Appears in the proceedings of the 13th ACM Workshop on Hot Topics in Storage (HotStorage '21)

A Machine Learning Framework to Improve Storage System Performance

Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok
Stony Brook University

ABSTRACT

Storage systems and their OS components are designed to accommodate a wide variety of applications and dynamic workloads. Storage components inside the OS contain various heuristic algorithms to provide high performance and adaptability for different workloads. These heuristics may be tunable via parameters, and some system calls allow users to optimize their system performance. These parameters are often predetermined based on experiments with limited applications and hardware. Thus, storage systems often run with these predetermined and possibly suboptimal values. Tuning these parameters manually is impractical: one needs an adaptive, intelligent system to handle dynamic and complex workloads. Machine learning (ML) techniques are capable of recognizing patterns, abstracting them, and making predictions on new data. ML can be a key component to optimize and adapt storage systems. In this position paper, we propose KML, an ML framework for storage systems. We implemented a prototype and demonstrated its capabilities on the well-known problem of tuning optimal readahead values. Our results show that KML has a small memory footprint, introduces negligible overhead, and yet enhances throughput by as much as 2.3×.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; **File systems management**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

Operating Systems, Storage Systems, Machine Learning, Storage Performance Optimization

ACM Reference Format:
Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. 2021. A Machine Learning Framework to Improve Storage System Performance . In *13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21), July 27–28, 2021, Virtual, USA.* ACM, New York, NY, USA, 9 pages. *https://doi.org/10.1145/3465332.3470875*

1 INTRODUCTION

Motivation. Computer systems are ever-changing. Storage performance depends heavily on workloads and precise system configuration [6, 56]. They include many parameters that can affect overall storage performance significantly [5, 7]. Yet, users often do not have the time or expertise to tune system parameters to optimize performance. Worse, the systems community is fairly conservative, resisting software changes that may cause instability or data loss. For that reason, many OS heuristics are relatively simple, and were developed based on human intuition after studying several workloads historically; but such heuristics cannot easily adapt to the ever-changing complex workloads. We propose a *versatile*, *low-overhead*, and *light-weight* system called *KML*, for conducting ML training and prediction for storage systems.

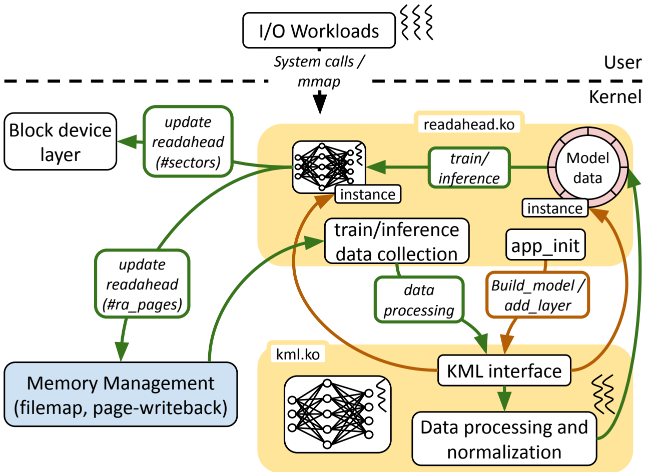

Figure 1: Kernel space training/inference architecture.

KML overview. Figure 1 shows KML's architecture where OS developers perform training and inference in the kernel. To show KML's usefulness, we describe a case study using KML to improve readahead in Section 4. In Figure 1, KML components are shown in yellow. We also show components specific to the readahead case study: the *memory management* subsystem is where we control readahead values using the *block*

HotStorage '21, July 27–28, 2021, Virtual, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8550-3/21/07...$15.00
https://doi.org/10.1145/3465332.3470875

device layer, as well as updating `ra_pages` for *open files*. The target component and how we apply ML to it depends on the problem at hand (see Section 3.2). In Figure 1, the green arrows show the execution flow when KML is enabled and is training and inferencing. Orange arrows denote the initialization flow of the readahead neural network. Users can configure when KML switches between training and inferencing. The green arrows show a closed-circuit flow in the system: when we take action on the system based on ML predictions, it affects the memory management system; thereafter, when the memory management's state changes, it affects future ML predictions.

Library design. We wrote KML's core ML parts from scratch. To make KML run inside the kernel, we implemented the required math and matrix operations, and other ML internals (see Section 2). KML currently supports the most commonly used layers and loss functions, and its versatile architecture is extensible.

OS integration. KML is an ML framework to run inside the OS. Thus, it was designed to conserve resources (see Section 3.1). KML's programming model is flexible which frees users from writing difficult kernel code directly (see Section 3.3).

2 MACHINE LEARNING LIBRARY DESIGN

One goal in designing KML is to provide modularity in building ML solutions. This requires (i) developing the required math functionality, (ii) creating a modular, extensible structure to implement new functionality, (iii) implementing back-propagation algorithms for training neural networks, and (iv) providing a flexible and high-fidelity environment for model development and debugging.

Math and matrix operations. Many standard math operations from `libc` are not available in the kernel. Hence, we implemented commonly used matrix manipulation and linear algebra functions. We also implemented must-have functions such as `logarithm`, `softmax`, and `logistic` from scratch using approximation algorithms.

Layer and loss functions. Neural networks use differentiable components, each performing a certain critical ML operation. For example, a fully connected layer performs matrix multiplication between the given input and the layer parameters (*i.e.*, weights), while a sigmoid layer performs the sigmoid function ($\frac{1}{1+e^{-x}}$) for given matrix (or vector). This component modularity is important in KML. For each layer type and loss function, we implemented a function for forward propagation (*i.e.*, inference), and another for back-propagation.

Inference and training. We perform inference by creating a computation directed acyclic graph (DAG) of the individual layers; we traverse this DAG during inference and propagate the outputs of previous layers to successive ones. We compute gradients using reverse mode automatic differentiation (*e.g.*, back-propagation [54]). We used the back-propagation [54] chain rule to compute the gradients for each parameter efficiently. Once gradients are computed, KML optimizes the neural network's parameters using Stochastic Gradient Descent (SGD) [61].

Extensibility. Other neural network components and loss functions can be easily added to KML. One has to implement 3 main functions, similar to the existing layer and loss functions: (i) building and initializing the layer, (ii) forward propagation for inferencing, and (iii) backward propagation for training.

Interoperability between kernel and user space. It is easier to develop and test software in user space than in the kernel. We enable quick and easy user space development for ML models that can then run seamlessly in the kernel. To ensure interoperability between user and kernel, we use the *exact* same code in both—only wrapping functions with a thin portability layer (see Section 3.3).

3 OPERATING SYSTEM INTEGRATION

In this section, we describe how KML is designed to: (i) reduce ML overheads, (ii) collect data efficiently and train on it with minimal system interference, and (iii) have a versatile architecture.

3.1 Reducing ML Overheads

OSs are sensitive environments where precious CPU and memory resources must be carefully managed. Therefore, any ML implementation inside the OS must be efficient. However, there is a critical trade-off between how much accuracy can be obtained from an ML model vs. how much CPU and memory is used for ML: more memory and CPU usage produces more accurate ML models. KML lets users balance these trade-offs.

Reducing computational overheads. ML computational overheads mainly arise when multiplying floating-point (FP) matrices. Most OSs disable FP operations in kernel space to reduce computational and context-switching overheads. To improve compatibility, KML supports multiple data types for matrix operations. One way to represent matrices compactly is using quantization [14, 18, 29, 31, 55]. Quantization can reduce both computational and memory overheads, but often reduces accuracy [34]. Another way to perform FP operations in a kernel is to use a fixed-point representation. Operations on fixed-point representations can be faster and do not require an FP unit in the running processor [12, 45]. However, fixed-point representations cannot emulate large ranges, which can lead to numerical instability issues [41].

To improve versatility, KML supports *integer*, *floating-point*, and *double* precision matrices. To enable and disable the FP unit (FPU) in the Linux kernel, we used `kernel_fpu_begin` and `kernel_fpu_end`, respectively. We minimize the number of code blocks using FPs, to reduce the number of context-switches that would also have to save and restore floating-point registers.

Reducing memory overheads. KML's memory consumption mainly comes from storing neural network specific data (*e.g.*, weights, biases, hyper-parameters, layer structures) and collecting training data. The size of the neural network specific data depends on the ML model depth and layer sizes. We designed KML to be as efficient as possible: we use a lock-free circular buffer to process and asynchronously train on input data. The circular buffer's size is configurable to cap memory usage. Since losing part of the training data could reduce the model's accuracy, users must carefully configure the circular buffer size based on the sampling rate of data collection. Section 4 evaluates total memory usage in our readahead case study. When the system is running under memory pressure, allocating memory might itself take a long time or even fail, which could hurt KML's performance and accuracy. KML thus supports memory *reservation* to ensure predictable performance and accuracy.

3.2 Data Collection and Async Training

Data normalization and processing is essential in ML. KML offers several data normalization and statistical functions: moving average, standard deviation, and Z-score calculation. The data normalization phase is usually computation-intensive and likely requires floating-point (FP) operations. Hence, we offload data normalization to a separate asynchronous kernel thread, which is also responsible for *training*. This avoids enabling FP operations in other thread contexts where we collect data for training: namely the I/O and data paths, which are highly sensitive to additional latencies.

KML creates a *training thread* during the model initialization stage. KML also abstracts all complicated data communications with the circular buffer. The only information users need to provide in the model-initialization code is a pointer to the model's training function. Using a separate training thread might have performance implications. If the training thread is not scheduled frequently enough, it could hurt KML's performance and accuracy, due to lost data samples. We recommend leaving at least one available CPU core for the training thread. KML currently supports only one asynchronous training thread, since our current prototype supports only chain computation graphs that have to be processed serially.

3.3 Versatile Architecture

KML development API. Kernel programming is challenging. We designed KML's development API to abstract away external functionality (*e.g.*, threading, memory-allocation). KML can be compiled in both user and kernel space with identical behavior. The KML development API has five parts: (i) system memory allocation, (ii) threading, (iii) logging, (iv) atomic operations, and (v) file operations. KML's development API has 27 functions to support KML's needs. For example, we implemented a simple `kml_malloc` wrapper API that calls `malloc` in the user level and `kmalloc` in the kernel.

Training in user space. KML runs identically in both user and kernel space to ease model development. Users can collect data using KML's data processing and normalization components and then train ML models on collected trace data in user space. Other adjustments, such as trying different neural network architectures or hyper-parameters, can also run in user space. Facilitating model development and debugging in user space accelerates both. When the neural network model is ready to be deployed, the user can save the model to a file that has a KML-specific file format. The user can then load the neural network model with a given path to the deployment file in the kernel module. In this paper, the target neural network model we discuss was trained and tested in user space and deployed for inference into kernel space. First, we make *KML API calls* to collect data inside the memory management system (*i.e.*, `filemap.c` and `page-writeback.c`). We then process, normalize, and store the collected data. Data collection is followed by feature selection. After we get the training data as our input, we start building our model to improve readahead performance. When we finalize the model design, we save the model to a file to be deployed later in the kernel. Users need to write or modify their kernel module using KML APIs (see Table 1). KML APIs define the interfaces between KML models and kernel.

```
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.

After we deploy the model, we configure KML for inference. When the system and the neural network model are operating in inference mode, execution proceeds as follows: (1) KML starts collecting data from the memory management component; (2) the collected data is processed and normalized, and the data processing and normalization unit generates suitable features; (3) features are passed to the KML engine for inference; (4) KML's engine inferences and generates predictions; and (5) finally, the KML application takes actions based on the predictions just made—*e.g.*, the KML application changes readahead sizes using block device layer `ioctls` and updates the readahead values in `struct files`.

Although the example demonstrated here focuses on the case where training is performed in the user space and the inference is performed in the kernel space, KML supports other modes of operation. KML can do either *training* or *inference* in user or kernel spaces. Also, one can switch between training and inference modes as needed to adapt automatically to ever-changing conditions in the kernel.

Training in kernel space. There are two critical reasons why we support in-kernel training: performance and accuracy. One of the most crucial parts of the machine learning flow is collecting the training data. Tracing OSs at a high sampling rate is challenging. Tracing tools like LTTng [46] use user-kernel shared memory and gather tracing data in user-space. However, research shows that these tracing tools introduce overhead—from 5% to 2×—and may lose tracing data to cap these tracing overheads [2]. We also plan to build a user-kernel co-operation mode for KML for cases that do not require high sampling rate data collection. In our readahead use case, we collected data and trained in user-space. We then deployed in the kernel. But, we also tried training same neural networks directly in the kernel without having separate data collection. Because our readahead use-case does not require high sampling rate data collection, both the in-kernel trained readahead model and the user-space one performed well.

In-kernel training also allows OS developers to build ML solutions using reinforcement learning [35]. Using reinforcement learning, we can build ML approaches that can adapt themselves based on the feedback from the system. For example, when we apply our readahead neural network on applications that use different file access patterns—and hence not represented in our training dataset—the readahead neural network may not perform as well. In that case, we can build a feedback system in the kernel and transform our readahead neural network model to reinforcement learning model.

Safety in KML's programming model. KML ensures that inline data collection operations of KML do not cause any deadlock. KML uses lock-free data structures to avoid deadlock and to reduce the overhead of data collection operations. KML uses dynamic memory allocation during the inference phase. Since the inference thread runs on a separate thread context and gets the collected data from lock-free data structures, even if it gets blocked during the memory allocation, it does not affect any other threads. Furthermore, KML's training is just computation that involves memory allocation and no I/O, hence cannot block or deadlock.

OS developers can be skeptical about ML solutions' stability. KML itself is just an OS framework for enabling ML in kernel and is not directly responsible for an ML solution's stability and accuracy. ML stability and accuracy, rather, depend on how these ML approaches are trained and their architecture. Nevertheless, we used standard techniques to validate the stability and accuracy of our ML solution, which we discuss in Section 4.

4 USE CASE: IMPROVING READAHEAD

We now explain how we improved readahead using KML. We start by describing the problem, then how we constructed the neural network, and finally illustrate our evaluation results.

Motivation. Readahead is a storage prefetching system. Its main objective is to improve I/O performance by caching "future" storage data based on tunable per-disk and per-file readahead values, as well as hints that users can provide through system calls such as `fadvise` and `madvise`. Users can tune the OS's use of certain files with these system calls (*e.g.*, hinting of expected sequential vs. random access). Our goal was to replace these manual or programmer-driven heuristics with an automated ML technique that quickly adapts to changing workloads to tune readahead values as needed.

Studying the problem. We studied the readahead problem empirically. We tested RocksDB [26] with four different workloads, 20 different readahead sizes (ranging from 8 to 1024), and two different storage media: NVMe SSD ("NVMe") and SATA SSD ("SSD"). We then built a mapping from the workload type to the readahead value that provided the best throughput. The results showed that no single readahead value maximized throughput for all workloads. Moreover, the relationship between readahead values was not linear and at times exhibited long tails. Clearly, traditional or naïve techniques could not optimize readahead values for each disk, file, and changing workload. We designed a readahead neural network model to classify which workload is running on the system; then, based on the neural network's workload type predictions, we configured the readahead size that we obtained through our experiments to optimize throughput.

Data collection. We collected training data from the Linux kernel using LTTng tracepoints [2, 46]. To model readahead behavior accurately, we used built-in kernel tracepoints (*e.g.*, `add_to_page_cache`, `writeback_dirty_page`). These tracepoints track file-backed pages. Next, we investigated the collected traces to find the correlations between file access patterns and workload types. We then chose data points that have high correlations with workload types. At runtime, these data points are collected on-the-fly by data-collection hook functions, which KML users need to implement. Readahead data collection functions record the `inode` number, page offset of the files that are accessed, and time difference from the beginning of the execution of KML kernel module.

Data pre-processing and feature extraction. One of our primary goals for adapting ML approaches into the storage stack is building more generalizable and adaptable solutions. Therefore, data pre-processing and normalization are crucial. In the readahead model, we process the collected data points every second and then extract features at runtime. We tried a total of eight features which we selected based on our domain expertise (storage workloads and readahead). We then experimentally narrowed them down to just five features that had the most predictive accuracy, also confirmed using Pearson correlation analysis [51]: (i) the number of tracepoints that were traced, (ii) the cumulative moving average of page offsets, (iii) the cumulative moving standard deviation of page offsets, (iv) the mean absolute page offset differences for consecutive tracepoints, and (v) the current readahead value. We then calculated the Z-score for each feature to normalize the input data of our readahead neural network model.

Neural network model. We designed our readahead neural network as a multi-class classification model. Our model has three linear layers, and these layers are connected with sigmoid activation functions to model the non-linearity exhibited by the readahead-vs-throughput curves we investigated. We used the cross-entropy loss function and optimized our network using an SGD optimizer [36, 53], configured with a (conventional) learning rate of 0.01 and a momentum of 0.99 [4]. We trained on the data we collected by running only four workloads (`readrandom`, `readseq`, `readreverse`, and `readrandomwriterandom`) on NVMe. We picked these training workloads because they were sufficiently diverse in sequentiality vs. randomness. We measured the performance of our neural network using k-fold cross-validation with $k = 10$, and found that our model reached an average accuracy of 95.5%.

Performance evaluation. Our evaluation goals were to prove that using ML-based solutions inside the OS can offer more generalizable and adaptable solutions than aging heuristics—and hence improve performance with minimal overhead. To prove that our neural network model can learn abstract patterns, we tested our model under the workloads we trained on as well as *never-seen before* workloads that include a complex mixed workload (`mixgraph`) [9]. Moreover, while training workloads ran on an NVMe, our evaluations tried those and new workloads on an SSD. The results show that the readahead neural network can reach up to 2.3× better performance (see summary Table 2). As Table 2 shows, only for `readseq` on NVMe, KML actually lost 4% performance; all other workloads improved. This is because `readseq` maxes out the underlying hardware throughput and there was little opportunity to improve throughput: KML's attempts to learn and optimize readahead only interfered with an already optimal throughput.

KML currently supports neural networks and decision trees. We have also implemented a decision tree for the readahead use-case to show how different ML approaches perform on the same problem. The readahead decision-tree model improved performance for SSD 55% and NVMe 26% on average. For brevity, we provide details only for the (*superior*) neural network results in this paper.

Another crucial part of our evaluation is to show that, with KML, we can build highly efficient and low overhead ML solutions for OS problems. Note that the numbers presented in this paper are specific to the readahead model: KML's neural network models' overhead is correlated with the ML model's complexity. Figure 2 shows how the readahead neural network changes the readahead size (Y2 axis) during RocksDB's `mixgraph` workload on an NVMe device, and how performance (ops/sec, Y1 axis) improves overall. We ran the same benchmark 15 times and averaged the performance improvements. As Figure 2 shows, there are fluctuations on the readahead size tuning. The reason for these fluctuations is that we clear the cache after every run, and that when the benchmark starts, read-access patterns are different than the rest of the execution. Still, while the readahead neural network predicts the readahead size inaccurately for short time periods, overall performance improvement is around 2.09×.

Benchmarks	**NVMe**	**SSD**
`readseq`	0.96×	1.02×
`readrandom`	1.65×	2.30×
`readreverse`	1.04×	1.12×
`readrandomwriterandom`	1.55×	2.20×
`updaterandom`	1.53×	2.22×
`mixgraph`	1.51×	2.09×

Table 2: KML readahead neural network model improved RocksDB I/O performance under six workloads across two device types: average performance gain for SSD was 82.5% and for NVMe was 37.3%.

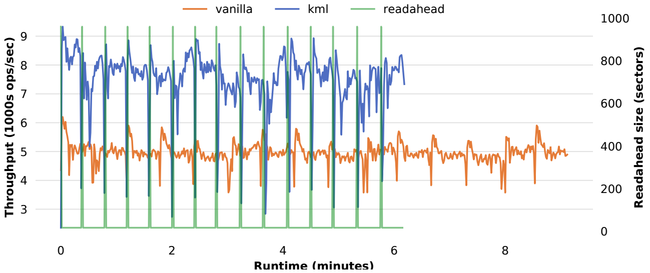

Figure 2: Timeline of performance comparison between running RockDB's `mixgraph` workload on vanilla and with KML optimizations enabled.

Our readahead model's average data collection and normalization overhead for each transaction was a mere 49 nanoseconds. The readahead neural network executes an inference in 21μs and one training iteration in 51μs on average. Since our readahead model's input data is designed to be processed and fed to the readahead neural network for every second, we run inference in a different thread context once a second. Therefore, the overhead of inference for our readahead model is 21μs for every second which is negligible. The readahead model consumed 3,916 bytes of dynamic memory to initialize the model and it temporarily used another 676 bytes of memory while inferencing. These results prove that KML is viable for solving I/O problems that often run in milliseconds.

5 RELATED WORK

Machine learning in systems and storage. In a follow up work to Mittos [30], the authors implemented a custom neural network to perform inference inside the OS's I/O scheduler queue, deciding synchronously whether to submit requests to the device or not (using binary classification) [31]. Their system trained offline using TensorFlow, only in user space, tested for inference only on SSDs, and could not be easily re-trained. Lastly, the two layers in their neural network were custom built. Conversely, KML offers a flexible architecture that can train, normalize, and infer repeatedly, online or offline, synchronously or asynchronously—with equal ease. KML can easily support any number of generalizable neural network layers and other ML models (*e.g.*, decision trees). Our experiments demonstrate more complex classification abilities on a wider range of devices.

Laga *et al.* [40] implemented Markov chain models to improve readahead performance in the Linux kernel. They presented 50% better I/O performance for a database system under TPC-H [64] benchmarks on SSDs. In comparison, we experimented with our readahead model using a broader range of workloads and storage media (NVMe and SSD), and our results show that our readahead model improved I/O throughput by as much as 2.3×. Moreover, our readahead model's kernel memory consumption is less than 4KB, compared to Laga *et al.*'s Markov model which consumed 94MB.

Some researchers have tried to integrate ML techniques into the OS task scheduler [12, 49]. But the performance improvements that they reported were negligible (0.1–6%). Nevertheless, there is a growing trend in using ML techniques to solve storage and OS problems: predicting index structures in key-value stores [17, 38], memory allocation [47], TCP congestion control [24], offline black-box optimization for storage parameters [8], database query optimization [37], local and distributed caching [60, 66] and cloud resource management [16, 19, 20].

Machine learning libraries for resource-constrained systems. A variety of ML libraries are available to help develop ML-enabled applications for various settings. Some of these ML libraries are mainstream libraries without a particular focus area, such as PyTorch [50], Tensorflow [1], and CNTK [15]. Other libraries are targeted at constrained or on-device environments, such as ELL [25], Tensorflow Lite [62], SOD [59], and Dlib [22]. ELL is targeted at deployment of ML models for inference, while the other libraries can also be used for inference. KML differs from these because it targets OS-level applications, where prediction accuracy is important but must be carefully balanced with overheads and resource consumption.

Adaptive readahead & prefetching. Readahead and prefetching techniques are well studied problems [21, 39, 57, 58] and are useful in distributed systems [11, 13, 23, 42–44, 48, 63]. Several works try to build statistical models to tune the systems [27, 57, 58]. The main limitation of statistical models is their inability to adjust to new or changing workloads and devices. Conversely, we have shown that our model can adapt to *never-seen before* workloads and different devices. Predicting individual I/O requests and file accesses based on patterns generated by the workloads is another way to improve readahead [3, 21, 33, 39, 65, 67, 69, 71]. Predicting file accesses using hand-crafted algorithms is a well-known approach. However, it does not scale with the number of workloads that such hand-crafted algorithms need to recognize. ML models, however, are more capable of scaling as long as we have the training data for workloads. Simulated environments also helped researchers to build solutions for readahead and prefetching [10, 28, 52, 70, 73]. However, simulations are limited to datasets that the models are trained and tested with, and they can be computationally intensive. Since the simulated environment models are not designed for resource-constrained environments, it is also difficult to port them to the kernel. Some use a user-space library to intercept file accesses [68] and even require application changes [72]. Conversely, KML does not require application changes and also intercepts `mmap`-based file accesses.

6 CONCLUSION

OSs and storage systems are evolving to support an ever diversifying range of devices and workloads. For best performance, storage systems need to optimize for specific workloads and devices. Traditional heuristics, however, cannot adapt quickly enough. Limitations of the existing solutions led us to develop KML—an ML framework for OSs that adapts quickly to optimize storage performance. Our preliminary results show that, for a readahead case study, we can improve I/O throughput by up to 2.3× with negligible overhead.

Future work. We plan to apply KML to other storage subsystems: *e.g.*, I/O schedulers, storage networking (*e.g.*, packet/flow scheduling), network file systems, and the page cache. We are expanding KML to support other ML approaches (*e.g.*, reinforcement learning [35], which can solve certain problems better than classification approaches). We also plan to support arbitrary computation DAGs (*e.g.*, Recurrent Neural Networks (RNNs) [35]) and Long Short-Term Memory (LSTM) [32]. This would require spawning several parallel training threads.

7 ACKNOWLEDGMENTS

We thank the ACM HotStorage anonymous reviewers and our shepherd Young-ri Choi for their helpful feedback. 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] 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)*, Santa Clara, CA, February 2020. USENIX Association.

[6] 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)*, pages 329–343, Santa Clara, CA, February-March 2017. USENIX Association.

[7] 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*.

[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 *USENIX Annual Technical Conference, (ATC)*, pages 893–907, Boston, MA, July 2018.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. Association for Computing Machinery.

[20] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. Association for Computing Machinery.

[21] 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.

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

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

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

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

[55] 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.

[56] 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)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

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

[58] 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.

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

[60] 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.

[61] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning*, pages 1139–1147, 2013.

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

[63] 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.

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

[65] 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.

[66] 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 *HotStorage '18: Proceedings of the 10th USENIX Workshop on Hot Topics in Storage*, Boston, MA, July 2019. USENIX.

[67] 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.

[68] 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.

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

[70] 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.

[71] 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.

[72] 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.

[73] 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.