LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network

Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim†,
Henry Hoffmann, and Haryadi S. Gunawi

University of Chicago †Surya University

Abstract

This paper presents LinnOS, an operating system that leverages a light neural network for inferring SSD performance at a very fine—per-IO—granularity and helps parallel storage applications achieve performance predictability. LinnOS supports black-box devices and real production traces without requiring any extra input from users, while outperforming industrial mechanisms and other approaches. Our evaluation shows that, compared to hedging and heuristic-based methods, LinnOS improves the average I/O latencies by 9.6-79.6% with 87-97% inference accuracy and 4-6µs inference overhead for each I/O, demonstrating that it is possible to incorporate machine learning inside operating systems for real-time decision-making.

1 Introduction

Predictable performance is an important requirement for today's and future systems [19, 51, 55, 65]. For data-center systems serving web search, email, and many other types of interactive services, predictable latency is even more important. On the bright side, faster and faster SSDs are available and becoming a dominant factor in the storage market [10]. On the negative side, SSD internal complexity continues to grow, and achieving highly predictable latency on modern flash devices remains an open challenging problem.

Due to the intrinsic NAND idiosyncrasies, modern flash devices behave like an "operating system," managing all of its internal resources with background operations such as garbage collection, buffer flushing, wear leveling, and read repairs. While important, these are the kinds of operations that pose a threat to latency predictability [15, 25, 27, 30, 49, 53, 71, 76], which is still a fresh problem faced by many storage industries in recent years [4, 31, 50, 57]. Furthermore, with a report that flash devices contribute to more than 19% of the total response time for some online applications [76], more solutions should be explored.

Because the device itself cannot mask the unpredictable latency, a vast amount of research has been devoted to this space. "White-box" approaches—that re-architect device internals [17, 33, 34, 36, 47, 61, 68, 71]—are powerful, but face a high barrier to adoption unless SSD vendors implement the recommendations. In the middle ground, "gray-box" methods suggest partial device-level modification combined with OS or application-level changes working together in taming the latency unpredictability [38, 39, 40, 58, 76, 77]. However, they also depend on the vendors' willingness to modify the device interface. Finally, more adoptable "black-box" techniques attempt to mask the unpredictability without modifying the underlying hardware and its level of abstraction. Some of them optimize the file systems or storage applications specifically for SSD usage [18, 37, 41, 42, 43, 54, 59, 69, 70], while some others simply use speculative execution [1, 5] but pay the cost of extra I/Os due to being oblivious to storage behaviors. Among all the approaches above, arguably, the most popular solution is speculative execution given its simplicity and capability to mitigate *every* slow I/O. For example, "hedged requests" [21], a form of speculative execution, is supported in many widely-used key-value stores today [1, 5, 8].

We take a new approach: let the device be the device (black-box) and do not redesign the file systems or applications, but *learn* the device behavior (*i.e.*, not be storage oblivious). The key to our approach is learning. Can we learn the behavior of the underlying device in a black-box way and use the results of the learning to increase predictability, so applications can know in advance whether their performance expectations can be fulfilled? This is a domain that machine learning can likely help. We introduce LinnOS, an operating system that has the capability of learning and inferring *per-I/O* speed with high accuracy and minimal overhead using a lightweight neural network. We show how LinnOS helps storage applications, in particular storage arrays/clusters with built-in failover logic (*e.g.*, flash RAID, Cassandra, MongoDB), achieve extreme latency predictability on unpredictable flash storage.

The biggest challenge for LinnOS is to be as effective and fine-grained as the popular approach, speculative execution, which can mitigate *every* slow I/O by sending a duplicate I/O to another node or device. Speculative execution's success in increasing predictability comes at the cost of poor resource utilization. The key to avoiding this cost is to know the current activities going on inside the devices and always schedule I/Os to those devices that will provide faster responses.

However, because keeping the abstraction barrier is a fundamental constraint, we need to learn to infer latency and make the inference highly usable. Achieving this requires *learning and inferring on a very fine, per-I/O scale* in a live fashion. To the best of our knowledge, there is no existing learning approach for I/O scheduling that supports such fine-grained learning due to the challenges of achieving per-I/O accuracy and fast online inference. To address this, LinnOS introduces three technical contributions.

First, LinnOS converts the hard latency inference problem into a simple *binary* inference ("fast" or "slow" speed). We take advantage of the typical latency distributions in system deployments, specifically, a behavior that forms a Pareto distribution with a high alpha number. In other words, most of the time (*e.g.*, >90%), the latency is very stable, but occasionally (*e.g.*, <10% of the time), the latency exhibits a long-tail behavior [16, 21, 45, 53]. The behavior of flash storage reflects the same distribution [15, 26]. In this simple view where users only want "slow" I/Os to become "fast," inferring the exact latencies is overkill. With this intuition, LinnOS comes with an algorithm that monitors the latency distribution of the current workload running on the flash device and computes a roughly optimal threshold that separates the slow and fast speed ranges.

Second, with the binary model, LinnOS employs a simple admission control for clustered storage applications. LinnOS makes a binary inference on every incoming I/O using a light neural network model that infers the I/O speed in advance in a black-box manner without any guide from the device nor application. If the I/O is inferred to be fast, LinnOS will submit it to the flash device; otherwise it will revoke the I/O and inform the application. With this timely and straightforward binary information, the storage application can quickly failover the I/O to another node or device that holds the same replica. Furthermore, resources are efficiently utilized because the original slow I/O has been revoked.

Third, LinnOS balances the accuracy and performance of the neural network. High accuracy but high inference time will lead to a significant per-I/O overhead, especially for modern SSDs. On the other hand, lowering inference time by lowering accuracy will lead to many false inferences that make storage performance hard to reason about.

For high accuracy, LinnOS profiles the latency of millions of I/Os submitted to the device (a natural "data lake"), which will be used to train the neural network. Furthermore, as we convert regression to a simple binary classification, the output accuracy is significantly improved (akin to the simplicity of "cat or dog" image classification). The next challenge is to decide the input features that matter most to improving accuracy. We will present our surprising findings. For example, "important-looking" features such as block offsets, read/write flags, or long history of writes do not play a significant role. In the end, the input features become tractable with only two types of information: the latencies of a few recently completed I/Os and the number of pending I/Os when those I/Os and the current, to-be-inferred I/O arrived.

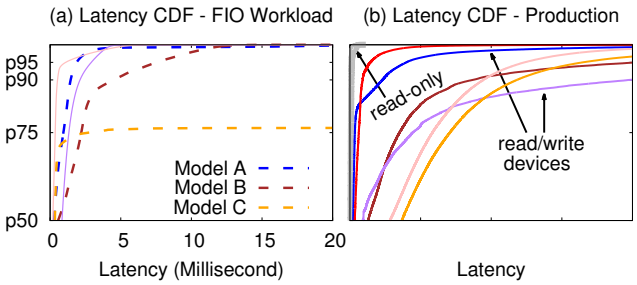

Figure 1: **Latency distribution.** *The figures show CDFs of block-level read latencies, as discussed in Section 2. For the left figure, we ran one FIO workload on five different SSD models (the five CDF lines). For the right figure, we plot the latencies of seven block-level traces obtained from four read-write servers (colored lines) and two read-only servers (bold gray lines) in Azure, Bing, and Cosmos clusters. The x-axis is anonymized for proprietary reasons. The traces are available from Microsoft with NDA.*

For performance, the challenge is to make an inference (admission decision) in sub-10µs, which is crucial as we target fine-grained live inference for fast storage devices. While using deeper models with more features can improve accuracy, it will hurt inference latency and would be too expensive for usage in the I/O layer. Through several design iterations, we cut the inference time to 4-6µs with minor accuracy loss, achieved with several methods: a 3-layer light neural network, weight quantization, and (optional) 2-threaded/2-core matrix multiplication.

Our evaluation shows that LinnOS supports a wide variety of black-box devices (10 device models tested) and works on real production traces without requiring any extra input from users (*e.g.*, hints about traces/devices or latency deadlines etc.), outperforms industrial approaches such as pure hedging, and beats simple and "advanced" heuristics that we design. Compared to these methods, LinnOS, complemented by hedging based on the learning outcome, further improves the average I/O latencies by 9.6-79.6% with 87-97% accuracy and only 4-6µs inference overhead for every I/O.

Overall, we show that it is plausible to adopt machine learning methods for operating systems to learn black-box devices. We conclude with many interesting discussions to explore in the future. LinnOS code is made public.

2 Background

Unpredictability. To motivate the problem, the colored lines in Figure 1a show read latency distribution in a read-write workload running on five different SSD models ranging from consumer SATA and NVMe SSDs to new data-center ones. Model A delivers fast and stable latencies up to about "p98" (the 98^{th} percentile), but models B and C exhibit larger la-

tency tails starting at p90 and p75, respectively. However, when the write operations are converted into read I/Os, the performance becomes highly stable without much latency tail (not shown in the figure). Figure 1b also confirms this in real production scenarios in Microsoft SSD-backed servers. The colored lines show block-level read latencies of read-write servers (more variability), and the gray lines for read-only servers (more predictability). All of these confirm how write-triggered garbage collection (GC), buffer flushing, and other internal operations are contending with user read I/Os. We only address read performance unpredictability because we found write latencies to be (surprisingly) stable as they are absorbed by the internal memory buffer on the device, hence not affected by internal contentions. Write latency spikes only happen when the buffer is full (rarely happened due to internal periodic flush).

Internal complexities. Inferring when a flash drive is exhibiting tail latency is hard given the internal complexities that factor into latency behavior. As a couple of examples, I/Os contend with each other if they fall into the same chip or channel, which depends on the hidden striping and partitioning logic; two user I/Os that go to separate channels might have different fates when one channel is occupied by GC data transfers between the chips in the channel. Our internal findings show that SSDs can have wide layouts (*e.g.*, 32 channels with four chips per channel) or deep layouts (*e.g.*, four channels with 16 chips per channel), where the latter will cause more channel contention. Some SSDs employ large write buffers from 256MB to as small as 12 MB and can periodically flush from every 3ms to as high as one second. As shown in Figure 1, this internal contention can affect from 1% to 25% of all read requests.

In this context, modern storage applications usually apply a "wait-then-speculate" approach that is agnostic about the device's internal complexities. For example, with hedging, applications wait for a timeout (*e.g.*, the p95 latency), then issue extra speculative I/Os, and use whichever is the faster response. Speculative execution works well for coarse-grained tasks (tens to hundreds of seconds), but is ineffective for flash storage since the waiting is costly when the expected response time is less than a few milliseconds (§5.3).

Machine learning. Before we tried machine learning techniques such as neural networks, we asked whether simple heuristics would be accurate enough in inferring per-I/O speed. For example, one might assume that a long I/O queue length implies longer latencies—a heuristic that works well for spinning disks [13, 62, 64]. However, for SSDs, due to the internal complexities, queue length is not highly correlated with delay (we did not find a high Pearson's correlation or Spearman's correlation between queue length and I/O latency). We also created a more "advanced" heuristic, but it did not yield a satisfying result (more in the evaluation section). While it is possible to keep crafting the right heuristic that can adapt to different workloads and device models, we decided to resort to machine learning. Recent operating and distributed systems research successfully employed machine learning for resource allocation and scheduling [22, 23, 24, 28, 48, 51, 52, 60]. A similar exploration targeting the I/O layer can lead to a powerful result, as we show in this paper.

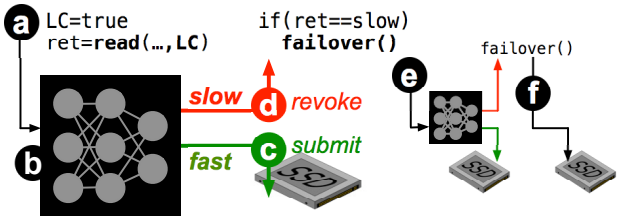

Figure 2: **Usage scenario.** *This usage scenario is explained in Section 3.1. "LC" implies latency critical.*

3 Overview

We now give the overview of LinnOS, its usage scenario, architecture, and challenges, followed by its design (§4).

3.1 Usage Scenario

LinnOS is beneficial for parallel, redundant storage such as flash arrays (cluster-based or RAID) that maintain multiple replicas of the same block, as illustrated in Figure 2. **(a)** With LinnOS, when a storage application performs an I/O via OS system calls, it can add a one-bit flag, hinting to LinnOS that the I/O is latency-critical (`LC=true`), *e.g.*, for interactive services. Such tagging of critical operations has been proposed many times [73, 76], but in our case, the bit is used to trigger LinnOS to infer the I/O latency. **(b)** Before submitting the I/O to the underlying SSD, LinnOS inputs the I/O information to the neural network model that it has trained, which will make a binary inference: fast or slow. **(c)** If the output is "fast," LinnOS *submits* the I/O down to the device. **(d)** Otherwise, if it is "slow," LinnOS *revokes* the I/O (not entered to the device queue) and returns a "`slow`" error code. **(e)** Upon receiving the error code, the storage application can failover the same I/O to another replica. **(f)** In the worst case where the application must failover to the last replica, this last retry will not be tagged as latency-critical so that the I/O will complete and not be revoked.

3.2 Overall Architecture

Figure 3 shows LinnOS's overall architecture, which consists of five main components.

(a) The model. At the center of LinnOS is the speedy inference model (Section 4.3) with a light neural network. The model's input features are information about the current outstanding I/Os and recently completed ones. The model infers

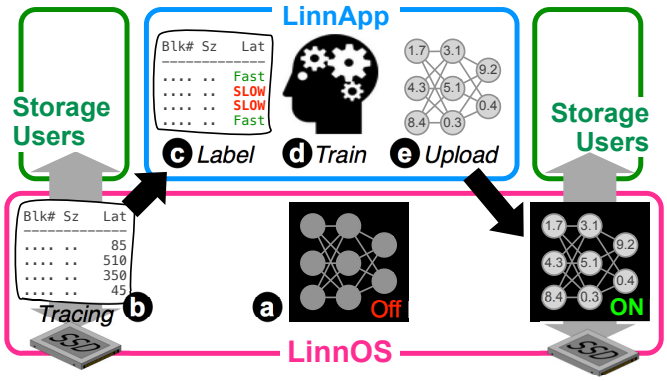

Figure 3: **LinnOS architecture.** *The figure displays LinnOS architecture including LinnApp, as summarized in Section 3.2. The two SSD pictures represent the same SSD instance; the left one depicts tracing/training and the right one live inference on the SSD.*

the speed of every incoming I/O individually. The model's output is the binary inference about the I/O (fast/slow).

(b) Tracing. To train the model, LinnOS uses the current live workload that the SSD is serving. To have a rich representative dataset, this can be done during normal busy hours. The I/O metadata (block offset, size, read/write) and their resulting latencies are recorded using `blktrace`. With millions of I/Os collected, this naturally forms the "data lake" of our model. The training data (the collected trace) is expected to be different than the "test data" (the I/Os that will be inferred when the model is activated).

(c) Labeling with inflection point analysis. The collected trace is then supplied to LinnApp, a supporting user-level application. LinnApp has three main jobs: labeling, training, and uploading trained weights to LinnOS. Because the model is designed to produce a binary output, the model must be trained with two labels, "fast" and "slow." Hence, given a latency distribution in the trace, LinnApp runs an algorithm (§4.2.1) that finds the "inflection point," a latency value that divides the fast and slow latency ranges.

(d) Training. With this inflection point, LinnApp labels the traced I/Os with "fast" and "slow" labels and proceeds with the training phase (using TensorFlow). We emphasize the labeling is done automatically *without* human input. This training phase can be run anywhere, on GPU or CPU nodes.

(e) Uploading weights. The training phase generates the weights for the neurons in the model that will be uploaded to LinnOS. Because using floating points is not well supported in OS kernel, the weights are converted to integers by quantization. The model is then activated, and LinnOS is ready to make inferences and revoke "slow" I/Os.

3.3 Challenges

Using a machine learning approach for making online, fine-grained inferences on I/O speed requires us to solve the following fundamental challenges.

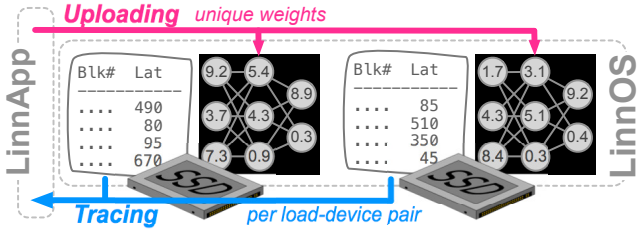

Figure 4: **Anticipating heterogeneity.** *The figure shows heterogeneous trained models, as mentioned in Section 3.3.*

High accuracy. The inference must be accurate. We should not revoke I/Os that can be served fast ("false revoke") or submit those that will be slow ("false submit"). Accuracy depends on careful output labeling and input features selection. If the label classification is too complicated, high accuracy is hard to achieve, *e.g.*, we find that classification by linear bucketing (0-10, 10-20µs, etc.) or exponential bucketing (0-1, 2-4µs, etc.) is hard to make accurate and should remain as a future work. However, the simple two-class approach (fast or slow) simplifies the output into a binary format, which helps the model achieve high accuracy.

Fast inference. For modern SSDs, while the raw NAND read latency is advertised to be below 100µs, we see that for typical production workload on data-center SSDs (Section 5), the actual user-perceived latency is above 200µs more than 50% of the time. Given this observation, we believe the challenge is to do decision-making in around 5µs, a $<$3% overhead per I/O. Fast inference depends on input preprocessing, the depth of the layers, neuron complexity, and feature representation. Using deep layers that tend to improve accuracy is not attractive in our problem domain. The input features must be minimized to include only the features that matter. Hence, we must balance accuracy and performance. Moreover, considering that operating systems run on CPUs, the models must be CPU-friendly [67].

Anticipating heterogeneity. In flash arrays (RAID or cluster-based), the user load is not always balanced, and all the flash hardware might not be homogeneous. Because this heterogeneity can lead to different latency distributions observed on different devices, we should not use one global latency value (*e.g.*, 1ms inflection point) to differentiate fast and slow speed for all the devices. For example, 3ms perhaps could be considered fast enough on slower SSDs or the ones with heavier user load. While we do not expect that the heterogeneity will be extreme (*e.g.*, a good storage system typically balances the load very well), heterogeneity is still important to address. For this reason, LinnApp collects per-device traces and trains the model for *every* load-device pair in the array (Figure 4). After the training phase completes, LinnApp supplies the model weights to all instances of LinnOS in a cluster-based array or to one instance of LinnOS in a RAID-based array. In the latter, LinnOS carries N trained

models for the N drives in the RAID. Furthermore, to anticipate workload changes over time, LinnApp occasionally recollects traces (*e.g.*, every few hours) to check if the inflection point has shifted significantly such that the model must be retrained.

4 LinnOS Design

In this section, we describe our solution to the challenges mentioned above. To the best of our knowledge, LinnOS is the first operating system that successfully infers I/O speed in a *fast*, *accurate*, *live*, *fine-grained*, and *general* fashion. The key to this is the "lightness" of the neural network model that LinnOS employs. This section presents the final design and the principal intuitions about how we get there. We will explain LinnOS design chronologically, from data collection (§4.1), labeling via inflection point analysis (§4.2), the model design (§4.3), and how to improve its accuracy (§4.4) and performance (§4.5), and summarize its advantages (§4.6).

4.1 Training Data Collection

This project started with a simple question: can we infer the performance of every I/O accurately? Since we use machine learning, accuracy depends on the amount of true-signal data available, the more, the better. Fortunately, I/O systems inherently can collect a large amount of data. Given low-overhead tracing tools and hundreds of KIOPS of workload that modern SSDs can serve, collecting a large amount of data for training is not an issue (a large "I/O data lake").

For every load-SSD pair to model, LinnApp collects traces of the real workload running on the drive. For example, for inferring a production workload performance on a particular SSD in deployment, an online trace will be collected. For every I/O, we collect five raw fields, the submission time, block offset, block size, read/write, and most importantly, the I/O completion time. Because the model input (Section 4.3) does not necessarily take the same raw fields, in this phase, we also convert the fields to the input feature format.

The main challenge here is to decide how long the trace should be. If the behavior of the training data (the latency distribution) is very different from that of the "test" data (the to-be-inferred I/Os), the inference accuracy will drop. In this work, we take a simple approach where we use a busy-hour trace (*e.g.*, midday). In the evaluation (§5.2), we show that for production workloads, a busy-hour trace well represents the other hours, *i.e.*, the inflection point does not deviate much. As mentioned above, to anticipate a dramatic shift in workload behavior, retracing and retraining can be done.

4.2 Labeling (with Inflection Point)

As we employ a supervised classification approach, the model must be trained with labels. If we label every I/O with the actual µs-level latency, there will be too many labels for our problem domain; a user might not care if the I/O is delayed by 1µs. Another option is to use a linear (0-10µs, 10-20µs, and so on) or exponential labeling (2-4µs, 4-8µs, and so on). While these fit better, the model is still hard to make accurate and fast after many design iterations. The accuracy only reached 60-70% because many times, an I/O that should fall into a specific group (*e.g.*, 128-256µs) is often mis-inferred to the neighbor groups (*e.g.*, 256-512µs)—"a Lhasa Apso dog can easily be misidentified as a Shih Tzu dog." This is perhaps why prior successes in auto-learning storage performance were only done at a coarse-grained level such as average latency or throughput aggregated for many requests [29, 66, 74].

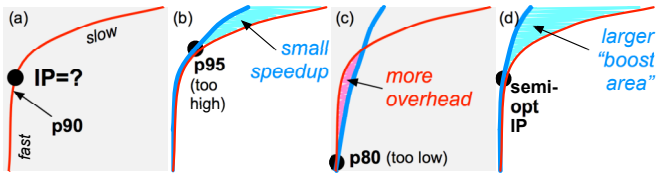

Figure 5: **Inflection point (fast/slow threshold).** *The figures show the results of using a higher, lower, or semi-optimum inflection point (IP) for the fast/slow threshold as explained in Section 4.2. The figure format is latency CDF, as in Figure 1.*

With all this mind and an understanding of how performance variance behaves in the field [15, 21, 26, 45, 49], we observe that latencies often form a Pareto distribution with a high alpha number [7]. As an example shown in Figure 5a, 90% of the time, the latency is likely stable, but in the other 10% of the time, it starts forming a long tail. Such a Pareto distribution clearly contrasts the fast and slow regions. Hence, a simple conjecture can be made that users only worry about the tail behavior, not the precise latency.

To separate the two regions, we need to find the "best" *inflection point* (marked with "**IP**=?" in Figure 5a) for maximizing the latency reduction. Setting the inflection point too relaxed (*e.g.*, the p95 latency in Figure 5b) will make LinnOS treat the relatively slow I/Os between p90 and p95 as "fast" (no failover), reducing the scope for effective retries, hence failing to cut many tail latencies, as highlighted by the small shaded area between the original and projected distributions (more in §4.2.1) in Figure 5b. On the other hand, setting the inflection point too low (*e.g.*, the p80 latency in Figure 5c) will make LinnOS revoke too many I/Os, including those that are supposed to be fast, which will induce unnecessary retry overhead as shown in Figure 5c.

An optimum inflection point implies that for every slow I/O that will be revoked, it is likely that the other replicas can serve it fast within the same time frame. Likewise, for every fast I/O, it should not be failed over. Finding this optimum point will deliver the maximum gap between the original tail-heavy and tail-free distributions, as shown by the large shaded area in Figure 5d. Finding an optimum value how-

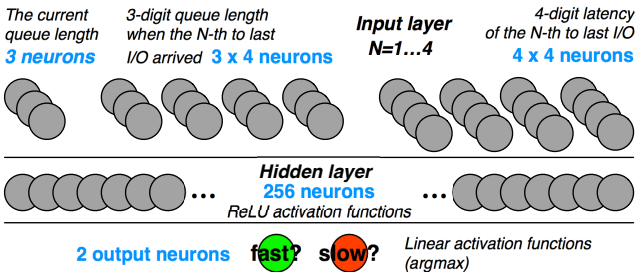

Figure 6: **Light neural network.** *The figure depicts LinnOS 3-layer neural network explained in Section 4.3.*

ever is hard in practice, fundamentally because of the many unknowns: we do not know which replica the request will be failed over to (application dependent); the training data is only an approximation of the future unknown test data; other variability such as CPU or network contention can factor into unknown retry overhead. The next section describes our best-effort algorithm in finding a semi-optimum inflection point for every workload-device pair.

4.2.1 Inflection Point Algorithm

First, during data collection, we collect t workload traces (T_1 to T_t) running on d devices (D_1 to D_d), respectively, where $t{=}{=}d$. Every trace T_i gives us the latency distribution of the workload running on the device (as in Figure 5a). To find the unique inflection point (IP) value for every $T_i{-}D_i$ pair, we run a user-space simulation based on random replica selection, with the assumption that latency delay is independent across the SSDs. For illustrative purposes, we use specific device numbers (*e.g.*, D_1) in our explanation below.

(1) For every $T_i{-}D_i$ pair, we pick a starting IP value where the slope of the CDF is one (likely entering the tail area). For example, if for D_1, T_1's 45-degree slope is at $y{=}p90.5$ and $x{=}1ms$, then the IP value is initially set to 1ms. **(2)** For the currently simulated device, D_1, we run a simulation of one million I/Os, ($r_{i=0..1000000}$) where each I/O request r_i takes a random latency value from T_1's real latency distribution. We then simulate LinnOS admission control: if the chosen latency is smaller than 1ms (the current IP), the r_i's new latency is set to be the same; else, if it is larger than 1ms, it will be revoked and failed over to another randomly selected node (*e.g.*, D_4) where a new random latency is picked from its trace, T_4, and the admission control is repeated (submit or revoke). We assume three replicas (configurable), hence a request can only be revoked a maximum of two times. **(3)** The simulation produces the new, optimized latencies for all the r_i in workload trace T_1 that previously went to only one device, D_1, but now can be redirected as if LinnOS admission control is activated. These optimized r_i latencies form the new CDF (as in the bold blue line in Figure 5d). Using the original and new CDFs, we can calculate the *area difference* (the shaded "boost area" in Figure 5d), which represents the latency gain if $1ms$ ($p90.5$) is used as the IP value. **(4)** Still, for D_1, we repeat all the steps above by moving $^+/_-0.1$ percentile within the $^+/_-10$ percentile ranges from the initial IP value. For every new IP value, the simulation gives a new boost area. We now can pick the IP^{max}, the IP value that gives us the *largest (positive) boost area*, which will be used as the fast-slow threshold in training the model for device D_1. **(5)** We repeat all the steps for other devices (D_2, D_3, etc.). At the end, for every $T_i{-}D_i$ pair, our algorithm generates a unique IP_i^{max} value. All these steps are repeated upon recalibration (§4.4).

4.3 Light Neural Network Model

Before we decided to build a light neural network model, we explored various learning methods such as logistic regression, decision trees, and random forests. We found that the accuracy only ranges from 17-84%, while a basic neural network can reach a better accuracy. Although it is possible to continue optimizing each of these methods to its full potential, we decided to start from an acceptable baseline that our initial neural model delivered. Below, we describe our final model (Figure 6), from input features, their representation, to the neural layers. We will emphasize how we use storage intuitions to design the model, as opposed to brute-force.

Input features. To infer the speed of every I/O, our model takes three inputs: **(a)** the number of pending I/Os when an incoming I/O arrives (in the number of 4KB pages, including the incoming I/O), **(b)** the latency of the R most-recently completed I/Os, where we set R as 4, and **(c)** the number of pending I/Os at the time when each of the R completed I/Os arrived. We now reason about these necessary inputs.

Deciding the first feature is straightforward—an I/O latency typically correlates with how many I/Os are currently pending. The unit we use here is the number of 4KB pending pages, and the reason is that the lowest granularity of striping inside SSDs is typically at the page level and the main contention is at channel and chip level.

While for disks, the first feature might be sufficient for inferring single-spindle performance, for SSDs, the other two features are required. In essence, to speculate whether the SSD is currently busy internally, we need to record a small piece of historical information, the latencies of the last four I/Os, as well as how many pending I/Os existed when those I/Os arrived. Put simply, if recent I/Os experienced a long delay without many pending I/Os, then the model could learn that there is likely an internal contention due to device-level activities such as GC, internal flushing, or wear leveling. In this case, the model will suggest revoking incoming I/Os until the number of pending I/Os drops substantially so that the device can provide fast responses despite heavy internal activity. Once the device resumes serving I/Os, the model can tell whether the device-level contention is over from the returned latency values.

Our features above look simple because we have removed

unnecessary features after many design iterations. For example, we surprisingly found that important-looking features such as block offsets, read/write flags, or long history of writes do not significantly improve accuracy. We make several conjectures. First, on read/write flags, although NAND-level read/write latencies differ, almost all medium/high-end SSDs employ write buffering. Thus, the problem of read-behind-write is no longer observable. More likely observable is read-behind-buffer-flush delays, which can be learned from our input features. Second, on block offsets, because we target production workloads and the fact that SSDs typically stripe incoming I/Os uniformly across all channels and chips (or with some bounded partitioning), the workload is likely to be evenly scattered, hence block offsets do not really matter for learning. In other words, scenarios where a batch of incoming I/Os with block offsets that simultaneously hit only one chip rarely happen in the field. Third, on history of writes, internal activities such as GC and buffer flush often happen in a short burst, hence they can be sensed by just observing the speed of the last four I/Os. These are surprising but fortunate findings because using just a small set of features will reduce the model's overhead.

Input format. The next challenge is to choose the right input format to be fed to the neurons. First, for the R value, if accuracy is the only important metric, we should record more completed I/Os (the higher R, the better), but it would prolong inference time as the number of neurons would increase. We found that R=4 suffices for balancing performance and accuracy.

In another simplification, we format the number of pending I/Os into three decimal digits. For example, the format for 15 pending I/Os is three integers {0,1,5}. Three digits suffice as device queue length of over 1,000 is rarely heard of. Similarly, for the latencies of the recent completed I/Os, we break the µs latency value into four digits. For example, a latency of a recent I/O that completed in 240µs will be formatted as four features {0,2,4,0}. Latencies larger than 9,999µs will be capped to {9,9,9,9}. In total, our model takes 31 input features, each a one-digit decimal number.

Reformatting the original integers into decimal digits is an effective trade-off. If we use bits and supply every bit to every neuron, there will be too many neurons that increase the model size and hurt inference time. On the other extreme, if every neuron takes a raw integer value, the neurons need to learn over a wide input range, which makes learning/training harder (*e.g.*, latency value can range from 1µs to over 9,999µs). With decimal digits, we make the neuron learning bounded within a small range of 0 to 9.

The network. The final model is a fully-connected neural network with only three layers ("light"), including one input/preprocess layer, one hidden layer, and one output layer, as shown in Figure 6. All the neurons are regular linear neurons (y=wx+b).

The input layer is supplied with the 31 features described above. The raw information from the block layer is converted to the feature format, in an offline way for training and an online way for live inference. For the latter, with some programming optimization, we can achieve O(1) preprocessing overhead. Next, the hidden layer consists of 256 regular neurons. This layer uses RELU activation functions for its low computation cost and ability to support non-linear modeling. More neurons will cause longer inference time and fewer neurons less accuracy. Lastly, the output layer has two neurons with linear activation functions. We use an `argmax` operator to convert the output to a binary decision (*e.g.*, {0.4,0.6} to {0,1}). Overall, this design makes the network lightweight and easy to integrate into the OS, while balancing inference accuracy and performance.

Preceding design iterations. Here we briefly describe how we reach the current design. We started by using the I/O offsets in binary format (32-bit) as the input features since the device FTL mapping basically uses I/O offsets to decide where the I/Os go, which defines the resource contention. This setting allows the learning models to achieve higher accuracies (up to 99% for some traces), however it has a heavy model and high inference overhead, which is impractical for real-time usage. We further trimmed the heavy model but could not find a reasonable tradeoff between generality and inference overhead. As a result, we took a step back from the fine-grained features and switched to more aggregate ones, and finally reached the current design.

4.4 Improving Accuracy

To further improve the model accuracy, we perform false-submit reduction via biased training, model recalibration via retracing/retraining, and inaccuracy masking with high-percentile hedging.

Reducing false submits. An accurate inference means LinnOS submits I/Os that will be fast (true negative) and revokes those that will be slow (true positive). Reversely, inaccurate cases can be categorized into **(a)** "false submit" (false negative) wherein the model believes the request will be served fast, making LinnOS submit the request to the device, but the request will take longer than the fast-slow threshold, or **(b)** "false revoke" (false positive) where the I/O is revoked, but in fact, it can be served fast by the device.

Using the same system intuition on typical latency distributions in the field (Section 4.2), we found that *reducing false submits* is far more important, while false revokes are more tolerable. When the storage devices of a cluster exhibit similar tail behavior (high-alpha Pareto), the *probability* that peer devices are simultaneously busy is relatively *small*. For example, with three replicas and P% busyness, the probability that all the replicas are busy around the same time is $(P/100)^3$ (*e.g.*, 0.000125 with 5% busyness). Another factor is that, with faster networks, a failover cost can be as low

as 1-6µs for flash arrays across PCIe or Fiber Channel or 5-40µs across Ethernet [3] (plus some negligible software overhead).

To summarize, the wrong inference penalty is small for false revokes but high for false submits. In the latter, the I/O will be "stuck" in the device and cannot be revoked. This motivates us to use *biased training* for reducing false submits by allowing more false revokes. We do this by customizing the categorical hinge loss function with a multiplier that puts more penalty weights for false submits, which makes the trained models favor false revokes.

Recalibrating. Another source of inaccuracy happens when the inflection point computed over the training data does not represent the same threshold of the "test" data (the workload during live inference). This can happen under significant workload changes that cause shifts in the latency distributions of the nodes in the cluster. Fortunately, our evaluation of production traces shows that latency distributions do not widely shift across hours (§5.2). However, to anticipate this scenario, re-tracing and re-computation of inflection point analysis can be done periodically every few hours. If in the new workload-device pair, the inflection point has shifted by five percentiles, LinnApp will retrain the model using the newly collected trace and re-upload the new trained weights to the device. Running `blktrace` during the busiest hour in the production workloads we use only generates 300 MB of data (85 KB/s of trace writes) and increases CPU overhead by 0.5% (only relevant parameters are traced).

Masking small inaccuracy. Our methods above managed to increase accuracy up to 98%. Just like other neural networks, achieving 100% accuracy is fundamentally hard and usually implies a lack of generality. Within the small inaccuracy, the long latency tail due to false submits still needs to be circumvented. This is where we marry learning and hedging [21]. When the false submit rate[1] (Section 5.4) is significant (*e.g.*, >5%), we use the rate as an indicator for the hedging percentile value. For example, if 6% of the inferences produce false submits, then p94 hedging will be applied. When the false submit rate is lower, we round up to conventional p95 hedging. Though sometimes this design issues extra I/Os, we show that it can further improve the performance (§5.3).

4.5 Improving Inference Time

A large part of deep neural network (DNN) research mainly focuses on how to structure even larger networks to achieve the highest possible accuracy [11]. Strict latency is often not a constraint. However, putting a neural network into the storage layer poses a unique challenge. Our goal is to reach around 5µs of inference time (as discussed in §3.3), and although the 3-layer design is fundamental to reach the goal, we made further optimizations.

Quantization. First, neuron weights are by default in floating points for improving accuracy, but it is an overkill for our purpose. Some of the major storage functionalities that define contention are striping and partitioning using `mod` operations over integers, which does not require ultra-high precision. Besides, floating point calculations are expensive and hard to manage inside the OS. Hence, we adopt DNN quantization by maintaining precision of three decimal points; the trained floating-point weights are converted to integers with precision of three decimal points. DNN quantization is a popular technique to reduce the space, power, and computation cost of DNN on mobile-platform and IoT devices, albeit some loss on accuracy [20, 32, 72]. In our case, the accuracy loss from quantization is less than 0.1%.

Co-processors. Second, using additional accelerators such as GPUs and TPUs may be possible in the future, but currently, they are optimized towards throughput and do not easily interact with host kernel code. If we move the inference to GPUs, the cross-communication would add more overhead. Furthermore, technology trends suggest that 100-200x improvement on inference latency can be foreseen in the near future with more advanced hardware [6]. This may make LinnOS faster in the future, especially as storage devices are also getting faster. However, until this technology arrives, we show that LinnOS can opportunistically use co-processors (if available) to reduce the average inference time from 6 to 4µs with 2-threaded optimized matrix multiplication using one additional CPU core.

4.6 Summary of Advantages

With all of the techniques, LinnOS delivers advantages in various dimensions, which we show in the evaluation.

- *Performance predictability*. The most important advantage is that LinnOS helps storage applications achieve predictable performance on flash arrays, outperforming other popular methods.
- *Automation*. LinnOS infers I/O operation latency by learning from millions of I/Os and automatically trains and produces neuron weights for different workloads and devices. Storage developers do not have to tweak and configure heuristics manually.
- *Generality*. To achieve predictability, LinnOS does not require device-level modification nor a heavy redesign of file systems or applications. Storage applications simply need to tag latency-critical I/Os. Failover/retry logic is already standard in many storage applications with data replicas.
- *Timeliness*. With fast inference, the application can failover as soon as the `slow` error code is returned, without the need to wait for a timeout.

[1] To clarify, different from conventional way of calculating false positive/negative, in this paper, the false submit rate is based on the submit decision and the actual resulting latency.

- *Efficiency*. With auto-revocation, LinnOS eliminates duplicate I/Os suffered in hedging. Some production systems do not use hedging for the same reason and instead use a more efficient method such as "tied requests," where clones are sent but when one of them is served, the duplicate is canceled [21]. Similar to this "clone-then-cancel" method, our "revoke-then-failover" also avoids duplicates. Furthermore, while some implementation of tied requests burdens the application layer [21], LinnOS supports I/O revocation inside the kernel.
- *Simplicity*. We do not require applications to supply an SLO value such as a deadline [14, 25, 63, 75]. I/O system calls today do not accept SLO info, arguably because setting the proper SLO is not easy [35, 46]. LinnOS simplifies this with an auto-tuned fast/slow binary classification.

4.7 Implementation Complexity

LinnOS extends Linux v5.4.8 in 2170 LOC within the block layer, mostly for the neural network model (written in C) and the simple revocation mechanism. The memory space needed for one neural network model (in total 8706 weights and biases) is 68 KB of kernel memory. LinnApp is written in 3820 LOC including data collection, analysis, labeling, training (using TensorFlow), and quantization. We make the source code public (Section A).

5 Evaluation

In this section, we first describe our evaluation setup (Section 5.1) and then present the results that answer the following important questions:

- *Stability* (Section 5.2): Is our inflection point algorithm stable enough for production workloads?
- *Latency predictability* (Section 5.3): Does LinnOS successfully deliver more predictable latencies compared to other methods?
- *Model accuracy* (Section 5.4): How accurate is the LinnOS neural network in inferring per-I/O speed?
- *Trade-offs* (Section 5.5): What are the performance and accuracy trade-offs in LinnOS?
- *Others* (Section 5.6): How does LinnOS work on other public traces? Can LinnOS support full-stack storage applications? What is the CPU overhead?

5.1 Setup

We present the evaluation workloads, devices, experiments, and methods to which we compare.

Workloads. Our ultimate goal is to evaluate whether LinnOS can help real production scenarios. We use SSD-level traces from Microsoft Azure (**AZ**), Bing Index (**BingI/BI**), Bing Select (**BingS/BS**), and Cosmos (**CO**) servers. Each server type contains I/O traces for six devices. The average trace contains 36 hours of I/O operations.[2] For training data, from each of the four server types, we pick the three busiest device traces and then pick the busiest hour (same three hours); we limit to three due to the number of (expensive) enterprise SSDs that we have (more below). For the "test data" that is dedicated for live experiments, we pick a random time slice from other busy hours, hence training and test data do *not* overlap. Overall, the training and test data do not occupy the entire available traces.

SSD devices. For performance evaluation, we show how much LinnOS helps flash arrays deliver predictable latencies. We prepared two flash arrays with consumer (**"C"**) and enterprise (**"E"**) configurations. The former connects an array of three homogeneous SM951 consumer-level SSDs, and the latter forms three *heterogeneous* enterprise-level SSDs, Intel P4600, Samsung PM1725a, and WD Ultrastar DC SN200. We assume every block is replicated three times across the devices, a typical setup for consumer-facing storage servers. For both configurations, the machine has a 2.6GHz 18-core (36-thread) Intel i9-7980XE CPU with 128GB DRAM. Unless otherwise stated, we do not use accelerators (§4.5). The overhead for failing over revoked I/Os is 15µs. For accuracy evaluation, beyond these four flash models, we also use Intel SSDSC1BG40, Intel SSDSC2BX01, Intel P3700, Intel P4510, Intel S3700, and Samsung 960 EVO, for a total of 10 models. Prior to this evaluation, all devices have been used for months with many workloads that reach the devices' full capacities, hence mimicking devices in the field.

The experiments. For performance evaluation, the experiments are performed with a storage application that executes the traces on the flash arrays, where all the devices serve read/write workloads. For example, in one experiment, the application simultaneously executes three different Azure traces on three separate SM951 devices in the consumer flash array and records the latencies of completed read I/Os. The application has a failover capability to complete revoked I/Os at other devices (as shown earlier in Figure 2). All read I/Os are marked as latency-critical.[3] We are also aware that the traces were collected on medium-end devices at Microsoft (in 2016). Hence, for our high-end flash array configuration, we have to mimic a heavier workload by re-rating the traces to be more intensive. Our methodology is that for each re-rated trace, the resulting baseline latency distribution (after running it on the high-end device) should be similar to the latency distribution in the original trace.

Table 1 shows the I/O characteristics of the re-rated traces. Typically, running these re-rated traces on our drives shows a low slack (<5% of all I/Os), where SSDs see no pend-

[2] The traces are available from Microsoft to the academic community with NDA.

[3] We assume that no write I/Os are latency critical as they are usually absorbed by write buffers. However, if needed, our techniques can be easily integrated with kernels/applications that support write re-routing (*e.g.*, AutoRAID, RAID+).

Test Trace	Avg IOPS	Max IOPS	R:W Ratio	Avg I/O Size Read/Write	Max I/O Size Read/Write
AZ/C	745	4.9K	27:73	24K/18K	64K/64K
BI/C	361	1.8K	17:83	57K/30K	64K/1M
BS/C	114	1.1K	22:78	163K/73K	2M/9M
CO/C	113	623	32:68	479K/121K	6M/32M
AZ/E	13K	31K	25:75	25K/17K	64K/64K
BI/E	2.4K	9.2K	23:77	55K/30K	512K/1M
BS/E	1.3K	4.3K	27:73	196K/73K	2M/9M
CO/E	2.5K	7.2K	22:78	430K/107K	7M/32M

Table 1: **I/O characteristics of re-rated traces (§5.1).** *The upper part (first four rows) is for the consumer-level flash array and the lower is for the enterprise-level one. Every max-IOPS value is measured within a 10-second window.*

ing I/Os for a few milliseconds, and noticeable burstiness (5-30%), where I/Os need to wait in the OS as the SSD queues are full. We believe this accurately emulates the slack and tail behaviors seen in real deployments. Also, the workload bursts across devices are highly correlated, which, in some cases, can cause inevitable long-tail behaviors that no failover can handle. However, in real runs we find that the internal busyness of the devices is not necessarily correlated due to device-level complexities, as LinnOS shows great improvement by evading the underlying device idiosyncrasies (§5.3). All the experiments are repeated three times, and no significant variance was observed.

Methods compared. We perform an extensive evaluation that compares eight methods: baseline, cloning, constant-percentile hedging (*e.g.*, at p95 latency), inflection-point hedging (with our algorithm), simple heuristic, advanced heuristic, LinnOS (by itself), and LinnOS with high-percentile hedging. Comparing LinnOS with white-box approaches [25, 30, 44, 56] is out of the scope of the paper because LinnOS targets black-box devices and we do not have access to an array of programmable devices.

5.2 Inflection Point (IP) Stability

One of the contributions in this paper is finding the semi-optimal fast/slow inflection point (IP) that brings a balance between timeliness and overhead (Figure 5 in Section 4.2). Table 2 shows the IP values our algorithm computed for every workload-device pair. The three numbers in every cell represent three different traces (from the same server type), each running on one of the SSDs in the flash array. As shown, the IP values widely range from p72 to p98, which highlights why a constant timeout value is *not* optimal and hurts performance. These IP values will be used for fast/slow labeling and training, which then generates a unique set of weights for each device.

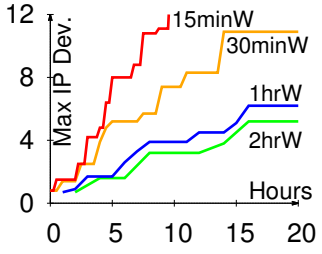

Figure 7: **IP stability.**

	Consumer	Enterprise
Azure	p73.3, p77.0, p91.4	p91.0, p93.2, p97.8
BingIndex	p80.0, p94.5, p98.5	p80.1, p83.3, p97.0
BingSelect	p72.0, p76.9, p87.2	p75.3, p83.7, p86.8
Cosmos	p73.4, p82.5, p84.1	p83.2, p84.8, p95.1

Table 2: **Inflection point (IP) settings.** *This table, as explained in Section 5.2, shows the IP values that our algorithm in Section 4.2.1 computed for every workload-device pair.*

We chose a busy hour (T=`1hr Window`) to collect the training data and calculate the IP values in that time slice. Figure 7 shows the stability of our methodology by plotting the max IP deviations in percentile (y-axis) within the next 20 hours (x-axis) for various T window values. For example, if the chosen hour exhibits p85 IP, but a subsequent hour exhibits p75 or p95 IP, then the deviation is 10 percentiles (y=10). The graph shows that if T=`1hrWindow`, the deviation is bounded within five percentiles in the next 15 hours, indicating that frequent retraining is unnecessary. If T is shorter (*e.g.*, `15minWindow`), the deviation is more apparent (needs frequent retraining, which typically converges within 15-20 minutes on CPUs, due to LinnOS's light model). If T is larger (`2hrWindow`), the gain is not significant. For generality, the figure is the result of our algorithm simulation on all the datasets (36 hours per trace, 24 traces, four server types).

The cost of delayed retraining depends on the deviation. Let us take an example of a model trained for p95 (5ms), but then the workload deviates such that the real IP is at p90 (10ms) because the workload becomes more write-intense. In this case, LinnOS (still using 5ms) will over-revoke many IOs that could have finished before 10ms (more false revokes). If the failover overhead is negligible, this will not cause much harm. Another scenario is when the workload deviates such that the IP moves up to p99 (3ms). Here, LinnOS would over-accept (more false submits) because 3-5ms latency is inaccurately considered "fast," but actually can be made faster. This is where LinnOS without retraining hurts.

5.3 Latency Predictability

We now evaluate LinnOS's success in achieving extreme latency predictability. Figure 8 shows the average I/O latencies (user-perceived) on the two flash arrays (consumer and enterprise) across the eight methods. In more detail, Figure 9 shows the latencies at specific percentiles (p80 to p99.99 in the x-axis). Below we dissect the strengths and weaknesses of every method. We start from the baseline, then we jump to "`LinnOS+HL`" (the best outcome), followed by the others.

Baseline. The `Base` lines in Figure 9 confirm unpredictability of flash storage with latencies that spike almost exponentially in between p95 to p99.99, increasing the average latencies to 1.3–6.5 times compared to our best cases

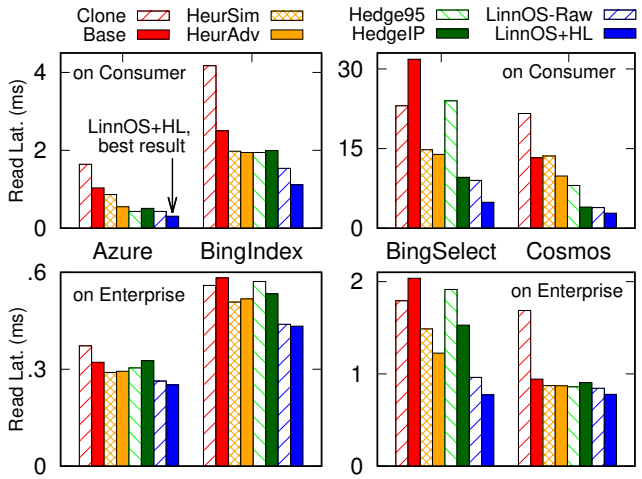

Figure 8: **Average latencies.** *The figures show that LinnOS consistently outperforms all other methods, as explained in Section 5.3. The top and bottom graphs represent experiments on the consumer and enterprise arrays, respectively.*

(Figure 8). Clearly, flash arrays with data redundancy should adopt tail-cutting methods to achieve higher predictability.

LinnOS+HL. This label represents the LinnOS method combined with high-percentile hedging for masking the small inaccuracy that is intrinsically hard to eliminate in a neural network (Section 4.4). That is, to compensate for the inaccuracies that cause false submits, our application sends a duplicate I/O after pX latency time has elapsed, where X is the smaller of 95 and (1 – false submit rate)×100. We use the false submit rates from the training process (Figure 10 in Section 5.4).

[Key outcome] → The average latencies in Figure 8 show that `LinnOS+HL` *consistently outperforms all other methods across different workloads and platforms.* On average, `LinnOS+HL` reduces latency by 9.6-79.6% compared to p95 hedging (`Hedge95`), 14.2-49.5% to hedging with our IP algorithm (`HedgeIP`), and 10.7-71.2% to an advanced heuristic (`HeurAdv`). These speed-ups are a product of the stable latencies; *in Figure 9,* `LinnOS+HL` *lines exhibit stable latencies even at extremely high percentiles, p99 to 99.99.* These results bring a positive conclusion that the downsides of LinnOS (a 15µs failover overhead including a 6µs per-I/O inference cost and the inaccuracies) are outweighed by its effectiveness in delivering predictable latencies.

LinnOS (Raw). Here we show LinnOS efficiency even without hedging (*i.e.*, revoke+failover without I/O duplication). The `LinnOS-Raw` bars in Figure 8 shows that LinnOS by itself is effective enough, only 1.3-45.7% worse than `LinnOS+HL`, and compared to p95 hedging, `LinnOS-Raw` reduces latency by 0.3-62.3%, and to an advanced heuristic, by 3.0-60.7%. Figure 9 details why adding hedging is useful. At high percentiles, above p99, `LinnOS-Raw` starts exhibiting high latencies (due to false submits). Learning from the "small-tail" behavior of hedging (*e.g.*, the `Hedge95` lines), we combined the best of the two in `LinnOS+HL`.

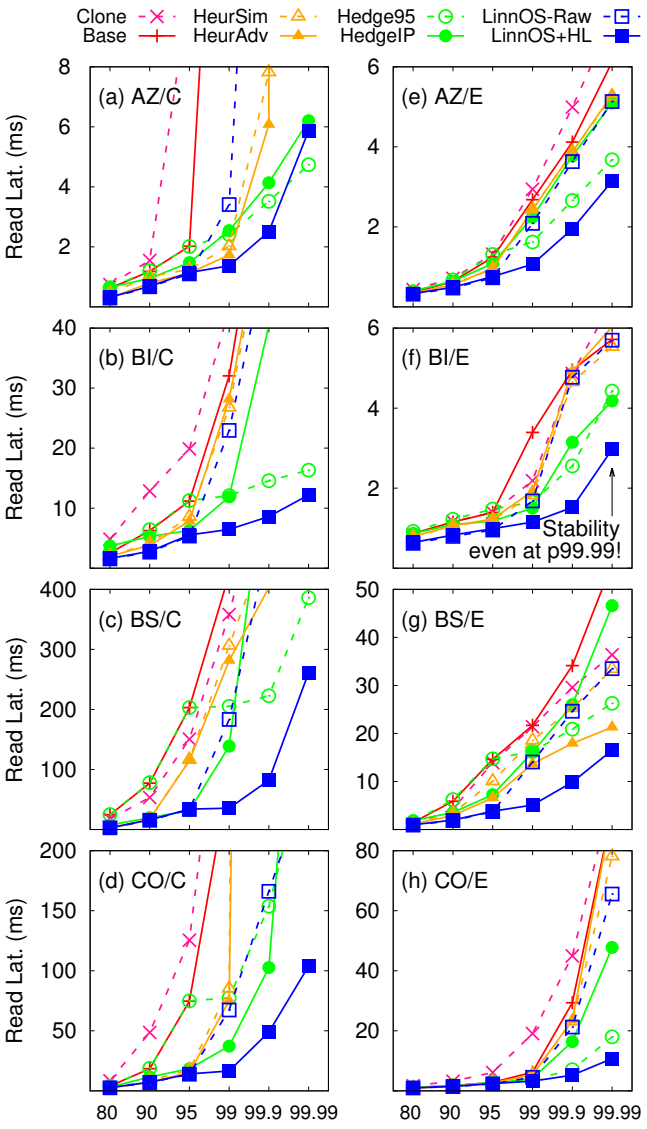

Figure 9: **Percentile latencies.** *Explained in Section 5.3, the figures show that LinnOS +HL delivers the most predictable latencies (y-axis) across all percentiles (x-axis), even at p99.99. In Figure (a), "AZ/C" means Azure running on consumer array.*

Hedging at p95. Sending a duplicate I/O after a p95-latency timeout has elapsed is a popular method used in the field [12, 21]. Figure 9 shows that, in general, this method is effective in cutting latency tail but generally incurs *higher* latencies than `LinnOS+HL`. This is because `Hedge95` needs to *wait* for the timeout to happen before sending the duplicate I/Os, while LinnOS returns a timely revocation that allows the application to failover quickly. As the implication, `Hedge95`, *on average, is slower* than `LinnOS+HL` or even

LinnOS-Raw (Figure 8).

Hedging at IP. Many of the IP values shown in Table 2 are below p95, which raises the question of whether hedging at IP would be better than at p95. The average values in Figure 8 show a mixed result. On the consumer devices, HedgeIP improves upon Hedge95 by 2x for heavy workloads BingS and Cosmos, but loses by up to 15% in light workloads Azure and BingI. Similarly, on enterprise devices, HedgeIP wins in BingS while slightly losing in the others. Upon further investigation, we see that, for example, in consumer devices, Azure and BingI latencies are generally fast (<2 and 10ms respectively, as shown by the y-axis in Figure 9a-b), hence are *sensitive* to the extra load from duplicate I/Os; HedgeIP in our experiments are sending more duplicates than Hedge95. Nevertheless, our experiments show that for most of the workloads, HedgeIP is more effective than Hedge95, hence systems with hedging can adopt our IP algorithm.

Simple heuristic. The first heuristic we wrote, "HeurSim," is based on a popular heuristic for spinning disks: if the device queue length (the number of outstanding I/Os) is larger than a threshold, the incoming I/O should be retried elsewhere [13, 62, 64]. For the threshold, we use a similar method as HedgeIP, but instead of using IP latency value, we use IP queue length. That is, we first profile the queue length distribution during tracing and then select the queue length at the IP percentile as the threshold for revoking. Figure 8 shows that HeurSim only gives a small improvement over the baseline and is far from the best case. In short, it is not smart enough to infer device-internal disruptions.

Advanced heuristic. We extend HeurSim to a more "advanced" heuristic, HeurAdv. For comparison fairness, we reuse the same intuition we had in building LinnOS and apply it to HeurAdv. An additional task that HeurAdv performs is scanning the last N completed I/Os (N=4, same as in LinnOS) and if this history shows a slow I/O ("slow" as defined in §4.2) but with a low queue length (less than the median), it will mark the drive as "internally busy." In this state, incoming I/Os will not be admitted unless the queue length drops to a low value (less than the lower-quartile queue length). The state will not be changed from "busy" to "normal" until it sees recent I/Os become fast ("fast" as defined in §4.2).

[2nd key outcome] → Figure 8 shows that HeurAdv improves upon HeurSim in most cases, but still loses from other methods. We would like to note that we spent several weeks tuning the heuristic to the "best" outcome we can achieve. Continued expansion and tuning of the heuristic is possible. However, the main difficulty that will arise is the *large design space* of parameters (normal/busy states, median and lower-quartile queue lengths, etc.) that must be optimally and *manually configured* for different workloads and devices. This is where we show that machine learning helps. The use of a lightweight neural network allows us to focus on deciding what features matter, but at the same time letting the model learn and reverse-engineer SSD behaviors. In our case, LinnOS neural network *auto-trains all the 8706 weights* for different devices and workloads.

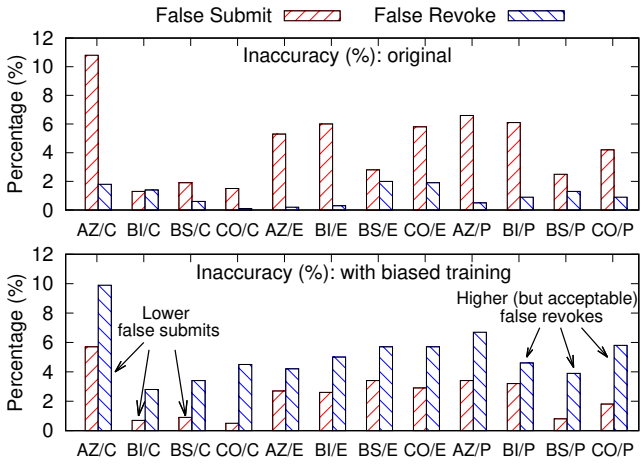

Figure 10: **Low inaccuracy.** *The figure shows the percentage of false submits and false revokes. Note that only false submits really matter (see Section 5.4). Additionally,* **"P"** *represents other device models that we can access from a public cloud. For graph readability, here for "P" we only show the results for one device model, while the observations stand across the rest. In total, the accuracy evaluation covers 10 device models (1C+3E+6P).*

Cloning. This method is essentially p00 hedging, sending a duplicate I/O for every I/O on the outset. Although SSDs are fast and have internal parallelism, Figure 8 shows that Clone is mostly worse than the baseline due to the 2x load.

5.4 (Low) Inaccuracy

We now measure LinnOS inaccuracy by counting the number of false submits and false revokes (Section 4.4). The live experiments can only measure the former but not the latter. This is because revoked I/Os are never submitted to the device, hence we never know whether the revoke is accurate or not. Thus, for this evaluation, we measure inaccuracy in an offline way using TensorFlow, just like the training phase. However, note that both the training and test data were collected from running the workloads on real flash arrays (*i.e.*, not simulated data). Just like before, we use 1-hour data sets for training and then pick three different 1-hour data sets for testing accuracy, and measure the average inaccuracy.

Figure 10 shows the inaccuracies before and after we use biased training. To recap Section 4.4, false submits are more dangerous than false revokes. Without bias, the top graph shows that the false submit rates (red bars) are high, between 1.3% to 10.8%. With biased training, as shown in the bottom graph, we successfully lower the false submit rates to 0.7-5.7%, by *shifting* the inaccuracies to false revokes, which are more tolerable as explained in Section 4.4. For example, let us assume an inferior scenario of p80 inflection point (*i.e.*,

Model:	A	B	C	D	E
Acc. (%)	−(3-12)	−(1-4)	+(1-2)	+(4-5)	+(8-12)
Perf. (µs)	−4	−1	+40	+94	+1670

Table 3: **Trade-offs balance.** *This table is explained Section 5.5. All the +/− of accuracy and performance values are compared to our final neural network model described in §4.*

20% slowness), which means the probability that all three replicas are slow is 0.008 ($(^{20}/_{100})^3$). Thus, although we have spiked up the false revokes to 2.8-9.7% in Figure 10b, *only 0.008* of these false revokes probabilistically will result in slow I/Os. Finally, as mentioned before, for masking the dangerous low inaccuracy (the 0.7-5.7% false submits), combining LinnOS with high-percentile hedging (`LinnOS+HL`) led to a powerful result.

5.5 Trade-offs

Table 3 shows some possible trade-offs between inference overhead and accuracy (models A-E, with accuracy involving both false submits and false revokes). On one hand, if lower overhead is preferred and some accuracy loss is acceptable, then one option is to trim the input features and the model. For example, in model B with R=3 (*i.e.*, including fewer history I/Os instead of R=4) can reduce the number of input features from 31 to 24 and lower inference overhead, −1µs, but it will bring some accuracy loss, −(1-4%), due to fewer inputs. Or, if even lower overhead is favorable, then in model A we can further cut the input features (R=2, 17 features) and use a slimmer hidden layer (from 256 neurons to 128), resulting in a lower inference time,−4µs, while bringing larger accuracy loss, −(3-12%).

If higher accuracy is needed, then we can bring in more features and heavier models. For example, in model C, by adding one more hidden layer to the model, we can gain +(1-2%) higher accuracy, while the inference overhead rises by +40µs. Taking a step further, we can involve more features (up to R=10 and 73 features) and more hidden layers (three layers with 256-512-256 neurons) to push the accuracy gain by +(4-5%), but an increased overhead, +94µs. The extreme model E includes block offsets in the input features (2048 features in total) and applies a model with five hidden layers (with 512 neurons each). For some traces, this model improves the accuracy by +(8-12)%, but its inference overhead, +1670µs, is extremely high for live inference.

5.6 Other Evaluations

5.6.1 Additional Performance Evaluations

Other possible manually-tuned heuristics. To get a sense of how much performance a heuristic can ultimately reach, we pick several 10-min slices from the traces and manually tweak the adjustable parameters of `HeurSim` and `HeurAdv` with various thresholds until an optimal outcome is achieved. In a nutshell, we start with the generic `HeurSim` and `HeurAdv`, evaluate them with the sliced traces, track the high-latency I/Os that are not revoked, update the thresholds to catch these I/Os without causing too many false revokes (*e.g.*, >15%), re-evaluate and repeat the entire process until an approximate optimum is observed. This approach is indeed capable of granting heuristics a further stretch. For example, we see a few cases where tweaked heuristics can outperform `LinnOS-Raw` by up to 20% at p95. However, this tuning procedure is onerous and impractical in real runs as the repeated manual tweaking is too slow to catch up with the fluctuation of incoming workloads.

LinnOS+H99. We also try `LinnOS+H99`, which employs p99 hedging that only generates 1% extra I/Os. Figure 11a shows one of its comparisons with `LinnOS+HL`. Generally, `LinnOS+H99` encounters a larger tail area due to longer waiting, but responds faster at lower percentiles due to less extra I/Os. With that, sometimes `LinnOS+HL` can show slightly worse average latencies (up to 3%) than `LinnOS+H99` (Figure 11b). However, in a large majority of our benchmarks, `LinnOS+HL` achieves 1.7-39.2% better average latencies than `LinnOS+H99`.

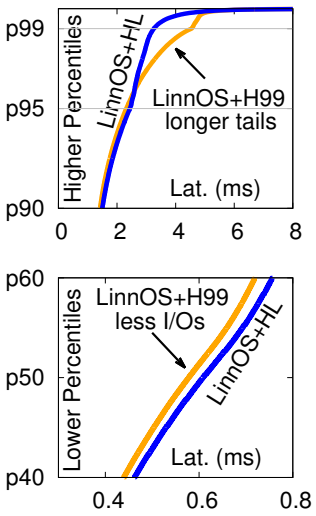

Figure 11: **LinnOS+H99.**

5.6.2 On Public Traces

Beyond our evaluation with Microsoft traces, Figure 12 shows a quick evaluation with the latest SSD traces published on the SNIA website [9] run on our consumer flash array. The result confirms that LinnOS also exhibits low inaccuracy (Figure 12a) and substantial latency improvement (Figure 12b).

5.6.3 MongoDB on Different Filesystems

To see how data applications can benefit from LinnOS, we set up a local MongoDB replica set on top of our three enterprise drives with homogeneous filesystem settings. For each type of filesystem, MongoDB receives 120K random read requests, and all drives run Microsoft traces as background noise when serving MongoDB requests as latency-critical I/Os. Here, we focus on high-percentile latency (*e.g.*, p99 latency) since the average latency is largely impacted by filesystem buffering, while the tail latency reflects the raw performance from the devices.

Figure 13 shows that with LinnOS, MongoDB achieves

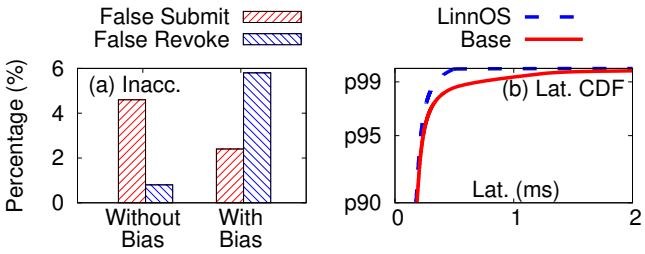

Figure 12: **On public traces.** *As explained in §5.6.2.*

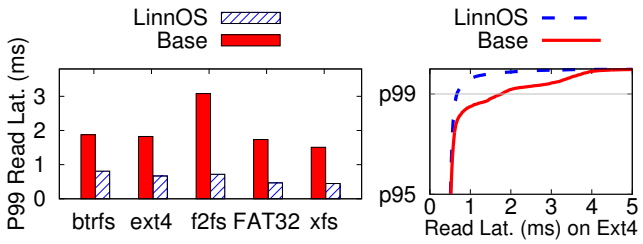

Figure 13: **MongoDB on different filesystems.** *This figure shows that LinnOS can easily help data applications achieve more predictable latency (§5.6.3).*

much more predictable performance. For example, with all underlying devices formatted with f2fs, LinnOS reduces the p99 latency by 76.7%. Moreover, LinnOS only requires minor changes to MongoDB and filesystems: 50 additional LOC. For example, the filesystems should directly return LinnOS's error code to the applications instead of conducting unnecessary self-checking, and MongoDB needs to be slightly modified to reuse its built-in failover logic.

5.6.4 Computation Overhead/Optimization

CPU overhead. A reasonable concern is that if the entire OS has many neural networks, then it will be CPU-intensive. Across all the benchmarks and SSDs, paired with a lightweight neural network, each device only costs 0.3-0.7% of the host CPU resource, making LinnOS practical for large-scale deployments.

Co-processors for acceleration. As mentioned in Section 4.5, additional processors can be utilized to speed up the inference. By utilizing one more CPU core, LinnOS can reduce the inference overhead by 36% (to 4µs), with the maximal CPU usage increased up to 1.4% per device.

6 Conclusion and Discussion

We have presented LinnOS, to the best of our knowledge, the first operating system capable of inferring the speed of every I/O to flash storage. We have shown the feasibility of using a light neural network in the operating system for making frequent, fine-grained, black-box live inferences. LinnOS outperforms many other methods and successfully brings predictability on unpredictable flash storage. We also believe that LinnOS's success leads to exciting discussions and questions that can spur future work:

On performance. Though LinnOS inference overhead (4-6µs) is less noticeable compared with the access latency of current SSDs (*e.g.*, 80µs), it could become problematic as SSDs march to 10µs latency range. Also, the consumption of computation resources can increase substantially as the IOPS grow. How to further lower the inference cost (*e.g.*, to 1µs) to support faster devices and higher throughput? Can advanced accelerators help accelerate OS kernel operations? Can near-storage/data processing help? Can we skip the inference when the outcome is highly assured (*e.g.*, the queue length is very low)? Can we cache the approximation results for popular predictions?

On masking the inaccuracy of machine learning. As machine learning (*e.g.*, LinnOS) can never achieve 100% accuracy, how should "ML-for-system" solutions mask the cases that machine learning fails to catch, while still benefiting from its generality? Is marrying learning and heuristic (*e.g.*, as in `LinnOS+HL`) a powerful option that exploits the advantages of both worlds?

On other integrations and extensions. One interesting question raised by LinnOS is why the latency behavior of SSDs—devices with complex idiosyncrasies—can be learned by the block layer with a few observable features. Understanding this can help other higher layers such as RAID, direct device access (SPDK), user/device-level filesystems, or distributed storage adopt our concept. Likewise, in lower layers, it is also a possibility in the future to have SSDs with latency inference capability built in. Although, arguably, one can say that the device already has full knowledge of its internals and does not need a black-box prediction, an argument can be made that SSD vendors can use the same machine learning method across different internal architectures. Hence, they do not need to re-develop the inference logic every time they modify the internal hardware, logic, and policies. Alternatively, SSD vendors can employ "gray-box" learning that incorporates some of the internal knowledge.

On precision. Can fast/slow inference be converted to a more precise latency inference, such as latency ranges (*e.g.*, 2-4µs, 4-8µs, ...), percentile buckets (*e.g.*, p0-p10, ..., p90-p100), or precise latency with high accuracy? Can model permutation or other machine learning techniques help?

7 Acknowledgments

We thank Tom Anderson, our shepherd, and the anonymous reviewers for their tremendous feedback and helpful comments. We also thank the Azure CSI group for providing the traces. This material was supported by funding from NSF (grant Nos. CCF-2028427, CNS-1405959, CNS-1526304, CNS-1764039, and CNS-1823032), ARO (W911NF1920321), DOE (DESC00141950003) and UChicago CERES Center, as well as generous donations from Dell EMC, Google, and NetApp.

References

[1] Cassandra - Speculative Execution for Reads / Eager Retries. https://issues.apache.org/jira/browse/CASSANDRA-4705.

[2] Chameleon. https://www.chameleoncloud.org.

[3] Ethernet: The High Bandwidth Low-Latency Data Center Switching Fabric. http://www.force10networks.com/whitepapers/pdf/F10_wp_Ethernet.pdf.

[4] GreyBeards on Storage. https://silvertonconsulting.com/gbos2/tag/tail-latency/.

[5] MongoDB - Basic Support for Operation Hedging in NetworkInterfaceTL. https://jira.mongodb.org/browse/SERVER-45432.

[6] New GraphCore IPU BenchMarks. https://www.graphcore.ai/posts/new-graphcore-ipu-benchmarks.

[7] Pareto Distribution. https://en.wikipedia.org/wiki/Pareto_distribution.

[8] Rapid Read Protection in Cassandra 2.0.2. https://www.datastax.com/blog/2013/10/rapid-read-protection-cassandra-202.

[9] SNIA I/O Trace Data Files. http://iotta.snia.org/traces.

[10] The Data Center Flash Storage Market Is Expected to Grow at a CAGR of Nearly About 17% during 2018-2024. https://prn.to/2z58q4L.

[11] The Evolution of Image Classification Explained. https://stanford.edu/~shervine/blog/evolution-image-classification-explained.

[12] Tuning Speculative Retries to Fight Latency. https://www.youtube.com/watch?v=uRJSuQofJWQ, 2016.

[13] Irfan Ahmad. Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2007.

[14] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[15] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the Performance Variation in Modern Storage Stacks. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[16] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[17] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[18] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[19] 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 ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[20] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[21] Jeffrey Dean and Luiz Andre Barroso. The Tail at Scale. *Communications of the ACM (CACM)*, 56(2), 2013.

[22] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[23] 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)*, 2014.

[24] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, P. Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-Learning Congestion Control. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[25] 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 ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[26] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S.

Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[27] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the 2017 EuroSys Conference (EuroSys)*, 2017.

[28] Henry Hoffmann. JouleGuard: energy guarantees for approximate applications. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[29] Chin-Jung Hsu, Rajesh K Panta, Moo-Ryong Ra, and Vincent W. Freeh. Inside-Out: Reliable Performance Prediction for Distributed Storage Systems in the Cloud. In *The 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016.

[30] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[31] Amber Huffman. Addressing IO Determinism Challenges at Scale with NVM Express Part 2: Renegotiating the Host/Device Contract. In *Proceedings of the 2017 Non-Volatile Memory Workshop (NVMW)*, 2017.

[32] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[33] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A Flexible, High-Performance Key-Value SSD. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA-23)*, 2017.

[34] Myoungsoo Jung, Wonil Choi, Miryeong Kwon, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Kandemir. Design of a Host Interface Logic for GC-Free SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 8(1), May 2019.

[35] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[36] Bryan S. Kim, Hyun Suk Yang, and Sang Lyul Min. AutoSSD: an Autonomic SSD Architecture. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[37] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[38] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *Proceedings of the 17th USENIX Symposium on File and Storage Technologies (FAST)*, 2019.

[39] Youngjae Kim, Junghee Lee, Sarp Oral, David A. Dillow, Feiyi Wang, and Galen M. Shipman. Coordinating Garbage Collection for Arrays of Solid-State Drives. *IEEE Transactions on Computers (TC)*, 63(4), April 2014.

[40] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-state Drives. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[41] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[42] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[43] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[44] Sungjin Lee, Ming Liu, SangWoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed Flash. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[45] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[46] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the 2016 EuroSys Conference (EuroSys)*, 2016.

[47] Chun-Yi Liu, Jagadish B. Kotra, Myoungsoo Jung, Mahmut T. Kandemir, and Chita R. Das. SOML Read: Rethinking the Read Operation Granularity of 3D NAND SSDs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[48] 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 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[49] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[50] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. Hiding the Microsecond-Scale Latency of Storage-Class Memories with Duplexity. In *Proceedings of the 2019 Non-Volatile Memory Workshop (NVMW)*, 2019.

[51] Nikita Mishra, Connor Imes, John D. Laferty, and Henry Hoffmann. CALOREE: Learning Control for Predictable Latency and Low Energy. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[52] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. ESP: A Machine Learning Approach to Predicting Application Interference. In *The 14th International Conference on Autonomic Computing (ICAC)*, 2017.

[53] Pulkit A. Misra, Mara F. Borge, Iigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the 2019 EuroSys Conference (EuroSys)*, 2019.

[54] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[55] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[56] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage System. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[57] Chris Petersen. Addressing IO Determinism Challenges at Scale with NVM Express. In *Proceedings of the 2017 Non-Volatile Memory Workshop (NVMW)*, 2017.

[58] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[59] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[60] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for OS-level power management. In *Proceedings of the 2009 EuroSys Conference (EuroSys)*, 2009.

[61] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gmez-Luna, and Onur Mutlu. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[62] Toby J. Teorey and Tad B. Pinkerton. A Comparative Analysis of Disk Scheduling Policies. *Communications of the ACM (CACM)*, 15(3), 1972.

[63] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2012.

[64] Elizabeth Varki, Arif Merchant, Jianzhang Xu, and Xiaozhou Qiu. Issues and Challenges in the Performance Analysis of Real Disk Arrays. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(6), 2004.

[65] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[66] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage Device Performance Prediction with CART Models. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2004.

[67] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and Inference with Integers in Deep Neural Networks. In *6th International Conference on Learning Representations (ICLR)*, 2018.

[68] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. Overcome the GC-induced Performance Variability in SSD-based RAIDs with Request Redirection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38(5), May 2019.

[69] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and Bo Mao. GC-aware Request Steering with Improved Performance and Reliability for SSD-based RAIDs. In *Proceedings of the 32th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[70] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Andy Rudoff, and Steven Swanson. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[71] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[72] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. BMXNet: An Open-Source Binary Neural Network Implementation Based on MXNet. In *Proceedings of the 2017 ACM on Multimedia Conference (ACMMM)*, 2017.

[73] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[74] Li Yin, Sandeep Uttamchandani, and Randy Katz. An Empirical Exploration of Black-Box Performance Models for Storage Systems. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2006.

[75] Hong Zhang, Kai Chen, Wei Bai, Dongsu Han, Chen Tian, Hao Wang, Haibing Guan, and Ming Zhang. Guaranteeing Deadlines for Inter-Datacenter Transfers. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[76] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[77] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

A Artifact Appendix

A.1 Abstract

We assemble an executable LinnOS workflow that runs on Chameleon Cloud Research Platform [2]. This self-contained artifact contains the major components and step-by-step instructions.

A.2 Artifact check-list

- **Program:** LinnOS with preprocess scripts [4].
- **Data set:** Example I/O traces.
- **Run-time environment:** Chameleon's shared Jupyter experiment environment.
- **Hardware:** A flash array with at least three SSDs.
- **Output:** Trained models for I/O prediction and latency CDF lines.
- **Experiments:** LinnOS workflow.
- **Expected experiment run time:** Several hours.
- **Public link:** https://www.chameleoncloud.org/experiment/share/15?s=409ab137f20e4cd38ae3dd4e0d4bfa7c

A.3 Description

A.3.1 How to access

Access the public link provided above and click the "Launch on Chameleon" botton (account required to access Chameleon resources), then see Readme.txt for a high-level description and LinnOS.ipynb for step-to-step instructions.

A.3.2 Hardware dependencies

Evaluating LinnOS requires a flash array with at least three SSDs, which are provided by the storage-hierarchy instances from Chameleon Testbed.

A.3.3 Data sets

The artifact contains some example I/O traces, which are used in the workflow for testing purposes.

A.4 Installation

Step-by-step installation instructions are available in the artifact.

A.5 Evaluation and expected result

Upon successful running, the workflow should produce a trained model, the accuracy outcome, and the I/O latency distribution of LinnOS and baseline. Please see readme.txt in the artifact for further details.

[4] Excluding the data collection and analysis code that may reveal sensitive information in Microsoft traces