Towards Machine Learning-Based I/O Scheduling

Technical Report FSL-25-07

Alexander Joukov
Stony Brook University

Gokul Kumar Manickavasagam
Stony Brook University

Ibrahim "Umit" Akgun
Stony Brook University

Michael Arkhangelskiy
Stony Brook University

Michael McNeill
Stony Brook University

Geoff Kuenning
Harvey Mudd College

Michael Mesnier
Independent

Erez Zadok
Stony Brook University

Abstract

Machine learning has been shown to enhance performance and decision-making within many of the Linux kernel's complex subsystems. This paper proposes that ML can significantly enhance file I/O, particularly in traditional storage devices like HDDs, which remain the dominant storage technology. In this work, we try two scheduling approaches: adaptive scheduler switching based on workload classification, and shortest-job-first I/O scheduling based on predicted latencies. Our experiments found that we could improve throughput by 7.5% and decrease average latency by up to 43.9%. We then discuss the significance of our results, considering both theoretical and experimental bounds.

1 Introduction

File I/O scheduling plays a critical role in overall system performance, particularly for storage-bound applications bottlenecked by disk operations. Efficient scheduling mechanisms are crucial for optimizing resource utilization, reducing latency, and enhancing throughput. Devices like HDDs remain the dominant storage technology [1]. I/O schedulers such as `Deadline` [2] and `CFQ` [3], built for HDDs, have proven effective in various scenarios by employing static heuristics to manage disk access. However, these fixed strategies often struggle to adapt to the diverse and dynamic workloads characteristic of modern computing environments, which can lead to suboptimal performance [4]. Figure 1 shows the service latency distributions of Filebench's `varmail` and `oltp` workloads [5], collected from our experiments using the default workload parameters. These distributions reveal distinctly different characteristics, further indicating that a one-size-fits-all scheduler based solely on static heuristics is unlikely to perform optimally under all conditions.

Recent advances in machine learning have demonstrated potential in optimizing complex systems, offering adaptive, data-driven solutions that can dynamically respond to changing workload patterns. ML techniques have already shown great performance improvements across Linux kernel subsystems [6–13], motivating us to investigate their applicability in enhancing file I/O scheduling.

We introduce two ML-driven approaches to optimize I/O scheduling. First, our adaptive scheduler-switching method leverages workload classification to dynamically choose the scheduler with the highest expected throughput.

Second, we propose a shortest-job-first (SJF) scheduling strategy that uses latency predictions to reorder I/O requests. Here, a neural network estimates the latency of each request, enabling the scheduler to prioritize those expected to complete quickly and thereby reducing the average latency.

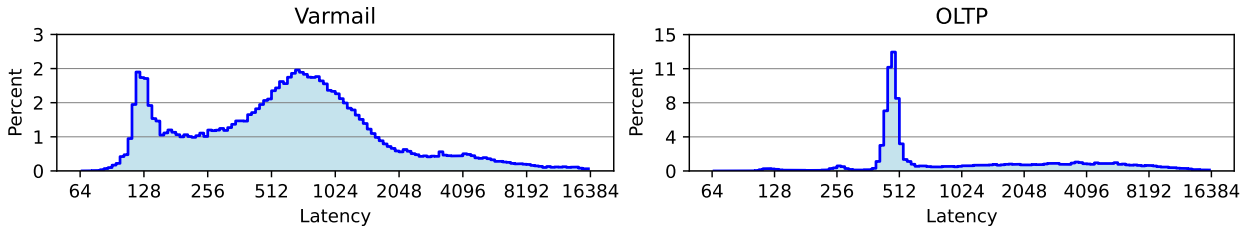

Figure 1: Latency pdfs of Filebench's `varmail` and `oltp` workloads (μs). The x-axes are log-scale and do not begin at 0.

2 Scheduler-Switching Design

Adaptive scheduler switching uses historic workload patterns to predict the ongoing workload, then uses per-workload scheduler history to switch to the scheduler with the highest expected throughput for the predicted workload.

2.1 Classification Model Setup

To predict the ongoing workload, we experimented with two different machine-learning models: decision trees and neural networks. Although decision trees are typically faster, adaptive scheduler switching is done asynchronously, so inference speed is not a significant concern. Thus, we instead opt for a slower but more robust neural-network approach.

For our input features, we concatenated data from N preceding requests, where N is an algorithm parameter. For each request, we recorded the operation (`read` or `write`), the Intel OST tag (described below), the request size in bytes, the latency, and the LBA delta (the difference between the current and previous disk LBAs).

To find the request latencies, we recorded the start times with a patched `Noop` scheduler and then recorded the end times in `blk_account_io_done_work`. A hash table mapped the I/O completions to previous dispatches, using the LBA as the key.

We used a custom Intel-OST 4.13.2 Linux kernel [14, 15] that enhances I/O tracing by tagging each request at the kernel level. It includes a modified Ext4 file system that attaches one of the following tags to each I/O operation. These tags are then passed within the `bio` structures as the requests traverse the block layer:

`META` (metadata)	`BLK_BMP` (block bitmap)
`SUPER` (superblock)	`INO_BMP` (inode bitmap)
`GROUP` (group descriptor)	`INODE` (inode info)
`DIRECT` (pointer)	`INDIRECT` (pointer)
`EXTENT` (extent info)	`XATTR` (extended attributes)
`DATA_DIR` (directory)	`JOURNAL` (journal block)

In addition to these file-system labels, Intel-OST also provides file-size labels that scale in powers of four, ranging from 4KB to 1GB—*e.g.*, `FILE_4KB`, `FILE_16KB`, ..., `FILE_1GB`—and `FILE_BULK` for files larger than 1GB.

We used the KML library [16] to perform efficient ML-based inferences directly in kernel mode. KML's CPU overhead is negligible for small-to-medium–sized models, so we do not focus on computational costs in this paper.

The neural-network architecture consists of an input layer containing all the features described above, M hidden layers, and an output layer with each node corresponding to a workload. The first hidden layer has L nodes. The next $M-1$ hidden layers have exponentially fewer nodes: the ith layer has $L/2^i$ nodes. The network is fully connected and contains leaky ReLU nonlinearities between all layers. A softmax function is applied to the outputs, normalizing the predictions and constraining them to lie between 0 and 1.

We tested a large number of combinations for the values of L, M, and N. We found that $L=128$ nodes in the first hidden layer, $M=4$ hidden layers, and saving $N=16$ previous requests performed sufficiently well. Increasing any of the hyperparameter values further provided minimal improvements (0–1%) in classification accuracy, while increasing CPU and memory usage significantly.

We trained and tested our models on the RocksDB [17] workloads `readrandomwriterandom`, `readseq`, `fillseq`, `mixgraph`, `readreverse`, `readwhilewriting`, `readrandom`, and `fill100k`.

2.2 Classification Results

Figure 2 shows a confusion matrix comparing the true workload and the workload our model predicted. A high value (using darker colors) across the main diagonal indicates successful differentiation. We can see that the model was generally successful, but it had difficulty differentiating between some groups of similar workloads that appeared within the dataset.

The `fillseq` and `fill100k` workloads formed one such group, while the `readwhilewriting`, `readrandom-writerandom`, `readrandom`, and `mixgraph` workloads formed another. This is not particularly surprising since the first group consists of large, sequential data writes while the second has highly randomized and unpredictable requests. We also notice that `readseq` and `readreverse` were easily differentiated due to our LBA Delta input feature.

As we demonstrate in §5, similar workloads tend to have the highest throughput under similar schedulers, meaning confusion among these groups causes little impact on performance. If we merge the `readwhilewriting`, `readrandomwriterandom`, `readrandom`, and `mixgraph` workloads accordingly, our final workload classification accuracy is 86%.

3 SJF Scheduling Design

A perfect shortest-job-first scheduling (SJF) algorithm optimizes for both total and average task execution latency [18]. When applied to HDDs, where I/O latency depends on mechanical movements, an SJF-like approach conceptually aligns with classical disk scheduling algorithms such as shortest-seek-time-first (SSTF) [18], which prioritizes requests that require the least movement from the current disk head position. Our ML-based SJF scheduler, `KML-IOSched`, aims to predict the overall service time, which encompasses seek time, rotational latency, and other system effects, rather than just seek distance. It thus does not have to account for undocumented and unpredictable disk behaviors [19].

While `KML-IOSched`'s goal of prioritizing requests with the shortest predicted latency is reminiscent of SSTF, it should also avoid the traditional drawbacks of seek-optimizing schedulers. SSTF, for instance, can lead to starvation for requests to tracks far from the current head position and may exhibit higher latency variance. We account for this possibility using an anti-starvation mechanism described in §3.4.

Our findings show that accurately predicting I/O latency for HDDs is challenging due to several interacting factors. For example, one might expect request sizes to correlate strongly with latency, but that relationship only achieves a low Pearson Correlation Coefficient of 0.03 (§3.3). This difficulty is not unique to HDDs; for instance, even for flash storage, which lacks mechanical components, predicting I/O latency accurately has been shown to be non-trivial due to internal device operations like garbage collection and wear-leveling [20].

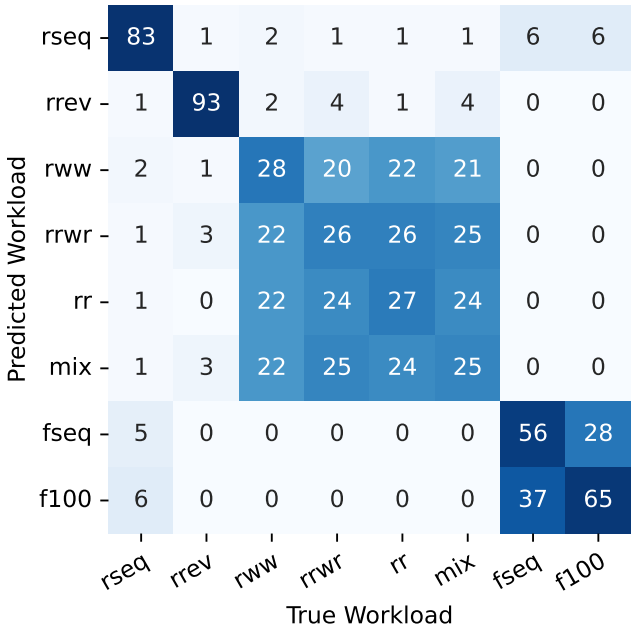

Figure 2: Classification confusion matrix for workloads `readseq` (`rseq`), `readreverse` (`rrev`), `readwhilewriting` (`rww`), `readrandomwriterandom` (`rrwr`), `readrandom` (`rr`), `mixgraph` (`mix`), `fillseq` (`fseq`), and `fill100k` (`f100`).

Our work on HDD latency prediction using ML should be viewed in the context of classical disk modeling. Previous researchers have developed mathematical models of disk behavior, often focusing on seek time and rotational latency [21–25]. These models have been instrumental in understanding HDD behavior. Work has also been done on modeling SSD behavior [26–30], but we do not focus on those in this work.

Our ML-based SJF approach differs from previous HDD models by attempting to learn these latency factors implicitly from I/O request features without requiring explicit parameterization of the drive's geometry, though it aims to achieve a similar goal of optimizing based on predicted service times. The KML library supports dynamic, online retraining [16]. In the future, `KML-IOSched` can integrate with that functionality, permitting us to adapt dynamically to changing conditions such as evolving workload patterns, aging, thermal variations, or the specific nuances of an individual storage device. This contrasts with static analytical models, which would require offline recalibration to account for such dynamic behaviors.

3.1 Initial Ordered Latency Prediction

Our initial latency-prediction models took six features as input: the current queue depth of the storage device, the latency of the previous request, the LBA Delta between the current and previous requests, and the tag, size, and operation of the current request. Testing these models on the RocksDB workloads yielded an average accuracy of 95% during 10-fold cross validation. An ablation study revealed that queue depth was the most significant feature, followed by the request tag. A high queue depth means a request must wait for many others to complete before it is dispatched; on the other hand certain tags (*e.g.*, for metadata) indicate much faster completions.

Although a 95% accuracy shows that accurate latency prediction is possible in a stable-order setting, this approach is inadequate for I/O scheduling for two reasons. First, our latency metric combines both wait and service time, so it depends on the service times of previous requests. Second, many input features—such as queue size, LBA delta, and the previous request's latency—are order-dependent. When

we change the dispatch order, these values shift and the prediction becomes meaningless. SJF requires latency predictions that are based solely on the service time of the current request, independent of order. As such, we no longer use this model.

3.2 Latency-Prediction Model Setup

Next, we ensured that only unordered features were included by using only the sizes, tags, and operations of the current request along with the N most recently completed requests. Because these N requests were finalized before the current request arrived, they avoided interference, yet still potentially provided non-local historic context that could improve our model's predictive performance. Expanding our feature set—such as integrating lifetime hints, which have been shown to boost write performance by up to 25% [31]—could further enhance prediction accuracy.

Given the exponential distribution of latencies, we first performed a logarithmic transformation on them. Then, to shield predictions from device performance fluctuations, we normalized the values against the previous N requests.

We tested both decision trees and neural networks using the KML library. Speed is critical in an SJF setting since predictions are performed synchronously on a per-request basis. However, we ultimately decided against using decision trees, which are typically faster than neural networks, because we found that their predictive capability was significantly lower than that of neural networks given these input features. In offline testing, the decision-tree model could only reach an accuracy of 76% given a large maximum tree depth of 18 whereas the neural network reached 92%.

Similar to our workload classification model, our latency-prediction model uses exponentially decreasing hidden-layer sizes. We experimented with various values of L, N, and M to maximize accuracy while keeping the average prediction time below the P5 latency (top 5% of *fastest* latencies), ensuring predictions occur faster than the rate of new requests. The optimal configuration was $L=32$, $M=4$, and $N=16$. These numbers differ from the workload classification model since that model focused on maximizing accuracy whereas this model balances accuracy and prediction time.

3.3 Latency-Prediction Results

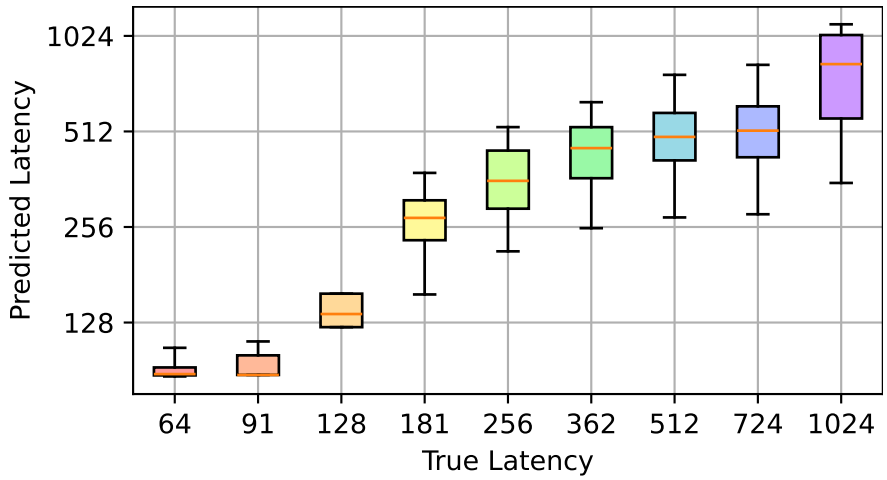

Figure 3: Predicted Latency (μs) vs. True Latency (μs). Axes are log-scale and do not begin at 0.

We believe that the Pearson Correlation Coefficient (PCC) between the true and predicted values is more important than absolute accuracy. The PCC measures the degree of the linear relationship between

two variables[32], with high values near 1 indicating close alignment. Since we are ordering requests, the relationship between two requests is more important than actually predicting a request's latency. Our model achieved a high PCC of 0.90 across the RocksDB and Filebench workloads, illustrated in Figure 3, demonstrating its ability to accurately sort requests by latency.

3.4 Scheduler Design

Even after minimizing ML inference time, we suspect that ML overhead could be a bottleneck in some workloads. Thus, our scheduler is designed with two queues: a FIFO queue and an ML priority queue. If requests are sparse over time, each request will be immediately dispatched from the FIFO queue without incurring overhead from inference. Otherwise, as the queue builds up, there is time to make latency predictions and dispatch them from the ML priority queue in smallest-predicted-latency order.

We use a minimal number of spinlocks when marking requests as "predicted" to prevent requests from being dispatched from both queues. Although we initially had additional spinlocks and aborted predictions if the request had already been dispatched from the FIFO queue, this increased overhead more than simply predicting for all requests. Other papers have also suggested that spinlocks in I/O schedulers can become severe bottlenecks [16].

Additionally, to prevent starvation, we added a variable increment to the predicted latency that increases by a fixed amount δ per request. For example, request i receives an extra $i \cdot \delta$, while request $i+1$ receives $(i+1) \cdot \delta$. This slightly raises the priority of older requests over time, ensuring that every request will eventually have the highest priority. The delta is orders of magnitude smaller than the average request latency to minimize its overall interference.

4 Benchmarking Environment

Component	**Configuration**
CPU	Intel(R) Xeon(R) Silver 4316 16 cores @ 2.30GHz
Memory	4GB DDR4
Storage	Western Digital 1TB HTS541010A9E662 HDD
Software	Ubuntu 18.04 Intel-OST Linux kernel 4.13.2 Filebench 1.5-alpha3, RocksDB 9.9.0

Table 1: Testing environment.

Our testing environment is given in Table 1. Intel OST runs on Linux kernel 4.13; we used three default single-queue schedulers (`Noop`, `CFQ`, and `Deadline`) and three default multi-queue schedulers (`Kyber`, `MQ-Deadline`, and `BFQ`) without modifying their parameters. We used a small VM configuration because the benchmarks do not demand larger amounts of memory or disk. We tested on Filebench and RocksDB workloads, using the default configurations given by the authors [17, 33].

We dropped caches and restarted `blktrace` before runs to prevent memory and storage interference. We also disabled readahead, limited the device queue depth to four, and disabled writeback delay to ensure that performance differences were due solely to scheduling rather than caching or internal disk reordering. These changes were made only for controlled evaluation and would not be required in production.

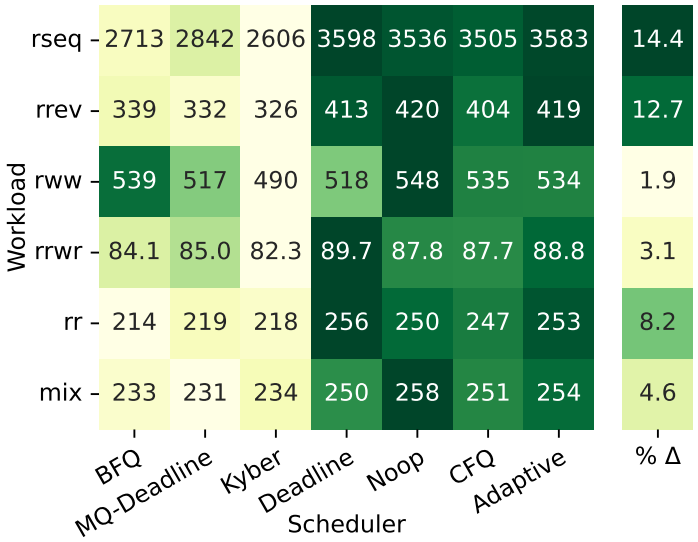

Figure 4: Throughputs (ops/sec) by scheduler of workloads `readseq` (`rseq`), `readreverse` (`rrev`), `readwhilewriting` (`rww`), `readrandomwriterandom` (`rrwr`), `readrandom` (`rr`), and `mixgraph` (`mix`).

5 Scheduler-Switching Evaluation

To compare the performance of our adaptive scheduler switching with the default schedulers, we first benchmarked each $\langle scheduler, workload \rangle$ combination to determine its expected throughput, using the RocksDB workloads `readseq`, `readreverse`, `readwhilewriting`, `readrandomwriterandom`, `readrandom`, and `mixgraph`. We ran each $\langle scheduler, workload \rangle$ combination 200 times and recorded the average for each, resulting in 95% confidence intervals of 1–4%.

In production, the adaptive model would run periodically—every few seconds or milliseconds—and switch to the highest-throughput scheduler. However, since switching between multi- and single-queue schedulers currently requires reboots, we simulated dynamic switching mathematically. Determining an optimal workload-classification frequency that balances responsiveness and overhead is future work.

We define the optimal scheduler for a workload of type u as s^*_u, and the throughput of that scheduler on a workload of type w as $T_{s^*_u}(w)$. Then, to find the expected throughput for a workload of type w, we consider the probability that it is predicted to be of type u (where u might or might not equal w). Then, the actual throughput $T(w)$ achieved for w can be found by summing across all predictions:

$$T(w) = \sum_{u \in workloads} P(predicted = u \mid true = w) \times T_{s^*_u}(w)$$

We can determine $P(predicted = u \mid true = w)$ from the corresponding entry of the confusion matrix.

Figure 4 presents our findings. As expected, adaptive scheduler switching did not achieve the highest throughput for any single workload, since its performance is bounded by the best scheduler for that workload. However, when averaging throughput across all workloads, adaptive scheduler switching outperformed every individual scheduler. Thus, in dynamic environments, our adaptive scheme is superior to statically selecting a single scheduler.

The "% Δ" column on the right side of Figure 4 represents the percent difference between the average for the workload across all schedulers and the throughput of adaptive scheduler switching. The average percent improvement is 7.5%, which can be significant for I/O-bound workloads.

A more varied set of workloads is likely to increase the throughput variance among the $\langle scheduler, workload \rangle$ pairs [34]. Thus, a more diverse and demanding workload set will likely yield an

even higher average percent improvement.

6 SJF Scheduling Evaluation

We now compare the performance of `KML-IOSched`, our SJF scheduler, to the default single-queue Linux kernel schedulers.

To determine the absolute experimental bounds of SJF scheduling, we designed our own Filebench workload consisting of two distinct request types: (1) long file reads with high service times and (2) "NULL" reads, which were made to an out-of-bounds disk location, effectively causing them to return immediately. Although NULL requests would not appear in a real workload, these instantaneous requests allowed us to test the upper bound of latency reduction achievable under ideal scheduling conditions. Each run consisted of a burst of 100 concurrent requests.

We evaluated scheduler performance under this workload by measuring the average latency of I/O requests. The results, normalized against the `Noop` scheduler, are as follows:

Scheduler	**Average Latency Change vs.** `Noop`
`CFQ`	+16.4%
`Deadline`	-7.9%
`KML-IOSched`	-43.9%

Our scheduler, `KML-IOSched`, significantly outperformed the other schedulers, achieving a 43.9% reduction in latency over `Noop`. The `Deadline` scheduler showed a modest improvement, likely due to its inherent mechanisms for reducing seek time. However, `CFQ` performed worse than `Noop`, indicating that its fair-queuing mechanism introduced unnecessary delays for this particular workload.

`KML-IOSched` correctly prioritized dispatching the instantaneous requests first. This means the model successfully inferred the importance of disk location, suggesting that the SJF-based approach can successfully learn workload characteristics without requiring domain-specific optimizations.

We did not observe substantial performance improvements in the default RocksDB and Filebench workloads. This may be due to two reasons. First, our model does not explicitly consider seek time. Seek time is partly captured by the LBA Delta, but modeling it further is challenging due to the complex, non-linear mapping between logical and physical disk locations [19]. Prior work also indicates that detailed seek-time estimates can add significant memory overhead [35]. Second, although our model predicts latency well at a broad scale, it struggles with fine-grained predictions in small request windows. It distinguishes well between requests with different tags (*e.g.*, inodes versus user data) but has difficulty differentiating between similar requests like successive user-level file reads, which dominate the built-in Filebench and RocksDB workloads.

7 Related Work

Hao *et al.* [20, 36] present two OS-level solutions for reducing tail latency in flash storage: LinnOS uses an ML-based model to predict per-chip queuing delays within a RAID scheduler, and MittOS has a mechanism for rejecting requests that will violate SLO deadlines. However, neither predicts latencies as a whole, instead identifying requests predicted to be in the 95–99^{th} latency percentiles. Moreover, LinnOS is limited to RAID systems. In contrast, `KML-IOSched` is general-purpose and capable of modeling the entire spectrum of I/O latencies.

Naweed *et al.* [37] evaluate the performance of a randomized algorithm that continuously switches schedulers to find one that performs well for the current workload. Their approach relies on significant

trial-and-error at run-time. Our workload classification model, conversely, can accurately determine the optimal scheduler with a single prediction.

Tarasov *et al.* [35] leverage a disk latency map to estimate the latency between LBAs, improving deadline enforcement and throughput. However, their approach requires about 400MB of memory for only 5GB of active storage space. Our model avoids significant memory overhead, requiring additional memory only on a per-request basis.

Popovici *et al.* [38] present a scheduler that uses a table-based model to estimate I/O latency. In contrast, we use machine learning to leverage more complex I/O feature vectors, including I/O tags.

8 Conclusion

We have demonstrated the performance of ML-driven I/O scheduling approaches. Our adaptive scheduler-switching model achieved 86% classification accuracy and delivered a 7.5% average throughput gain, while our SJF model reached a Pearson Correlation Coefficient of 0.90 and reduced average latency by up to 43.9%. Thus, data-driven techniques have the ability to significantly outperform static heuristics.

9 Future Work

(1) We have only tested ML-based scheduling on hard disks because previous work suggests that flash storage benefits less from scheduling. Nonetheless, we believe that modeling garbage collection on flash storage could still yield significant improvements. (2) Although we show that our ML model has sufficient features to accurately predict request latency on its own, incorporating mathematically-modeled seek time predictions [21] could offer further benefits. (3) Finally, unsupervised workload-classification techniques can generalize adaptive scheduler switching, improving performance for out-of-distribution workloads.

References

[1] Horizon Editorial. HDD remains dominant storage technology, January 2025. URL *https://horizontechnology.com/news/hdd-remains-dominant-storage-technology-1219/*.

[2] Seetharami R. Seelam, Rodrigo Romero, Patricia J. Teller, and William Buros. Enhancements to Linux I/O scheduling. In *Proceedings of the Linux Symposium*, volume 2, pages 175–192, Ottawa, Ontario, Canada, July 2005. Linux Foundation. URL *https://www.kernel.org/doc/mirror/ols2005v2.pdf*.

[3] Jens Axboe. Cfq IO scheduler, 2007. URL *https://brick.kernel.dk/axboe-lca2007.pdf*.

[4] Yunus Ozen and Abdullah Yildirim. Performance comparison and analysis of Linux block I/O schedulers on SSD. *Sakarya University Journal of Science*, 23:106–112, June 2019. doi: 10.16984/saufenbilder.477446. URL *https://pdfs.semanticscholar.org/e8c5/ab5ce789b8a67b7b0980c097cd8f15e3689a.pdf*.

[5] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[6] Ibrahim "Umit" Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. Improving storage systems using machine learning. *ACM Transactions on Storage (TOS)*, 19(1):1–30, Jan 2023. doi: 10.1145/3568429.

[7] Shi Qiu, Weinan Liu, Yifan Hu, Jianqin Yan, Zhirong Shen, Xin Yao, Renhai Chen, Gong Zhang, and Yiming Zhang. GeminiFS: A companion file system for GPUs. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 221–236, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL *https://www.usenix.org/conference/fast25/presentation/qiu*.

[8] Wenbin Zhou, Zhixiong Niu, Yongqiang Xiong, Juan Fang, and Qian Wang. 3L-Cache: Low overhead and precise learning-based eviction policy for caches. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 237–254, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL *https://www.usenix.org/conference/fast25/presentation/zhou-wenbin*.

[9] Han Dong. Tuning Linux kernel policies for energy efficiency with machine learning. *Red Hat Research Quarterly*, 5(1):20–24, May 2023. URL *https://research.redhat.com/blog/article/tuning-linux-kernel-policies-for-energy-efficiency-with-machine-learning/*.

[10] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. Towards a machine learning-assisted kernel with LAKE. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 846–861, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575697. URL *https://doi.org/10.1145/3575693.3575697*.

[11] 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, Melbourne, Victoria, Australia, 2005. IEEE. doi: 10.1109/TENCON.2005.300837.

[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, pages 67–74, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380690. doi: 10.1145/3409963.3410492. URL *https://doi.org/10.1145/3409963.3410492*.

[13] Damian Valles and Stan McClellan. Using machine learning to optimize Linux networking. *Linux Journal*, 298:128–138, May 2019. URL *https://linuxjournal.rubdos.be/ljarchive/LJ/298/12625.html*.

[14] Michael P. Mesnier and Jason B. Akers. Differentiated storage services. *SIGOPS Oper. Syst. Rev.*, 45(1):45–53, February 2011. ISSN 0163-5980. doi: 10.1145/1945023.1945030. URL *https://doi.org/10.1145/1945023.1945030*.

[15] Michael Mesnier. Open storage toolkit from intel labs, 2015. *https://sourceforge.net/projects/intel-iscsi/*.

[16] Ibrahim 'Umit' Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. A machine learning framework to improve storage system performance. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)*, pages 94–102, Virtual, July 2021. ACM. doi: https://doi.org/10.1145/3465332.3470875.

[17] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL *https://www.usenix.org/conference/fast20/presentation/cao-zhichao*.

[18] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Hoboken, NJ, 10th edition, 2018. ISBN 9781119456339.

[19] Dave Anderson. You don't know jack about disks: Whatever happened to cylinders and tracks? *Queue*, 1(4):20–30, June 2003. ISSN 1542-7730. doi: 10.1145/864056.864058. URL *https://doi.org/10.1145/864056.864058*.

[20] 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)*, pages 173–190, Banff, Alberta, November 2020. USENIX Association. URL *https://www.usenix.org/conference/osdi20/presentation/hao*.

[21] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27: 17–28, 1994.

[22] John S. Bucy, Gregory R. Ganger, et al. The DiskSim simulation environment version 3.0 reference manual. Technical report, Carnegie Mellon University, January 2003. *www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-CS-03-102.pdf*.

[23] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, , and John Wilkes. On-line extraction of SCSI disk drives parameters. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–156, Ottawa, Canada, 1995. ACM.

[24] N. Talagala, R. Arpaci-Dusseau, and D. Patterson. Micro-benchmark based extraction of local and global disk characteristics. Technical Report CSD-99-1063, UC Berkeley, 1999.

[25] D. Shi, Y. Yu, and H. Yeom. Shedding light on the black-box structural modeling of modern disk drives. In *15th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'07)*, pages 410–417, Istanbul Turkey, 2007. IEEE Computer Society.

[26] Jihun Kim, Joonsung Kim, Pyeongsu Park, Jong Kim, and Jangwoo Kim. SSD performance modeling using bottleneck analysis. *IEEE Computer Architecture Letters*, 17(1):80–83, 2018. ISSN 1556-6056. doi: 10.1109/LCA.2017.2779122.

[27] Sterling Hansen, Morgan Githinji, Etienne Elie, and Charles Anyimi. Modeling compute storage quality of service and latency using sigmoid functions. In *Proceedings of the 2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 813–818, Austin, TX, May 2022. IEEE. doi: 10.1109/ISCAS48785.2022.9937953.

[28] Krishna T. Malladi, Mu-Tien Chang, Dimin Niu, and Hongzhong Zheng. FlashStorageSim: Performance modeling for SSD architectures. In *Proceedings of the 2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–2, Shenzhen, China, Aug. 2017. IEEE. ISBN 978-1-5386-3487-5. doi: 10.1109/NAS.2017.8026860.

[29] Yang Hu, Zhiming Zhu, Shuangwu Zhang, Chao Ren, and Hao Luo. SSDsim: Simulation tool of SSD's internal hardware and software behavior. GitHub repository, 2011. URL *https://github.com/huaicheng/ssdsim*. Version 2.0; accessed 2025-05-21; *https://github.com/huaicheng/ssdsim*.

[30] Jinsoo Yoo and Youjip Won. Modeling I/O latency of SSDs. In *Proceedings of the 4th IEEE International Conference on Network Infrastructure and Digital Content (IC-NIDC)*, Friendship Hotel, Beijing, China, September 2014. IEEE.

[31] Jens Axboe. Add support for write life time hints. lwn.net, 2017. URL *https://lwn.net/Articles/726477/*.

[32] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, England, 3rd edition, 2007. ISBN 9780521880688.

[33] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, Boston, MA, February 2021. USENIX Association. ISBN 978-1-939133-20-5. URL *https://www.usenix.org/conference/fast21/presentation/dong*.

[34] Ben Cane. Improving Linux system performance with I/O scheduler tuning, 2017. URL *https://www.cloudbees.com/blog/linux-io-scheduler-tuning*.

[35] Vasily Tarasov, Gyumin Sim, Anna Povzner, and Erez Zadok. Efficient I/O scheduling with accurately estimated disk drive latencies. In *In Proceedings of the 8th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'12)*, pages 36–45, Pisa, Italy, 2012. The George Washington University.

[36] 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 Twenty-Sixth ACM Symposium on Operating Systems Principles*, SOSP '17, pages 1–16, Shanghai, China, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132774.

[37] Asad Naweed, Joe Di Natale, and Sarah J Andrabi. Modification and evaluation of Linux I/O schedulers. Technical report, University of North Carolina at Chapel Hill, 2014. URL *https://www.cs.unc.edu/~sandrabi/Project_work/ModificationandEvaluationLinuxIOSchedulers.pdf*.

[38] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Robust, portable I/O scheduling with the Disk Mimic. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, pages 297–310, San Antonio, TX, June 2003. USENIX Association. URL *https://www.usenix.org/legacy/events/usenix03*.