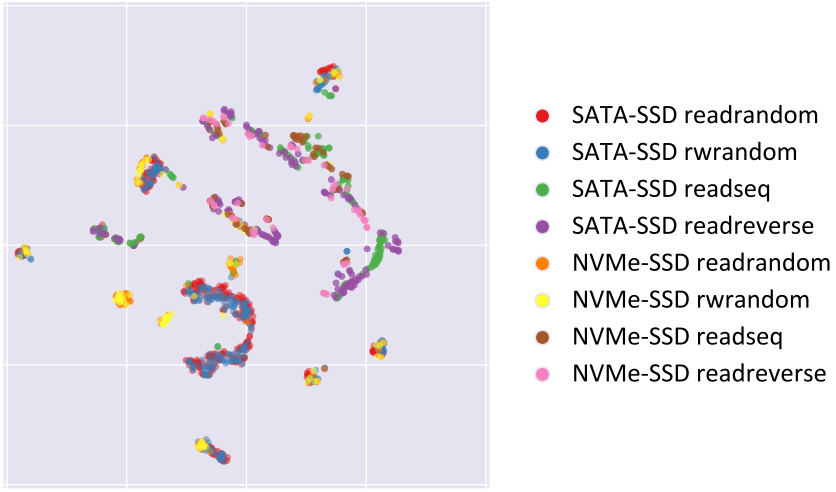


Figure 5.4: t-SNE visualization of readahead normalized features that are generated from both NVMe-SSD and SATA-SSD traces. Axes are intentionally omitted because the dimensions are generated by t-SNE and do not represent any specific data.

Why we chose machine learning for this use case After studying the readahead problem, we wanted to explore whether machine learning would be suitable for solving this problem or whether more traditional heuristics could still work. Therefore, while extracting features from collected traces, we visualized the features to investigate what type of patterns and clusters the data has. Figure 5.4 shows a t-SNE [184] visualization of normalized features that are generated from both NVMe-SSD and SATA-SSD traces. t-SNE is a visualization technique that applies dimension reduction and is often used for representing high-dimensional data and cluster identification. We can observe that sequential and random workloads are somewhat separated; alas, data points from the same workload type are distributed over multiple clusters, overlapping clusters of other types. Worse, random workloads' clusters overlap with some sequential workloads' clusters, because RocksDB's warm-up phases involve mostly sequential accesses—another source of dynamism. All these findings strongly suggest that workload classification for the readahead problem would be fairly challenging using traditional heuristics. Hence, we felt motivated to explore ML solutions to solving the readahead problem.

Building neural network model We modeled the readahead problem as a classification problem and designed a neural network with three linear layers (with hidden layer sizes of 5 and 15), using sigmoid non-linearities in between layers, and with a cross-entropy loss method as the loss function. We used an SGD optimizer [152, 103], and set a learning rate of 0.01 and a momentum of 0.99 after trying different values; all these values are common in the literature [20]. We also used Tune [122] to optimize the learning rate and momentum. Our readahead neural network trains on the aforementioned input data and predicts the workload type. We trained on the following four types of RocksDB workloads on NVMe-SSD because they provide a diverse combination of random and sequential operations: (i) readrandom, (ii) readseq, (iii) readrandomwriterandom, and (iv) readreverse. Class frequencies were close, suggesting that classification accuracy is a good metric to evaluate the performance, with the least frequent class being 21.4% and most frequent class being 28.8%.

We tested the neural network's performance with the aforementioned data via k-fold cross validation with $k = 10$, and found out that it achieved an average accuracy of 95.5%. We also analyzed the contribution of each feature to the classification performance; we randomized the order of a feature of interest across samples in the validation dataset, and then calculated the 10-fold validation perfor-

mance [26]. Using Pearson correlation analysis [142], we found that two features were highly correlated: the cumulative moving standard deviation and the cumulative moving mean of page offsets. Including both would have over-emphasized their importance in this analysis, so we excluded the cumulative standard deviation of page offsets. Cross validation results were 69.6%, 76.4%, 42.6%, and 89.1% for number of transactions, cumulative moving mean of page offsets, mean absolute page offset differences, and current readahead value, respectively. This shows that mean absolute page offset differences is the most important feature, because randomizing its order reduced the validation results the most (down to 42.6%)—followed by number of transactions, cumulative moving mean of page offsets, and finally the currently used readahead value.

After obtaining classification predictions, we set the empirically determined optimal readahead sizes according to the predicted workload type. In Section 5.3.4, we evaluate the readahead neural network not only on workloads we trained on but also on workloads that were *not* included in the training data and workloads running on different devices (NVMe vs. SATA SSDs).

Figure 5.4 shows that the same type of workloads for SATA-SSD vs. NVMe-SSD are not placed in the same clusters all the time. We use neural network input data that is generated only from an NVMe-SSD to train the readahead neural network; nevertheless, we still get significant performance improvement even for SATA-SSDs (see Section 5.3.4). This indicates that our readahead neural network is indeed learning higher-level abstractions about the workloads, one that traditional heuristics would struggle with.

Finally, we also experimented with the readahead neural network using TPC-H [181] queries running on MySQL [139] to show how our readahead neural network behaves on completely different types of workloads and applications and how generalizable the models are.

Decision-tree models We also built a decision-tree (DT) model for workload type classification based on the same features and training data. The readahead DT model contains 59 nodes with a maximum depth of 9 (see Figure 5.5). We tested the prediction accuracy of this DT using the same procedure with the readahead neural network (10-fold cross-validation), and observed that it results in an average prediction accuracy of only 75.4%. In the readahead decision-tree model, decisions are made based on features. For example, the decision at the root node is whether the Z-score of the mean absolute page offset was less than or equal to -0.349 (represented as $X[3] <= -0.349$). Even though the worst case of classi-

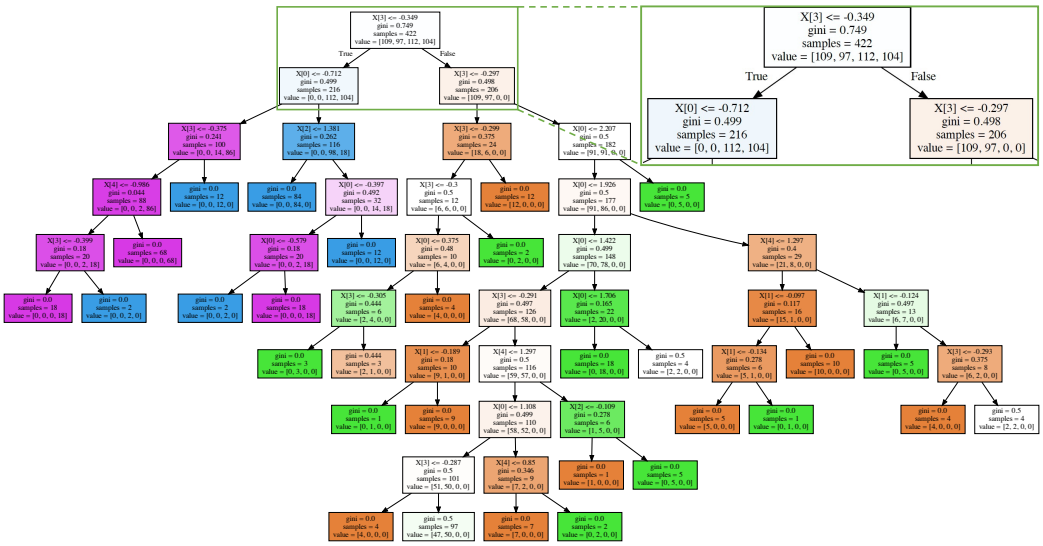


Figure 5.5: A readahead decision tree is built to classify RocksDB workloads running on a NVMe-SSD backed device. Colors denote workload classes: orange for `readrandom` workload, green for `rw-random`, blue for `readseq`, and purple for `readreverse`.

fying a particular readahead workload takes nine IF-THEN decisions, we can observe that the readahead decision-tree model can separate sequential from random workloads in only two levels of decision making; however, this speed of recognition comes at a significant cost of accuracy. As mentioned in Section 5.1.5, KML supports DTs because DTs trees are more explainable than neural networks and run considerably faster. Although the decision trees are more explainable, it is still hard to interpret the readahead decision tree model. The reason is that the values at each node have been normalized to avoid overfitting and numerical instability, and such normalization loses the original values. It is possible that given a normalized input data, we can get the original value and improve the explainability of the decision-tree path. Nevertheless, even with an improve explainability, the readahead neural network model proved more accurate. While it would be useful to have both high predictive power and explainability, faced with a choice between the two, we believe that prediction accuracy that leads to improved throughput is more valuable to end users than explainability. We evaluated the readahead DT using the same procedure as the neural networks (Section 5.3.4).

Readahead in per-file basis So far, we have shown how we approach the readahead problem when a single I/O workload is accessing one device. Storage system developers recognize the challenge of handling mixed storage workloads running on the same system—a common occurrence [12]. In that case, readahead values cannot be set at the device level, as that would be suboptimal in mixed workloads. Instead, readahead values should be set at a higher abstraction level, on a per-file basis. To show our neural network's versatility, we use the same model to tune readahead sizes not only on a per-disk basis but also on a per-file basis. Whereas before we ran inference every second and set one readahead value for an entire device, here we ran inference every second on *each* open file and set a readahead value directly in Linux's `struct file`. We evaluated the per-file basis approach and found that it could predict and improve I/O throughput for *mixed workloads* better than both the vanilla and per-disk basis approaches (see Section 5.3.4).

5.2.2 Use Case: NFS rsize

Problem definition Networked storage systems such as NFS are popular and heavily used. NFS is used for storing virtual machine disks [131], hosting NoSQL databases [165], and more. A misconfiguration of NFS can hurt performance. We experimented with different applications using NFS and found out that one critical NFS configuration parameter is the `rsize`—default network read-unit size. Hence, we focus on predicting an optimal NFS `rsize` value based on workload characteristics.

Studying the problem We tested NFS using the same methodology as for readahead. The only difference here is tuning `rsize` instead of readahead. We used NFSv4 for all of our tests. The NFSv4 implementation we used supports only seven different `rsize` values (4K–256K). However, in the NFS use case, there are additional external factors not present in the readahead problem that can affect I/O performance (*e.g.*, NFS server configuration, network speed, and number of clients connected to the same server). We experimented with four different RocksDB benchmarks under different NFS server configurations and network conditions. We configured our server with two different NFS mount point options—one backed by NVMe-SSD and one backed by SATA-SSD. We injected artificial network delays to simulate slower networks. Our experiments revealed that random and sequential workloads require different `rsize` values to achieve

optimal performance.

Data collection We enabled NFS and page-cache related kernel tracepoints to collect training data (*e.g.*, `nfs4_read`, `nfs4_readpage_done`, `vmscan-_lru_shrink_inactive`, and `add_to_page_cache`). Unlike the readahead neural network model, we collected data from tracepoints not only to model page cache behavior, but also network conditions. Similarly studying these traces, we chose our feature set and placed our hook functions. Our feature set includes eight features (described below) which are calculated using the following five data points: (i) time difference from the beginning of execution for each tracepoint transaction, (ii) NFS file handles, (iii) file offsets in NFS requests, (iv) page offsets of the files that were accessed, and (v) number of reclaimed pages during LRU scans.

Data preprocessing & normalization We applied the same data preprocessing and normalization techniques that we used for the readahead neural network. The NFS neural network model consists of eight features which are computed every second: (1) number of tracepoint transactions, (2) average time difference between each `nfs4_read` and `nfs_readpage_done` matching pair, (3) average time difference between each consecutive `nfs4_read` request, (4) average time difference between each consecutive `nfs4_readpage_done` request, (5) mean absolute requested offset difference between each consecutive `nfs4_read` request, (6) mean absolute page offset difference between each consecutive `add_to_page_cache`, (7) average number of reclaimed pages, and (8) current `rsize`.

Neural network model We trained and tested our NFS neural network model using the same methodology as the readahead problem; for brevity, we detail only the differences between the neural network models. We approached the NFS problem as a workload characterization problem and constructed our NFS neural network model with four linear layers (with hidden layer sizes of 25, 10, and 5) with sigmoid activation functions in between. Similar to the readahead neural network, we used cross entropy as the loss function and SGD as the optimizer. We evaluated the NFS neural network model and found out that it results in a prediction accuracy of 98.6% (using 10-fold cross-validation).

5.3 Evaluation

Our evaluation proceeds as follows: First, we explain our evaluation goals in Section 5.3.1. We then describe the testbed design and benchmarks that we used to evaluate the readahead and NFS `rsize` neural networks in Section 5.3.2. In Section 5.3.3 we provide performance details regarding KML's training and inference. Section 5.3.4 shows how the readahead ML models improve performance. Finally, in Section 5.3.5, we present our evaluation of the `rsize` neural network model for NFS.

5.3.1 Evaluation Goals

Our primary evaluation goal is to show that using ML techniques inside the OS can be used to to tune parameters dynamically and improve storage systems' performance.

We start by showing the practicality of using ML in kernel space. We evaluate KML's system overheads in terms of (i) data collection overhead, (ii) training cost, (iii) inference cost, and (iv) memory usage. Then, we evaluate both readahead and NFS neural network models to show how they improve the I/O performance and quickly adapt the system in the presence of changing workloads and conditions. To show that our models can learn abstract workload patterns, we first present the generalization power of our models by testing it on workloads *not* included in the training dataset. Next, we present benchmarks on a device type that was *not* used in the data collection phase or training. We also built a decision tree model for the readahead problem to have *comparable* results since decision trees are more explainable, still popular, and closer in operation to traditional heuristics.

Furthermore, we evaluate KML's versatility by applying the readahead neural network model on a per-file basis. This demonstrates KML's ability to optimize individual I/Os in a mixed workload. Lastly, we evaluate our readahead ML models' behavior when they mispredict and how quickly they recover.

5.3.2 Testbed

We ran the benchmarks on two identical Dell R-710 servers, each with two Intel Xeon quad-core CPUs (2.4GHz, 8 hyper-threads), 24GB of RAM and an Intel 10GbE NIC. In some experiments, we intentionally configured the system with only 1GB of memory to force more memory pressure on the I/O system; but we also show experiments with the full 24GB of system RAM. We used the CentOS

7.6 Linux distribution. We developed KML for Linux kernel version 4.19.51, the long-term support stable kernel; we added our readahead ML models to this kernel and used it in all experiments. Because HDDs are becoming less popular in servers, especially when I/O performance is a concern, we focused all of our experiments on SATA and NVMe SSDs. We used Intel SSDSC2BA200G3 200GB as our SATA-SSD device and a Samsung MZ1LV960HCJH-000MU 960GB as our NVMe-SSD device, both formatted with Ext4. These two devices were used exclusively for RocksDB databases. To avoid interference with the installed CentOS, the two servers have a dedicated Seagate ST9146852SS 148GB SAS boot drive for CentOS, utilities, and RocksDB benchmark software. We used 10GbE switches to connect the machines (useful for NFS experiments). We observed an average RTT time of 0.2 milliseconds.

Benchmarks We chose RocksDB's `db_bench` tool to generate diverse workloads for evaluating the readahead and NFS `rsize` neural networks. RocksDB [64] is a popular key-value store and covers an important segment of realistic storage systems; `db_bench` is a versatile benchmarking tool that includes a diverse set of realistic workloads. Workloads can be run individually or in series, and the working set (database) size can be easily configured to generate more I/O pressure on a system. On the 1GB RAM systems, we configured a RocksDB database of twice the size (2GB). The two main reasons why we choose this configuration are (1) to ensure that benchmarks can generate enough I/O operations that would not be merely cached in memory and (2) to reduce the time of executing all benchmarks considerably. Nevertheless, one may consider a system with only 1GB RAM as not a realistic system configuration. Therefore, we also executed all the benchmarks in this chapter with a 56GB RocksDB database running on the same system configured with 24GB RAM. The results are showing similar improvements and there are no significant performance-trend differences (see Section 5.3.4). Neverthless, because we ran experiments with more RAM and for a longer period of time, we noticed some interesting findings which are explained in Section 5.3.4.

To demonstrate that our ML models can learn from and optimize for different types of real-world workloads, we chose the following six popular yet different `db_bench` workloads: (1) readrandom, (2) readseq, (3) readrandomwriterandom (alternating random reads and writes), (4) readreverse, (5) updaterandom (read-modify-write in random offsets), and (6) `mixgraph` (a complex mix of sequential and random accesses, based on Facebook's realistic data that follow certain Pareto and power-law distributions [35]).

We trained our readahead neural network on traces that contain only four of these workloads: readrandom, readseq, readreverse, readrandomwriterandom—all running only on the NVMe-SSD. These four tend to be the simpler workloads, because we wanted to see whether KML can train on simpler workloads yet accurately predict on more complex workloads not trained on. This also ensures a balanced representation of randomness and sequentiality in the training dataset.

After the training phase completed, we tested our models on all six workloads as well as different devices. This was done to show that our models not only perform accurate predictions on the training set samples, but they also generalize to two new and complex workloads (updaterandom and `mixgraph` as well as a different device (SATA-SSD))—which were excluded from the training data. We evaluated mixed workloads by running two concurrent `db_bench` instances, each on a separate RocksDB database and using a different workload profile, both stored on the same device. We kept the hardware configuration the same as before (1GB RAM) to increase system and page-cache pressure.

We also experimented with our readahead network model using TPC-H [181] queries running on MySQL [139], to evaluate how generalizable and effective the readahead neural network is to an entirely different workload. In this chapter, we do *not* claim that our readahead neural network model will work universally to optimize readahead values for all possible workloads. Rather, these use cases are meant to demonstrate the KML framework's versatility. With more workloads and datasets, one can build a wide range of ML models to optimize many storage problems.

5.3.3 KML's Overheads

An ML model's overhead depends on its architecture. Generally, deeper or higher-dimensional neural networks consume more memory and CPU than, say, decision-tree models. It is vital that an ML component, especially one that may run inside the kernel, consume as little CPU and memory as possible. Next, we evaluate the readahead neural network overheads.

Data gathering overheads The only inline operations that readahead neural network inserts directly in the data path are data collection probes. Hence it is vital for these probes to be optimized. Figure 5.6(C) shows how the data collection CPU overheads (percentage) change with subsampling window sizes. When there is no subsampling in the system ($X = 1$), the CPU overheads of data collection

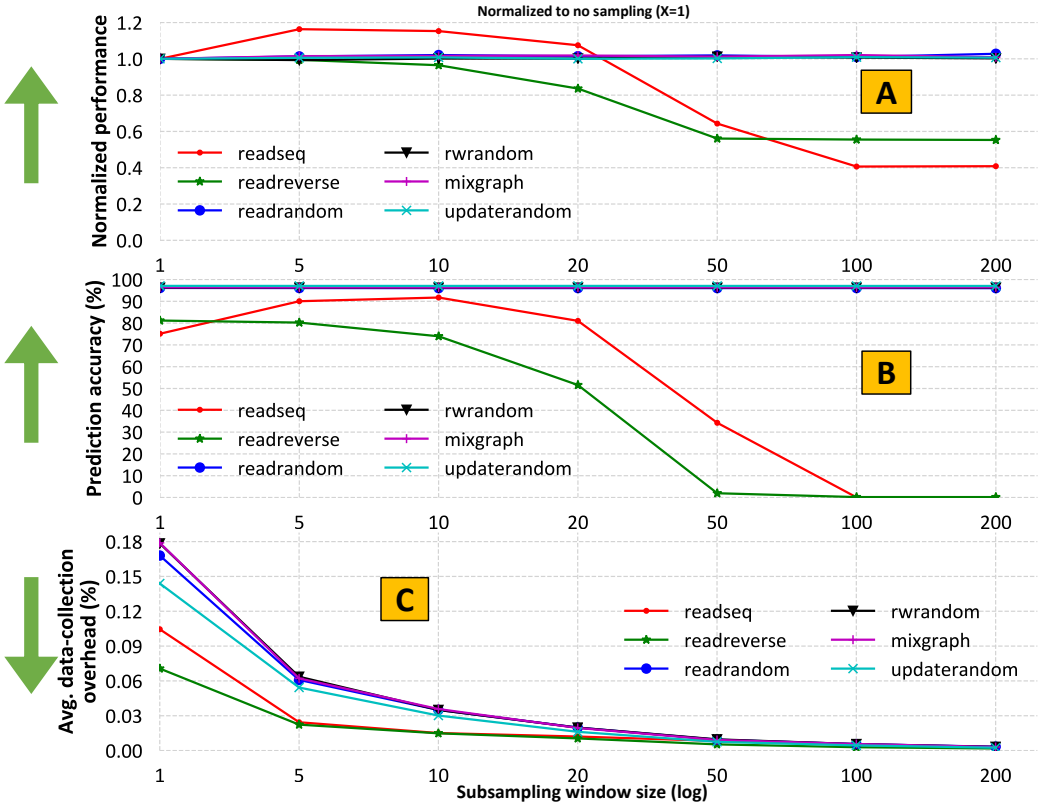


Figure 5.6: Performance (A), prediction accuracy (B) and CPU overheads (C) in seven different subsampling window sizes for the per-disk readahead neural network. Upward green arrows denote that higher is better.

probes is as high as 0.18%. Although this is a fairly low overhead considering the multiplicative I/O benefits we report, this overhead can be reduced further by increasing the subsampling window. However, increasing the subsampling windows size can hurt prediction accuracy and performance improvements, as less data is available to make rapid predictions. See Figure 5.6(A) and (B). Figure 5.6(B) shows that workloads with a lot of randomness in them were the least affected, because randomness is still predicted as random even with fewer samples; yet we can reduce the already small CPU overheads even more.

The figure further shows that only sequential workloads are affected by subsampling window changes: generally, as the sampling window widens, prediction accuracy and normalized performance worsen. However, we noticed an unexpected behavior for the `readseq` workload. Increasing the subsampling window size from one to five or ten *actually improved* both prediction accuracy and performance; this is because `readseq` keeps the I/O subsystem busy at near maximum bandwidth, and increasing subsampling window size reduced short-term noise that resulted in more frequent mispredictions.

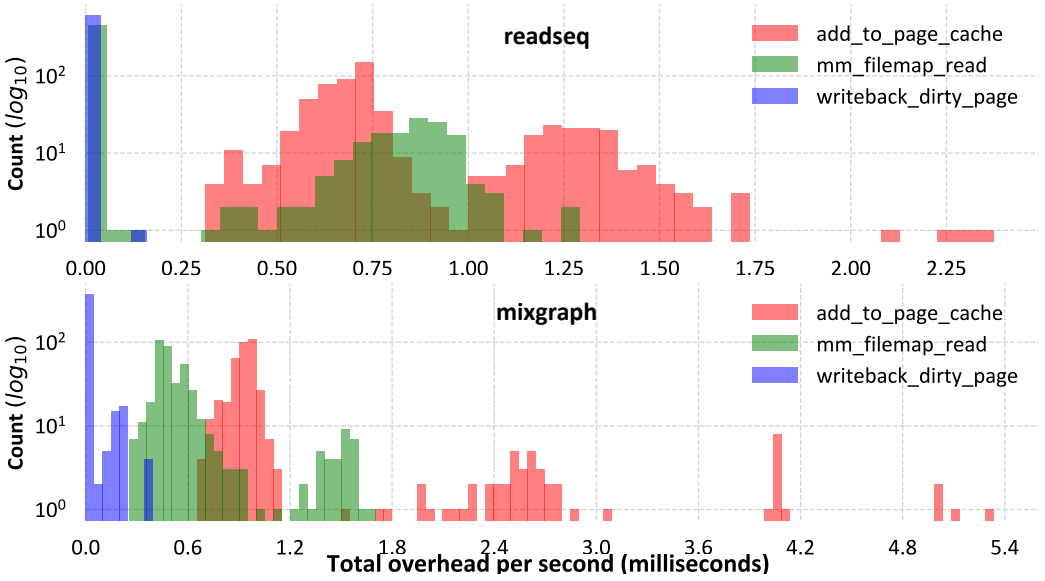


Figure 5.7: Distribution of total data collection overhead (milliseconds) in every second when `readseq` and `mixgraph` workloads are running.

We can also observe that the data collection overheads depend on the workload type. For example, `readseq` workload's average data sampling frequency per-

second is around 30K but its data collection overhead is still lower than `mixgraph` workload which has 20K average data sampling frequency per-second. The reason that data collection overheads change based on the workload type, is due to the sudden I/O bursts resulting in some cache misses. In Figure 5.7, we show the histograms of the data collection overheads for the `readseq` and `mixgraph` workloads. We can observe that the `mixgraph` histogram shows that data collection overheads for all data points are higher than `readseq`. In addition, `mixgraph`'s data collection histogram displays outliers of `add_to_page_cache` data collection point: these result due to cache misses caused by sudden I/O bursts.

Inference/training overheads The readahead neural network performs inference (prediction) and changes the block-layer readahead value in 21μs on average (std. dev. $< 10\%$). This action executes in a separate, asynchronous kernel thread, once in every second. Hence, it has negligible impact on the overall OS performance. When the readahead neural network runs in per-file mode, KML runs inferences an average 135 times a second (*i.e.*, one per open file): inferencing for all open files consumes 1.7ms on average. We measured that the readahead decision tree inference takes only 8μs (using the same feature vector). The readahead neural network and decision tree have the same data pre-processing and normalization implementation—the only difference between them is in the inference part. Overall, these overheads are fairly small and acceptable, considering the multiplicative I/O performance benefits they enable.

As discussed in Section 5.2.1, our readahead neural network prototype offloads training to the user level. We measured the time to perform one training iteration in user level at 51μs on average; this training iteration includes the forward pass, back-propagation, and weight update stages.

Memory overheads The readahead neural network allocates 3,916 bytes of dynamic memory during the model's initialization phase. While inferencing, KML temporarily allocates 676 bytes before returning the inference results. This overall memory footprint is negligible in today's multi-GB systems. The readahead decision tree occupies only 2,432 bytes of dynamic memory during initialization. The decision tree model does not allocate dynamic memory during inference. Lastly, the kernel module `readhead.ko` has a binary memory footprint of 432KB and the kernel module `nfs.ko` is 636KB, while the KML framework itself (`k-Mlib.ko`) has a memory footprint of 5.5MB.

Practicality and scalability Our vision is KML could enable a future where traditional heuristics are gradually replaced with ML-based approaches to improve storage and network I/O performance. In Section 5.3.4 we demonstrate, for example, that our readahead neural network model improves I/O performance by as much as 2.3×, but consumes less than 0.2% additional CPU cycles: we believe this is a fairly acceptable trade-off for most users. Nevertheless, we tested this model with 100 concurrent inferences and found that both overheads and I/O improvements have scaled linearly; hence KML's benefits still outweigh its overheads.

5.3.4 Readahead Evaluation

Readahead background There are two places in the Linux kernel where readahead is defined: the block layer and the file system level. When a file is opened, the VFS initializes an open `struct file` and copies the readahead value for that file from the corresponding block layer. Upon a page fault for that file, the page-cache layer uses the value stored in the file to initiate reading-ahead the desired number of sectors of that file. However, the readahead value in the file structure is initialized only once when the file is opened. So when KML changes the block layer readahead value, the Linux kernel does not copy the new value to any file already opened. This means that open files may continue to use a sub-optimal readahead value, even if better values are available (*e.g.*, due to workload changes). That is why we implemented a mechanism that changes the readahead size for *open files* when KML changes the disk-level readahead value. This propagates newer readahead values to each open file, improving our adaptability. Conversely, if KML mispredicts the workload type and changes the readahead size to a sub-optimal value, short-term performance degradation can happen, which might hurt overall performance.

Back-to-back workloads on NVMe Figure 5.8 shows four workloads running back to back with each subfigure comparing a vanilla run (colored orange) to our KML-enabled readahead run (colored blue). The readahead value was left at the default value (*i.e.*, 256) at the start of both vanilla and KML-enabled runs, but when the next workload started, it used the last readahead value from the previous workload's run (*e.g.*, the readahead value at the end of the leftmost subfigure is the same at the start of the subfigure immediately to its right). This experiment evaluates KML's ability to optimize the readahead values when the I/O workload

Figure 5.8: Running four back-to-back RocksDB workloads in order from left to right: readsequential, readrandom, readreverse, then mixgraph. Here, we started with the default readahead value; thereafter, the last value set in one workload was the one used in the next run. For each of the four graphs, we show their Y axes (throughput, different scales). The readahead value is shown as the Y2 axis for the rightmost graph (d) and is common for all four. Each workload ran 15–50 times in a row, to ensure we ran it long enough to observe patterns of mis/prediction and reach steady-state. Again, we see KML adapting, picking optimal readahead values, occasionally mis-predicting but quickly recovering, hence overall throughput was better.

may change every few minutes. The X axes indicate the run time in minutes. The Y axes indicate throughput in thousands of ops/sec (higher is better), and have different scales for each experiment. The Y2 axes show the readahead values used or predicted by KML over time in terms of number of sectors (denoted with a green line and using the same scale). Each workload ran 15–50 times in a row, so it ran long enough to observe mis/predictions patterns. As seen in Figure 5.8, KML adapts quickly to changing workloads by tuning the readahead value in about one second.

Although we observe some mis/prediction patterns, seen as sudden spikes, overall throughput still improved across all four runs, averaging 63.25% improvement: 140% improvement for readrandom, 2% for readsequential, 109% for `mixgraph`, and 12% for readreverse. We note that even a small improvement in throughput can yield significant cumulative energy and economic cost savings for long-running servers [119].

Read-sequential workloads Out of the six workloads we ran, Figure 5.9 shows the one where KML performed the worst: read-sequential. Reading data sequentially directly from the raw SATA-SSD is nearly 1,000× faster than the `mixgraph` workload, and nearly 400× faster with the NVMe-SSD. Here, there is little opportunity for KML to improve throughput for a sequential workload that reads at speeds near the maximum throughput of the physical device.

Read-reverse workloads As we can see from the fluctuating green line (readahead values in Figure 5.8) KML mispredicts readreverse as readseq and changes the readahead value to something sub-optimal. These two workloads both access files sequentially—one reading forward and one backward. Interestingly, readseq and readreverse are quite close from a feature representation perspective, which explains the mispredictions. But since both of these workloads access files sequentially, their optimal readahead values are also quite close to each other. Thus, even when KML mispredicts readreverse as readseq or vice versa, this had a small overall impact on performance.

Figure 5.9: Readahead neural network performance improvements (×) for RocksDB benchmarks on SATA-SSD and NVMe-SSD across all six workloads, normalized to vanilla (1.0×).

Summary of readahead neural network results We summarize all readahead neural network results in Figure 5.9. We observe that the average throughput improvement for NVMe-SSD is ranging from 0% to 65%. We saw greater improvements in the SATA-SSD case, ranging from 2% to 130% (2.3×). Lastly, we ran the complex `mixgraph` workload on NVMe-SSD with the system memory set to the maximum (*i.e.*, 24GB) and the database size set to be relatively large, 65GB (compared to a 2GB baselines database size). This experiment ran for nearly an hour (48.5 minutes) and resulted in an average throughput improvement of 38%.

Mixed workloads Mixed workloads are considered a challenging optimization problem [12]. In Figure 5.10, we present a timeline performance comparison using the readahead neural network model running on a per-disk vs. per-file basis. The per-file mode performs better overall because readahead values are set for

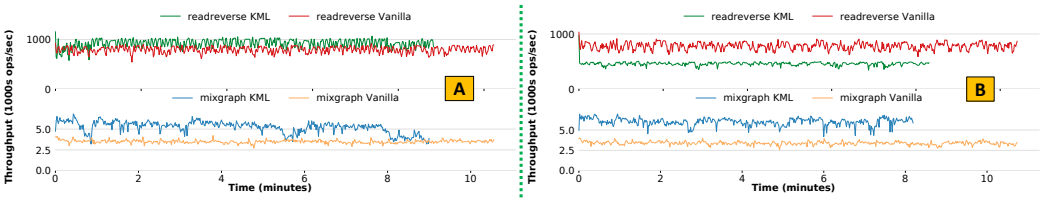


Figure 5.10: Mixed workloads results on a timeline, comparing the readahead neural network model running on per-file basis ('A', left) vs. per-disk basis ('B', right).

each open file independently. Conversely, in the per-disk mode, a single readahead value is set at the disk level and hence uniformly on all open files: a readahead value good for one workload is likely to be sub-optimal for other open files. One reason why the per-disk mode cannot predict workload types correctly is that when different workloads are mixed—even sequential ones or ones with regular patterns—the mix looks more like a purely random workload at the disk level.

Figure 5.11: Mixed workloads results. We ran sequential and random workload combinations on the same NVMe-SSD device. Each unique combination is tested with the readahead neural network running in per-disk basis (kml disk) and per-file basis (kml file) and compared against vanilla results. The model running in per-file basis outperformed both vanilla and per-disk modes.

Figure 5.11 shows overall mixed workloads performance comparisons. Per-file mode performed overall better in every combination of mixed workloads. If we compare only the sequential parts of the mixed workload combination (orange bars in Figure 5.11), in per-disk mode, we observe significant performance degradation. However, in per-file mode, we can observe performance improvements for both the sequential and random (blue bars in Figure 5.11) parts of the mixed workload combination. The reason why per-disk mode performs better for the random parts of the mixed workload combinations is for the same reason: mix-

ing workloads looks more random-like at the disk level. KML predicts these as readrandom or readrandomwriterandom which coincidentally fits this part of the workload, but significantly hurts non-random workloads.

Figure 5.12: Readahead decision tree performance improvements (×) for RocksDB benchmarks on SATA-SSD and NVMe-SSD devices across all six workloads, normalized to vanilla (1.0×).

Decision tree evaluation In addition to the neural network model, we implemented a decision tree model for the readahead problem to compare the two ML approaches on the same problem. We tested the readahead decision tree the same way. Figure 5.12 shows that there is a performance improvement for workloads with a random component. For the readahead decision tree, we measure average throughput improvement for random workloads on NVMe-SSD as ranging from 48% to 59%; and in the SATA-SSD case, ranging from 99% to 119% (2.19×). While good, the neural network model yielded greater improvements, as discussed above.

The DT model, however, degraded performance for sequential workloads. it degraded performance for sequential workloads on NVMe-SSD by 15–40%; and in the SATA-SSD case, by 36–73% worse. We investigated this performance degradation. Figure 5.13 shows the readseq workload running on a RocksDB instance stored on an NVMe-SSD. Here, the readahead decision tree predicts the workload correctly in the first three minutes, despite some fluctuations. Afterwards, the decision tree model's predictions fluctuate wildly, and at around minute 10 it consistently makes wrong predictions. Overall, this was somewhat expected for our I/O optimization problem: neural network models, while more complex to train and use, are more adaptable than decision-trees [78]. Specifically, when

Figure 5.13: Performance timeline graph for tuning with KML decision tree while running readseq workload on NVMe-SSD.

the DT model mispredicts, and system conditions change (*i.e.*, I/O activity), the DT model continues to mispredict, and it cannot recover as quickly as the more adaptable neural network model.

Figure 5.14: Readahead neural network performance improvements ($\times$) for TPC-H queries on SATA-SSD and NVMe-SSD devices, normalized to vanilla ($1.0\times$).

TPC-H benchmarks As we mentioned in Section 5.3.2, we evaluated our readahead neural network model—trained on RocksDB workloads—on TPC-H queries running on MySQL database (both NVMe-SSD and SATA-SSD cases). This intends to show the model's accuracy limitations when presented with vastly different workload and application combinations. Figure 5.14 shows performance improvements as much as 39% for most query types. For query 11, however, the readahead neural network failed to characterize the workload correctly and resulted in a 53% performance reduction. Nevertheless, overall TPC-H performance

still improved by 6%. We expect that neural network models trained on more traditional SQL database workloads would likely yield even better predictions across most similar databases.

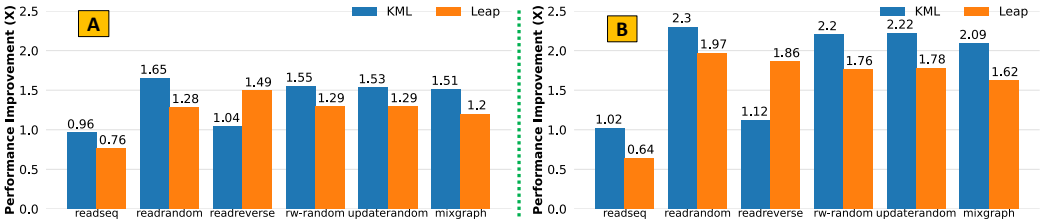


Figure 5.15: Performance improvement comparisons between LEAP [10] and KML for RocksDB benchmarks on NVMe-SSD ('A', left) and SATA-SSD ('B', right).

Comparison with LEAP Data prefetching and caching is a well-studied problem with many heuristics developed to optimize I/O transactions. We compared our readahead neural network with a recent data-prefetching heuristic, *LEAP* [10]. We evaluated both LEAP and our readahead neural-network model with the same setup that we used to evaluate KML with RocksDB workloads running on NVMe-SSD and SATA-SSD. We have integrated LEAP to work with a local page cache. LEAP integration took only 243 LoC and was mostly a straightforward data-aggregation code. Our readahead neural network achieves 16% better average throughput improvements than LEAP, when workloads are executed on NVMe-SSD. When running workloads on SATA-SSD, the readahead neural network model's average performance gain is 22% better than LEAP.

Figure 5.15 shows these results. We highlight two main takeaways. First, LEAP causes a significant performance reduction for `readseq` workloads (-24% for NVMe-SSD and -36% for SATA-SSD). Conversely, our readahead neural network either improves the I/O performance across all the RocksDB workloads or keeps the performance close to the same as running without the optimization. It is important that any optimization technique that helps one workload would not hurt another.

Second, there is only one workload where LEAP's performance was better than our readahead neural network's performance: `readreverse`. The main reason why LEAP outperformed us in the `readreverse` workload is that LEAP is directly in charge of choosing pages that will be stored in memory. Conversely,

our readahead neural network tunes only readahead value in the block layer. Thus, LEAP can fetch pages in descending order while our readahead neural network relies on the readahead subsystem—which generally cannot handle reading "ahead" in reverse order.

Large memory experiments To test our readahead neural network model's abilities on significantly different hardware setup, we experimented with a 56GB RocksDB database running on 24GB RAM configuration. This represents a more realistic storage server scenario. Overall, we observed that performance improvement trends have not changed. However, the larger memory experiments took a significantly longer time which exposed numerical instabilities in our normalization phase. We originally used `float`s to compute normalization statistics. Over the course of longer-running experiments, we lost precision in numerical statistics. We fixed this problem simply by switching to `double` floats. We measured that switching to `double`s did not add any extra computational overheads thanks to modern CPUs' advanced floating-point units.

In addition, we also adjusted our weighted-moving average. This adjustment was needed because the large RAM size affected the number of transactions per second which is one of our key features. Since this setup used a larger RAM, we can keep fetching and updating KV pairs without writing them back for a longer period of time in the beginning of benchmarking. As a result, we can perform more transactions per second. This type of significant changes in hardware or software setup can affect the features and their extraction process (*e.g.*, moving averages). Such significant changes in features can cause mispredictions which leads to performance degradation.

We fixed this by adjusting the weighted moving average. We initially considered the runtime input data to contribute to the moving average equally as training data (*e.g.*, a uniform moving average). Then, we tuned the moving average weight to 10%, meaning that we only take one-tenth each new sample into the moving average. This ensures that sudden spikes in activity do not disturb the moving average too much—keeping its change smoother. We reached this final value by testing different weights using binary search. In the future, we plan to integrate a feedback control mechanism to adapt the moving average weight automatically in case of drastic changes in hardware or software conditions. After the change, we tested the readahead neural network model with different storage devices, memory sizes, workloads, mixed workloads, applications: it consistently performed significantly better than baseline and LEAP.

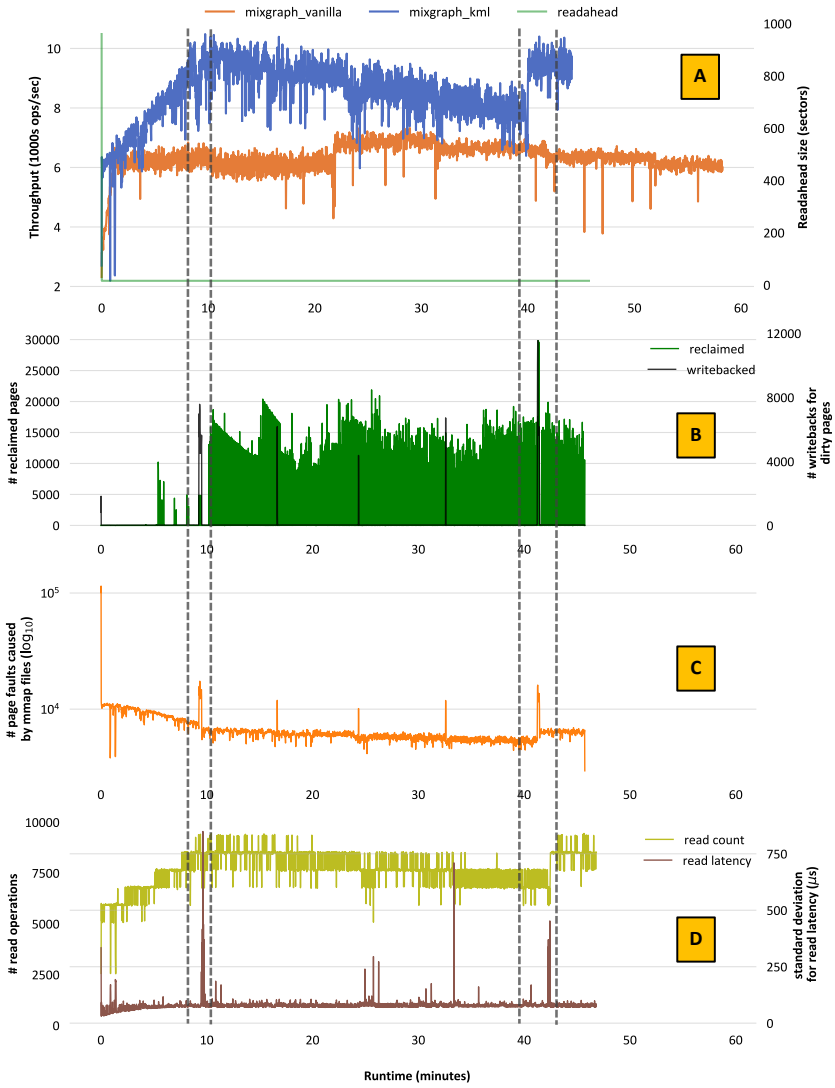


Figure 5.16: Throughput analysis for running `mixgraph` on 24GB memory with a 56GB RocksDB database. In (A) we show throughput timeline and improvements for `mixgraph` running with KML. We can see three phases of `mixgraph`'s execution, demarcated by double vertical dashed lines: (1) startup, (2) stabilize and gradually decline, and (3) restabilize. We explain these phases and why throughput changes by showing page-reclamation numbers (B), triggering writeback operations for dirty pages (C), and the number of page faults taking place due to file operations from the OS's perspective. In (D) we show the number of read operations and their standard deviation operations from RocksDB's perspective.

By running experiments with larger memory and database sizes, we also experimented with how KML behaves over long-term executions. Since these experiments took many hours and even days, we could evaluate the readahead neural network behavior under different phases of the page cache. In Figure 5.16 we show a `mixgraph` workload running on the large memory and database setup. We see three phases separated by double vertical dashed lines.

First, the startup phase took around nine minutes to fill up the entire page cache while the readahead neural network was in inference mode and optimizing the readahead size for the storage device. We observe that the startup phase for running the `mixgraph` workload without a readahead neural network took around one minute due to poor use of the page cache with a sub-optimal readahead size and resulted in a stable-looking, but sub-optimal throughput.

In the second phase, stabilization starts after filling the entire page cache and beginning to trigger some page reclamation processes. In this stabilization phase, we observed staircase-like throughput reductions, which are correlated with spikes in write-back dirty page requests (see in Figure 5.16).

Third, a *re-stabilization* phase starts with sudden spike in the write-back activity of reclaimed pages. This frees a large number of pages: we can observe a sudden spike in page faults which are related to `mmap`ed files. This page-fault spike also indicates that a lot of new pages loaded into memory. Overall, this improves performance with newly loaded data in the page cache being accessed.

Finally, We can notice that all these phase changes create variation in read latency for the `mixgraph` workload (see Figure 5.16 D). Even though all these variations and sudden spikes occur in the I/O subsystem, our readahead neural network successfully predicted the workload and tuned the readahead size.

5.3.5 NFS Evaluation

Figure 5.17 shows the NFS `rsize` neural network performance improvements using the same evaluation techniques of readahead. Throughout these experiments, we ran multiple iterations of the same workloads. Since `rsize` is a mount point parameter for NFS, our NFS neural network can tune `rsize` values only in the beginning of the iteration. (We plan to fix the Linux kernel to permit `rsize` to change dynamically.) Hence, in sequential workloads, if the NFS neural network makes even one misprediction, it will affect the entire iteration, leading to performance degradation. Nevertheless, in random workload cases, we still measured around 15× performance improvement; in separate experiments (not shown for brevity), performance improvements for random workloads reached up to 20×.

Figure 5.17: Performance improvements (×) for RocksDB benchmarks on SATA-SSD and NVMe-SSD devices across all six workloads running on NFS, normalized to vanilla (1.0×).

This demonstrates the significant potential of KML.

5.4 Future Work

We are adding ML techniques to KML, such as reinforcement learning [97], which can be a better fit for solving certain OS problems. To support more advanced ML approaches (*e.g.*, Recurrent Neural Networks (RNNs) [191]) and Long Short-Term Memory (LSTM) [84]), we are extending KML to support arbitrary computation DAGs. We also plan to integrate user-kernel co-operated design into KML. Finally, loading an unverified ML model into a running kernel opens up new attack surfaces. We are exploring known techniques to digitally sign and certify loadable models [125, 104].

Chapter 6

Proposed and Future Work

In this chapter we outline our proposed work that we intend to include in this thesis. We then discuss the possible future work that is extends beyond this thesis.

6.1 Proposed Work

We have already demonstrated that ML solutions can be first-class citizens in operating systems and storage systems. In this thesis proposal, we plan to apply ML approaches to other I/O and network subsystems to make this argument more robust. We specifically chose these components because I/O is always the slowest component of any system. Nevertheless, with a small computational overhead that will be spent on ML, we can improve the I/O performance of these components significantly. Moreover, these components have a lot of tunable parameters. In addition, they have to be self-adaptive because of the workloads' ever-changing nature. One of the possible research problems that we will be working on is predicting I/O latencies in the block layer to investigate and build a better I/O scheduler. Another proposed research project is building an assistive system for TCP/IP or BBR congestion control and fairness to help these subsystems better adapt to the environment and traffic conditions.

We also plan to extend KML's capabilities with reinforcement learning (RL). RL is one of the most important plans for KML. Because storage systems and operating systems are constantly evolving, the likely best ML solutions for these systems use RL models.

TCP congestion control Researchers have been trying to optimize TCP congestion control via ML [3, 92, 61, 60, 30, 185]. However, they approached the problem using off-the-shelf ML frameworks. To make this setup work, they had to collect data from the kernel, move them to user-space, feed it to ML frameworks, and pass back the inference results to kernel-space. Therefore, the solutions imposed huge overheads (*e.g.*, more than 100% even). Considering KML's current capabilities, we can build ML models for improving TCP congestion control algorithms' performance, latency, and fairness with a much lower and acceptable computational overhead. We also theorize that we can tune the network layer to optimize for applications. To this end, we can build an ML model to classify workloads by their network usage characteristics. Hence we can apply application-specific optimizations to the network layer.

I/O latency predictor Large-scale storage applications are designed with consideration of application-specific QoS metrics. One of the most critical QoS metrics is I/O latency tails. Predicting the latency of individual I/O requests is crucial to building latency-sensitive large-scale applications. First, we plan to utilize Intel Labs' Open Storage Toolkit [91] to collect block traces with data access type hints to investigate the correlations between data access types and I/O latencies. We then try to build an ML model which takes advantage of knowing data access type hint to predict latencies for I/O requests. Finally, we would like to use I/O latency predictions to build latency-aware I/O schedulers. Researchers have worked on similar problems on a specific setup that is either targeting particular hardware [79, 80] or designed for simulation and not practical in the real world [36]. In addition, to our best knowledge, there has not been any research trying to approach I/O latency prediction with data type hints.

6.2 Future work

KML opens countless possibilities for ML applications for operating systems. There are more than a thousand knobs For example, on Linux 4.19.51+ `sysctl -a` alone reports 1,917 tunables; for `net.ipv4` reports 435 tunables. I/O subsystems can be tuned by multiple ML models. Researchers will face interesting challenges while tuning multiple knobs concurrently using KML. Multi-objective optimizations will be an interesting research area which we leave to future work.

Our next possible future work is about integrating federated learning into KML. In a modern data center environment, learning locally about the storage

systems will not be practical, because modern workloads are fairly dynamic and evolving; even a single machine can be running different workloads, which are also constantly changing. To solve these challenges, ML models in OSs and storage systems should learn distributively without sharing data. Security and privacy in federated learning are well-studied research fields.

We believe that KML can be a starting point for practical ML applications for storage and operating systems. In the long run, machine learning models can assist the storage systems' components and get us one step closer to truly and fully self-adaptive systems.

Chapter 7

Conclusions

Operating systems and storage systems have to support many ever-changing workloads and devices. To provide the best performance, we have to configure storage system knobs based on workloads' needs and device characteristics. Unfortunately, current heuristics cannot adapt to workload changes quickly enough and require constant development efforts to support new devices. We propose KML to solve these problems—an ML framework inside the OS that adapts quickly to optimize storage performance. KML enables finer granularity optimizations for individual files in even mixed workloads—a challenging problem.

We first built a tracing framework for operating systems to collect input data for building ML models. It is necessary to have a low-overhead and high-fidelity data-collection framework to model operating system problems correctly. We then implemented a machine learning framework that is tailored for operating systems. KML provides a low-overhead, easy-to-use ML development environment for operating system developers to build machine learning models for optimizing OS I/O components. We have supported our vision of substituting heuristics with ML models by implementing two use-cases: tuning `readahead` size (both per-disk and per-file basis) and NFS `rsize`.

Our preliminary results show that, for a readahead problem, we can boost I/O throughput by up to 2.3× with a mere 0.2% CPU overhead and 4KB memory usage. For the NFS `rsize` problem, the improvement was up to 15×. These I/O throughput improvements far outweigh the small memory and CPU overheads of KML.

Our thesis is that tuning I/O subsystems knobs with a fine granularity using ML models is promising and can bring significant improvements. KML is still missing a couple of crucial capabilities (*e.g.*, reinforcement learning); and we need

more ML models to improve different I/O subsystems to provide more proofs for generalizability.

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] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, pages 265–283, Savannah, GA, November 2016.

[3] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 632–647, 2020.

[4] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael P. Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: Versatile cluster-based storage. In *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*. USENIX, 2005.

[5] Rishabh Agarwal, Nicholas Frosst, Xuezhou Zhang, Rich Caruana, and Geoffrey E Hinton. Neural additive models: Interpretable machine learning with neural nets. *arXiv preprint arXiv:2004.13912*, 2020.

[6] 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.

[7] 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.

[8] Ibrahim 'Umit' Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. A machine learning framework to improve storage system performance. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage (HotStorage '21)*, Virtual, July 2021. ACM.

[9] Ibrahim Umit Akgun, Geoff Kuenning, and Erez Zadok. Re-animator: Versatile high-fidelity storage-system tracing and replaying. In *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR '20)*, Haifa, Israel, June 2020. ACM.

[10] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857, 2020.

[11] Ahmed Amer, Darrell DE Long, J-F Pâris, and Randal C Burns. File access prediction with adjustable accuracy. In *Conference Proceedings of the IEEE International Performance, Computing, and Communications Conference (Cat. No. 02CH37326)*, pages 131–140. IEEE, 2002.

[12] George Amvrosiadis, Ali R. Butt, Vasily Tarasov, Erez Zadok, Ming Zhao, Irfan Ahmad, Remzi H. Arpaci-Dusseau, Feng Chen, Yiran Chen, Yong Chen, Yue Cheng, Vijay Chidambaram, Dilma Da Silva, Angela Demke-Brown, Peter Desnoyers, Jason Flinn, Xubin He, Song Jiang, Geoff Kuenning, Min Li, Carlos Maltzahn, Ethan L. Miller, Kathryn Mohror,

Raju Rangaswami, Narasimha Reddy, David Rosenthal, Ali Saman Tosun, Nisha Talagala, Peter Varman, Sudharshan Vazhkudai, Avani Waldani, Xiaodong Zhang, Yiying Zhang, and Mai Zheng. Data storage research vision 2025: Report on NSF visioning workshop held may 30–june 1, 2018. Technical report, National Science Foundation, February 2019. *https://dl.acm.org/citation.cfm?id=3316807*.

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

[14] 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.

[15] 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 '04)*, pages 45–58, San Francisco, CA, March/April 2004. USENIX Association.

[16] 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 '04)*, pages 129–143, San Francisco, CA, March/April 2004. USENIX Association.

[17] 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.

[18] 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.

[19] 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.

[20] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

[21] 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*.

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

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

[24] 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*.

[25] 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.

[26] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[27] 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-ala.html*. Visited May 15, 2020.

[28] 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.

[29] 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.

[30] Yi Cao, Arpit Jain, Kriti Sharma, Aruna Balasubramanian, and Anshul Gandhi. When to use and when not to use bbr: An empirical analysis and evaluation study. In *Proceedings of the Internet Measurement Conference*, pages 130–136, 2019.

[31] Zhen Cao, Geoff Kuenning, and Erez Zadok. Carver: Finding important parameters for storage system tuning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, February 2020. USENIX Association.

[32] Zhen Cao, Vasily Tarasov, Hari Raman, Dean Hildebrand, and Erez Zadok. On the performance variation in modern storage stacks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 329–343, Santa Clara, CA, February-March 2017. USENIX Association.

[33] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, July 2018. USENIX Association. Data set at *http://download.filesystems.org/auto-tune/ATC-2018-auto-tune-data.sql.gz*.

[34] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *USENIX Annual Technical Conference, (ATC)*, pages 893–907, Boston, MA, July 2018.

[35] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST)*, pages 209–223, 2020.

[36] Chandranil Chakraborttii and Heiner Litz. Learning i/o access patterns to improve prefetching in ssds. *ICML-PKDD*, 2020.

[37] 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.

[38] Hui Chen, Enqiang Zhou, Jie Liu, and Zhicheng Zhang. An rnn based mechanism for file prefetching. In *2019 18th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pages 13–16. IEEE, 2019.

[39] Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Machine learning for load balancing in the linux kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, Tsukuba, Japan, 2020. Association for Computing Machinery.

[40] 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*.

[41] 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.

[42] Giovanni Cherubini, Yusik Kim, Mark Lantz, and Vinodh Venkatesan. Data prefetching for large tiered storage systems. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 823–828, November 2017.

[43] Jungwook Choi, Swagath Venkataramani, Vijayalakshmi Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, and Pierce Chuang. Accurate and efficient 2-bit quantized neural networks. In *Proceedings of the 2nd SysML Conference*, 2019.

[44] CNTK, September 2020. *https://github.com/microsoft/CNTK*.

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

[46] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, Shanghai, China, 2017.

[47] 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*.

[48] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, November 2020.

[49] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R. Aberger, Kunle Olukotun, and Christopher Ré. High-accuracy low-precision training, 2018. arXiv preprint arXiv:1803.03383.

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

[51] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[52] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

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

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

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

[56] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *USENIX Annual Technical Conference*, pages 261–274, 2007.

[57] 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.

[58] dlib C++ Library, September 2020. *http://dlib.net/*.

[59] Bo Dong, Xiao Zhong, Qinghua Zheng, Lirong Jian, Jian Liu, Jie Qiu, and Ying Li. Correlation based file prefetching approach for hadoop. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 41–48. IEEE, 2010.

[60] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. {PCC}: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, 2015.

[61] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018.

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

[63] Embedded Learning Library (ELL), January 2020. *https://microsoft.github.io/ELL/*.

[64] Facebook. RocksDB. *https://rocksdb.org/*, September 2019.

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

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

[67] 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.

[68] Cory Fox, Dragan Lojpur, and An-I Andy Wang. Quantifying temporal and spatial localities in storage workloads and transformations by data path components. In *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, pages 1–10. IEEE, 2008.

[69] 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*.

[70] Gaddisa Olani Ganfure, Chun-Feng Wu, Yuan-Hao Chang, and Wei-Kuan Shih. Deepprefetcher: A deep learning framework for data prefetching in flash storage devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3311–3322, 2020.

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

[72] 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.

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

[74] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 1737–1746, Lille, France, 2015.

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

[76] 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.

[77] 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.

[78] Lawrence O Hall, Xiaomei Liu, Kevin W Bowyer, and Robert Banfield. Why are neural networks sometimes much more accurate than decision trees: an analysis on a bio-informatics problem. In *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme-System Security and Assurance (Cat. No. 03CH37483)*, volume 3, pages 2851–2856. IEEE, 2003.

[79] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting millisecond tail tolerance with fast rejecting SLO-aware OS interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 168–183, Shanghai, China, October 2017.

[80] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Banff, Alberta, November 2020. USENIX Association.

[81] 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.

[82] 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.

[83] 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.

[84] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[85] 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.

[86] 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*.

[87] Haiyan Hu, Yi Liu, and Depei Qian. I/o feature-based file prefetching for multi-applications. In *2010 Ninth International Conference on Grid and Cloud Computing*, pages 213–217. IEEE, 2010.

[88] 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 '11)*, San Jose, CA, February 2011. USENIX Association.

[89] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[90] 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.

[91] Intel Labs. Open Storage Toolkit. *https://sourceforge.net/projects/intel-iscsi/*.

[92] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059. PMLR, 2019.

[93] Jeya Vikranth Jeyakumar, Joseph Noor, Yu-Hsi Cheng, Luis Garcia, and Mani Srivastava. How can i explain this to you? an empirical study of deep neural network explanation methods. *Advances in Neural Information Processing Systems*, 2020.

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

[95] 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 '05)*, pages 337–350, San Francisco, CA, December 2005. USENIX Association.

[96] Chet Juszczak. Improving the write performance of an NFS server. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, San Francisco, California, 1994. USENIX Association.

[97] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, pages 237–285, 1996.

[98] 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.

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

[100] 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.

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

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

[103] Jack Kiefer, Jacob Wolfowitz, et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[104] Doowon Kim, Bum Jun Kwon, Kristián Kozák, Christopher Gates, and Tudor Dumitras. The broken shield: Measuring revocation effectiveness in the windows code-signing PKI. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 851–868, 2018.

[105] 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.

[106] 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.

[107] 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.

[108] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A learned database system. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, January 2019.

[109] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[110] Thomas M. Kroeger and Darrell D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *USENIX Annual Technical Conference*, pages 105–118, Boston, MA, June 2001.

[111] 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.

[112] Arezki Laga, Jalil Boukhobza, M. Koskas, and Frank Singhoff. Lynx: A learning Linux prefetching mechanism for SSD performance model. In *5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, August 2016.

[113] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Deep convolutional neural network inference with floating-point weights and fixed-point activations, 2017. arXiv preprint arXiv:1703.03073.

[114] Sangmin Lee, Soon J Hyun, Hong-Yeon Kim, and Young-Kyun Kim. Aps: adaptable prefetching scheme to different running environments for concurrent read streams in distributed file systems. *The Journal of Supercomputing*, 74(6):2870–2902, 2018.

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

[116] 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.

[117] Daixuan Li and Jian Huang. A learning-based approach towards automated tuning of ssd configurations. *arXiv preprint arXiv:2110.08685*, 2021.

[118] 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.

[119] Z. Li, A. Mukker, and E. Zadok. On the importance of evaluating storage systems' $costs. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, 2014.

[120] Shuang Liang, Song Jiang, and Xiaodong Zhang. Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 64–64. IEEE, 2007.

[121] Jianwei Liao, Francois Trahay, Guoqiang Xiao, Li Li, and Yutaka Ishikawa. Performing initiative data prefetching in distributed file systems for cloud computing. *IEEE Transactions on cloud computing*, 5(3):550–562, 2015.

[122] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[123] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, June 2016.

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

[125] Linux. Linux kernel module signing facility. *https://www.kernel.org/doc/html/v4.19/admin-guide/module-signing.html?highlight=signing*, January 2021.

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

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

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

[129] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for C++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 541–556, Lausanne, Switzerland, March 2020.

[130] 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 '16), Santa Clara, CA, February 2016. USENIX Association.

[131] Paul Manning. Best practices for running vmware vsphere on network attached storage. *https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-nfs-bestpractices-white-paper-en.pdf*, 2009.

[132] 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.

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

[134] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, Haifa, Israel, June 2010.

[135] Anusha Nalajala, T Ragunathan, Sri Harsha Tavidisetty Rajendra, Nagamlla Venkata Sai Nikhith, and Rathnamma Gopisetty. Improving performance of distributed file system through frequent block access pattern-based prefetching algorithm. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–7. IEEE, 2019.

[136] Atul Negi and P Kishore Kumar. Applying machine learning techniques to improve Linux process scheduling. In *TENCON 2005-2005 IEEE Region 10 Conference*, pages 1–6. IEEE, 2005.

[137] 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.

[138] 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.

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

[140] 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.

[141] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS 2019)*, pages 8024–8035, Vancouver, BC, Canada, December 2019.

[142] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58(347-352):240–242, 1895.

[143] 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.

[144] 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.

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

[146] 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.

[147] 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.

[148] Yiming Qiu, Hongyi Liu, Thomas Anderson, Yingyan Lin, and Ang Chen. Toward reconfigurable kernel datapaths with learned optimizations. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 175–182, 2021.

[149] 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.

[150] Gabriëlle Ras, Marcel van Gerven, and Pim Haselager. Explanation methods in deep learning: Users, values, concerns and challenges. In *Explainable and interpretable models in computer vision and machine learning*, pages 19–36. Springer, 2018.

[151] Natarajan Ravichandran and Jehan-François Pâris. *Making early predictions of file accesses*. PhD thesis, University of Houston, 2005.

[152] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[153] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[154] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, (ISCA)*, pages 561–574, Toronto, ON, Canada, June 2017.

[155] Wojciech Samek, Grégoire Montavon, Sebastian Lapuschkin, Christopher J Anders, and Klaus-Robert Müller. Toward interpretable machine learning: Transparent deep neural networks and beyond. *arXiv e-prints*, pages arXiv–2003, 2020.

[156] 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.

[157] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating performance and energy in file system server workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '10)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[158] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *SIGMETRICS*, June 1998.

[159] Elizabeth AM Shriver, Christopher Small, and Keith A Smith. Why does file system prefetching work? In *USENIX Annual Technical Conference, General Track*, pages 71–84, 1999.

[160] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. Running neural networks on the nic. *arXiv preprint arXiv:2009.02353*, 2020.

[161] Filippo Sironi, Davide B Bartolini, Simone Campanoni, Fabio Cancare, Henry Hoffmann, Donatella Sciuto, and Marco D Santambrogio. Metronome: operating system level performance management via self-adaptive computing. In *Proceedings of the 49th Annual Design Automation Conference*, pages 856–865, 2012.

[162] 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.

[163] SOD - An Embedded, Modern Computer Vision and Machine Learning Library, September 2020. *https://sod.pixlab.io/*.

[164] Gagan Somashekar and Anshul Gandhi. Towards optimal configuration of microservices. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 7–14, 2021.

[165] Kalyanasundaram Somasundaram. The impact of slow nfs on data systems. *https://engineering.linkedin.com/blog/2020/the-impact-of-slow-nfs-on-data-systems*, June 2020.

[166] 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.

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

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

[169] 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.

[170] Pradeep Subedi, Philip Davis, Shaohua Duan, Scott Klasky, Hemanth Kolla, and Manish Parashar. Stacker: An autonomic data movement engine for extreme-scale data staging-based in-situ workflows. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 920–930. IEEE, 2018.

[171] 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.

[172] 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.

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

[174] 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.

[175] 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.

[176] 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.

[177] TensorFlow lite, January 2020. *https://www.tensorflow.org/lite*.

[178] 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.

[179] 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.

[180] Nancy Tran and Daniel A Reed. Automatic arima time series modeling for adaptive i/o prefetching. *IEEE Transactions on parallel and distributed systems*, 15(4):362–377, 2004.

[181] Transaction Processing Performance Council. TPC benchmark H (decision support). *www.tpc.org/tpch*, 1999.

[182] 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.

[183] Ahsen J Uppal, Ron C Chiang, and H Howie Huang. Flashy prefetching for high-performance flash drives. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[184] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[185] Santiago Vargas, Rebecca Drucker, Aiswarya Renganathan, Aruna Balasubramanian, and Anshul Gandhi. Bbr bufferbloat in dash video. In *Proceedings of the Web Conference 2021*, pages 329–341, 2021.

[186] 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.

[187] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage (HotStorage '18)*, Boston, MA, July 2018. USENIX.

[188] 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.

[189] Gary AS Whittle, J-F Pâris, Ahmed Amer, Darrell DE Long, and Randal Burns. Using multiple predictors to improve the accuracy of file access predictions. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.*, pages 230–240. IEEE, 2003.

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

[191] Wikipedia. Recurrent neural network. *https://en.wikipedia.org/wiki/Recurrent_neural_network*.

[192] 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.

[193] Jiwoong Won, Oseok Kwon, Junhee Ryu, Dongeun Lee, and Kyungtae Kang. ifetcher: User-level prefetching framework with file-system event monitoring for linux. *IEEE Access*, 6:46213–46226, 2018.

[194] Fengguang Wu, Hongsheng Xi, and Chenfeng Xu. On the design of a new Linux readahead framework. *Operating Systems Review*, 42:75–84, 2008.

[195] Chenfeng Xu, Hongsheng Xi, and Fengguang Wu. Evaluation and optimization of kernel file readaheads based on markov decision models. *The Computer Journal*, 54(11):1741–1755, 2011.

[196] Xiaofei Xu, Zhigang Cai, Jianwei Liao, and Yutaka Ishiakwa. Frequent access pattern-based prefetching inside of solid-state drives. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 720–725. IEEE, 2020.

[197] 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.

[198] Gala Yadgar, MOSHE Gabel, Shehbaz Jaffer, and Bianca Schroeder. Ssd-based workload characteristics and their performance implications. *ACM Transactions on Storage (TOS)*, 17(1):1–26, 2021.

[199] Chuan-Kai Yang, Tulika Mitra, and Tzi-cker Chiueh. A decoupled architecture for application-specific file prefetching. In Chris G. Demetriou, editor, *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pages 157–170. USENIX, 2002.

[200] 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.

[201] 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.

[202] 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.

[203] Shengan Zheng, Hong Mei, Linpeng Huang, Yanyan Shen, and Yanmin Zhu. Adaptive prefetching for accelerating read and write in nvm-based file systems. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 49–56. IEEE, 2017.

[204] 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 '05)*, pages 323–336, San Francisco, CA, December 2005. USENIX Association.