KML: Using Machine Learning to Improve Storage Systems

Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, Andrew Burford,
Michael McNeill, Michael Arkhangelskiy, and Erez Zadok
Stony Brook University

Technical Report FSL-21-02

Abstract

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—essentially 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 overall 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 paper, we describe our proposed ML architecture, called KML. We developed a prototype KML architecture and applied it to two problems: optimal readahead and NFS read-size values. Our experiments show that KML consumes little OS resources, adds negligible latency, and yet can learn patterns that can improve I/O throughput by as much as 2.3× or 15× for the two use cases respectively—even for complex, never-before-seen, concurrently running mixed workloads on different storage devices.

1 Introduction

Computer hardware, software, storage, and workloads have been changing for decades—an accelerating trend in recent years. Storage performance heavily depends on workloads and the precise system configuration [7, 68]. Storage systems and OSs include many parameters that can affect overall performance [6, 8]. Yet, users often do not have the time or expertise to tune these parameters. Worse, the storage and OS communities are very conservative: they resist making significant changes to algorithms and software to prevent instability or data loss. Thus, many techniques currently used were historically developed based on 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. Yet correctly setting them is important and difficult when workloads change: too little readahead wastes potential throughput and too much readahead pollutes caches—both hurting performance. 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 complex and changing workloads. In this paper, we experimented with a variety of modern workloads and many different values of readahead: we found that no single readahead value is optimal for all workloads. Another example of tunable parameters in the network settings is the default read-size (`rsize`) parameter in NFS: if it is set too small or too 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 [18, 44], for database query optimization [43], improved caching [73], cache eviction policies [78], I/O scheduling [34], and more.

In this paper we describe our ML approach to improve storage performance by dynamically adapting to changing workloads. We designed and developed a versatile, low-overhead, light-weight system called *KML*, for conducting ML training and prediction in 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 in the user level, then upload the same model to run identically in the kernel.

In this paper 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 seconds under changing and even mixed workloads. This paper makes several 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 many additional and future ML techniques;
4. We apply KML to two important optimization problems (readahead and `rsize` values); and
5. We evaluate our solutions using multiple, complex, and even mixed workloads, as well as two different storage devices. We demonstrate throughput improvements as high as 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-seen-before workloads or devices. And finally, we show that KML has negligible overheads or memory footprint, making it suitable for embedding into storage systems.

The rest of this paper is organized as follows. Section 2 provides brief ML background. Section 3 describes KML's main design. Section 4 describes our two use cases (readahead and NFS `rsize`). Detailed evaluation of KML and two use cases are in Section 5. We survey related work in Section 6 and conclude in Section 7.

2 Background and Motivation

We now explain how ML can help storage systems and address potential concerns regarding integrating ML into storage systems and OSs.

Our vision. KML is designed to replace OS-level storage system heuristics and system parameter tuning. To this end, a KML application first observes the target OS component by collecting data from probes that were placed in the kernel. The data collected from these probes is used to train an ML model using the functionality provided by KML. The KML application then makes predictions using this trained model and tunes the system. We implemented our use cases with an *observe-and-tune* paradigm to reduce overhead introduced by the ML models. Therefore, we do not impose extra overheads on storage components that require low latency and predictable performance. The two KML use-cases that we detail in this paper are ML models developed (1) to tune readahead sizes on a per-disk and per-file basis and (2) to tune NFS `rsize` value. We chose these two examples because: (i) their storage components can significantly benefit from fine-tuned parameters, (ii) they require adaptation to a variety of different workloads, which is crucial in providing optimal performance, and (iii) it proves that adding asynchronous ML computation (see Section 5.5) will *not* negatively impact critical I/O paths.

Unexpected behavior & explainability of ML. Concerns regarding the potentially unexpected behavior and explainability of ML applications are justified when it comes to tuning OS parameters, where 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 may hamper the performance of the system where it is deployed. This type of issue is difficult to troubleshoot due to the long-standing explainability problems that plague ML models. KML currently supports two ML models: neural networks and decision trees. Predictions made by decision trees are more easily explainable because they can be represented and visualized as successive IF-THEN statements, bisecting the range of the features considered. Meanwhile, deep neural networks are considerably more challenging to explain and verify. However, recent research is dedicated to improving these aspects of ML [38, 67]. While we plan to address possible unstable behavior using feedback-based control algorithms in future work, we currently focus on demonstrating that ML can tune storage system parameters *better* than existing heuristics. In Section 5.3, we compare the performance and accuracy of the neural network vs. decision tree models that we built.

3 KML's Architecture

We begin by discussing high-level KML design choices and why we chose to implement a new machine learning framework instead of porting an existing one.

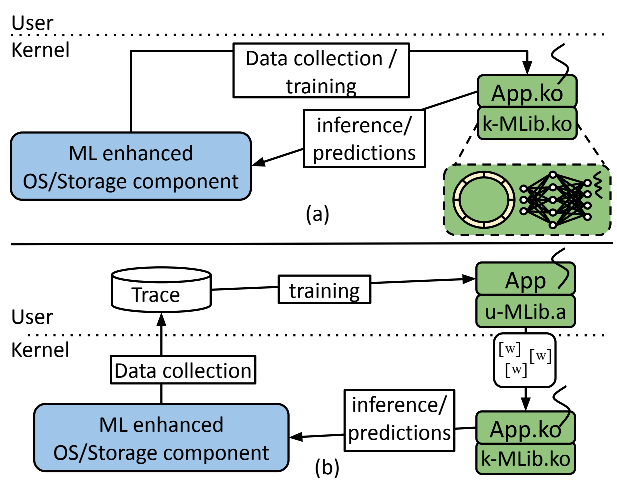

Figure 1: Two different operational modes that we built to achieve a high efficiency ML framework for tuning OS-level storage systems: (a) kernel space training and inference and (b) offline user space training and kernel space inference.

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. KML has a highly modular design: the core ML code base is shared by both user and kernel space. However, operation mode (a) is designed for performance and accuracy whereas operation mode (b) is built to support ease of ML development. We recommend using mode (a) when the problem at hand requires processing data with a high collection rate. Mode (a) also allows training and running inference in kernel space without any prior data collection and processing. Mode (a) also enables more advanced ML techniques (*e.g.*, Reinforcement Learning).

Conversely, mode (b) helps OS developers iterate over their ML model design in user space with different features, architectures, and hyper-parameters until a stable and accurate model is reached. Experimenting with and debugging a model is much easier in user space. Once a model is built in user space, the same model can be loaded into the kernel without modification.

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

3.1 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 compiled from the same KML source code. We developed a wrapper layer for the KML development API; thus, KML's core code base is uniform across both user and kernel APIs. Having this abstraction speeds up development, eases debugging, and facilitates extensibility (see Section 3.3). While these abstractions make it easy to build ML solutions for storage systems, we recognize that developing KML-based solutions requires a good understanding of OS and storage system internals.

Low overhead. Among the most critical challenges in making ML approaches practical for storage systems is reducing computational and memory overheads. ML solutions have 3 phases that consume much memory/CPU resources: (i) inference, (ii) training and (iii) data processing & normalization. We detail our design choices to reduce these overheads in Section 3.4.

Safe programming model. Even a single new line of code integrated into an OS kernel can introduce safety and security concerns. For that reason, we explain how KML avoids instability or security breaches in Section 3.5.

3.2 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, the library is easily debuggable in user space by ensuring identical behavior of all ML functionality also in kernel space.

Mathematical and matrix operations. Most machine learning algorithms heavily rely 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.

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 [57], or sigmoid are essential components of many neural networks regardless of the end goal; loss functions are also fairly standard across many applications. In addition, both layers and loss functions implement two main functionalities, one during the inference (forward) phase and another during the back-propagation phase. Our initial implementation is focused on these common components and modes of operation, implementing the forward and back-propagation functionality for the layers and loss functions included in KML.

Implementing new layers and loss functions in KML requires implementing the aforementioned functionality for the layer/loss function to be added, as well as a function for initializing the data structures related to the layer/loss (*e.g.*, a weight matrix in the fully-connected layer).

Inference and training. When stacked together, the elements of a conventional neural network may be represented as a DAG. From this perspective, 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 [65]—where the chain rule allows for efficient computation of the gradients. KML includes Stochastic Gradient Descent (SGD) which uses the gradients computed using back-propagation to optimize the neural network weights.

3.3 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)).

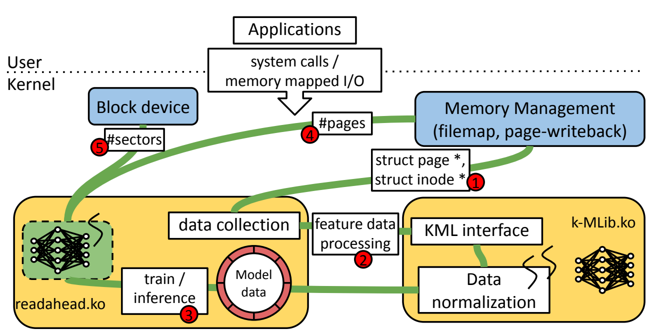

Figure 2: KML kernel space training and inferencing system architecture.

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 readahead problem specific. The green line represents execution and data flow. Numbered boxes refer to transitions happening between the components.

As we mentioned in Section 2, we designed our use cases based on the *observe-and-tune* principle. In its first stage, the readahead module needs to observe and collect data. Since our target component is the memory management 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 configured for training, it trains on the normalized data, and the execution flow ends here for this processing cycle. However, if the readahead module is configured for inference, it feeds the normalized data to the readahead neural network model and tunes the target components based on the prediction made by the model.

The way the KML application optimizes target components is highly dependent on the problem and its solution. In this use case, the readahead module updates readahead sizes using block layer `ioctl` calls (Figure 2(5)). In addition, we developed another approach for tuning readahead which changes readahead sizes on a per-file basis (Figure 2(4)). When the readahead module is running in inference mode, execution flow forms a *closed-circuit*. After the readahead module changes readahead sizes, memory management state changes. When the memory management state changes, the input for the readahead neural network model changes and then predictions change. If users want to train and inference in the kernel, they have to control the switching between training and inference carefully. We have implemented the readahead use case using both in-kernel training and offline training in user space. In this section, we focus on how execution and data flow happens in KML's in-kernel training and inference mode. We provide features and the ML model details for the readahead use case in Section 4.1.

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 OS's and storage systems with high accuracy and sampling rates is challenging [2]. Nevertheless, research shows that tracing tools like LTTng [54] can bring overhead down to as little as 5%. Additionally, traces may still be inaccurate due to data loss. LTTng collects trace data in user/kernel circular buffers. This means that under heavy sampling loads, trace events could be overwritten and lost before they are processed by user space threads. While this problem may still occur in our modes of operation, operating in kernel space gives us more control over thread scheduling to reduce loss of sampled events. Considering the high data sampling rate needed in our use cases, placing data processing and normalization in user space results in more data loss than in the kernel. Still, we believe a user-kernel co-operated design may be beneficial in some cases. That is why we are exploring implementing a user/kernel co-operated design for use cases that do not require a high data sampling rate. Thanks to KML's modular architecture, supporting a user/kernel co-operated design can be achieved without fundamental design changes.

In the long term, we believe that reinforcement learning can be used to achieve truly *self-adaptive and learned* storage systems [40]. Hence, kernel training is critical to reaching this goal. One of KML's current limitations in the inability to adapt to file-access patterns not present in the training data set; reinforcement learning would make this possible.

Training in user space. Building ML solutions is an iterative development life cycle. Hence, to find essential features and build accurate neural network models, we need to run multiple data analyses and train/test our ML model with different architectures and hyper-parameters. Recognizing the advantages of kernel training, building this iterative development life cycle in the kernel is challenging. To facilitate faster model development and debugging, KML offers offline user space training and kernel inference operation mode (see Figure 1(b)). Since KML's user and kernel space libraries are accessed by the same APIs and compiled from the same uniform code base, porting trained ML models to kernel space for inference is effortless.

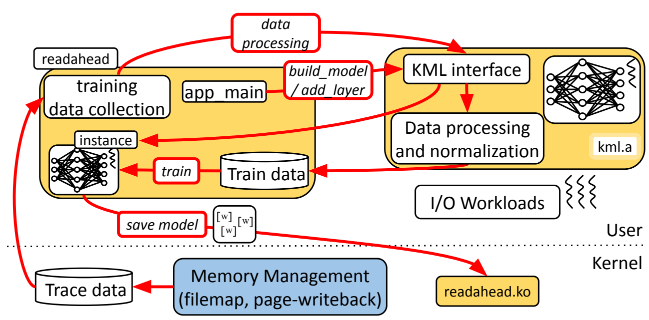

Figure 3: KML user space training and kernel space inferencing system architecture.

Figure 3 shows how the readahead model works with this mode of operation. Components highlighted in yellow represent KML-specific implementations. The red arrows denote the offline data collection and training paths.

First, we built our training data set by tracing the memory management subsystem. We used Re-Animator [2] to collect data from the target storage component. Next, we moved on to the feature-extraction phase. This is where user-space training was useful, because we could run various analyses, test different features, and implement numerous data normalization techniques without re-running experiments. After we finalized the feature selection, we trained and tested the readahead ML model in user space. The training phase often requires iterating over different

ML architectures and testing with multiple hyper-parameters. When the readahead ML model is ready for real-time testing, the only step the user needs to take to deploy the trained model is to save the readahead ML model to a KML-specific deployment file and load it into the readahead kernel module. KML APIs provide all the functionality necessary for building, training, and deploying ML models in-kernel (see examples in Table 1). The rest of the in-kernel execution is identical to how KML runs when it is configured for inference in kernel training/inference mode (Figure 2).

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

Table 1: KML API examples

We implemented a wrapper layer to abstract external functionality away from KML. This way KML can be compiled and used in both user and kernel space with the same API. KML's development API falls into five categories: (i) memory management, (ii) threading, (iii) logging, (iv) atomic operations, and (v) file operations. KML's development API provides a total of 30 functions to support KML's functionality.

3.4 Computational & Memory Overheads

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

Reducing computational overheads. Matrix manipulation is a computationally intensive ML building block that relies on floating-point (FP) operations. However, OSs often disable the floating-point unit (FPU) in the kernel because of the overhead incurred by context-switching. To address this, we considered three approaches: (1) quantization, (2) fixed-point representations, and (3) enabling the floating-point unit in kernel space. Quantization provides compact representation, allows developers to compute matrix manipulation operations, and does not require an FPU [15, 19, 31, 34, 66]. Quantization can help reduce computational and memory overheads, but it reduces accuracy [37]. 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 [13, 52]. However, fixed-point representation works within fixed ranges which can result in numerical instability [47]. For these reasons, KML enables the FP register usage in the Linux kernel using the `kernel_fpu_begin` and `kernel_fpu_end` functions. To avoid context-switch overheads, we minimize the number of code blocks that use FPs.

Reducing memory overheads. There are three factors affecting KML's memory consumption: (1) ML model-specific data (*e.g.*, metadata for model structure, weights and biases), (2) KML's internal memory allocations during training and inference, and (3) data collection for both training and inference. ML model-specific data and KML's internal memory usage depend highly on the number of layers, layer sizes, and layer types in the ML model. 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 plays a significant role in the ML model's performance and accuracy. Users need to configure the size of the circular buffer in consideration of 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 extract predictable performance and accuracy, even under memory pressure. This allows KML to operate without fear of memory allocation lagging or failing, which would otherwise hurt performance and accuracy.

Data processing & asynchronous training. To make ML solutions generalizable, Data normalization is often used as a powerful tool. To this end, KML supports data normalization functionalities such as moving average, standard deviation, and Z-score calculation. However, data normalization often requires heavy FP computation. One of our main principles while building ML solutions for storage systems is avoiding inline computation on I/O and data paths. Thus, KML supports offloading training, inference, and data normalization to a separate *asynchronous thread*. This thread communicates with other KML components (*e.g.*, data collection) over a lock-free circular buffer. KML also provides development APIs to make this communication seamless.

We recommend allocating one CPU core to the training thread. This ensures it gets scheduled frequently enough to keep up with processing and training—otherwise, crucial training data may be lost. Currently, KML supports only one asynchronous training thread. This is because our prototype supports only chain computation graphs that must be serially processed (*i.e.*, the previous layer's output is processed and handed to the next layer as input).

3.5 Safety & Security

Both the training and inference phases for machine learning solutions are computationally intensive. Except for model initialization and ML model-saving operations, there is no I/O involved in KML APIs. KML's impact on the safety and stability of storage components is limited to memory allocation and concurrency. Memory allocations in both user and kernel space can use locking mechanisms. Therefore, there might be unexpected latencies or deadlocks. To avoid these problems, KML allocates memory only in the asynchronous training thread. KML also uses a lock-free circular buffer for data communication and reserves additional memory for extra safety.

Obtaining predictable accuracy and performance from ML solutions depends on how the ML model is architected. We applied standard (*e.g.*, N-fold) validation techniques to ensure the stability of our ML solutions (see Section 4). From a security perspective, we recognize that loading an unverified ML model into a running kernel opens up new attack surfaces. We are exploring solutions to digitally sign and certify loadable models [42, 53].

3.6 Implementation

KML contains 10,703 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 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.

4 Use Cases

We now detail our two use cases: (1) readahead neural network and decision tree model 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.

4.1 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. See Figure 3. After implementing KML's base ML functionality, we focused on the readahead problem: predicting optimal readahead values in the face of dynamic workloads.

Studying the problem. We experimented with running 4 different RocksDB [28] benchmarks, each with 20 different readahead sizes (8–1024), and attempted to determine the readahead sizes that yield the best performance (in ops/sec) for each workload. This became our training data, which can help predict readahead values for *other* workloads and environments. This investigation revealed that each workload has a unique behavior and requires a different readahead size to reach optimal performance. We further investigated the correlations between file access patterns and performance. This allowed us to zero in on the exact nature of the information that was needed to model the readahead problem.

Data collection. We used LTTng [54] to collect trace data, which we then used for finding useful features for the readahead problem. We captured most page cache tracepoints [22] (*e.g.*, `add_to_page_cache`, `writeback_dirty_page`). We collected and processed over 20GB of traces by running multiple 10-minute RocksDB [28] 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 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 at the places where the hook functions were called.

Data preprocessing & normalization. Data preprocessing and normalization are essential to ensure that our readahead neural network is generalizable. We summarize the input data at one-second intervals. The first input feature we selected for our model was the number of transactions taking place each second. The next two features were the calculated cumulative moving mean and the cumulative moving standard deviation of page offsets. Another important feature was the mean absolute page offset differences for successive transactions. We used the `inode` number to filter only RocksDB file accesses. Before we fed these features to our readahead neural network, we applied Z-score normalization to each feature.

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 [41, 64], and set a learning rate of 0.01 and a momentum of 0.99 after trying different values; these values are common in the literature [4]. Our readahead neural network trains on the aforementioned input data and predicts the workload type. We trained on the following four types of RocksDB workloads on NVMe-SSD because they provide a diverse combination of random and sequential operations: (i) readrandom, (ii) readseq, (iii) readrandomwriterandom, and (iv) readreverse. Class frequencies were close, suggesting that classification accuracy is a good metric to evaluate the performance, with the least frequent class being 21.4% and most frequent class being 28.8%.

We tested the neural network's performance with the data described above via k-fold cross validation with $k = 10$, and found out that it achieved an average accuracy of 95.5%. We also performed an analysis to understand the contribution of each feature to the classification performance. One technique is to randomize the order of the feature of interest across samples in the validation dataset, and then calculate the 10-fold validation performance [5]. Using Pearson correlation analysis [62], 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 hampered 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. Next, in Section 5.3, 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). We also experimented with the readahead neural network using TPC-H [76] queries running on MySQL [60] to show how our readahead neural network behaves on completely different types of workloads and how generalizable the models are.

Decision-tree models. We also built a decision-tree model for workload type classification based on the same features and training data. The readahead decision tree contains 59 nodes with a depth of 9. We tested the prediction accuracy of the decision tree using the same procedure with the readahead neural network (10-fold cross-validation), and observed that it results in an average prediction accuracy of 75.4%. As mentioned in Section 2, we included decision trees in KML because decision trees are relatively more explainable than neural networks and run considerably faster. We evaluated the readahead decision tree using the same procedure as the neural networks (Section 5.3).

Readahead in per-file basis. So far, we have shown how we approach the readahead problem when a single workload is running on a device. Storage system developers recognize the challenge of handling mixed storage workloads running on the same system—a common occurrence. In that case, the readahead problem needs to be addressed at a finer granularity than setting one readahead value for the entire block device. Instead, readahead values have to be set at a higher logical layer—on a per-file basis. To show the usefulness of our neural network model, we use the same neural network model to tune readahead size 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 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 (Section 5.3).

4.2 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 [56], hosting NoSQL databases [72], 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-block size. Hence, we focus on the NFS `rsize` problem: predicting an optimal `rsize` value based on workload characteristics.

Studying the problem. We tested NFS using the same methodology we used 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. After studying these traces, we chose our feature set and placed our hook functions. Our feature set includes eight features (described below) which are calculated using the following five data points: (i) time difference from the beginning of execution for each tracepoint transaction, (ii) NFS file handles, (iii) file offsets in NFS requests, (iv) page offsets of the files that were accessed, and (v) number of reclaimed pages during LRU scans.

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

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

5 Evaluation

Our evaluation proceeds as follows: First, we explain our evaluation goals in Section 5.1. We then describe the testbed design and benchmarks that we used to evaluate the readahead neural network in Section 5.2. Section 5.3 shows how the readahead ML models improve performance. In Section 5.4, we present our evaluation of the `rsize` neural network model for NFS. Finally, in Section 5.5 we provide performance details regarding KML's training and inferencing.

5.1 Evaluation Goals

Our primary evaluation goal is to show that using ML techniques inside the OS can unlock the hidden potential of tuning parameters such as readahead for files and block devices, and `rsize` for NFS.

To this end, we evaluate both readahead ML models and the NFS neural network to show how these models improve the I/O performance and quickly adapt the system in the presence of changing workloads. To show that our neural network models are capable of learning 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 nor training. Moreover, 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 workloads when they are mixed together.

One of the main concerns with ML approaches is unpredictable behavior due to mispredictions. We evaluate what happens when the readahead ML models mispredict and how quickly they recover. Finally, we evaluate KML's own system overheads, in terms of (i) training cost, (ii) inference cost, and (iii) memory usage.

5.2 Testbed

We ran the benchmarks on two identical Dell R-710 servers, each with two Intel Xeon quad-core CPUs (2.4GHz, 8 hyper-threads), 24GB of RAM and an Intel 10GbE NIC. In some experiments, we intentionally configured the system with only 1GB of memory to force more memory pressure on the I/O system; but we also show experiments with the full 24GB of system RAM. We used the CentOS 7.6 Linux distribution. We developed KML for Linux kernel version 4.19.51, the long-term support stable kernel released on June 15, 2019; we added our readahead ML models to this kernel and used it in all experiments. Because HDDs are becoming less popular in servers, especially when I/O performance is a concern, we focused all of our experiments on SATA and NVMe SSDs. We used Intel SSDSC2BA200G3 200GB as our SATA-SSD disk and a Samsung MZ1LV960HCJH-000MU 960GB as our NVMe-SSD disk, both formatted with Ext4. These two devices were used exclusively for RocksDB databases. To avoid interference with the installed CentOS, the two servers had a dedicated Seagate ST9146852SS 148GB SAS boot drive for CentOS, utilities, and RocksDB benchmark software. We used 10GbE switches to connect our machines. 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 [28] 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 size can be easily configured (*e.g.*, increased to generate more I/O pressure on a system).

To demonstrate that our ML models can learn from and optimize for different types of workloads, we chose the following six popular yet different `db_bench` workloads to run: (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 [10]). This diverse set of workloads tests reads and writes, sequential and random patterns, and even complex mixed patterns; this provides a good coverage of many possible real-world workloads users could face.

We trained our readahead neural network on traces that contain only four of these workloads: readrandom, readseq, readreverse, readrandomwriterandom—all running 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 was done, 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 three new and complex workloads (updaterandom and two variants of `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,

Figure 4: Running four back-to-back RocksDB workloads in order from left to right: readsequential, readrandom, readreverse, then mixgraph. Here, we started with the default readahead value; thereafter, the last value set in one workload was the one used in the next run. For each of the four graphs, we show their Y axes (throughput, different scales). The readahead value is shown as the Y2 axis for the rightmost graph (d) and is common for all four. Each workload ran 15–50 times in a row, to ensure we ran it long enough to observe patterns of mis/prediction and reach steady-state. Again, we see KML adapting, picking optimal readahead values, and occasionally mis-predicting, yet overall throughput was better.

each on a separate RocksDB database, both stored on the same device. We kept the hardware configuration the same as before to increase system and page-cache pressure.

We also experimented with our readahead network model using TPC-H [76] queries running on MySQL [60] to evaluate how generalizable the readahead neural network is. TPC-H is a well-known database benchmarking framework. We tested our readahead neural network on this setup to show the limits of our model. Because `db_bench` benchmarks and TPC-H queries generate different I/O patterns, we anticipate that models built with the former may mispredict for the latter. In this paper 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. Developing universal solutions for these problems requires more comprehensive datasets and potentially more powerful or complex models, which is beyond the scope of this paper. Figure 5 demonstrates the performance improvements as a result of predictions made by the readahead neural network—detailed further below.

5.3 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 suboptimal value, short-term performance degradation can happen, which

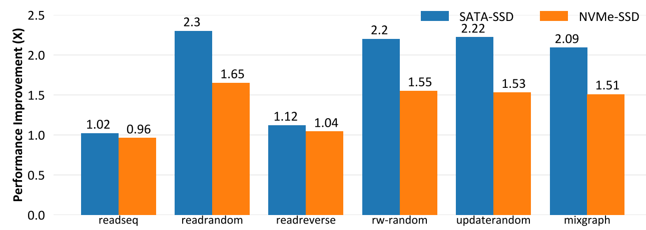

Figure 5: Readahead neural network performance improvements (×) for RocksDB benchmarks on SATA-SSD and NVMe-SSD across all six workloads, normalized to vanilla (1.0×).

might hurt overall performance.

Back-to-back workloads on NVMe. Figure 4 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 set to 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 on complex systems where the I/O workload may change every few minutes. The X axes indicate the run time in minutes. The Y axes indicate throughput in thousands of ops/sec (higher is better), and have different scales for each experiment. The Y2 axes show the readahead values used or predicted by KML over time in terms of number of sectors (denoted with a green line and using the same scale). Each workload ran 15–50 times in a row, so it ran long enough to observe mis/predictions patterns. As seen in Figure 4, KML adapts quickly to changing workloads by tuning the readahead value in about one second.

Although we observe some mis/prediction patterns, overall throughput improved across all four runs, averaging 63.25% improvement: 140% improvement for readrandom, 2% for readsequential, 109% for `mixgraph`, and 12% for readreverse. The potential benefits are clear: even a small percentage improvement in throughput can yield significant cumulative energy and economic cost savings for long-running servers [49].

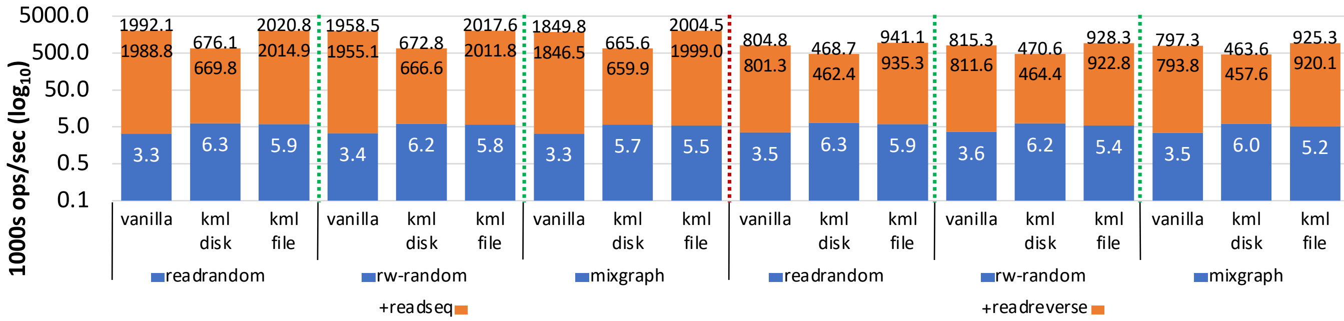

Figure 6: Mixed workloads results. We ran sequential and random workload combinations on the same NVMe-SSD device. Each unique combination is tested with the readahead neural network running in per-disk basis (kml disk) and per-file basis (kml file) and compared against vanilla results. Our model running in per-file basis outperformed both vanilla and per-disk modes.

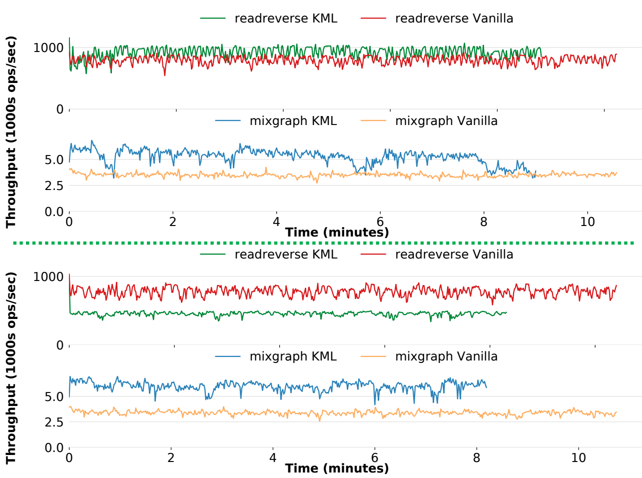

Figure 7: Mixed workloads results on a timeline, comparing readahead neural network model running on per-file basis vs. per-disk basis. Top (above the green dotted line) graph shows per-file basis; bottom half shows per-disk basis.

Read-sequential workloads. Out of the six workloads we ran, Figure 5 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. 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 4) KML mispredicts readreverse as readseq and changes the readahead value to something suboptimal. 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 if KML mispredicts readreverse as readseq or vice versa, it had a small overall impact on performance.

Summary of readahead neural network results. For brevity, we summarize all readahead neural network results in Figure 5. We observe that the average throughput improvement for NVMe-SSD is ranging from 0% to 65%. We saw greater improvements in the SATA-SSD case, ranging from 2% to 130% (2.3×). Lastly, we ran the complex `mixgraph` workload on NVMe-SSD with the system memory set to the maximum (*i.e.*, 24GB) and the database size set to a relatively large 65GB. 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. In Figure 7, we present a timeline performance comparison using the readahead neural network model running on a per-disk vs. per-file basis. 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 to all open files: a readahead value good for one workload is likely to be bad for a second one running concurrently. One reason why the per-disk mode cannot predict workload types correctly is that when different workloads are mixed—even sequential ones or ones with regular patterns—the mix looks more like a purely random workload at the disk level. Figure 6 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 6), 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 6) 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 aforementioned reason (mixing workloads looks more random-like at the disk level); KML predicts these as readrandom or readrandomwriterandom which coincidentally fits this part of the workload, but significantly hurts non-random workloads.

Decision tree evaluation. In addition to the neural network model, we implemented a decision tree model for the readahead problem to compare the two ML approaches on the same problem.

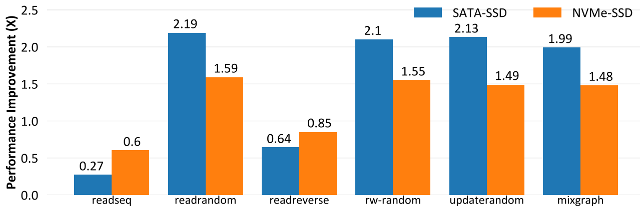

Figure 8: Readahead decision tree performance improvements (×) for RocksDB benchmarks on SATA-SSD and NVMe-SSD devices across all six workloads, normalized to vanilla (1.0×).

Figure 9: Performance timeline graph for tuning with KML decision tree while running readseq workload on NVMe-SSD.

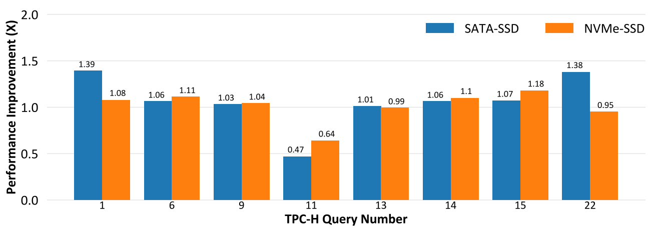

Figure 10: Readahead neural network performance improvements (×) for TPC-H queries on SATA-SSD and NVMe-SSD devices, normalized to vanilla (1.0×).

We tested the readahead decision tree the same way. Figure 8 shows that there is a performance improvement for workloads with a random component. However, performance degrades for sequential workloads. We investigated this performance degradation. Figure 9 shows the readseq workload running on a RocksDB instance stored on an NVMe-SSD. Here, the readahead decision tree predicts the workload correctly in the first three minutes, despite some fluctuations. Afterwards, the decision tree model's predictions fluctuate wildly, and at around minute 10 it consistently makes completely wrong predictions. For the readahead decision tree, we measure average throughput improvement for random workloads on NVMe-SSD as ranging from 48% to 59%; and in the SATA-SSD case, ranging from 99% to 119% (2.19×). However, the readahead decision tree model degrades performance for sequential workloads on NVMe-SSD, ranging from 15% to 40%; and in the SATA-SSD case, ranging from 36% to 73%. Overall, this was somewhat expected: it is known that neural network models, while more complex to build and use, are generally more accurate and adaptable than decision-tree models [32].

TPC-H benchmarks. As we mentioned in Section 5.2, we evaluated our readahead neural network model—trained on `db_bench` RocksDB workloads that run on NVMe-SSD—on TPC-H queries (also NVMe-SSD). This was to show our

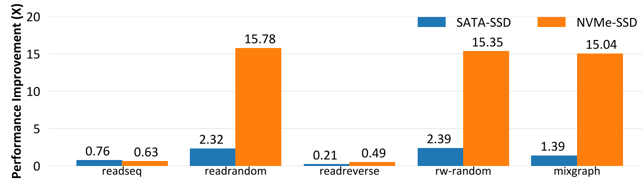

Figure 11: Performance improvements (×) for RocksDB benchmarks on SATA-SSD and NVMe-SSD devices across all six workloads running on NFS, normalized to vanilla (1.0×).

model's accuracy limitations when presented with vastly different workload and application combinations. See Figure 10. We measured performance improvements as much as 39% for most query types. For query 11, however, the readahead neural network failed to characterize the workload correctly and resulted in a 53% performance reduction. Nevertheless, overall TPC-H performance still improved by 6%.

5.4 NFS Evaluation

Figure 11 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), performance improvements for random workloads reached up to 20×. This demonstrates the significant potential of KML.

5.5 KML's Own Overheads

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

CPU overheads. On average, data processing and normalization transactions took 22 nanoseconds for the readahead use-case and 36 nanoseconds for the NFS use-case. This means that data processing, which is the only inline operation in our readahead neural network, adds a negligible latency on the running system compared to typical I/O latencies. Such low overhead is crucial in enabling high sampling rates.

The readahead neural network performs inference (prediction) and changes the block-layer readahead value in $21\mu s$ on average. This is a small number and executes in a separate, asynchronous kernel thread. Hence, it has negligible impact on the overall performance of the OS. When the readahead neural network runs in per-file mode, KML runs inferences an average 135 times (*i.e.*, the number of open files) every second; running inference for all open files consumes 1.7ms on average. We measured that the readahead decision tree inference takes only 8μs. The readahead neural network and decision tree have the same data pre-processing and normalization implementation—the only difference between them is in the inference part.

As discussed in Section 4.1, our readahead neural network prototype offloads training to the user level. We measured the time to perform one training iteration in user level at 51μs on average; this training iteration includes the forward pass, back-propagation, and weight update stages.

Memory overheads. The readahead neural network allocates 3,916 bytes of dynamic memory during the model's initialization phase. While inferencing, KML temporarily allocates 676 bytes before returning the inference results. This overall memory footprint is negligible in today's multi-GB systems. The readahead decision tree occupies only 2,432 bytes of dynamic memory during the initialization. No dynamic memory allocation happens during the inference for the readahead decision tree. Lastly, the kernel module `readhead.ko` has a memory footprint of 432KB and the kernel module `nfs.ko` is 636KB, while KML entire framework (`k-Mlib.ko`) has a memory footprint of 5.5MB.

6 Related Work

Machine learning in systems and storage. In follow-up work to Mittos [33], 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 [34]. There are notable differences between that system and our KML. That system was trained offline using TensorFlow and exclusively trained in user space. Additionally, inferences were tested only on NVMe-SSDs, and the model could not be easily re-trained. Finally, each of the two layers in their neural network were custom built. Conversely, KML provides a more flexible architecture. Training, retraining, normalization, repeated inference—all are possible and accomplished with ease in any combination of online, offline, synchronous, or asynchronous settings. Additionally, KML easily supports an arbitrary number of generalizable neural network layers, and experiments demonstrate more expressive classification abilities on a more diverse set of devices.

Laga *et al.* [46] improved readahead performance in the Linux Kernel with Markov chain models, netting a 50% I/O performance improvement in TPC-H [76] queries on SATA-SSDs. In contrast, our experimentation was conducted on a wider selection of storage media (NVMe-SSD and SATA-SSD) and workloads. In TPC-H, we show improvements up to 39% despite TPC-H being a completely new workload for our readahead model. Moreover, our results illustrate that our readahead model can improve I/O throughput by as much as 2.4×—all while keeping memory consumption under 4KB, in comparison to Laga *et al.*'s much larger 94MB Markov chain model.

Some research has attempted to apply ML techniques to OS task scheduling [13,59], with negligible reported performance improvements (0.1–6%). Nevertheless, it is becoming increasingly popular to apply ML techniques to storage and OS problems including, but not limited to: memory allocation [55], TCP congestion [26], predicting index structures in key-value stores [18,44], offline black-box storage parameter optimization [9], local and distributed caching [73, 78], database query optimization [43], cloud resource management [17, 20, 21], and more.

Machine learning libraries for resource-constraint systems. A myriad of ML libraries exist—some general purpose and others specialized or designed with constraints in mind. Popular general-purpose ML libraries include Tensorflow [1], PyTorch [61], and CNTK [16]. Conversely, libraries like ELL [27], Tensorflow Lite [74], SOD [71], and Dlib [24] specialize in constrained or on-device environments, KML differentiates itself by targeting OS-level applications. 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 [23, 45, 69, 70] and see use in distributed systems [12, 14, 25, 48, 50, 51, 58, 75]. Many have attempted to build statistical models to optimize and tune systems [29, 69, 70]. 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 [3, 23, 36, 45, 77, 79, 82, 84]. Predicting file accesses using hand-crafted algorithms is a reasonable first approach. However, such manual labor simply cannot keep up with the breadth of workload complexity possible on modern systems. Conversely, as long as we have training data, ML models can adapt and optimize much faster. Simulations are also viable solutions for readahead and prefetching problems [11, 30, 63, 83, 86]. 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 [81] or to require application-level changes [85]. In contrast, KML requires no application changes and is capable of intercepting `mmap`-based file accesses.

Finally, while techniques exist to improve NFS performance, to the best of our knowledge, none are performed automatically using ML [39].

7 Conclusion

Operating systems and storage systems have begun 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 may not 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 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×.

Future work. We plan on using KML to tune knobs for other OS subsystems: *e.g.*, packet and I/O schedulers, networking, and the page cache. We are enriching KML's capabilities by supporting other ML techniques such as reinforcement learning [40], which can be a better fit for solving OS problems. To support more advanced ML approaches (*e.g.*, Recurrent Neural Networks (RNNs) [80]) and Long Short-Term Memory (LSTM) [35]), we are extending KML to support arbitrary computation DAGs.

8 Acknowledgments

This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; and NSF awards CCF-1918225, CNS-1900706, CNS-1729939, and CNS-1730726.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[17] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource

central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, Shanghai, China, 2017.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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