# Improving Storage Systems Using Machine Learning

IBRAHIM UMIT AKGUN, ALI SELMAN AYDIN, ANDREW BURFORD, MICHAEL MCNEILL, MICHAEL ARKHANGELSKIY, and EREZ ZADOK,
Stony Brook University

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 article, we describe our proposed ML architecture, called KML. We developed a prototype KML architecture and applied it to two case studies: optimizing readahead and NFS read-size values. Our experiments show that KML consumes less than 4 KB 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.

CCS Concepts: • **Software and its engineering** → **File systems management**; • **Computing methodologies** → *Machine learning;*

Additional Key Words and Phrases: Operating systems, storage systems, Machine Learning, storage performance optimization

**ACM Reference format:**
Ibrahim Umit Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. 2023. Improving Storage Systems Using Machine Learning. *ACM Trans. Storage* 19, 1, Article 9 (January 2023), 30 pages.
https://doi.org/10.1145/3568429

## 1 INTRODUCTION

Computer hardware, software, storage, and workloads are constantly changing. Storage performance heavily depends on workloads and the precise system configuration [14, 82]. Storage systems and OSs include many parameters that can affect overall performance [13, 15, 104]. Yet, users often do not have the time or expertise to tune these parameters. Worse, the storage and OS

This work was made possible in part thanks to Dell-EMC, NetApp, Facebook, and IBM support; a SUNY/IBM Alliance award; and NSF awards CNS-1729939, CNS-1900706, CCF-1918225, CNS-1951880, CNS-2106263, CNS-2106434, and CNS-2214980.
Authors' address: I. U. Akgun, A. S. Aydin, A. Burford, M. McNeill, M. Arkhangelskiy, and E. Zadok, Department of Computer Science, New Computer Science Building, Stony Brook University, Stony Brook, NY 11794-2424; emails: iakgun@cs.stonybrook.edu, aaydin@cs.stonybrook.edu, aburford@cs.stonybrook.edu, mmcneill@cs.stonybrook.edu, markhangelsk@cs.stonybrook.edu, ezk@cs.stonybrook.edu.

1553-3077/2023/01-ART9 $15.00
https://doi.org/10.1145/3568429

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 [4]. 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.

**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 [24, 50], for database query optimization [49], improved caching [90], cache eviction policies [97], I/O scheduling [40], and more.

In this article, 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, lightweight system called *KML*, for conducting ML training and prediction for storage systems. 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.

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 1 second under changing and even mixed workloads. Overall, our approach to storage systems optimization using ML is a continuous *observe-and-tune* paradigm.

This article 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)
(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 (4 KB), well worth the overall I/O improvements.

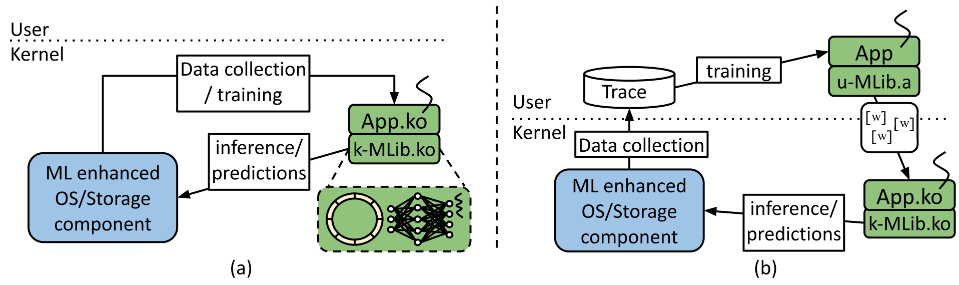

Fig. 1. Two different operational modes that we built to achieve a high efficiency ML framework for tuning OS-level storage systems: (a) kernel space training and inference and (b) offline user space training and kernel space inference.

Next, Section 2 describes KML's design. Section 3 describes our two use cases (readahead and NFS `rsize`). A detailed evaluation of KML and two use cases are in Section 4. We survey related work in Section 5 and conclude in Section 6.

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

### 2.1 Design Overview

*Easy to develop and extend.* In Figure 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 2.3). 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 and 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 and 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 2.4.

## 2.2 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 [66], 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, and so on.

*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 [78]. KML also includes **Stochastic Gradient Descent** (**SGD**) which uses the gradients computed using back-propagation to optimize the neural network weights.

## 2.3 KML's Modular Design

We now elaborate on KML's operation modes: (i) in-kernel training and inference (see Figure 1(a)) and (ii) user space training and in-kernel inference (see Figure 1(b)).

*Training in kernel space.* We use the readahead use case to describe how KML works in kernel training and inference mode. Figure 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 dataflow. Numbered boxes refer to transitions happening between the components.

As we mentioned in Section 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 2, ①). The readahead module then extracts features and transfers them to the KML framework to be normalized (Figure 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

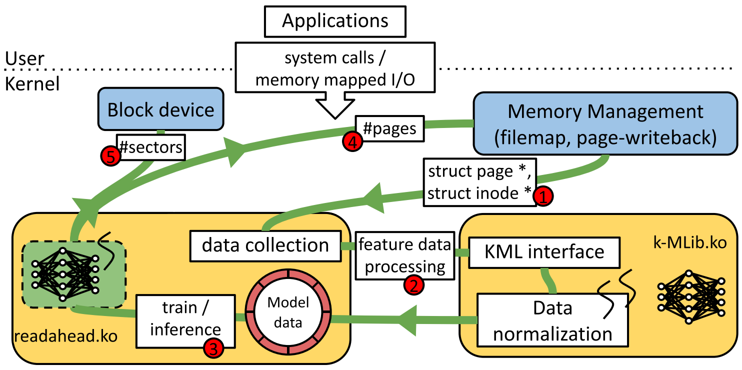

Fig. 2. KML kernel space training and inferencing architecture.

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 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 2, ④) or a per-device basis (Figure 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 [5]. Nevertheless, tracing tools like LTTng [63] 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 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 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 [5]. 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 model in user space, varying several hyper-parameters; we used

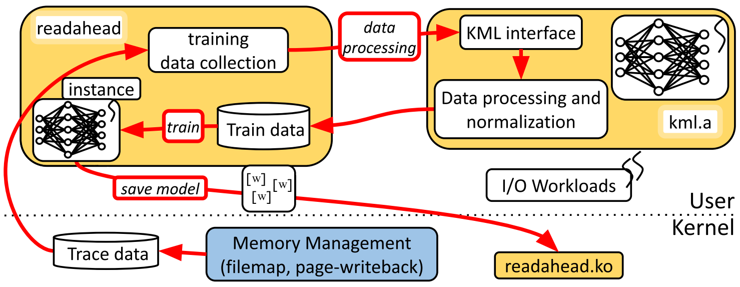

Fig. 3. KML user-space training and kernel-space inference architecture.

Table 1. KML API Examples

```
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);
```

Tune [59] 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 2.6); Table 1 presents a few examples of the KML API.

## 2.4 Computational and 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 [21, 25, 37, 40, 79]. Quantization can help reduce computational and memory overheads, but it reduces accuracy [43]. 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 [19, 60]. However, fixed-point representation works within fixed ranges which can result in numerical instability [53]. 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 and 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 4.3, we evaluate the impact of subsampling windows on overheads, prediction accuracy, and overall I/O performance.

## 2.5 Stability and 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 [3]. 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 [3, 44, 74, 80]. 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.

### 2.6 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).*

## 3 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.

### 3.1 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 3). Our goal is to predict optimal readahead sizes while running under dynamic I/O workloads.

*Studying the problem.* We experimented with running four different RocksDB [34] benchmarks, each with 20 different readahead sizes (8–1,024), 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 [63] to collect trace data, which we then used for finding useful features for the readahead problem. We captured most page cache tracepoints [28] (e.g.,

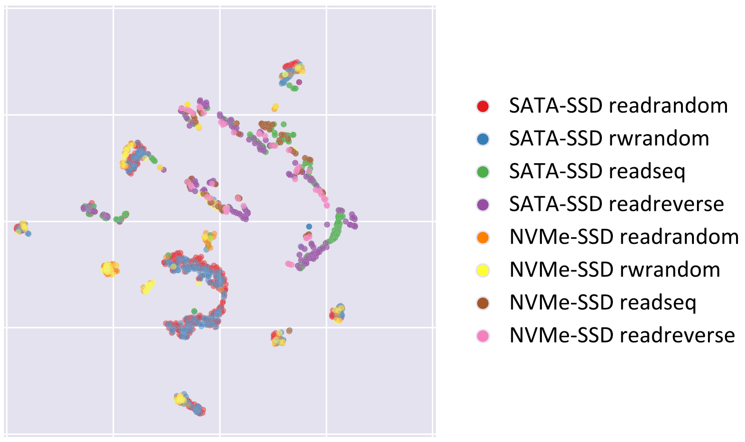

Fig. 4. t-SNE visualization of readahead normalized features that are generated from both NVMe-SSD and SATA-SSD traces. Axes are intentionally omitted because the dimensions are generated by t-SNE and do not represent any specific data.

`add_to_page_cache`, `writeback_dirty_page`). We collected and processed over 20 GB 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 and normalization.* We summarize the input data at 1-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.

Building ML solutions for OS problems requires domain expertise on the target OS module. To this end, we have investigated what features best fit the readahead problem. For example, our own intuition led us to select features based on how fast I/O requests can be processed and what type of access patterns emerge. Similar features have been used for workloads characterization or other purposes before [9, 77, 81, 84]. During the feature-extraction period for the readahead problem, we tried various features and reduced them using feature importance analysis [71, 72].

*Why we chose ML for this use case.* After studying the readahead problem, we wanted to explore whether ML 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 4 shows a t-SNE [96] 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 [47, 76], 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 [10]. We also used Tune [59] to optimize the learning rate and momentum. We approached the readahead problem by modeling it as a regression problem. Due to the large search space for readahead sizes, we could reach a similar prediction accuracy only with large regression models. Thus, the large regression model for the readahead problem has higher computational and memory overheads, which conflicted with our desire and vision of designing efficient ML approaches for storage systems. 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 the 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 performance [11]. Using Pearson correlation analysis [71], 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. For example, the optimal readahead value for `readrandom` is 16 and for `readseq` is 640. Experiment details and the optimal readahead values for all the workloads are included in our code base, released via GitHub. In Section 4.4, 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 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

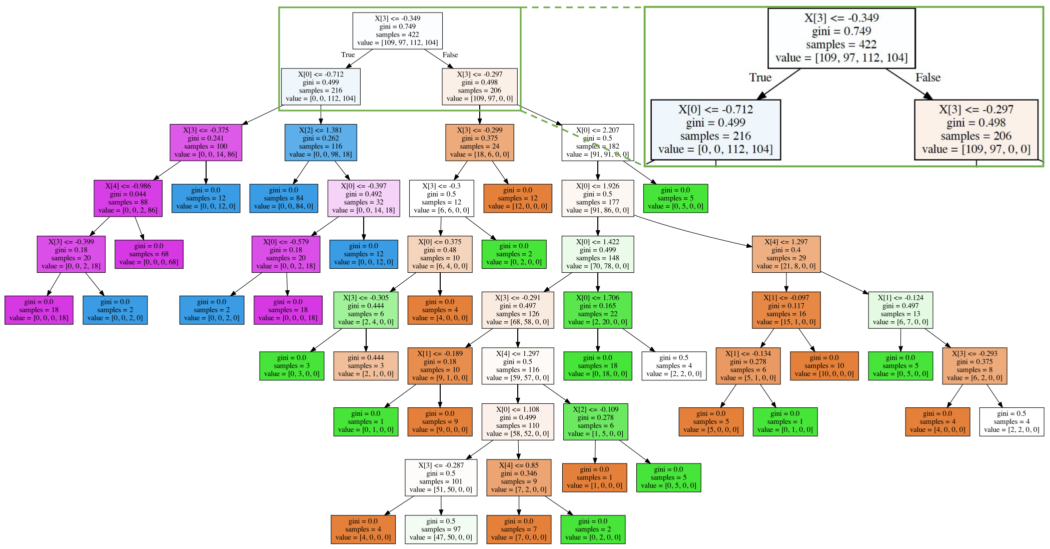

Fig. 5. A readahead decision tree is built to classify RocksDB workloads running on a NVMe-SSD backed device. Colors denote workload classes: orange for `readrandom` workload, green for `rw-random`, blue for `readseq`, and purple for `readreverse`.

performance improvement even for SATA-SSDs (see Section 4.4). 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 [94] queries running on MySQL [69] 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). 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 DT 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 classifying a particular readahead workload takes nine IF-THEN decisions, we can observe that the readahead DT 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 2.5, KML supports DTs because DT trees are more explainable than neural networks and run considerably faster. Although the DTs are more explainable, it is still hard to interpret the readahead DT 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 DT path. Nevertheless, even with an improved 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 4.4).

*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 [8]. 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 4.4).

## 3.2 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 [65], hosting NoSQL databases [89], 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 and 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).

## 4 EVALUATION

Our evaluation proceeds as follows: First, we explain our evaluation goals in Section 4.1. We then describe the testbed design and benchmarks that we used to evaluate the readahead and NFS `rsize` neural networks in Section 4.2. In Section 4.3, we provide performance details regarding KML's training and inference. Section 4.4 shows how the readahead ML models improve performance. Finally, in Section 4.5, we present our evaluation of the `rsize` neural network model for NFS.

### 4.1 Evaluation Goals

Our primary evaluation goal is to show that using ML techniques inside the OS can be used 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 DT model for the readahead problem to have *comparable* results since DTs 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.

### 4.2 Testbed

We ran the benchmarks on two identical Dell R-710 servers, each with two Intel Xeon quad-core CPUs (2.4 GHz, 8 hyper-threads), 24 GB of RAM, and an Intel 10 GbE NIC. In some experiments, we intentionally configured the system with only 1 GB of memory to force more memory pressure on the I/O system; but we also show experiments with the full 24 GB 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 200 GB as our SATA-SSD device and a Samsung MZ1LV960HCJH-000MU 960 GB 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 148 GB SAS boot drive for CentOS, utilities, and RocksDB benchmark software. We used 10 GbE 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 [34] 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 1 GB RAM systems, we configured a RocksDB database of twice the size (2 GB). Choosing the dataset size to be twice the size of memory is a well known "rule of thumb" configuration to create a realistic strorage cache behavoir [91]. 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 1 GB RAM as not a realistic system configuration. Therefore, we also executed all the benchmarks in this article with a 56 GB RocksDB database running on the same system configured with 24 GB RAM. The results are showing similar improvements and there are no significant performance-trend differences (see Section 4.4). Nevertheless, because we ran experiments with more RAM and for a longer period of time, we noticed some interesting findings which are explained in Section 4.4.

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 [12]).

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 (1 GB RAM) to increase system and page-cache pressure.

We also experimented with our readahead network model using TPC-H [94] queries running on MySQL [69], to evaluate how generalizable and effective the readahead neural network is to an entirely different workload. In this article, 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.

## 4.3 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, DT models. It is vital that an ML

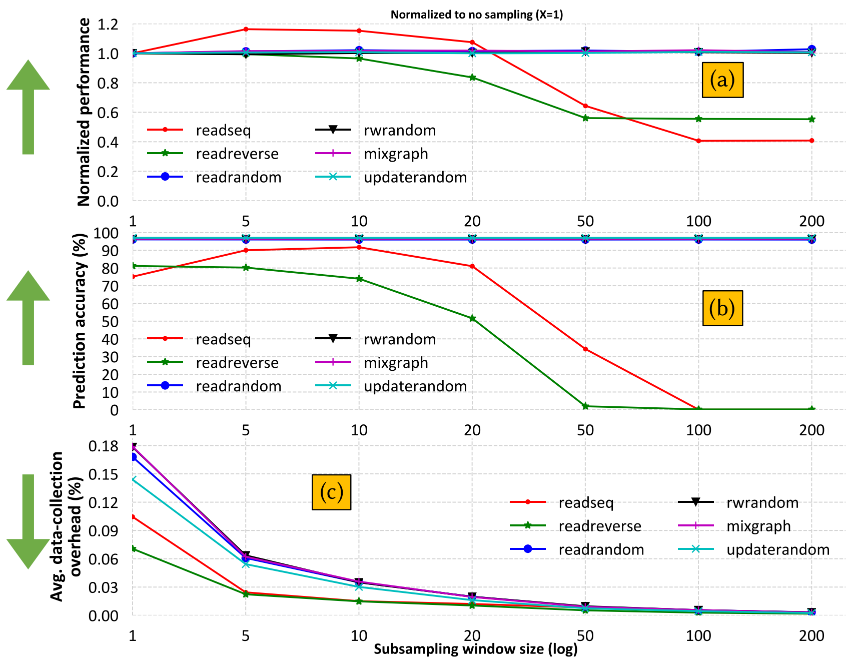

Fig. 6. Performance (a), prediction accuracy (b), and CPU overheads (c) in seven different subsampling window sizes for the per-disk readahead neural network. Upward green arrows denote that higher is better.

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 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 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 6(a) and (b). Figure 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 1 to 5 or 10 *actually improved* both prediction accuracy and performance; this is because `readseq` keeps the I/O subsystem busy at near maximum bandwidth, and increasing the subsampling window size reduced short-term noise that resulted in more frequent mispredictions.

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

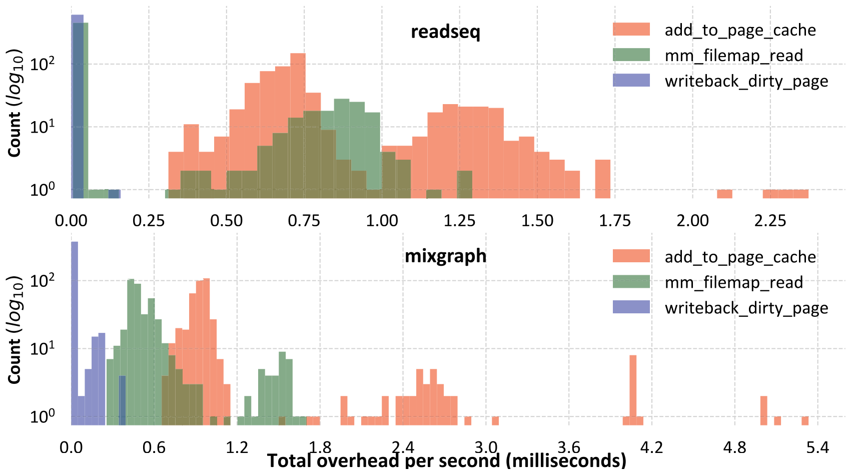

Fig. 7. Distribution of total data collection overhead (milliseconds) in every second when `readseq` and `mixgraph` workloads are running.

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 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 21 $\mu$s 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 1.7 ms on average. We measured that the readahead DT inference takes only 8 $\mu$s (using the same feature vector). The readahead neural network and DT have the same data preprocessing 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 3.1, our readahead neural network prototype offloads training to the user level. We measured the time to perform one training iteration in user level at 51 $\mu$s 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-gigabyte systems. The readahead DT occupies only 2,432 bytes of dynamic memory during initialization. The DT model does not allocate dynamic memory during inference. Lastly, the kernel module `readhead.ko` has a binary memory footprint of 432 KB and the kernel module `nfs.ko` is 636 KB, while the KML framework itself (`k-Mlib.ko`) has a memory footprint of 5.5 MB.

*Practicality and scalability.* Our vision is that KML could enable a future where traditional heuristics are gradually replaced with ML-based approaches to improve storage and network I/O

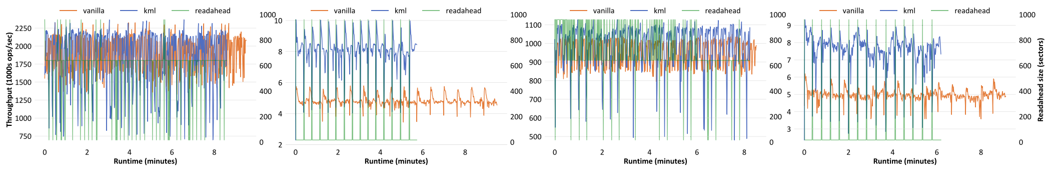

Fig. 8. Running four back-to-back RocksDB workloads in order from left to right: `readsequential`, `readrandom`, `readreverse`, then `mixgraph`. Here, we started with the default readahead value; thereafter, the last value set in one workload was the one used in the next run. For each of the four graphs, 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. Periodic spikes that we observed in `readrandom` and `mixgraph` denote the experiments' starting points because of multiple iterations of benchmarks. Again, we see KML adapting, picking optimal readahead values, occasionally mis-predicting but quickly recovering, hence overall throughput was better.

performance. In Section 4.4, 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 tradeoff 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.

## 4.4 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.* To show the readahead model's ability to adapt to changing workloads, we experimented with running different workloads back to back. We observed how the readahead model reacted to the workload changes and tuned readahead values. Figure 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 may change every few minutes. The $X$ axes indicate the runtime 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

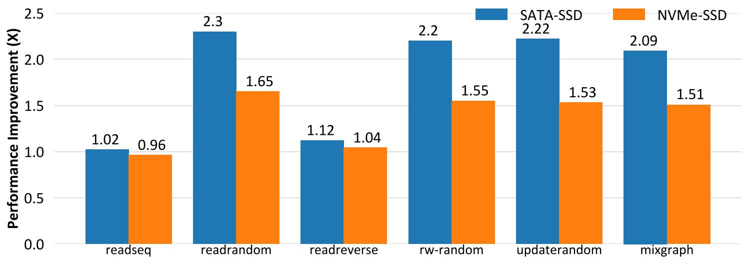

Fig. 9. Readahead neural network performance improvements (×) for RocksDB benchmarks on SATA-SSD and NVMe-SSD across all six workloads, normalized to vanilla (1.0×).

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 8, KML adapts quickly to changing workloads by tuning the readahead value in about 1 s.

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 [56].

*Read-sequential workloads.* Out of the six workloads we ran, Figure 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 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.

*Summary of readahead neural network results.* We summarize all readahead neural network results in Figure 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., 24 GB) and the database size set to be relatively large, 65 GB (compared to a 2 GB baseline 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 [8]. In Figure 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 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

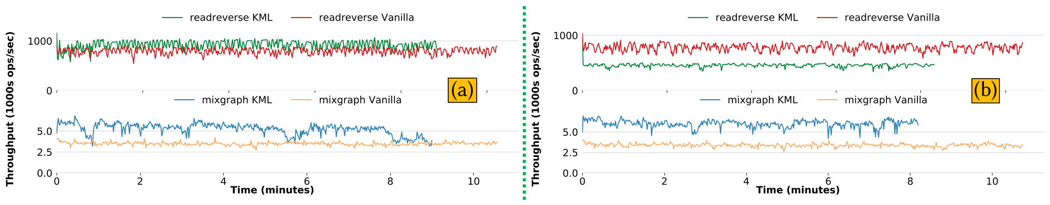

Fig. 10. Mixed workload results on a timeline, comparing the readahead neural network model running on per-file basis ("a", left) vs. per-disk basis ("b", right).

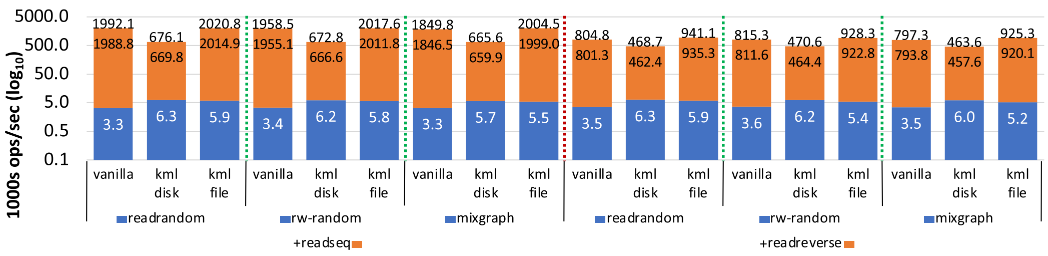

Fig. 11. Mixed workload results. We ran sequential and random workload combinations on the same NVMe-SSD device. Each unique combination is tested with the readahead neural network running in per-disk basis (kml disk) and per-file basis (kml file) and compared against vanilla results. The model running in per-file basis outperformed both vanilla and per-disk modes.

mixed—even sequential ones or ones with regular patterns—the mix looks more like a purely random workload at the disk level.

Figure 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 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 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: mixing workloads looks more random-like at the disk level. The per-disk readahead ML model predicts these as `readrandom` or `readrandomwriterandom`, which coincidentally fits this part of the workload, but hurts non-random workloads. However, the per-file readahead ML model *improves both the sequential and random part of the mixed workloads*. Thanks to KML's versatile architecture, we adapted the readahead ML model to two different working modes and *improved* page cache performance for mixed workloads; these are considered challenging tests for storage systems.

*DT evaluation.* In addition to the neural network model, we implemented a DT model for the readahead problem to compare the two ML approaches on the same problem. We tested the readahead DT the same way. Figure 12 shows that there is a performance improvement for workloads with a random component. For the readahead DT, 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%.

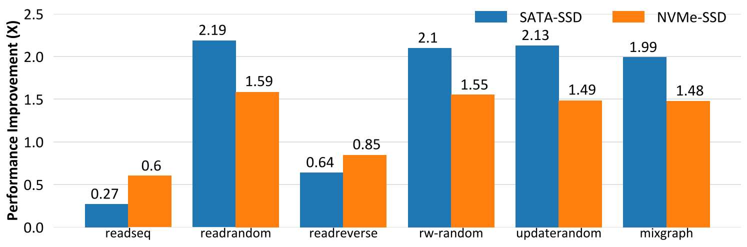

Fig. 12. Readahead decision tree performance improvements (×) for RocksDB benchmarks on SATA-SSD and NVMe-SSD devices across all six workloads, normalized to vanilla (1.0×).

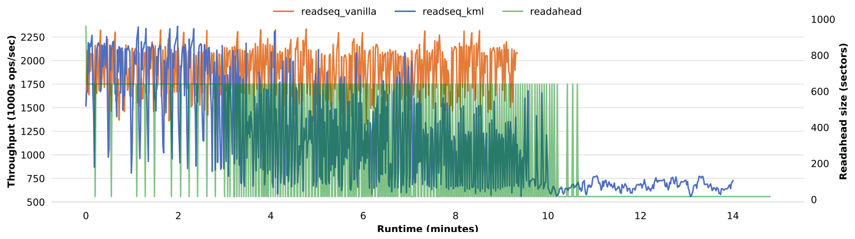

Fig. 13. Performance timeline graph for tuning with KML decision tree while running readseq workload on NVMe-SSD.

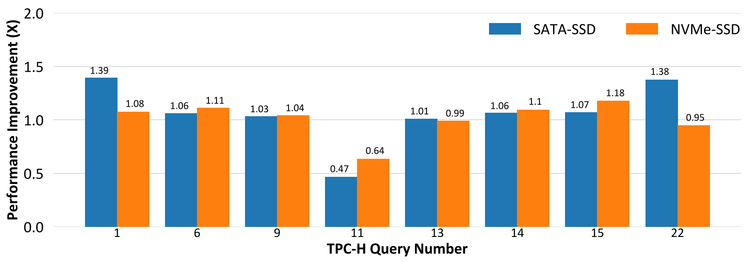

Fig. 14. Readahead neural network performance improvements (×) for TPC-H queries on SATA-SSD and NVMe-SSD devices, normalized to vanilla (1.0×).

We investigated this performance degradation. Figure 13 shows the readseq workload running on a RocksDB instance stored on an NVMe-SSD. Here, the readahead DT predicts the workload correctly in the first 3 minutes, despite some fluctuations. Afterwards, the DT 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 DTs [38]. Specifically, when 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.

*TPC-H benchmarks.* As we mentioned in Section 4.2, 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 14 shows performance improvements as much as 39% for most query types. For query 11, however, the readahead

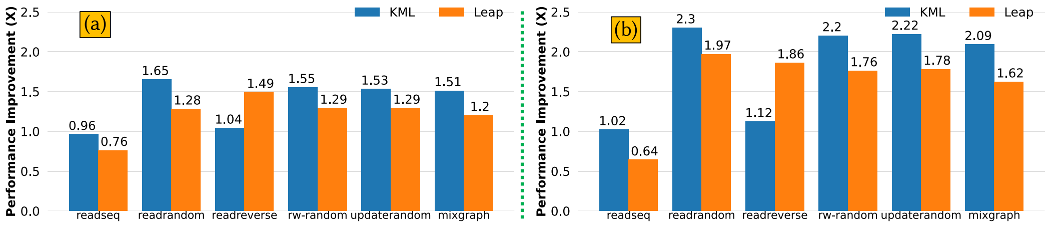

Fig. 15. Performance improvement comparisons between LEAP [6] and KML for RocksDB benchmarks on NVMe-SSD (“a”, left) and SATA-SSD (“b”, right).

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.

*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* [6]. 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 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 56 GB RocksDB database running on 24 GB 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 `floats` 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 `doubles` 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, and applications; it consistently performed significantly better than baseline and LEAP.

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 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 9 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 1 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 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 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.

## 4.5 NFS Evaluation

Figure 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×. This demonstrates the significant potential of KML.

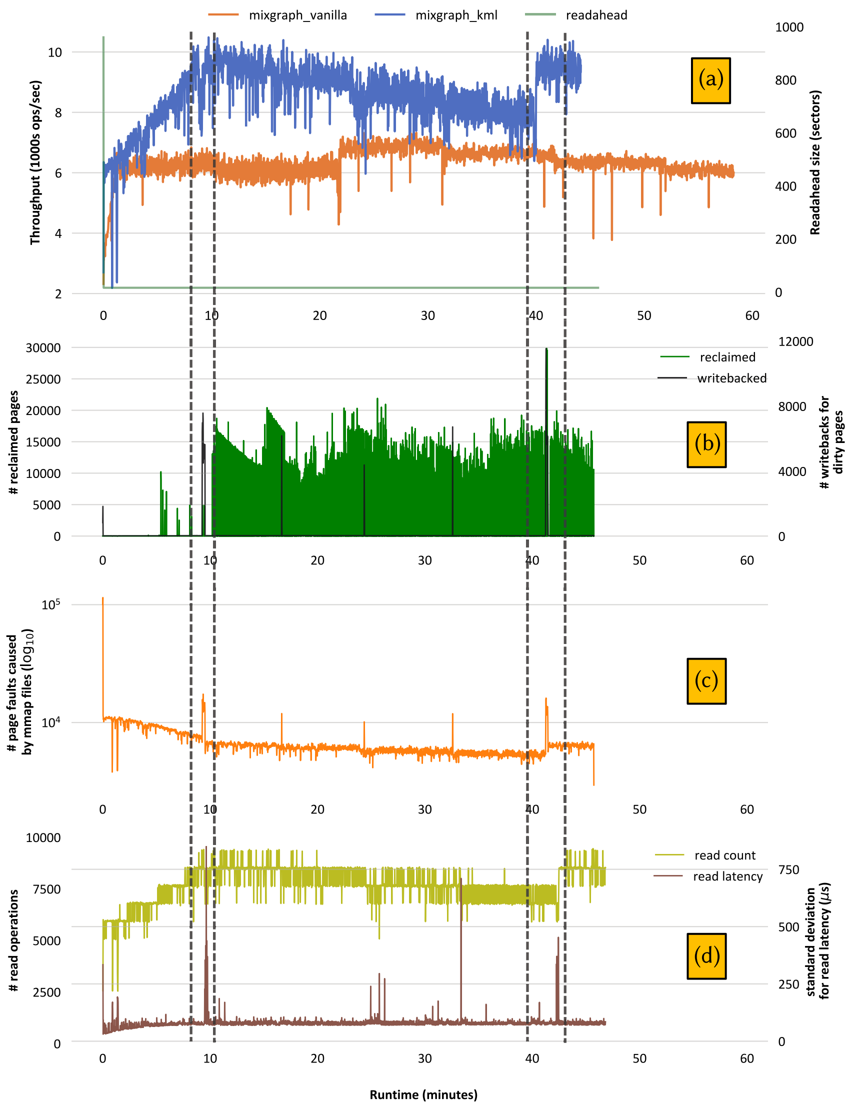

Fig. 16. Throughput analysis for running `mixgraph` on 24 GB memory with a 56 GB RocksDB database. In (a) we show the throughput timeline and improvements for `mixgraph` running with KML. We can see three phases of `mixgraph`'s execution, demarcated by double vertical dashed lines: (1) startup, (2) stabilize 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.

## 5 RELATED WORK

*ML in systems and storage.* In follow-up work to Mittos [39], 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 [40]. There are notable differences between that system and our KML. That system was trained offline using

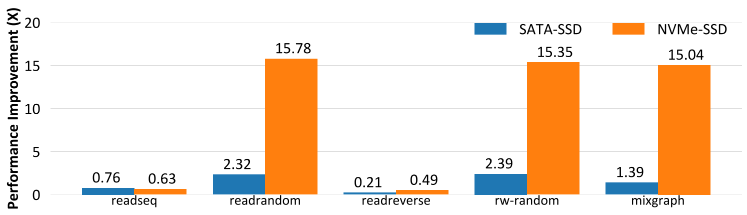

Fig. 17. Performance improvements (×) for RocksDB benchmarks on SATA-SSD and NVMe-SSD devices across all six workloads running on NFS, normalized to vanilla (1.0×).

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. [52] improved readahead performance in the Linux Kernel with Markov chain models, netting a 50% I/O performance improvement in TPC-H [94] 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 4 KB, in comparison to Laga et al.'s much larger 94 MB Markov chain model.

Parameter tuning for storage and operating systems has been a challenge and researchers approached this problem using control theory [86] and data distribution analysis for storage clusters [2]. Some research has attempted to apply ML techniques to OS task scheduling [19, 68], 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 [55], memory allocation [64], TCP congestion [32], building smart NICs [85], predicting index structures in key-value stores [24, 50], offline black-box storage parameter optimization [16], reconfigurable kernel datapaths [73], local and distributed caching [90, 97], database query optimization [49], and cloud resource management [23, 26, 27, 88].

*ML 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 [1], PyTorch [70], and CNTK [22]. Conversely, libraries like ELL [33], Tensorflow Lite [92], SOD [87], Dlib [30], and Tiny Training Engine [61] specialize to run on resource-constrained or on-device environments, KML differentiates itself by targeting OS-level applications and is 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 [29, 51, 83, 84] and see use in distributed systems [18, 20, 31, 54, 57, 58, 67, 93]. Many have attempted to build statistical models to optimize and tune systems [35, 83, 84]. 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 [7, 29, 42, 51, 95, 98, 101, 103]. Predicting file accesses using handcrafted 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 [17, 36, 75, 102, 106]. 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 [100] or to require application-level changes [105]. 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 [45].

## 6 CONCLUSION

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. Our preliminary results show that, for a readahead problem, we can boost I/O throughput by up to 2.3× without imposing significant CPU/memory overheads. For the NFS `rsize` problem, the improvement was up to 15×. These I/O throughput improvements far outweigh the small memory and CPU consumption of KML.

*Future work.* We plan on using KML to tune knobs for other OS subsystems, e.g., packet and I/O schedulers, and networking. We are adding ML techniques to KML, such as reinforcement learning [46], which can be a better fit for solving certain OS problems. To support more advanced ML approaches (e.g., **Recurrent Neural Networks** (**RNNs**) [99]) and **Long Short-Term Memory** (**LSTM**) [41]), 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 [48, 62].

## REFERENCES

[1] 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. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*. 265–283.

[2] 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. 2005. Ursa minor: Versatile cluster-based storage. In *Proceedings of the FAST '05 Conference on File and Storage Technologies, 2005.* USENIX.

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

[4] Ibrahim 'Umit' Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. 2021. A machine learning framework to improve storage system performance. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage (HotStorage'21).* ACM, Virtual. https://doi.org/10.1145/3465332.3470875

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

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

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

[8] 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. 2019. *Data Storage Research Vision 2025: Report on NSF Visioning Workshop Held May 30–June 1, 2018*. Technical Report. National Science Foundation. https://dl.acm.org/citation.cfm?id=3316807.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*. ACM, New York, NY, 53–64. https://doi.org/10.1145/2254756.2254766

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

[11] Leo Breiman. 2001. Random forests. *Machine Learning* 45, 1 (2001), 5–32.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*. 209–223.

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

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

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

[16] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. 2018. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *USENIX Annual Technical Conference (ATC'18)*. 893–907.

[17] Chandranil Chakraborttii and Heiner Litz. 2020. Learning I/O access patterns to improve prefetching in SSDs. *ICML-PKDD* (2020).

[18] Hui Chen, Enqiang Zhou, Jie Liu, and Zhicheng Zhang. 2019. An RNN based mechanism for file prefetching. In *2019 18th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES'19)*. IEEE, 13–16.

[19] Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. Machine learning for load balancing in the Linux kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*. Association for Computing Machinery. https://doi.org/10.1145/3409963.3410492

[20] Giovanni Cherubini, Yusik Kim, Mark Lantz, and Vinodh Venkatesan. 2017. Data prefetching for large tiered storage systems. In *2017 IEEE International Conference on Data Mining (ICDM'17)*. 823–828. https://doi.org/10.1109/ICDM.2017.99

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

[22] CNTK 2020. CNTK. (Sept. 2020). https://github.com/microsoft/CNTK.

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

[24] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/dai.

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

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

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

[28] Mathieu Desnoyers. 2016. Using the Linux Kernel Tracepoints. (2016). https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.

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

[30] Dlib 2020. dlib C++ Library. (Sept. 2020). http://dlib.net/.

[31] Bo Dong, Xiao Zhong, Qinghua Zheng, Lirong Jian, Jian Liu, Jie Qiu, and Ying Li. 2010. Correlation based file prefetching approach for hadoop. In *2010 IEEE 2nd International Conference on Cloud Computing Technology and Science*. IEEE, 41–48.

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

[33] ELL 2020. Embedded Learning Library (ELL). (Jan. 2020). https://microsoft.github.io/ELL/.

[34] Facebook. 2019. RocksDB. (Sept. 2019). https://rocksdb.org/.

[35] Cory Fox, Dragan Lojpur, and An-I Andy Wang. 2008. 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*. IEEE, 1–10.

[36] Gaddisa Olani Ganfure, Chun-Feng Wu, Yuan-Hao Chang, and Wei-Kuan Shih. 2020. 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 (2020), 3311–3322.

[37] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*. 1737–1746.

[38] Lawrence O. Hall, Xiaomei Liu, Kevin W. Bowyer, and Robert Banfield. 2003. 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)*, Vol. 3. IEEE, 2851–2856.

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

[40] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on unpredictable flash storage. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/hao.

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

[42] Haiyan Hu, Yi Liu, and Depei Qian. 2010. I/o feature-based file prefetching for multi-applications. In *2010 9th International Conference on Grid and Cloud Computing*. IEEE, 213–217.

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

[44] Jeya Vikranth Jeyakumar, Joseph Noor, Yu-Hsi Cheng, Luis Garcia, and Mani Srivastava. 2020. How can I explain this to you? An empirical study of deep neural network explanation methods. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)*, Curran Associates Inc., Red Hook, NY, 12.

[45] Chet Juszczak. 1994. Improving the write performance of an NFS server. In *Proceedings of the USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, San Francisco, CA, 1. http://dl.acm.org/citation.cfm?id=1267074.1267094.

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

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

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

[49] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A learned database system. In *9th Biennial Conference on Innovative Data Systems Research (CIDR'19)*.

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

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

[52] Arezki Laga, Jalil Boukhobza, M. Koskas, and Frank Singhoff. 2016. Lynx: A learning Linux prefetching mechanism for SSD performance model. In *5th Non-Volatile Memory Systems and Applications Symposium (NVMSA'16)*. 1–6.

[53] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2017. Deep convolutional neural network inference with floating-point weights and fixed-point activations. (2017). arXiv:1703.03073. arxiv.org.

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

[55] Daixuan Li and Jian Huang. 2021. A learning-based approach towards automated tuning of SSD configurations. arXiv:2110.08685. arxiv.org.

[56] Z. Li, A. Mukker, and E. Zadok. 2014. 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)*.

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

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

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

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

[61] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256 KB memory. arXiv:2206.15472. arxiv.org.

[62] Linux. 2021. Linux Kernel Module Signing Facility. (Jan. 2021). https://www.kernel.org/doc/html/v4.19/admin-guide/module-signing.html?highlight=signing.

[63] LTTng. 2019. LTTng: An Open Source Tracing framework for Linux. (April 2019). https://lttng.org.

[64] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based memory allocation for C++ server workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 541–556.

[65] Paul Manning. 2009. Best Practices for running VMware vSphere on Network Attached Storage. (2009). https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-nfs-bestpractices-white-paper-en.pdf.

[66] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML'10)*. 807–814.

[67] Anusha Nalajala, T. Ragunathan, Sri Harsha Tavidisetty Rajendra, Nagamlla Venkata Sai Nikhith, and Rathnamma Gopisetty. 2019. 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'19)*. IEEE, 1–7.

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

[69] Oracle Corporation. 2020. MySQL. (May 2020). http://www.mysql.com.

[70] 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. 2019. 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'19)*. 8024–8035.

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

[72] Karl Pearson. 1901. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, 11 (1901), 559–572.

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

[74] Gabriëlle Ras, Marcel van Gerven, and Pim Haselager. 2018. Explanation methods in deep learning: Users, values, concerns and challenges. In *Explainable and Interpretable Models in Computer Vision and Machine Learning*. Springer, 19–36.

[75] Natarajan Ravichandran and Jehan-François Pâris. 2005. *Making Early Predictions of File Accesses*. Ph.D. Dissertation. University of Houston.

[76] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The Annals of Mathematical Statistics* (1951), 400–407.

[77] Chris Ruemmler and John Wilkes. 1994. An introduction to disk drive modeling. *Computer* 27, 3 (1994), 17–28.

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

[79] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. 2017. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 561–574.

[80] Wojciech Samek, Grégoire Montavon, Sebastian Lapuschkin, Christopher J. Anders, and Klaus-Robert Müller. 2021. Toward interpretable machine learning: Transparent deep neural networks and beyond. *ArXiv* abs/2003.07631 (2021). arxiv.org.

[81] Jiri Schindler, Sandip Shete, and Keith A. Smith. 2011. Improving throughput for small disk requests with proximal {I/O}. In *9th USENIX Conference on File and Storage Technologies (FAST'11)*.

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

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

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

[85] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. 2020. Running neural networks on the NIC. arXiv:2009.02353. arxiv.org.

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

[87] SOD 2020. SOD—An Embedded, Modern Computer Vision and Machine Learning Library. (Sept. 2020). https://sod.pixlab.io/.

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

[89] Kalyanasundaram Somasundaram. 2020. The Impact of Slow NFS on Data Systems. (June 2020). https://engineering.linkedin.com/blog/2020/the-impact-of-slow-nfs-on-data-systems.

[90] Pradeep Subedi, Philip Davis, Shaohua Duan, Scott Klasky, Hemanth Kolla, and Manish Parashar. 2018. 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*. IEEE, 920–930.

[91] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. 2011. Benchmarking file system benchmarking: It *IS* rocket science. In *Proceedings of HotOS XIII: The 13th USENIX Workshop on Hot Topics in Operating Systems*.

[92] TensorFlow Lite 2020. TensorFlow Lite. (Jan. 2020). https://www.tensorflow.org/lite.

[93] Nancy Tran and Daniel A. Reed. 2004. Automatic ARIMA time series modeling for adaptive I/O prefetching. *IEEE Transactions on Parallel and Distributed Systems* 15, 4 (2004), 362–377.

[94] Transaction Processing Performance Council. 1999. TPC Benchmark H (Decision Support). (1999). www.tpc.org/tpch.

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

[96] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9, 86 (2008), 2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html.

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

[98] Gary A. S. Whittle, J.-F. Pâris, Ahmed Amer, Darrell D. E. Long, and Randal Burns. 2003. Using multiple predictors to improve the accuracy of file access predictions. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'03)*. IEEE, 230–240.

[99] Wikipedia. 2022. Recurrent neural network. https://en.wikipedia.org/wiki/Recurrent_neural_network.

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

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

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

[103] Xiaofei Xu, Zhigang Cai, Jianwei Liao, and Yutaka Ishiakwa. 2020. Frequent access pattern-based prefetching inside of solid-state drives. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE'20)*. IEEE, 720–725.

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

[105] Chuan-Kai Yang, Tulika Mitra, and Tzi-cker Chiueh. 2002. A decoupled architecture for application-specific file prefetching. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, Chris G. Demetriou (Ed.). USENIX, 157–170.

[106] Shengan Zheng, Hong Mei, Linpeng Huang, Yanyan Shen, and Yanmin Zhu. 2017. Adaptive prefetching for accelerating read and write in NVM-based file systems. In *2017 IEEE International Conference on Computer Design (ICCD'17)*. IEEE, 49–56.

Received 1 June 2022; revised 1 June 2022; accepted 13 September 2022