IONET: Towards an Open Machine Learning Training Ground for I/O Performance Prediction

Daniar H. Kurniawan, Levent Toksoz, Mingzhe Hao, Anirudh Badam[‡], Tim Emami[*], Sandeep Madireddy[†], Robert B. Ross[†], Henry Hoffmann, Haryadi S. Gunawi,

University of Chicago [‡]Microsoft Research [*]NetApp [†]Argonne National Laboratory

Abstract

Low and stable latency is a critical key to the success of many services, but variable load and resource sharing in a modern cloud environment introduces resource contention that in turn increases the unpredictability of the systems which often cause a "tail latency problem." As one of the main building-blocks of a complex request-chain, understanding the I/O request becomes an important topic to help parallel storage applications achieve performance predictability and to reduce the tail latency. This paper presents IONET, ML-based per-I/O latency predictor capable of achieving 80-97% inference accuracy and sub-10µs inference overhead for each I/O. IONET's light-weight NN models demonstrate that this line of research is practical and incorporating the models inside operating systems for real-time decision-making is a feasible solution to achieve latency stable systems.

1 Introduction

Today, our modern cloud environment has become more complex than ever which then introduces various degree of performance instability at scale. Interactive services (such as search engine, email, social media, etc.) get affected the most by this instability. Thus, improving the performance predictability is becoming an important step towards developing a better aforementioned systems. Moreover, since the I/O request is the main building-blocks of any services, improving the predictability of I/O latency is becoming even more important. Moreover, faster and faster SSDs are available and becoming a dominant factor in the storage market [6], but the internal complexity continues to grow; thus, I/O latency predictability still becomes an open problem to this day.

With the growing innovation in NAND technology led by private companies, modern flash devices are lacking of transparency when it comes to managing its internal resources. The background operations such as garbage collection, buffer flushing, wear leveling, and read repairs, are the kinds of operations that pose a threat to latency predictability [9, 17, 20, 21, 43, 45, 63, 64], which is becoming a problem faced by many storage industries in recent years [3, 22, 44, 50]. Furthermore, with a report that flash devices contribute to more than 19% of the total response time for some online applications [64], more solutions should be explored.

There have been substantial amount of research being done to mitigate the unpredictable latency at the SSD. "White-box" approaches—that re-architect device internals [11, 24, 25, 28, 42, 56, 60, 63]—are powerful, but less likely to be widely adopted by the SSD vendors. Then there is the "gray-box" method which suggested partial device-level modification combined with OS or application-level changes [35–37, 53, 64, 65]. However, that solution depends on the vendors' willingness to modify the device interface. Finally, there is "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 [12, 30, 38–40, 46, 54, 61, 62], while some others simply use speculative execution [2, 4] which will generate extra I/Os and extra latency due to the waiting time. Among all the approaches above, arguably, the most popular solution is speculative execution given its simplicity and capability to mitigate *every* slow I/O [2, 4, 5, 14].

In general, the problem of I/O performance instability at scale raises many questions. How will a new workload perform on a large cluster of storage devices? How will the workload performance change if the cluster deploys heterogeneous storage devices with complex tiering/caching (e.g., mixes of persistent memory, SSDs, and disks)? How does scale affect performance variability? Finding answers is more difficult today as storage devices are adding increasing internal complexity. Most disks and SSDs are black-box devices with a vast number of different models and internal policies invisible to higher-level systems.

Many works extract specific internal characteristics of disks [13, 16, 51, 52, 55, 59] or SSDs [8, 10, 15, 29, 31, 32, 34, 41] and further utilize them with analytic methods. These works raise valuable ideas, but they stop short in answering the questions above. Another line of work is the use of machine learning methods to predict workload performance, [19, 26, 27, 33, 47, 48, 57], but they do so at a coarse-grained, aggregate level (e.g., minute- or hourly-level performance). We believe that achieving useful performance predictions requires a much more fine-grained prediction and modeling. That is, we would like to intervene on a request-by-request basis and improve performance predictability not just in aggregate, but for each individual request. With the vast amount of I/O-level datasets that can be collected today and the advances in machine learning methods and platforms, we take up this challenge.

To this end, we introduce IONET, ML-based per-I/O latency predictor capable of achieving 80-97% inference accuracy and sub-10µs inference overhead for each I/O. With IONET framework, we can build models of storage devices in such a way that we can predict the latency of every I/O of a full-workload running on a target storage cluster without actually running it on the cluster. The are two biggest challenges for IONET project. First, how to provide the proof of concept that ML-based per-IO latency prediction is feasible? Second, how to get a vast collection of real-production I/O traces? We collected the traces from our industry partners under various NDAs. To the best of our knowledge, IONET is the first one to introduce a framework for training I/O latency predictor.

IONET introduces three technical contributions. First, IONET provides a proof of concept that per-IO latency prediction using machine learning techniques is feasible. Second, IONET's framework empowers new type of research in the storage community such as "I/O performance prediction". Third, IONET's evaluation provides a complete analysis of the training result that give many insights about how the models behave across different traces and devices, and also how to improve the model's accuracy. By doing so, we believe we can expand the storage community to include modeling and theoretical people. Overall, we show that it is plausible to adopt machine learning methods for operating systems to learn black-box devices.

The rest of the paper is organized as follows: We first show the vision of IONET project (Section §2). In Section §3 we explain the worklflow of IONET project. Next, in Section §4 we write the details of our dataset. Then, in Section §5 and §6, we explain the devices used to replay the traces and the design of 4 sample models used in this project. Following that, Section §8 explains our detailed evaluation. Finally, we conclude with many interesting discussions at Section §9 and §10 to explore in the future.

2 The Vision: The IONet Project

With machine learning methods, IONET project will enables us to build models of storage devices in such as way that we can predict the latency of every I/O of a full workload running on a target storage cluster without running it on the cluster. IONET achieves this by acquiring a vast collection of training and test datasets, that we will build, maintain,

ML methods	Accuracy
Logistic regression	49-51%
Decision tree	45-69%
Random forest	16-84%
Linear, dense DNN	63-78%
A custom IONET DNN	93-99%

Table 1: **Initial Benchmark of various ML-methods.** *Among various machine learning techniques, the use of neural networks seems to be very promising with its high accuracy.*

and publish the datasets to support this new research area. With the current datasets, IONET is able to predict the per-I/O latency on individual and heterogeneous SSDs.

I/O LATENCY DATASETS (BENCHMARKS): "Benchmarks shape a field" [49]. In the ML/visualization community, the large ImageNet benchmarks have spurred research in image recognition. Similarly, we would like to provide benchmarks for fostering storage research in ML-based per-IO latency prediction. In the storage community, workload benchmarks are available in the form of traces [7, 27, 57], but to evaluate whether a predictor is accurate, one must run the traces through the predictor on many varieties of storage devices, which not all researchers have access to. Thus, we will assemble a large benchmark for evaluating prediction models (and make it publicly accessible) in the following way. First, in terms of workloads, we will use the traces acquired from our partner institutions; we name these traces $T_i = 1..I$. Second, with help from our industry partners, we have a collection of more than 20 heterogeneous SSD devices (and plan to collect more); we label them $S_j = 1..J$. For each combination of trace and device, we can evaluate latencies. Thus, there will be $I \times J$ latency files (potentially >500 files) that can be used to train and test per-IO latency predictors. This way ML/storage researchers are not burdened to obtain any Ti or Sj , as they would simply pick any latency file to test their predictor.

I/O MODELING AND PREDICTING VIA MACHINE LEARNING: Given the datasets above, IONET enables us to design and train an ML-based model that can be used in the following way. When a new customer has a new workload (e.g., "W_{99}") and would like to predict the latency of every I/O in this workload when running on a popular SSD (*e.g.*, "S_2" that we already profiled), then we would give to the customer our predictor model "M_2" with parameters that have been trained before (i.e., "M_2" was trained with running $T_{1..I}$ on S_2). The customer can keep their workload private ("W_{99} in this case) and does not have to purchase S2, yet. They would simply input every I/O in "W_{99}" to the model "M_2", which will output for every I/O the predicted latency (specifically a fine-grained latency range).

To show a feasibility of predicting every I/O latency, we start with a simple experiment: can a machine learning method guess accurately whether the resulting latency will fall below or above a latency threshold (*e.g.*, 5ms)? In other words, we start with a binary classification. Table 1 shows our initial effort in trying out many machine learning methods, and a promising initial result can be seen in the use of neural networks. Furthermore, in the next one year, we will re-design the model aiming for a more realistic non-binary classification (e.g., exponential latency buckets of 0-1µs, 2-4µs, 4-8µs, and so on). We will also train and test the model across all the workloads and SSDs we have ($I \times J$ profiles).

The complete vision is summarized below:

1. We collect as many raw IO traces ($T_i = 1..X$) as possible from our industry partners. For proprietary traces, we will rerate or resize the trace (*e.g.* increase I/O intensity by 2x, resize every I/O to be 2x larger, etc.)
2. We gain access to as many storage devices ($S_j = 1..Y$) from various vendors.
3. Our team at UChicago will rerun every trace on every device, and collect the block-level trace (*e.g.* with the blktrace tool). For instance, with $X = 20$ and $Y = 20$, we will have 400 "trace profiles".
4. With approval from industry partners, we publish these 400 profiles and publish the traces in SNIA IOTTA [1]. The minimum goal is to reach a dataset with at least 1 billion I/Os.
5. The end goal, the community doesn't have to rely on an expensive storage device and hard-to-get I/O traces in order to develop a better ML-based I/O predictor.

3 The Workflow

This is the workflow of how IONET's pipeline can be used by the community.

1. Cutting the raw traces
   First, we have to determine the desired duration (*e.g.* 10 minutes). Then, IONET scripts can find a 10 minute section from each of the raw traces that satisfy one of these following options: the busiest (highest IOPS), biggest bandwidth, largest I/O's size (in average), and most random write. Afterwards, we can cut the raw traces based on the start-time and end-time of the desired section.
2. Modifying (filter / rerate / resize) the traces
   IONET has a trace-editor script that can do filtering, rerating, and resizing the input traces. This is helpful to create a diversification among the traces to match any workload profile that we want. For example, we want to train a model on a 100x bigger workload, we can achieve that by resizing the traces to 100x.
3. Analyzing the modified-traces
   IONET's trace-analyzer can analyze the characteristics of the given trace. The generated output are CDF of Bandwidth, CDF of IOPS, and CDF of Interarrival time. This is useful to check whether the modified-traces satisfy the workload profile that we want.
4. Replaying the traces
   A trace-replayer script will replay all input traces to the given devices. The replayed-traces will record the latency of every single I/O (including the latency of multiple sub-I/Os that was splitted from a single big I/O). Then we can analyze the replayed traces using IONET's trace-analyzer to get these following characteristics: length of the traces (in seconds), I/O count, IOPS, read vs write percentage, read throughput, write thoughput, whisker plot of the read and write request, bucketed read and write size, etc. Furthermore, we called the replayed-trace as trace **profile**.
5. Parsing the traces
   The goal of this process is to get a clean dataset for training and testing the model. Given the replayed-traces (`profile`) from the previous step, we use IONET trace parsers to extract the desired input features that match to each of the IONET models (detailed in Section §6).
6. Training the models
   Lastly, we train the IONET models against the datasets that we have. The outputs are accuracy, FN, FP, TP, TN, and training latency. Those output are easily adjustable as needed. Finally, we can analyze the training result with IONET's graph-generator scripts. The scripts are capable of parsing the training output and dynamically generate thousands of graphs based on various combinations of the `TraceType`, `EditOption`, `DeviceName`, and `ModelID`. Those variables are detailed in Table 5.

	Length (hour)	Total IO (million)
`Raw Azure`	142	29.8
`Raw BingI`	75	39.6
`Raw BingS`	21	5.5
`Raw Cosmos`	120	25.1

Table 2: **Characteristic of the raw traces acquired from Microsoft cloud infrastructure (Section §4).** *These raw traces will be cut to a shorter duration to reduce the training time.*

4 The Data

We have four raw-traces from Microsoft, which are named `Azure`, `BingI`, `BingS`, and `Cosmos` (detailed in Table 2). These traces were collected by replaying production requests on a

	Azure	**BingI**	**BingS**	**Cosmos**
IO Count	97016	102396	391375	167807
IOPS	161.70	170.67	652.29	304.22
R : **W** (%)	8 : 92	100 : 0	100 : 0	20 : 80
Total **R** (GB)	157.28	755.49	7690.93	2401.68
Total **W** (GB)	507.36	0	0	6182.95

Table 3: **Characteristic of the chosen 10 minutes long section of each raw traces.** *This is the characteristic of the* `original` *(not rerated nor resized) traces section. Rerating and resizing the traces will result in a different IOPS and different I/O size. The R denotes read operation, while W is write operation.*

	Brand	**Size**	**Type**	**Code**
`nvme0n1`	*SG*	128G	PCIe	MZVPV128HDGM
`nvme1n1`	*IN*	2T	PCIe	SSDPEDKE020T7
`nvme2n1`	*SG*	1.6T	PCIe	MZPLL1T6HEHP
`nvme3n1`	*WD*	1.6T	PCIe	HUSMR7616BHP301
`sdd`	*SG*	2T	SATA	SSD-850-PRO
`sde`	*SG*	128G	PCIe	MZHPV128HDGM

Table 4: **Specifications of the devices.** *These are the devices that we use to replay the selected traces. The SG is the abbreviation of Samsung, IN is Intel, while WD is Western Digital.*

private cluster that hosted the services purely for the sake of trace collection. The requests were logged in real-time in a small proportion of servers from the entire cluster. The collected requests were then replayed on top of the servers in a test cluster. The test cluster was setup in a such condition so that the trace collection would not interfere with production services. Microsoft trace logging system was used for the instrumentation and trace collection at the file system, cache, and SSD levels.

Furthermore, the raw traces have these following fields: submission time, block offset, block size, read/write, and most importantly, the I/O completion time. These raw traces then get modified and cut to 10 minutes long sections based on the highest IOPS. This trace modification process is part of the IONET workflow as explained in the previous section. For this experiment, we pick 10 minutes long section because that length is sufficient to train a model with low traning latency and considerably high accuracy. However, we also believe that the longer the trace length used as the training data, the better the accuracy.

The details of the `original` (unedited) traces, that have been cut to 10 minutes long, can be found in Table 3. Based on the IOPS and the total byte (written and read), we can categorize the traces into two different classes, light and heavy. The `Azure` and `BingI` are classified into the **light-workload** class, while the `BingS` and `Cosmos` are into the **heavy-workload** class. In average, the heavy-workload traces are 3x faster and 11x bigger than the light-workload traces.

5 The Devices

We have six devices in total, which being used to replay the selected traces (Table 3). In general, we categorize the devices into two classes, consumer and enterprise class. The enterprise-level devices (`nvme1n1`, `nvme2n1`, and `nvme3n1`) are faster and generally have higher workload tolerance than the consumer-level (`nvme0n1`, `sdd`, `sde`). The detail specifications of these devices are shown at Table 4.

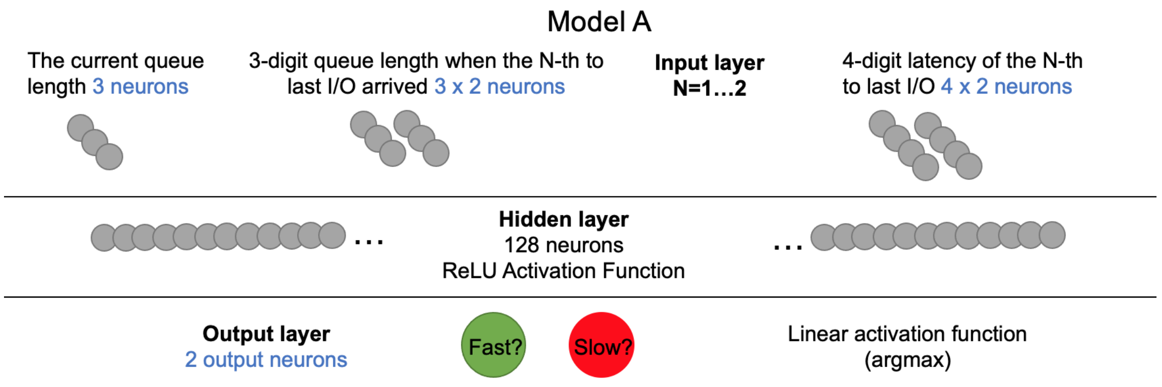
6 The Models

These are the architecture and the intuition behind the four IONET models that we have.

6.1 Model A

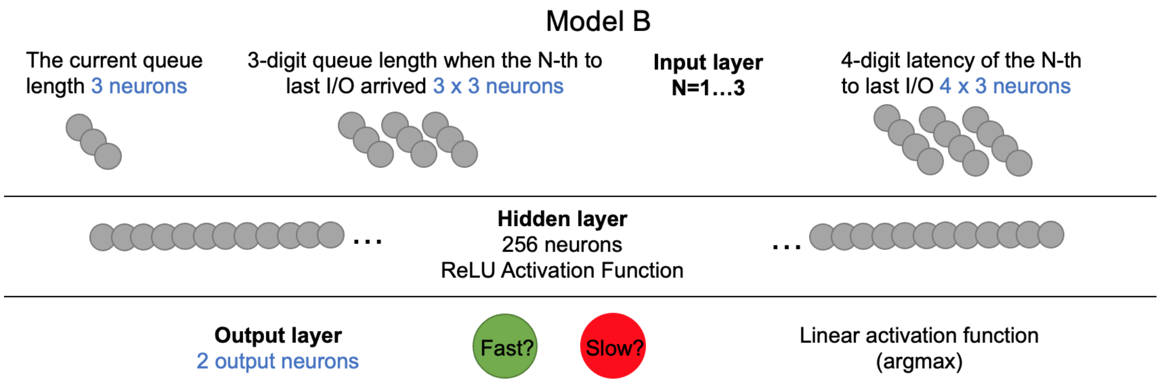
Overall theme of the `Model-A` is to be as light weight as possible without sacrificing too much accuracy. This model has three main input features with total dimension size of 17 where each dimension represents one-digit decimal number. Total dimension size along with its constituent input feature dimension sizes are chosen to reflect the very light weight nature of the model. These main input features are: (a) the number of pending I/Os when an incoming I/O arrives, (b) the set of historical latency values ranging from the latency of most-recently completed I/O to that of second most-recent, (c) the set of number of pending I/Os starting at the time of second most-recently completed I/O and ending at the time of most-recently completed I/O. The first feature is chosen since there is usually a correlation between I/O latency and the number of pending I/Os. The number of 4KB pending pages are the chosen unit for the first feature as the lowest granularity of striping inside SSDs is typically at the page level. The first feature is represented by using three decimal digits and hence occupies three dimensions of the seventeen dimensional input feature vector.

Recording small amount of historical information about latency values and number of pending IO sizes can be helpful at understanding the internal operations and busyness of the SSD drive. Having a long delay without a lot of pending I/Os might indicate internal contention due to device-level activities such as GC, internal flushing, or wear leveling. Thus that is why these variables are incorporated into the model as second and third feature. In the `Model-A`, the historical information in second and third feature only looks up to information of last two I/Os, reflecting the very light weight theme of this model. The second feature has 2 elements in it (i.e latency of most-recently completed I/O and latency of second most-recently completed I/O) and each of these elements in μs are represented by using 4 decimal digits which in total occupies eight dimensions of the seventeen dimensional input feature vector. Similar to second feature, third



Model A
The current queue length 3 neurons
3-digit queue length when the N-th to last I/O arrived 3 x 2 neurons
Input layer N=1...2
4-digit latency of the N-th to last I/O 4 x 2 neurons
Hidden layer 128 neurons ReLU Activation Function
Output layer 2 output neurons
Fast?
Slow?
Linear activation function (argmax)

Figure 1: **Neural Network Architecture of Model A**. *Explained in Section §6.1*



Model B
The current queue length 3 neurons
3-digit queue length when the N-th to last I/O arrived 3 x 3 neurons
Input layer N=1...3
4-digit latency of the N-th to last I/O 4 x 3 neurons
Hidden layer 256 neurons ReLU Activation Function
Output layer 2 output neurons
Fast?
Slow?
Linear activation function (argmax)

Figure 2: **Neural Network Architecture of Model B.** *Explained in Section §6.2*

feature has also 2 elements in it (i.e the number of pending I/Os at the time when second most recently completed I/O arrived and the number of pending I/Os at the time when most recently completed I/O arrived). However these two elements are formatted using 3 decimal digits each and in total makes up the remaining six dimensions of the seventeen dimensional input feature vector [18].

In general, decimal digit based representation of the input feature vector intends to achieve balance between inference time and accuracy. Using a input representation that is based on binary digits (or any other digit systems that has lower base than that of decimal) would result in larger input dimension and hence larger number of corresponding neurons in machine learning model which in return increases the inference time. On the other hand directly using numbers as inputs without dividing them into smaller decimal digit features, or using digit systems that has larger base than that of decimal can result in a harder training/learning process.

Just like the very light design choices made in the input features and their formatting, machine learning architecture used in this model also tries to accomplish that without sacrificing too much accuracy by utilizing a fully-connected neural network model with only three layers. First the raw information from the block layer is preprocessed with optimized $O(1)$ preprocessing overhead to create the 17 dimensional input feature vector described above. The resultant inputs are send to the input layer with 17 neurons. The input layer is followed by the hidden layer with 128 neurons and reLU activation function. The result obtained from the hidden layer is passed through the output layer with 2 neurons and linear activation function. The final output is converted to a binary decision using the argmax operator.

6.2 Model B

`Model-B` has a slightly less light weight design than `Model-A` does, but it still prioritizes lightness over accuracy. Reflecting this, Input vector of `Model-B` has total dimension size of 24 with three main input features that are same as `Model-A` but with the historical information in second and third feature extending up to the third most recent I/O. Thus `Model-B` utilizes larger window of historical information compared to that of `Model-A`. The size contributions of each feature in `Model-B` are as follows: the first feature is represented by 3 decimal digits and makes up the three dimensions of the

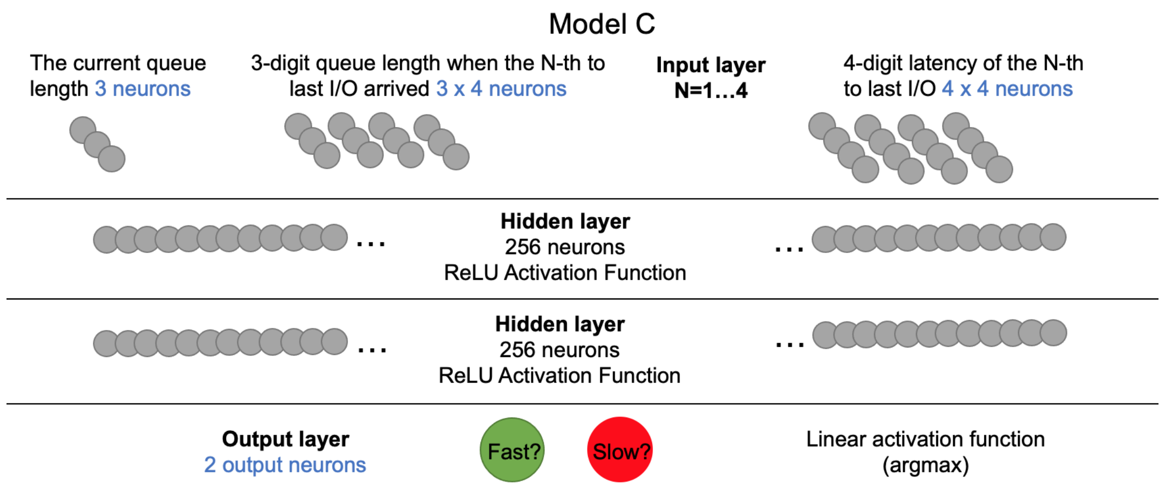

Model C
The current queue length 3 neurons
3-digit queue length when the N-th to last I/O arrived 3 x 4 neurons
Input layer N=1...4
4-digit latency of the N-th to last I/O 4 x 4 neurons
Hidden layer 256 neurons ReLU Activation Function
Hidden layer 256 neurons ReLU Activation Function
Output layer 2 output neurons
Fast?
Slow?
Linear activation function (argmax)

Figure 3: **Neural Network Architecture of Model C.** *Explained in Section §6.3*

24 dimensional input feature vector, the second feature has three elements in it (i.e latency of most-recently completed I/O, ... and latency of third most-recently completed I/O) and each are represented by 4 decimal digits, forming the 12 (4x3) dimensions of the total; finally the third feature has also three elements in it (i.e the number of pending I/Os at the time when third most recently completed I/O arrived, ... and the number of pending I/Os at the time when most recently completed I/O arrived) and each are represented by 3 decimal digits, making up the remaining 9 (3x3) dimensions.

Similar to machine learning architecture used in `Model-A`, `Model-B` also uses a fully-connected neural network model with three layers. After raw inputs are reprocessed and formed in to the the 24 dimensional input feature vector, they are send to the input layer with 24 neurons. Following the input layer comes hidden layer with 256 neurons and reLU activation function. This hidden layer has larger number of neurons compared to that of `Model-A` to provide more information processing power. The last layer in the model is the output layer and it has 2 neurons and linear activation function. The binary decision is obtained by applying argmax operator to the output.

6.3 Model C

This model relaxes the restriction on lightness further and puts slightly more emphasize on the accuracy over lightness. `Model-C` follows the same feature structure as that of `Model-A` and `Model-B` but with a larger window of historical information in the second and third feature, extending up to the fourth most recent I/O.

In general models A to D, explore the trade off between accuracy and light weightiness of the model by changing number of input feature dimensions in the way of presenting more or less historical IO information, and by utilizing neural network architectures with various number of neurons and layers. Recurring pattern here is if a model is aiming for a high accuracies it tends to use a lot of historical I/O information along with heavier neural network model that have large number of layers and neurons. On the other hand the model that is aiming for lightness utilizes as few historical I/O information as possible with a very light network that has very few layers and nodes.

Input vector of `Model-C` has total dimension size of 31. The composition of this dimension size is as follows: 3 decimal digits are used to represent the first feature and hence it contributes three dimensions to the total size, the second and third feature utilizes historical information up to the fourth most recent I/O and hence they both include 4 I/0 elements (i.e latency of most-recently completed I/O, ... and latency of fourth most-recently completed I/O for the second feature; and the number of pending I/Os at the time when fourth most recently completed I/O arrived, ... and the number of pending I/Os at the time when most recently completed I/O arrived for the third feature) and each element in the second feature is represented by 4 decimal digits, contributing the next 16 (4x4) dimensions and each element in the third feature is represented by 3 decimal digits, contributing the remaining 12 (4x3) dimensions.

This model uses fully-connected neural network with four layers and these layers are: input layer with 31 neurons, two hidden layers with 256 neurons and reLU activation function, and finally output layer with 2 neurons and linear activation function. Raw inputs are preprocessed to form the 31 dimensional input feature vector, before entering into the input layer. Argmax operator is used to obtain the binary decision.

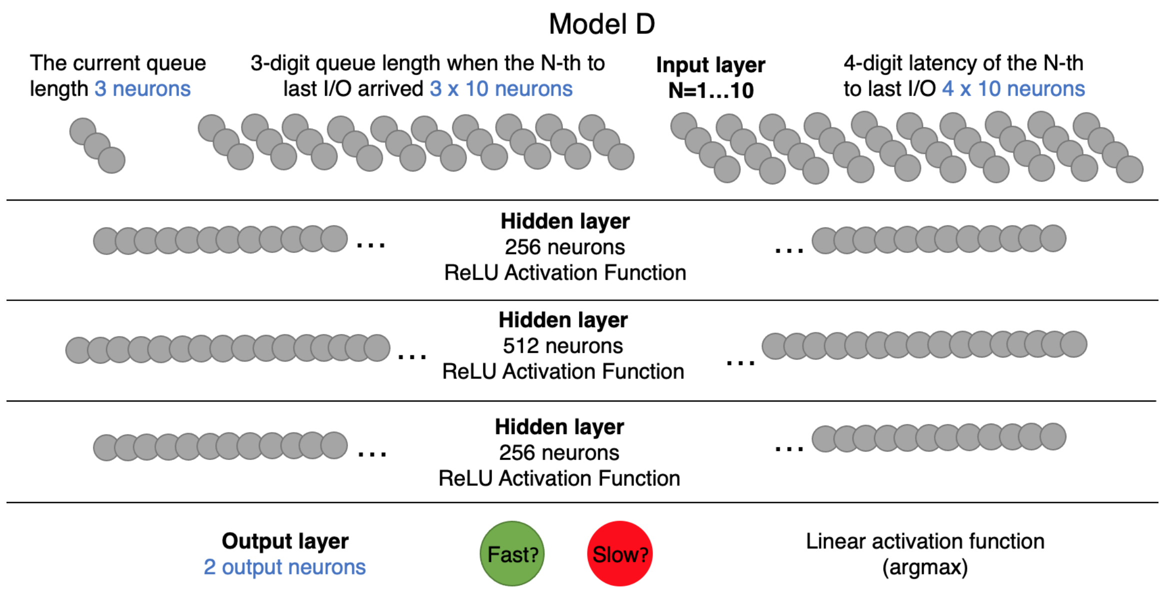

Model D
The current queue length 3 neurons
3-digit queue length when the N-th to last I/O arrived 3 x 10 neurons
Input layer N=1...10
4-digit latency of the N-th to last I/O 4 x 10 neurons
Hidden layer 256 neurons ReLU Activation Function
Hidden layer 512 neurons ReLU Activation Function
Hidden layer 256 neurons ReLU Activation Function
Output layer 2 output neurons
Fast?
Slow?
Linear activation function (argmax)

Figure 4: **Neural Network Architecture of Model D.** *Explained in §6.4*

6.4 Model D

This model is accuracy oriented heavy model with a large inference time. Following the general trend, it has the same input features as the other models but historical information in the second and third features extend up to the tenth most recent I/O. Thus `Model-D` utilizes the largest window of historical information among all other models, making it the most feature rich model which in return contributes the heaviness of the model and negatively impacts the inference time. It‘s input feature vector has 73 dimensions with following size distribution: first feature contributing 3 dimensions to the total with its 3 digit encoded representation, both second and third features contain 10 historical I/O information with second feature using 4 decimal digit based representation and third feature using 3 decimal digit based representation, making up the remaining 70 (3x10 + 4x10) dimensions.

The `Model-D` utilizes relatively heavy neural network with large number of layers and neurons. Especially, it has the largest number of hidden layers and neurons compared to that of other models. This provides `Model-D` with more processing power but further increases the inference time. More specifically, `Model-D` has the following machine learning architecture: It has one input layer with 73 neurons, three hidden layers with 256, 512, 256 neurons respectively, each with reLU activation function and one output layer with 2 neurons. Input feature vector is obtained by preprocessing the raw inputs and final binary decision is obtained by using the argmax operator.

	Values
`ModelID`	[`Model-A`, `Model-B`, `Model-C`, `Model-D`]
`TraceType`	[`Azure`, `BingI`, `BingS`, `Cosmos`]
`DeviceName`	[`nvme0n1`, `nvme1n1`, `nvme2n1`, `nvme3n1`, `sdd`, `sde`]
`EditOption`	[`rerated-10x`, `rerated-100x`, `resized-10x`, `resized-100x`]

Table 5: **All possible values of the experiment variables.** *The evaluation at Section §8 explains the relation between these variables.*

7 The Setup

The experiment is run on a machine which has 2.6GHz 18-core (36-thread) Intel i9-7980XE CPU with 128GB DRAM (no accelerators). Moreover, all devices have been used for months with many workloads that reach the devices' full capacities, hence mimicking devices in the field. Furthermore, these are four variables that we used to vary our experiment:

1. `TraceType`: indicates which trace that we use as the input dataset. All traces are explained in Section §4.
2. `DeviceName`: shows which device that we use to replay the traces. All devices are detailed in Section §5.
3. `ModelID`: identifies which model that we use. All models are described in Section §6.
4. `EditOption`: points out the type of editing that was done on the input trace. The unedited trace is identified as `original` trace.

The values of these four variables are shown in Table 5.

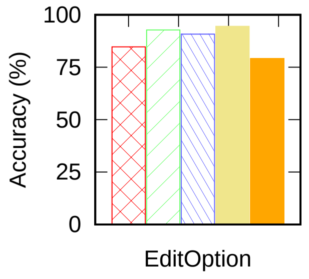


Figure 5: **Accuracy across EditOption.** `TraceType =Azure, DeviceName =nvme0n1, ModelID =Model-D`

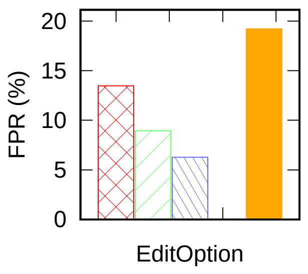


Figure 6: **FPR across Edit-Option.** `TraceType =BingS, DeviceName =sde, ModelID =Model-D`

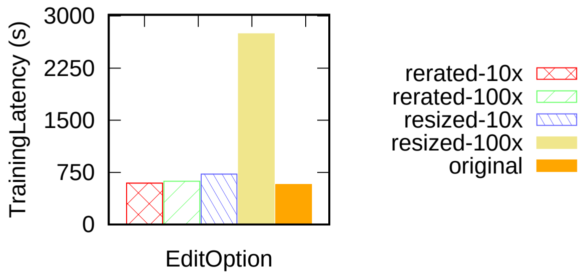


Figure 7: **Training latency across EditOption.** `TraceType =BingS, DeviceName =nvme0n1, ModelID =Model-D`

When analyzing the date, we will use those variables in the x-axis. The y-axis will be the numerical data, such as accuracy, False Positive Rate/FPR, and training latency. In total, the script generates 416 2D-bar graphs for each y-value. However, we can't include all the graph in the Section §8; and *WLOG*, we carefully pick the graph that represent the general trends. Therefore, for completeness, all figures in Section §8 always include the combination of the variables where the data was taken from.

8 Results

We analyze the result based on all possible combination of x-axis and y-axis variations as explained in Section §7. There are four possible variables (Table 5) for the x-axis which are `EditOption`, `TraceType`, `ModelID`, and `DeviceName`. Each of the x-axis has these three possible y-axis values, acuracy, FPR (False Positive Rate), and training latency. In general, there are two type of bar graph, 2D bar graph and 3D bar graph. We explain our hyphotheses of the graph's trend by referring to all quantitative data that we have, including the extra data/measurement that we acquire during the trace-replaying process and the training process.

8.1 Two-Dimensional Analysis

Below are the analysis of our experiments in the form of 2D bar graph.

8.1.1 Accuracy across Different EditOption

Editing (resizing/rerating) the traces in Figure 5 increase the accuracy. This trend is closely related to the accuracy of the `original` (unedited) trace and the device that we use. When the `original` accuracy is low ($<$ 90%), the edited (resized/rerated) traces will likely increase the accuracy. However, when the `original` accuracy is already high ($>$90%), rerating/resizing the traces will less likely increase the accuracy but rather decrease it. In addition, the device `nvme0n1` on this experiment is not overloaded by the workload, thus rerating/resizing the traces will have a positive impact towards the accuracy. The overloaded device can be verified from its average I/O latency that is more than 300x higher than the average I/O latency of the unedited trace.

8.1.2 FPR across Different EditOption

Based on Figure 6, we can see that the edited (resized/rerated) traces have less FPR (False Positive Rate) compared to the `original` (unedited) trace. In addition, the bigger the resize/rerate factor, the lower the FPR. The reason behind it is that the `original` (unedited) trace is too light for this device. Thus, increasing the workload's intensity and size (by rerating/resizing) will push the device to achieve its optimal performance and better predictability (low FPR).

8.1.3 Training Latency across Different Edit-Option

Figure 7, we can see that rerating/resizing the traces are generally have higher training latency than the `original` (unedited) trace. Furthermore, the `resized-100x` has highest training time than the other edit options because resizing to 100x generates much larger dataset (approximately about 4 times bigger).

8.1.4 Accuracy across Different DeviceName

We can see from Figure 8 that the consumer-level devices (`nvme0n1`, `sdd`, and `sde`) have higher accuracy compared to the enterprise-level devices (`nvme1n1`, `nvme2n1`, and `nvme3n1`). The reason is that the resized `Azure` trace is still too light for the enterprise-level devices, thus they are in the suboptimal condition that make it hard to predict. In other words, the accuracy of the enterprise-level devices can be improved by using more intense workload. Regarding the accuracy of the consumer-level devices, the device `nvme0n1` has the highest accuracy because the resize factor is just right (not too light nor too heavy) such that the device could achieve its optimal performance.

8.1.5 FPR across Different DeviceName

The general trend at Figure 9 shows that enterprise-level devices have higher FPR than the consumer-level devices. This is due to the fact that the consumer-level devices could

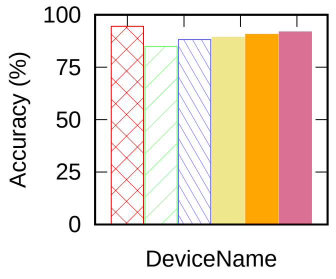


Figure 8: **Accuracy across DeviceName.** `ModelID =Model-D, EditOption =resized-100x, TraceType =Azure`

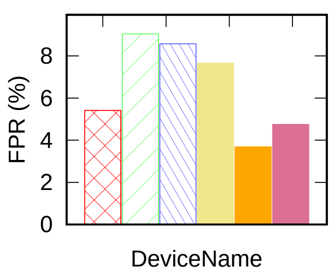


Figure 9: **FPR across DeviceName.** `ModelID =Model-C, EditOption =rerated-100x, TraceType =Azure`

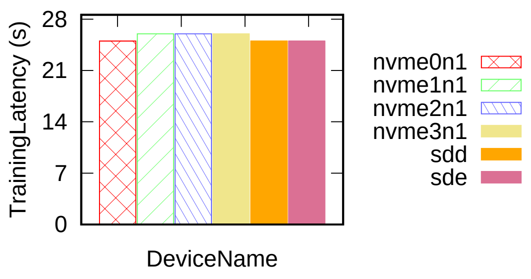


Figure 10: **Training latency across DeviceName.** `ModelID =Model-C, EditOption =resized-100x, TraceType =Azure`

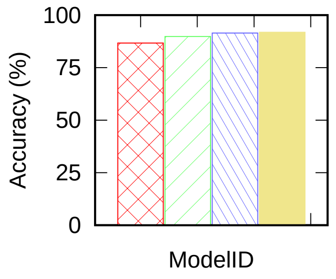


Figure 11: **Accuracy across ModelID.** `TraceType =Azure, EditOption =resized-100x, DeviceName =sde`

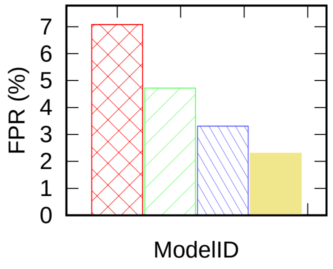


Figure 12: **FPR across ModelID.** `TraceType =Azure, EditOption =resized-10x, DeviceName =sdd`

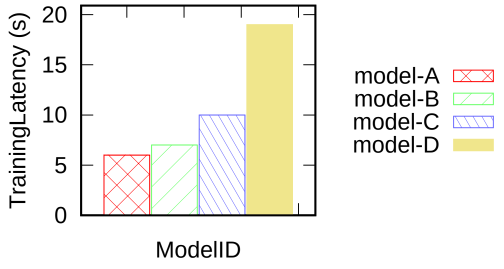


Figure 13: **Training latency across ModelID.** `TraceType =Azure, EditOption =resized-10x, DeviceName =sde`

achieve better (more optimal) performance on the current workload (`rerated-100x` traces). On the other hand, currently, the enterprise-level devices are in the suboptimal condition and require more intense workload to increase their predictability. Therefore, the enterprise-level devices have a higher FPR (less accuracy) than the consumer-level devices.

8.1.6 Training Latency across Different DeviceName

Figure 10 shows that the training latency is not affected by the type of the devices. This is due to the main resources being consumed during the training process are the processing units (CPU and GPU). Since all devices (SSDs) are co-located on the same machine; thus, the training latency of each device is similar because they use the same processing units.

8.1.7 Accuracy across Different ModelID

Figure 11 shows that the accuracy correlates highly with the complexity of the models. The more complex the model, the higher the accuracy. As detailed in the Section §6, the complexity of the models are gradually increasing from `Model-A` (has 17 input features) to `Model-D` (has 73 input features). Therefore, `Model-D` as the heaviest model achieves better accuracy than the others as it can accommodate more historical information of the I/O traces.

8.1.8 FPR across Different ModelID

Figure 12 shows that the FPR correlates highly with the complexity of the models. As detailed in the Section §6, the complexity of the models are gradually increasing from `Model-A` to `Model-D`. Therefore, `Model-D` as the heaviest model achieves better accuracy (smaller FPR) than others.

8.1.9 Training Latency across Different ModelID

Figure 13 depicts that the more complex the model, the higher the training latency. Thus, `Model-D` has the highest training latency because it is the most complex (heaviest) model as explained in in the Section §6. On the other hand, `Model-A` as the lightest model with only 17 input features achieves the shortest training time.

8.1.10 Accuracy across Different TraceType

Based on Figure 14, `BingS` and `Cosmos` have less accuracy compared to the `Azure` and `BingI` because the device (`sde`) was overloaded by `BingS` and `Cosmos` traces. Without any rerate/resize, `BingS` and `Cosmos` are categorized as heavy-workloads traces. Given the fact that the unedited version of `BingS` and `Cosmos` traces are already bigger and more intense, resizing the traces to 100x make the `BingS` and `Cosmos` became a super-heavy workload which easily overload the device `sde`.

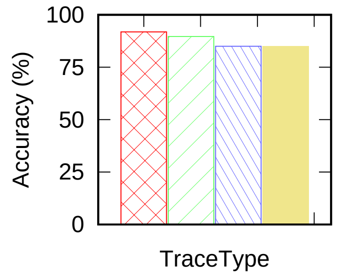


Figure 14: **Accuracy across TraceType.** `ModelID =Model-D, EditOption =resized-100x, DeviceName =sde`

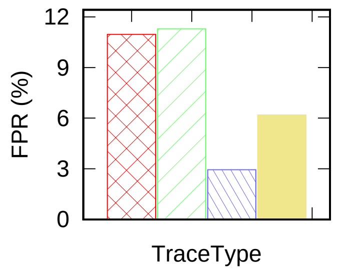


Figure 15: **FPR across TraceType.** `ModelID =Model-C, EditOption =rerated-10x, DeviceName =nvme3n1`

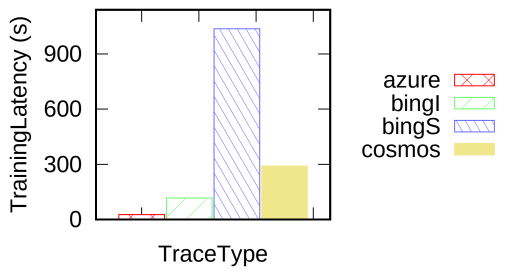


Figure 16: **Training latency across TraceType.** `ModelID =Model-C, EditOption =resized-100x, DeviceName =nvme2n1`

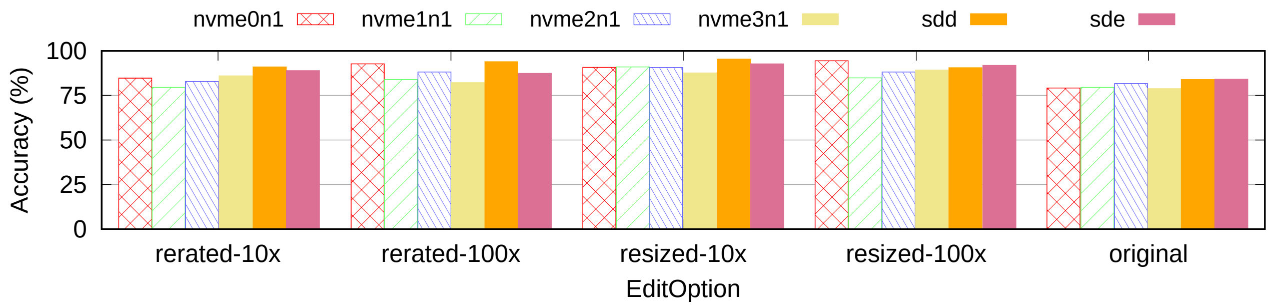


Figure 17: **Accuracy across different DeviceName and EditOption.** `ModelID =Model-D, TraceType =Azure`

8.1.11 FPR across Different TraceType

Figure 15 shows that light-workload traces (`Azure` and `BingI`) have higher FPR (lower accuracy) than heavy-workload traces `BingS` and `Cosmos`. In this experiment, we use `nvme3n1` which is an enterprise level device. `Azure` and `BingI` traces are too light for `nvme3n1` so `BingS` and `Cosmos` traces (which fall into the heavy-workload category) are able to push the device to give better performance. Thus they have lower FPR and can better predict device behavior.

8.1.12 Training Latency across Different TraceType

As shown in Figure 16, `BingS` has the longest training latency because `BingS` trace has the highest number of IO than the others, thus it needs longer training time. Referring to the Table 3, the IO count of `BingS` = 391375; `Cosmos` = 167807; `BingI` = 102396; and `Azure` = 97016. In general, the more the IO, the longer the training latency.

8.2 Three-Dimensional Analysis

Below are the analysis of our experiments in the form of 3D bar graph.

8.2.1 Accuracy across Different DeviceName and EditOption

Figure 17 shows that the rerated or resized traces lead to a better accuracy regardless of the device that we use because the the heavier the workload, the better the accuracy. That correlation holds as long as the device doesn't get overloaded by the workload.

8.2.2 Accuracy across Different EditOption and TraceType

The trends on the Figure 18 are determined by the accuracy of the `original` traces. When the `original` accuracy is low ($< 90\%$), the edited (resized/rerated) traces will likely have better accuracy. However, when the `original` accuracy is already high ($>90\%$), rerating/resizing the trace will less likely increase the accuracy but rather decrease it as shown by the bar chart at `BingI` and `Cosmos`.

8.2.3 Accuracy across Different ModelID and EditOption

Based on Figure 19, the `original` (unedited) trace has less accuracy compared to the edited (rerated/resized) traces. The higher the rerate/resize factor, the higher the accuracy as long as the device doesn't get overloaded. The overloaded device can be verified from its average I/O latency that is more than 300x higher than the average I/O latency of the unedited trace.

8.2.4 Accuracy across Different ModelID and DeviceName

The general trend on Figure 20 shows that enterprise-level devices are harder to predict than the consumer-level devices. The easiest device to predict is device `nvme0n1`, while device `nvme1n1` is the hardest one. This is due to the fact that

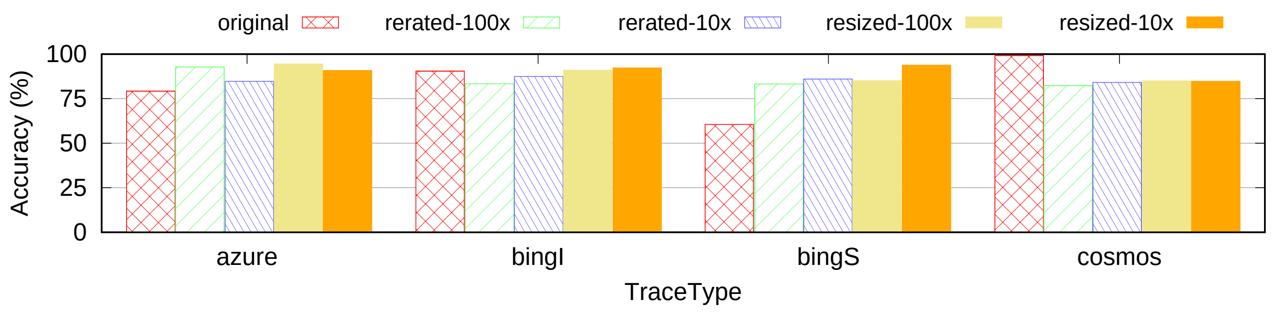


Figure 18: **Accuracy across different EditOption and TraceType.** `ModelID =Model-D`, `DeviceName =nvme0n1`

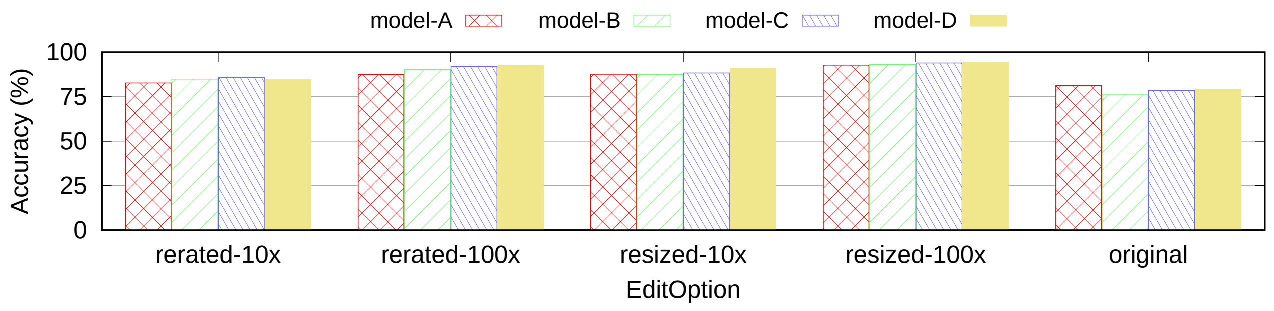


Figure 19: **Accuracy across different ModelID and EditOption.** `TraceType =Azure`, `DeviceName =nvme0n1`

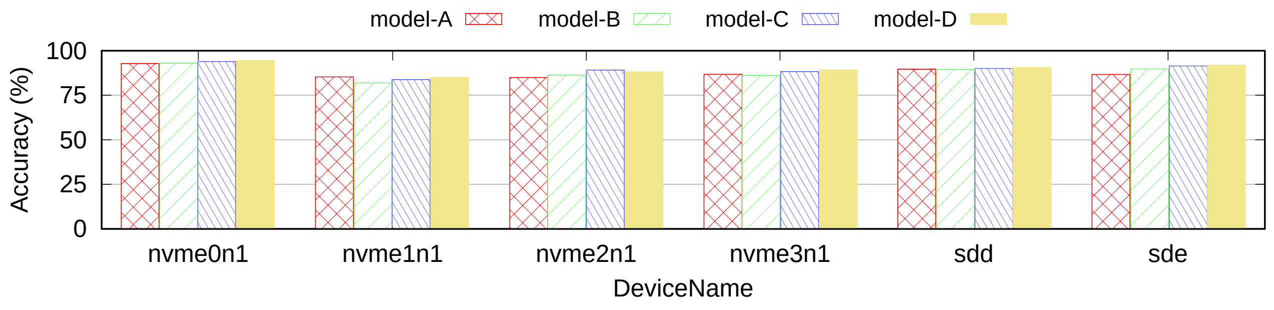


Figure 20: **Accuracy across different ModelID and DeviceName.** `EditOption =resized-100x`, `TraceType =Azure`

the consumer-level devices achieve optimal performance on the current edited traces (`resized-100x`). On the other hand, currently, the enterprise-level devices are in the suboptimal condition, thus it requires more intense workload to improve the accuracy.

8.2.5 Accuracy across Different ModelID and TraceType

Based on the datasets' characteristics, here are the average latency of the traces used in the experiment at Figure 21; `Azure` = 0.12 ms; `BingI` = 0.22 ms; `BingS` = 0.88 ms; and `Cosmos` = 51.76 ms. It shows that `Cosmos` average latency is much higher than others which is a strong indication that the device (`sde`) is overloaded. Since the device predictability will decrease as the device get overloaded, the `Cosmos` accuracy achieves far less accuracy than others.

8.2.6 Accuracy across Different TraceType and DeviceName

From Figure 22, we can see that the light-workload traces (`Azure` and `BingI`) running on the consumer-level devices (`nvme0n1`, `sdd`, and `sde`) have higher accuracy compared to other traces on any devices. The reason is that the rerate/resize factor on those traces are just enough such that the consumer-level devices can run optimally. In addition, the average accuracy at the enterprise-level devices could not reach as high accuracy as the prior case because the traces are either too light or too heavy. Referring to the input datasets' characteristics, the average latency of `BingS` and `Cosmos` traces on device `sde` are 156 ms and 4747 ms, while the average latency of `Azure` and `BingI` traces on the same device are only 5 ms and 2 ms respectively. Based on those numbers, we can conclude that `BingS` and `Cosmos` traces are overloading the device. Since the model accuracy will de-

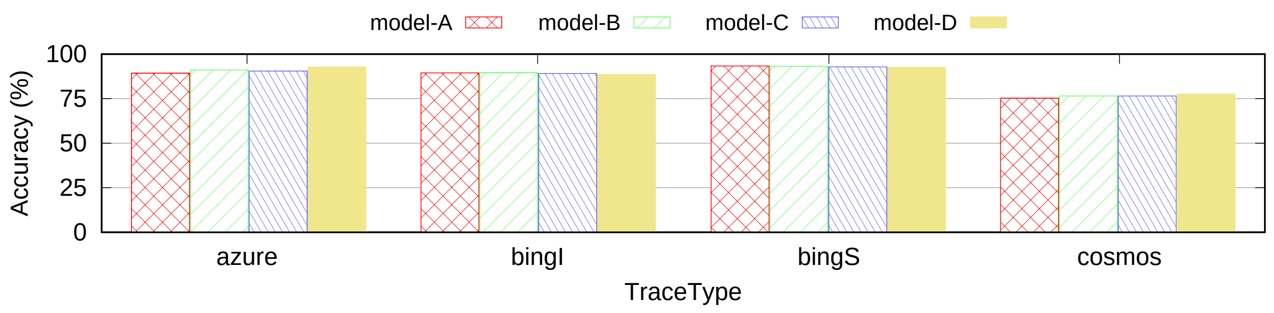


Figure 21: **Accuracy across different ModelID and TraceType.** `EditOption =resized-10x`, `DeviceName =sde`

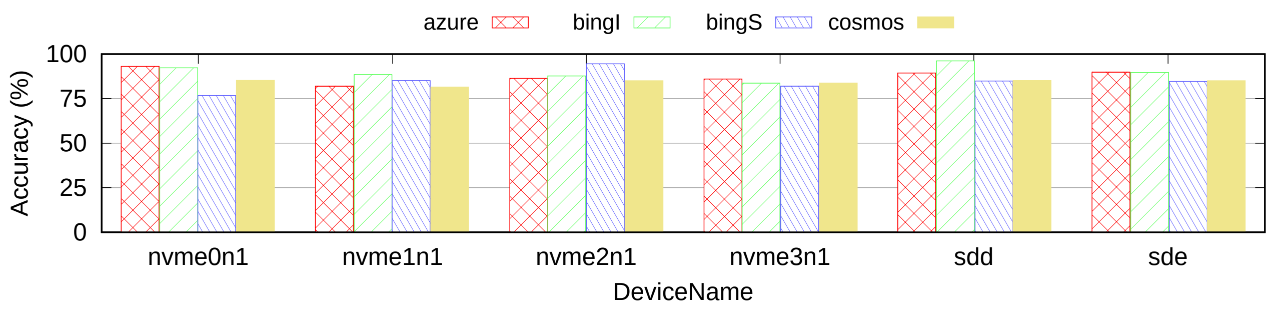


Figure 22: **Accuracy across different TraceType and DeviceName.** `EditOption =resized-100x`, `ModelID =Model-B`

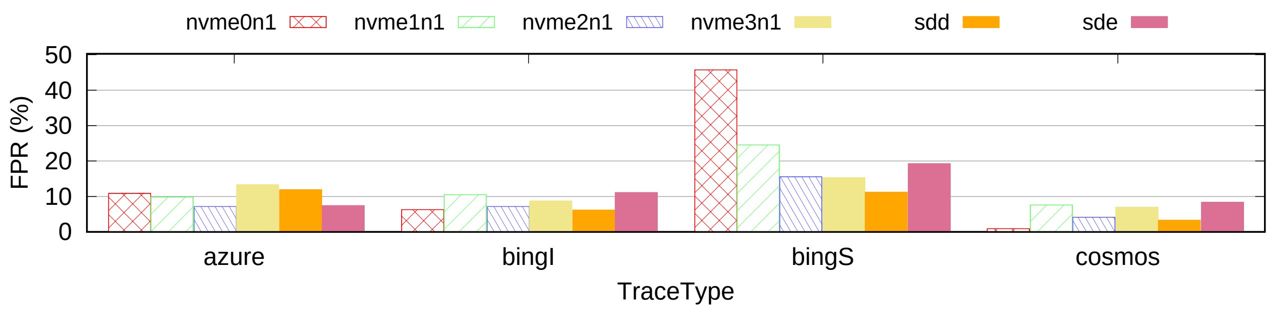


Figure 23: **FPR across different DeviceName and TraceType.** `EditOption =original`, `ModelID =Model-D`

crease as the device get overloaded, the use of `BingS` and `Cosmos` traces on this experiment will cause the model to get less accuracy.

8.2.7 FPR across Different DeviceName and TraceType

From Figure 23, we can see that `BingS` tends to have higher FPR (less accuracy) compared to other traces. Given the fact that the `original` (unedited) version of `BingS` trace is already heavy (3.5x more I/O than the other traces), rerating/resizing it to 100x will cause the device to get overloaded easily. Since an overloaded device tends to have a poor predictability, the FPR of `BingS` trace is expected to be higher than the other traces.

8.2.8 FPR across Different ModelID and DeviceName

Figure 24 depicts the general trend that the enterprise-level devices tend to have higher FPR and it is harder to predict than the consumer-level devices. The reason is the enterprise-level devices require heavier workload to achieve optimal performance. In this case, the resized `BingI` is still too light for the enterprise-level devices; thus the devices perform in a suboptimal condition which make the prediction less accurate (high FPR).

8.2.9 FPR across Different EditOption and DeviceName

Figure 25 shows that the rerated/resized by a factor of 100 tends to result in a lower FPR regardless of the device that we use. Since the workload intensity is closely correlated with the predictability of the device, the more intense the workload, the better the accuracy. Furthermore, the edited traces on Consumer-level devices tend to get a lower FPR regardless of the rerate/resize factor because the device predictability increases on a heavier workload. Meanwhile, the enterprise-level devices in general are in the suboptimal con-

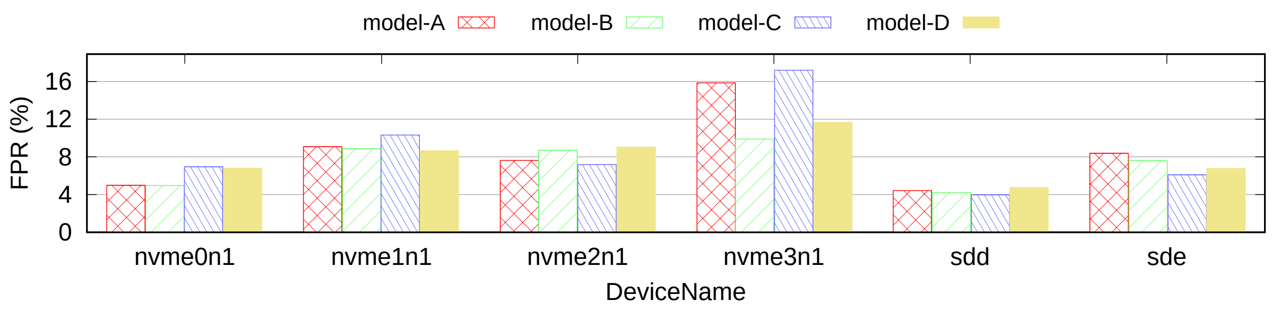


Figure 24: **FPR across different ModelID and DeviceName.** `EditOption =resized-100x`, `TraceType =BingI`

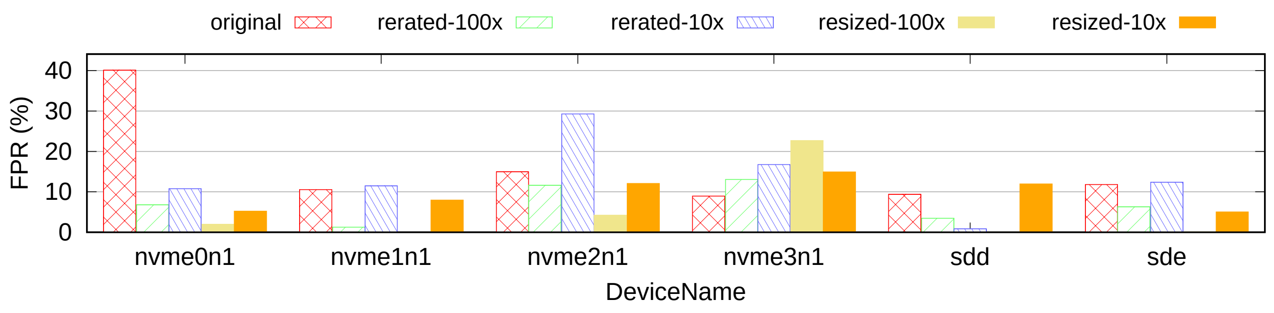


Figure 25: **FPR across different EditOption and DeviceName.** `ModelID =Model-A`, `TraceType =BingS`

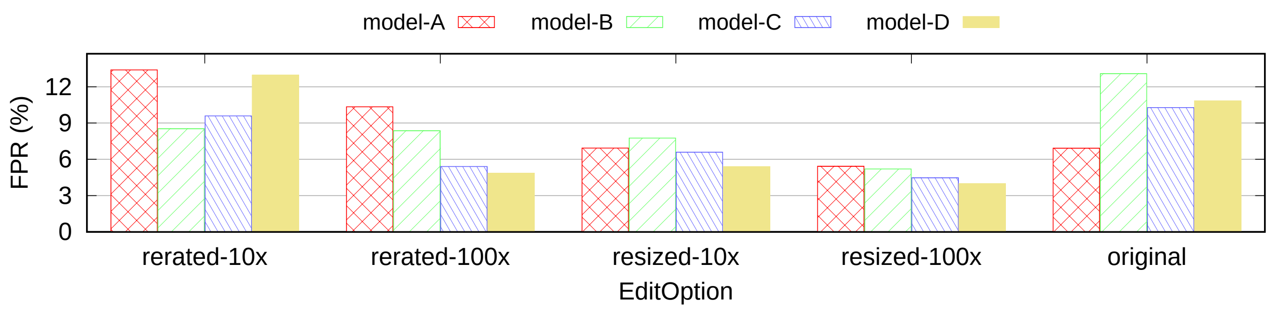


Figure 26: **FPR across different ModelID and EditOption.** `TraceType =Azure`, `DeviceName =nvme0n1`

dition, thus they require more intense workloads to improve the accuracy.

8.2.10 FPR across Different ModelID and EditOption

Based on Figure 26, `Model-D` tends to get smaller FPR (better accuracy) compared to other models. This is because `Model-D` is the heaviest model with 73 input features (which can accommodate more historical information of the I/O traces). Moreover, the `original` (unedited) trace tends to have higher FPR compared to the edited (rerated/resized) and resizing/rerating the traces to 100x tends to result in a lower FPR regardless of the model because the heavier the workload, the better the device predictability, hence the higher the accuracy as long as the device does not get overloaded. Overloaded devices can be detected with an average I/O latency that is more than 300x higher than the average I/O latency of the unedited trace.

8.2.11 FPR across Different EditOption and TraceType

The trends on Figure 27 are determined by the FPR of the `original` bar. When the `original` FPR is high (>10%), editing (resizing/rerating) the traces will likely decrease its FPR. Meanwhile, when the `original` accuracy is low (<10%), rerating/resizing the traces will likely increase the FPR as shown by the bar chart at `Azure` and `BingI`.

8.2.12 FPR across Different ModelID and TraceType

Figure 28 depicts that `Cosmos` FPR is higher (less accuracy) than the others. Based on the dataset characteristics, here are the average latency of the traces used in this experiment; `Azure` = 0.12 ms; `BingI` = 0.22 ms; `BingS` = 0.88 ms; and `Cosmos` = 51.76 ms. It shows that `Cosmos` average latency is much higher than others. In other words, the `Cosmos` trace overloads the device `nvme1n1` which then cause a hard-to-predict behavior. Furthermore, the FPR correlates highly

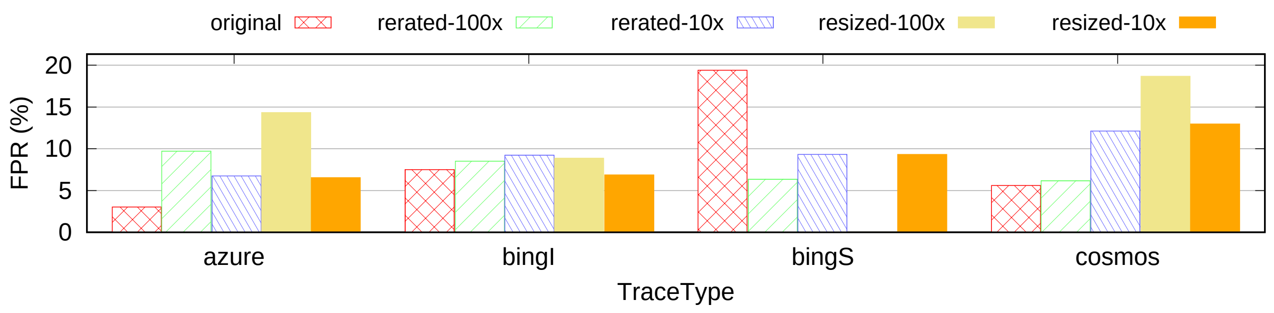


Figure 27: **FPR across different EditOption and TraceType.** `ModelID` =`Model-B`, `DeviceName` =`nvme1n1`

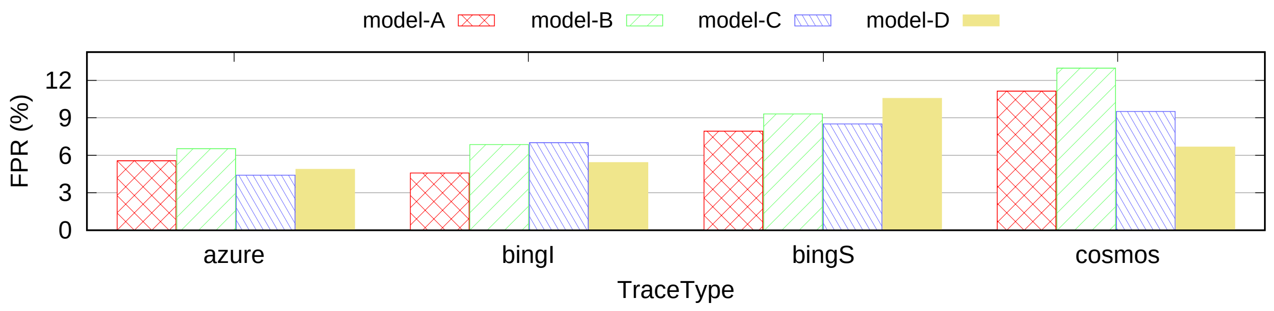


Figure 28: **FPR across different ModelID and TraceType.** `EditOption` =`resized-10x`, `DeviceName` =`nvme1n1`

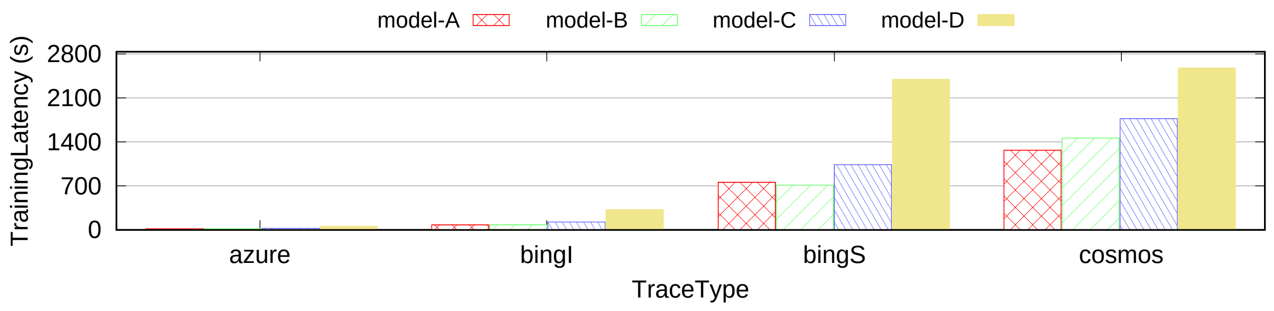


Figure 29: **Training latency across different ModelID and TraceType.** `EditOption` =`resized-100x`, `DeviceName` =`sdd`

with the complexity of the model. As detailed in the Section §6, the complexity of the models is gradually increasing from `Model-A` to `Model-D`. Thus, the `Model-D` as the heaviest model achieves better accuracy (smaller FPR) than others.

8.2.13 Training Latency across Different ModelID and TraceType

Based on Figure 29, `Model-D` has the highest training latency because it is the most complex (heaviest) model with 73 input features. As explained in the Section §6, `Model-A` is the lightest model with only 17 input features so it has the shortest training time. Moreover, the heavy-workload traces (`BingS` and `Cosmos`) tend to have higher training time since they generate bigger dataset (4 times bigger than the light-workload traces).

8.2.14 Training Latency across Different DeviceName and TraceType

Based on Figure 30, the training latency of `BingS` bar-cluster is the highest because `BingS` has the highest number of IO (Table 3). In general, the training latency will likely decrease when there is less IO. In addition, the training latency of a certain `TraceType` is similar in all devices because the devices does not introduce major resource contention during the training.

8.2.15 Training Latency across Different DeviceName and EditOption

Based on Figure 31, we can see that the `resized-100x` traces need longer training time because they generate much larger dataset (4x more than the other `EditOption`). Meanwhile, the other edit options have similar training latency because they use similar size datasets.

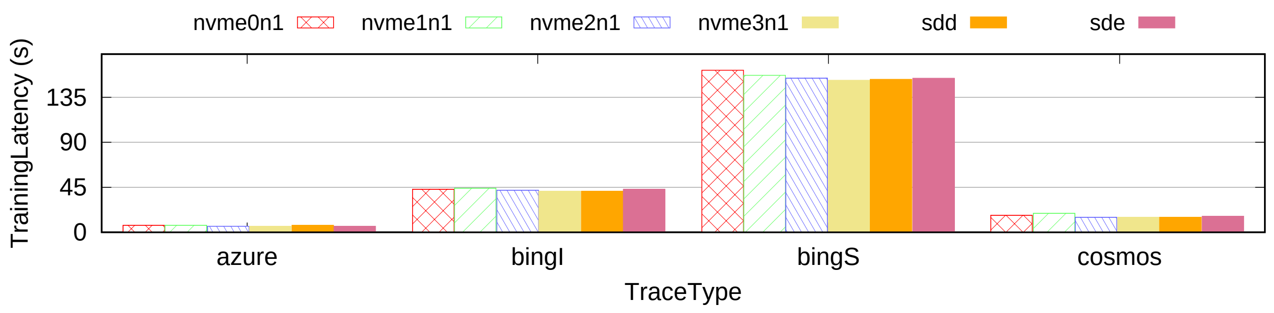


Figure 30: **Training latency across different DeviceName and TraceType.** `EditOption` =`rerated-10x`, `ModelID` =`Model-B`

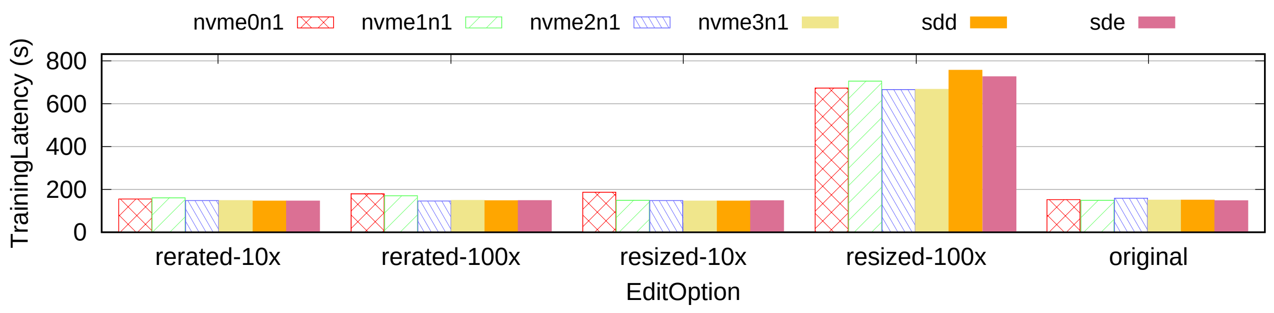


Figure 31: **Training latency across different DeviceName and EditOption.** `ModelID` =`Model-A`, `TraceType` =`BingS`

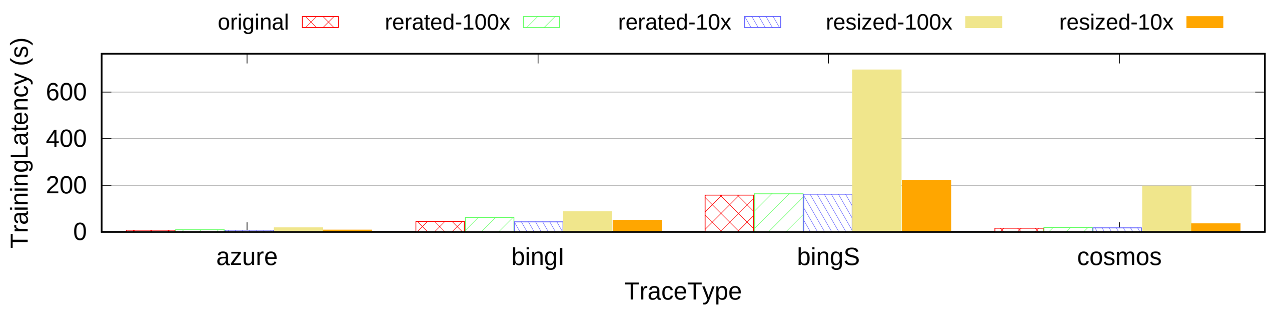


Figure 32: **Training latency across different EditOption and TraceType.** `ModelID` =`Model-B`, `DeviceName` =`nvme0n1`

8.2.16 Training Latency across Different EditOption and TraceType

Figure 32 shows that the training latency increases when the trace is `resized-100x`. The reason is that the dataset's size increases (4x bigger) when we resize the trace by 100. In addition, `BingS` has the highest number of IO (Table 3), thus it needs longer training time. And, the training latency will more likely decrease when the IO count becomes smaller.

8.2.17 Training Latency across Different ModelID and EditOption

Figure 33 depicts that the training latency will be higher when the model has higher complexity because the more complex the model, the more processing/computation need to be done to compute the neurons' weight. In addition, we can see that the `resized-100x` trace needs longer training time because it generates much larger dataset (4x bigger).

8.2.18 Training Latency across Different ModelID and DeviceName

Based on Figure 34, the type of the devices doesn't provide much difference to the training latency. Therefore, the bar-cluster per `DeviceName` has similar trends.

9 Related Work

One of the main factors of the tail latency problem is the unpredictability of the systems in a modern cloud environment. Currently, there has been a lot of research on the topic of unpredictable latency on SSDs. Multiple studies show how to reduce SSD latency by utilizing internal characteristics of the disks using an analytics method. The difference between IONet and other related research is that IONet framework allows us to model storage devices in such a way that we

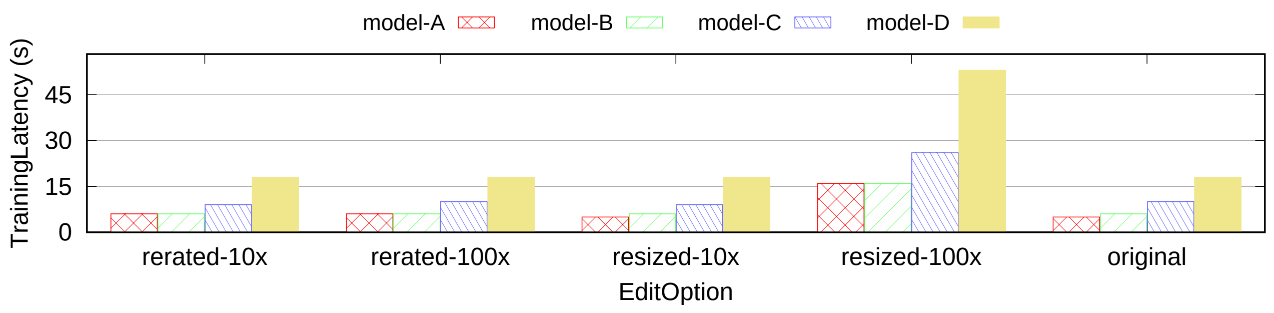


Figure 33: **Training latency across different ModelID and EditOption.** `TraceType =Azure`, `DeviceName =nvme3n1`

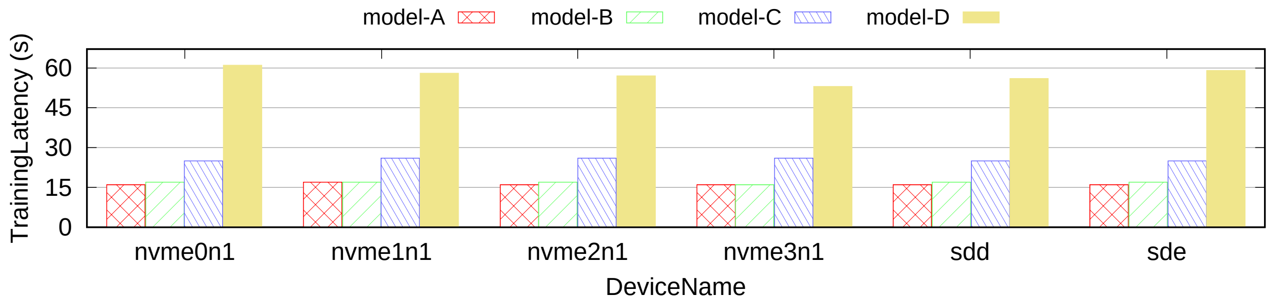


Figure 34: **Training latency across different ModelID and DeviceName.** `EditOption =resized-100x`, `TraceType =Azure`

can predict the latency of every I / O of a full workload running on a target storage cluster without actually running it on the cluster. IONet also only causes very little inference overhead thus it is feasible to integrate the models inside operating systems. Below are the summaries of related works of this paper.

Rails [54] introduces an approach based on redundancy that physically separates reads from writes to achieve read-only performance in the presence of writes. It says that SSDs frequently block in the presence of writes, exceeding hard-drive latency and leading to unpredictable performance. The proposed solution eliminates the high latencies, therefore providing read-only response time that is low and predictable. **FlashShare** [64] says that replacing the storage devices of servers with Ultra-Low-Latency (ULL) SSDs does not typically reduce the latency of I/O services, especially when co-running multiple applications. It proposes a holistic cross-stack approach, which can significantly reduce I/O interferences among co-running applications at a server without any change in applications. **TTFlash** [63] describes a garbage collection system that eliminates GC-induced tail latencies by circumventing GC-blocked I/Os. It ensures that the GC is only 1.0 to 2.6× slower than the no-GC case, while a base approach suffers from 5–138× GC-induced slowdowns. **Harmonia** [37], similar with ttFlash, proposes a Global Garbage Collection (GGC) mechanism using an approach to improve response times and reduce performance variability for a RAID array of SSDs. It states that the frequency of garbage collection (GC) activity is directly correlated with the pattern, frequency, and volume of write requests, and scheduling of GC is controlled by logic internal to the SSD.

KAML [24] states that modern solid state drives (SSDs) do not need to restrict host programs to conventional block I/O interfaces, leading to under-optimal performance and under-usage of resources. It presents the key-addressable, multi-log SSD with a key-value interface that uses a novel multi-log architecture and stores data as variable-sized records rather than fixed-sized sectors. The results show that KAML improves the performance of online transaction processing (OLTP) workloads by 1.1x - 4.0x, and NoSQL key-value store applications by 1.1x - 3.0x. **Swan** [30] says that the main source of performance degradation is garbage collection (GC). It proposes a solution that can reduce the performance interference caused by GC at SSD-level and AFA software-level. This approach ensures the storage bandwidth always matches the full network performance without being interfered by AFA-level GC. **Z-Map** [58] designs a novel space management and address mapping scheme for flash which manages flash space at granularity of Zone (multiple numbers of flash blocks). It classifies data before it is permanently stored into Flash memory thus isolating different workloads and reducing garbage collection overhead. **Gauge** [23] introduces a data-driven diagnostic tool for exploring the latent space of supercomputing job features, understanding behaviors of clusters of jobs, and interpreting I/O bottlenecks. Gauge can detect families of applications and spot strange I/O behavior that may require further finetuning and optimization of these applications, hardware provisioning, or further investigation. Gauge provides novel information that leads to new insights, but it still requires guidance from a domain expert.

10 Conclusion

In this paper, we introduce IONET: ML-based per-I/O latency predictor capable of achieving 80-97% inference accuracy and sub-10µs inference overhead for each I/O. The neural network is built by training real-life production systems dataset in order to learn SSDs as black-box devices. Furthermore, this paper provides a complete analysis of the training result that give many insights about how the models behave across different traces and devices, and also how to improve the model's accuracy. With IONET framework, we can build models of storage devices in such a way that we can predict the latency of every I/O of a full-workload running on a target storage cluster without actually running it on the cluster. In the future, IONET will allow storage-system researchers to benefit from and contribute to the IONET project, which then spurs more solutions in this new research space.

References

[1] http://iotta.snia.org.

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

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

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

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

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

[7] Cristina L. Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H. Campbell. Metadata traces and workload models for evaluating big storage systems. In *UCC*, 2012.

[8] Simona Boboila and Peter Desnoyers. Performance Models of Flash-based Solid-state Drives for Real Workloads. In *MSST '11*.

[9] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the Performance Variation in Modern Storage Stacks. In *FAST '17*.

[10] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing. In *HPCA-17*.

[11] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-defined Network-attached Key-value Drives. In *ASPLOS '19*.

[12] 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 *SIGMOD '15*.

[13] M. Cordle, C. Rea, J. Jury, T. Rausch, C. Hardie, E. Gage, and R. H. Victora. Impact of radius and skew angle on areal density in heat assisted magnetic recording hard disk drives. *AIP Advances*, 8, 2018.

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

[15] Peter Desnoyers. Analytic Models of SSD Write Performance. In *TOS '14*.

[16] J. Gim and Y. Won. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. *ACM Transactions on Storage*, 6, 2010.

[17] 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 *SOSP '17*.

[18] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *OSDI '20*, 2020.

[19] hao Wu, Cheng Ji, Qiao Li, Congming Gao, Riwei Pan, Chenchen Fu, Liang Shi, and Chun Jason Xue. Maximizing i/o throughput and minimizing performance variation via reinforcement learning based i/o merging for ssds. *IEEE Transactions on Computers*, 69(1), 2020.

[20] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *EuroSys '17*.

[21] 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 *FAST '17*.

[22] Amber Huffman. Addressing IO Determinism Challenges at Scale with NVM Express – Part 2: Renegotiating the Host/Device Contract. In *NVMW '17*.

[23] Mihailo Isakov, Eliakin del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip H Carns, Robert B Ross, and Michel A Kinsy. Hpc i/o throughput bottleneck analysis with explainable local models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'20, 2020.

[24] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A Flexible, High-Performance Key-Value SSD. In *HPCA-23*.

[25] 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.

[26] Won-Kyung Kang and Sungjoo Yoo. Value prediction for reinforcement learning assisted garbage collection to reduce

long tail latency in ssd. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10), 2020.

[27] Swaroop Kavalanekar, Bruce L. Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *IISWC*, 2008.

[28] Bryan S. Kim, Hyun Suk Yang, and Sang Lyul Min. AutoSSD: an Autonomic SSD Architecture. In *USENIX ATC '18*.

[29] Jae-Hong Kim, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs). In *MASCOTS '09*.

[30] 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 *USENIX ATC '19*.

[31] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-Aware I/O Management for Solid State Disks (SSDs). In *TC '11*.

[32] Jihun Kim, Joonsung Kim, Pyeongsu Park, Jong Kim, and Jangwoo Kim. SSD Performance Modeling Using Bottleneck Analysis. *IEEE Computer Architecture Letters*, 17(1):80–83, 2018.

[33] Jihun Kim, Joonsung Kim, Pyeongsu Park, Jong Kim, and Jangwoo Kim. Ssd performance modeling using bottleneck analysis. *IEEE Computer Architecture Letters*, 17(1), 2018.

[34] Joonsung Kim, Pyeongsu Park, Jaehyung Ahn, Jihun Kim, Jong Kim, and Jangwoo Kim. SSDcheck: Timely and Accurate Prediction of Irregular Behaviors in Black-Box SSDs. In *MICRO-51*.

[35] 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 *FAST '19*.

[36] 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.

[37] 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 *MSST '11*.

[38] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *ASPLOS '17*.

[39] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *SOSP '17*.

[40] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *FAST '15*.

[41] Nanqinqin Li, Mingzhe Hao, Xing Lin, Huaicheng Li, Levent Toksoz, Tim Emami, and Haryadi S. Gunawi. Fantastic SSD Internals and How to See/Use Them. In *In submission.*, 2020.

[42] 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 *ASPLOS '19*.

[43] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *OSDI '18*.

[44] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. Hiding the Microsecond-Scale Latency of Storage-Class Memories with Duplexity. In *NVMW '19*.

[45] Pulkit A. Misra, María F. Borge, Iñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *EuroSys '19*.

[46] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *NSDI '17*.

[47] Jun Nemoto and Gregory R. Ganger. On io latency prediction accuracy and automated load balancing in consolidated vm environments. In *IC2E*, 2016.

[48] Qais Noorshams, Axel Busch, Andreas Rentschler, Dominik Bruhn, Samuel Kounev, Petr Tuma, and Ralf H. Reussner. Automated modeling of i/o performance and interference effects in virtualized storage systems. *ICDCS Workshops*, 2014.

[49] David Patterson. Technical Perspective: For Better or Worse, Benchmarks Shape a Field. *Communications of the ACM*, 55(7), 2012.

[50] Chris Petersen. Addressing IO Determinism Challenges at Scale with NVM Express. In *NVMW '17*.

[51] J. Schindler and G. R. Ganger. Automated disk drive characterization. In *CMU SCS Technical Report CMU-CS-99-176*, 1999.

[52] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, C. Faloutsos, and G. R. Ganger. On multidimensional data and modern disks. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'05, 2005.

[53] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *OSDI '14*.

[54] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent Flash Performance through Redundancy. In *USENIX ATC '14*.

[55] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based extraction of local and global disk characteristics. Technical Report UCB/CSD-99-1063, EECS Department, University of California, Berkeley, 1999.

[56] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *ISCA '18*.

[57] Muhammad Wajahat, Aditya Yele, Tyler Estro, Anshul Gandhi, and Erez Zadok. Distribution fitting and performance modeling for storage traces. In *MASCOTS*, 2014.

[58] Qingsong Wei, Cheng Chen, Mingdi Xue, and Jun Yang. Z-map: A zone-based flash translation layer with workload classification for solid-state drive. *ACM Transactions on Storage*, 11, 2015.

[59] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of scsi disk drive parameters. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1995.

[60] 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.

[61] 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 *IPDPS '18*.

[62] 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 *SOSP '17*.

[63] 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 *FAST '17*.

[64] 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 *OSDI '18*.

[65] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *FAST '12*.