Quick Generation of SSD Performance Models Using Machine Learning

Mojtaba Tarihi, Soheil Azadvar, Arash Tavakkol, Hossein Asadi, Hamid Sarbazi-Azad

Abstract—Increasing usage of *Solid-State Drives* (SSDs) has greatly boosted the performance of storage backends. SSDs perform many internal processes such as out-of-place writes, wear-leveling, and garbage collection. These operations are complex and not well documented which make it difficult to create accurate SSD simulators. Our survey indicates that aside from complex configuration, available SSD simulators do not support both *sync* and *discard* requests. Past performance models also ignore the long term effect of I/O requests on SSD performance, which has been demonstrated to be significant.

In this paper, we utilize a methodology based on machine learning that extracts history-aware features at low cost to train SSD performance models that predict request response times. A key goal of our work is to achieve real-time or near-real time feature extraction and to achieve practical training times so our work can be considered as part of solutions that perform online or periodical characterization such as adaptive storage algorithms. Thus, we extract features from individual read, write, *sync*, and *discard* I/O requests and use structures such as exponentially decaying counters to track past activity using $O(1)$ memory and processing cost. To make our methodology accessible and usable in real-world online scenarios, we focus on machine learning models that can be trained quickly on a single machine. To massively reduce processing and memory cost, we utilize feature selection to reduce feature count by up to 63%, allowing a feature extraction rate of 313,000 requests per second using a single thread. Our dataset contains 580M requests taken from 35 workloads. We experiment with three families of machine learning models, a) decision trees, b) ensemble methods utilizing decision trees, and c) *Feedforward Neural Networks* (FNN). Based on these experiments, FNN achieves an average R^2 score of 0.72 compared to 0.61 and 0.45 for the Random Forest and Bagging, respectively, where $R^2 \in (-\inf, 1)$ of 1 indicates a perfect fit. However, while the random forest model has lower accuracy, it uses general processing hardware and can be trained much faster, making it viable for use in online scenarios.

Index Terms—Performance Prediction, Solid State Drives, Machine Learning, Neural Networks

1 Introduction

Current computer workloads require fast and responsive storage backends, which often rely on *Solid-State Drives* (SSDs) with superior performance to mechanical *Hard Disk Drives* (HDDs). Due to the ever-increasing storage requirements of computing workloads, it is essential to design fast and efficient hardware backends. This is especially important as storage is the bottleneck in many applications. Prediction of device latency and throughput can aid in offline and online configuration and re-configuration of backends [1], [2], [3] and implementation of performance-aware algorithms [4], [5].

Accurate prediction of SSD performance is, however, very challenging. SSDs are composed of many pages and must perform complex operations such as out-of-place writes, wear-leveling, and garbage collection to perform I/O requests. Fig. 1 demonstrates that long-running random workloads (with 4K requests and 8K requests respectively) greatly increase SSD response time as the free SSD capacity is consumed[1]. The only way to recover SSD performance in such scenarios is to invalidate SSD data using *discard* commands [6].

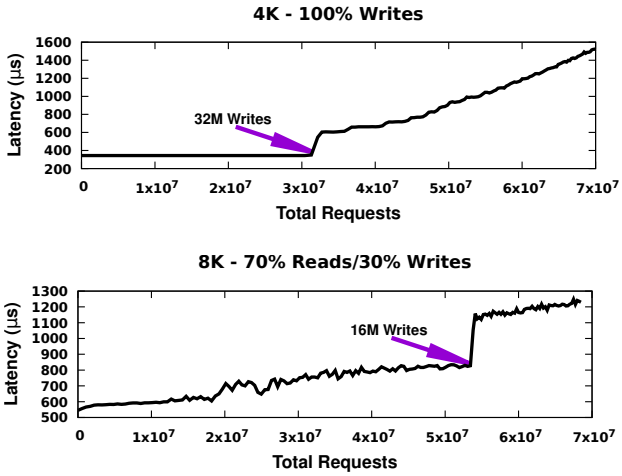


Fig. 1: SSD response time greatly increases as it is saturated by two highly random workloads.

Three main tools exist to quickly predict workload performance, a) real hardware experiments on representative

- *Mojtaba Tarihi, Soheil Azadvar, Hossein Asadi, and Hamid Sarbazi-Azad are associated with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.*
- *Hamid Sarbazi-Azad is also associated with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran*
- *Arash Tavakkol is associated with the Department of Computer Science, ETH Zurich, Switzerland.*
E-mails: tarihi@ce.sharif.edu, azadvar@ce.sharif.edu, arash@tavakkol.ch, asadi@sharif.edu, azad@ipm.ir

1. The points where the total volume of writes reaches the advertised SSD capacity (128GB) are shown on both charts. It must be noted that the device under test, a 128GB Samsung 850 Pro has 192GB raw flash capacity (a.k.a. overprovisioned capacity).

workloads [7], [8], b) hardware simulators [9], [10], [11], [12], and c) performance prediction models [13], [14], [15]. As past workload activity affects future SSD performance [6], [16], [17], [18], approaches such as sampling and replaying of short intervals [7], [8] cannot be used. Furthermore, real and simulated experiments must perform preconditioning [12], [19], [20] and any model must consider past request history when predicting response time. Due to these limitations, we believe it is very challenging to quickly perform small and representative hardware experiments.

Simulators are a widely-used alternative to real experiments in many fields. However, implementation of accurate SSD simulators is challenging due to poor availability of hardware details and the high complexity of SSD devices. Detailed simulators such as MQSim [12] and SSDSim [11] require more than sixty parameters, which include hardware structure, timing, and firmware behavior while almost *none* of these parameters are published by manufacturers. Furthermore, the sheer complexity of SSDs means that even well-developed simulators have very severe limitations. For example, out of four well-known simulators [10], [11], [12], [21], a) *none* of them support all the four main command types of *read*, *write*, *sync*, and *discard*, and b) *none* of them support asymmetric flash topologies such as our device under test, a 128GB Samsung 850 Pro, which uses two pairs of distinct flash chips with a raw capacity of 192GB.

Modeling can forgo many implementation details while offering sufficient accuracy. However, an SSD performance model must **a)** consider I/O workload characteristics such as spatial and temporal locality, which can affect SSD performance by an order of magnitude[2], **b)** track past I/O activity due to its impact on performance [4], [5], [14], [15], [18], and **c)** analyze *sync* and *discard* commands, which can greatly affect SSD performance. To our knowledge, *no* previous SSD response time models have fulfilled any of these requirements.

In this paper, we present a methodology using machine learning models to generate response time prediction models for a target SSD device based solely on recorded traces. Our methodology, a) does not require the user to know device-specific parameters, relying on I/O traces instead, b) utilizes space-efficient data structures such as decaying counters and simulated queue to extract request history so that spatial and temporal locality can be estimated, c) analyzes *sync* and *discard*, which have key impact on SSD performance, d) uses neural networks and ensemble models for the first time to predict storage hardware response time, and e) performs feature scoring to compare feature importance and to remove under-performing and redundant features.

In the proposed methodology, as shown in Fig. 2, traces are analyzed in **Feature Extraction** to generate per-request *feature vectors* describing request type, past activity, spatial, and temporal locality. We utilize the space-efficient and high performance exponentially decaying counters in place of tracking methods based on sliding windows which require a potentially unbounded amount of space.

We split the output of feature extraction into three subsets: training, validation, and testing. The training and validation sets are used in the next stage called **Model and Feature Selection**. This process selects model and their parameters so practical training time and acceptable accuracy can be achieved. Also, feature selection eliminates features with limited effect on accuracy to drastically reduce feature extraction and model training costs.

The process of feature selection is repeated multiple times to eliminate unnecessary features. These models are then used in the final evaluation to predict trace response times. Three main types of machine learning models are used: a) decision trees, b) *Feedforward Neural Networks* (FNNs), and c) ensemble models based on decision trees. The three ensemble models tested are: Adaboost, Bagging[3], and Random Forest. Out of these models, only decision trees have previously been used to predict HDD [13] or SSD [14], [15] response times. Alternatives such as *Support Vector Regression* (SVR) models and *Recurrent Neural Network* (RNN) were also considered but were not included due to their low learning rate even with hardware acceleration.

Compared to past performance modeling methods relying on synthetic traces [14], [15] or HDD trace replay [15], we utilize a comprehensive set of traces recorded from a real SSD running various workloads. These traces include 580M requests and originate from workloads such as workstation, server, database, and synthetic benchmarks. All four main request types of *read*, *write*, *sync*, and *discard* are captured in these traces.

We calculate prediction accuracy by two metrics, a) R^2 score or the *Coefficient of determination* ($R^2 \in (-\inf, 1)$), and b) *Mean Absolute Error* (MAE). An R^2 score of 1.0 refers to a perfect fit and a score of 0.0 is achieved when every prediction is equal to the average trace response time. Our experiments demonstrate that while feature selection has negligible effect on model accuracy, it can boost learning rate on models utilizing decision trees by over six times while greatly reducing their memory requirements, allowing ensemble methods to run their predictors concurrently. Finally, our FNN model achieves an average R^2 score of 0.72 against 0.61 and 0.45 for random forest and bagging models, respectively. Overall, we can highlight the following major **contributions** in this work:

- We introduce a comprehensive set of features that can be calculated with constant memory and $O(1)$ per-request cost using space-efficient data structures. Through feature selection, we eliminate over 60% of features to reach the throughput of 313,000 feature vectors per second.
- For the first time, we use FNNs and ensemble methods to predict SSD response time. Compared to decision trees, which have been previously used for this purpose [14], [15] and achieve an extremely poor R^2 score, FNN, random forest, and bagging models achieve an average R^2 score of 0.72, 0.61, and 0.45, respectively.
- The final random forest model can be trained at a rate of ≈11000 requests per second by using 10 threads, meaning that as much as ten seconds can be used to train a model over a hundred thousand requests. This, coupled with the high feature extraction rate

2. This will be discussed in Sec. 2.

3. Bootstrap Aggregating

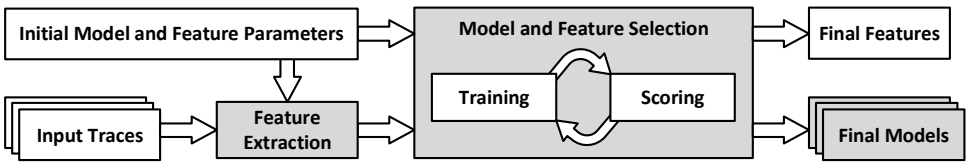


Fig. 2: The overall process for generating prediction models. I/O traces are used to generate feature vectors which are split into the training, validation, and testing sets. The training and validation process are used to select the final model and feature set.

allows periodic re-training of performance prediction models on live systems for use in performance-aware models.
- Our work predicts SSD performance utilizing all four main request types of *read*, *write*, *sync*, and *discard*. No past works on modeling [14], [15], [18] or simulation [10], [11], [12], [21] of SSDs considers all four request types.
- Unlike past studies that rely either on synthetic workloads [14], [22] or replays of real workloads originally from HDDs [15], [18], [23] for training and evaluation, we utilize 35 traces containing 580M requests recorded from a real SSD running database, server, workstation, and benchmark applications [4]

In the remainder of this paper, we describe our feature extraction methodology in Sec. 2. Sec. 3 describes the selection process for machine learning models and feature types and parameters. Sec. 4 compares the accuracy of the selected four models at predicting response time. A survey of past related research is done in Sec. 5. Finally, we conclude the paper in Sec. 6 and discuss the future directions of our work.

2 Extraction of Features

Modern flash-based SSDs are constructed out of NAND flash memory, which is read and written in *page* granularity but must be erased in larger, multi-page *erase blocks* before being written to. SSD firmware performs many internal operations in order to maximize device performance and endurance while presenting SSDs as standard storage devices to the host computer. These operations include mapping logical to physical addresses, invalidation of pages holding old values, garbage collection of partial or fully invalidated erase blocks, and wear leveling. Due to these internal operations, past workload behavior has long-lasting performance effects [6], [16], [17], [18] and any SSD performance models must account for past activity.

While flash-based SSDs store data as electrical charges and do not suffer from the slow, mechanical seek operations of HDDs, inter-request distance can greatly affect SSD performance. Table 1 compares the bandwidth of a number of HDD and SSD devices under three synthetic benchmarks. In this table, a) **2MB Sequential** is composed of 2MB blocks that exactly follow each other and have maximum spatial locality, b) **2MB Random** is also entirely made of 2MB blocks but with random starting addresses, and c) **4K Random** is created by sending 4KB requests to random addresses, resulting in very poor spatial locality. The results demonstrate clearly that it is important to have features that consider request type and measure workload spatial locality.

4. This work is essential as available SSD traces lack response time [24] or are recorded from *Embedded Multi-Media-Card* (eMMC) devices [25]

TABLE 1: Comparison of random and sequential performance for a number of storage devices. All data are taken from StorageReview [26].

Device	Capacity	Bandwidth Under Test (MB/s)					
		2MB Sequential		2MB Random		4K Random	
		Read	Write	Read	Write	Read	Write
Samsung 850 PRO SSD	2TB	496	472	486	473	40	116
Corsair Neutron XT	960MB	514	471	403	467	36	74
WD Black HDD	6TB	215	215	78	107	0.3	0.8
WD Blue Hybrid HDD	4TB	143	143	54	66	0.2	0.5

A workload with high spatial locality makes accesses to pages that have close addresses, meaning they are placed in the same erase blocks and are invalidated together. This results in more efficient garbage collection operations and lower erase rate. High temporal locality can also be exploited by SSD write buffering and *Flash Translation Layer* (FTL) policies [18]. In other words, both spatial and temporal locality must be taken into account when generating SSD performance models.

While for qualitative description, a workload may be described as being *Random*, *Sequential*, or *Bursty*, binary or numerical *features* must be generated for use in machine learning. Due to the highly bursty nature of I/O workloads, placing I/O requests in time-based intervals may result in extremely dense or sparse intervals. Such request grouping strategies discard valuable per-request information and lead to varying request density across intervals, which will likely require weighting for intervals [8], [27]. To avoid all these issues, we choose to perform per-request feature extraction and response time prediction.

To make this methodology practical, features must be extracted in a way that minimizes computation and memory costs. After evaluating many previously introduced quantitative measures [8], [28], [29], [30], we come up with a set of features which can be calculated with $O(1)$ per-request processing time with constant memory utilization. Such requirements reject many algorithms as *Stack Distance* [28] or the *Hurst Exponent* [31] (used in [32]) due to the non-constant memory/processing costs. Similarly, unless *Shannon Entropy* (used in [8], [29]) is always used with a constant number of elements, computationally expensive logarithmic operations are needed.

For the rest of this section, we begin by describing the three main feature extraction methods used in this study. We then describe all the features used in this study and their

initial parameters. Finally, we describe our implementation of feature extraction.

2.1 Feature Extraction Methods

We use three main methods to generate features for device performance models: a) **Queue-Based Spatial Distance**, which uses a simulated queue to measure the per-request minimum *spatial distance* and classify each request, b) **Decaying Counters** that measure key properties of past requests with a configurable decay rate so recent requests are assigned more importance, and c) **Locality Score**, which uses an array of decaying counters to measure temporal locality.

2.1.1 Queue-Based Spatial Distance

To analyze spatial locality, it is necessary to compare request address and length against preceding requests. In this method, shown in Fig. 3 and first introduced by Ahmad *et al.* [33], the address of each incoming request (R_N) is compared against a fixed number (Q) of preceding requests. We refer to the minimum address difference as *Spatial Distance*. The spatial distance is a representative of workload spatial locality and is resistant to disruptions caused by multiple concurrent I/O streams.

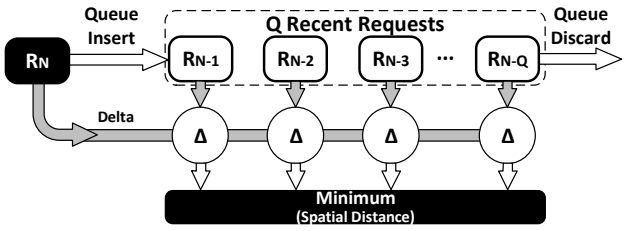


Fig. 3: Calculation of minimum *Spatial Distance*, R_i is the ith request and A_i is its address.

To classify requests based on their spatial distance, Tarihi *et al.* [30] compare the spatial distance against request length and a parameter called the *randomness threshold* to classify requests into four categories: *Sequential*, *Random*, *Overlapped*, and *Strided*. Overlapped requests intersect one of the preceding requests whereas Sequential requests immediately follow one. Strided requests have a spatial distance less than the *randomness threshold*, and the remaining requests are classified as Random. A more formalized definition of these request types as calculated in this paper can be seen in Equation 1 to Equation 4. In these equations, i is the current request, A_i is the request address, L_i is request length, $md(i,k)$ is the truncated spatial distance, k_{min} is k where $md(i,k)$ is minimum, RT is the randomness threshold, SD_i is the spatial distance, and each request is compared against Q previous requests. The reason for using a minimum distance of $2\times RT$ for random requests is to clearly differentiate the spatial distance of random requests while avoiding the huge numbers resulted by subtracting very large address differences.

$$\Delta_{i,k} = A_i - (A_k + L_k) \tag{1}$$

$$md(i,k) = \begin{cases} 2 \times RT & \text{if } RT < \Delta_{i,k} \\ 2 \times RT & \text{if } \Delta_{i,k} < 0 \text{ and } A_i < A_k \\ 0 & \text{if } \Delta_{i,k} < 0 \text{ and } A_k \leq A_i \\ \Delta_{i,k} & \text{if } 0 \leq \Delta_{i,k} \leq RT \end{cases} \tag{2}$$

$$SD_i(Q) = min_{k=i-Q}^{i-1} md(i,k) \tag{3}$$

$$\begin{cases} Overlapped & \text{if } SD_i(Q) = 0 \text{ and } \Delta_{i,k_{min}} < 0 \\ Sequential & \text{if } SD_i(Q) = 0 \text{ and } \Delta_{i,k_{min}} = 0 \\ Strided & \text{if } 0 < SD_i(Q) < RT \\ Random & Otherwise \end{cases} \tag{4}$$

2.1.2 Decaying Counters

Not only SSD *state space* is significantly larger than that of HDDs, it is also affected by workload history [6], [16], [17], [18]. Furthermore, as user request and processing requirements vary over time [34], so does the spatial and temporal locality of the I/O workload [35] which requires feature measurements to be constantly updated. To track past workload activity in a simple manner, we rely on exponentially decaying counters.

Exponentially decaying counters require a single counter to aggregate past workload activity while assigning higher importance to more recent requests. The rate of *forgetting* about past requests can be controlled by the *decay rate*. This method has a very small cost of keeping a single value and its latest update time and can be iteratively updated with a single multiplication and accumulation operation. In contrast, sliding windows require the storage of an unbounded number of requests for tracking the requests falling within the window. This greatly reduces the methodology cost.

Equation 5 calculates the value of an exponentially decaying counter. In this equation, i events have arrived before t, t_k ($1 \leq k \leq i$ and $t_k \leq t$) is the arrival time of the event k, and b denotes the exponential decay rate. As time passes by, a) more events arrive (i increases), and b) current counter value decays exponentially. The decaying counter $x(t)$ does not make use of a scaling factor (such as a in $a \times e^{-b\Delta t}$) as the output of decaying counters are stored in independent feature vector dimensions and thus, are unaffected by linear scaling and offsets.

$$x(t) = \sum_{k=1}^{i} e^{-b(t-t_k)} \tag{5}$$

The benefit of using Equation 5 is the simplicity of iterative updates especially in our case where counter values are updated and read upon the arrival of I/O requests. As seen in Equation 6, counter value at $t_{i+1} = t_i + \Delta t$ can be calculated based on the previous counter value. Thus, on arrival of each I/O request, new values of decaying counters can simply be calculated based on Δt and the existing counter value.

$$x(t_{i+1}) = x(t_i + \Delta t) = e^{-b\Delta t} x(t_i) + e^{-b(t_{i+1} - t_{i+1})} = e^{-b\Delta t} x(t_i) + 1 \tag{6}$$

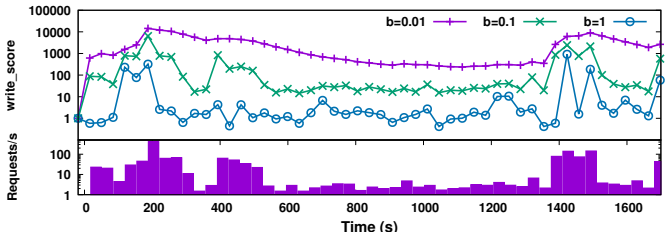


Fig. 4: Calculation of `write_score` using different decay rates (b) (top) and write request arrival rate (per seconds) over time (bottom), y scale for both is logarithmic.

Further expanding on this concept, weighted exponential score is defined in Equation 7 where the score of request k is scaled by its length, L_k. In our work, L_k is the request size, used to calculate request length-dependent features such as the weighted write score (`write_score_w`).

$$x(t) = \sum_{k=1}^{i} L_k e^{-b(t-t_k)} \tag{7}$$

We utilize order-based decaying counters to study temporal locality. These counters decay based on request order k instead of arrival time t_k. The decaying counter can be calculated and updated with the α decay rate with no regard to time. Thus iterative updates can be simplified to $x'(i+1) = \alpha x'(i) + 1$. We use this form of decaying counter in Sec. 2.1.3 to estimate temporal locality.

The behavior of decaying counters is demonstrated in Fig. 4 by plotting `write_score` with multiple values of b over the first 30 minutes of an I/O workload. `write_score` is one of features used in our study[5] to characterize past I/O write activity and is incremented by one on arrival of each write request while decaying by time. The bottom part of Fig. 4 shows the arrival rate of write requests (per second) during these 30 minutes. It can be seen that a high decay rate causes `write_score` to quickly fall down and resemble the bottom histogram whereas low decay rates follow a much smoother pattern.

2.1.3 Locality Score

Temporal locality is the probability of re-access to recently accessed data. Many storage workloads are well-known for having a highly active or *hot* subset of data [30], [35], [36] and presence of hot data allows for specific performance optimizations [18].

While individual decaying counters are sufficient for tracking request count and length, it is essential to use multiple counters to estimate temporal locality. To accomplish this, we use an array of decaying counters in an arrangement similar to Bloom Filters. On each request, a counter is selected using its hashed request address and is incremented (Fig. 5). We use the *Murmur3* hashing algorithm due to its popularity and speed. The selected decaying counter is updated using Equation 6 and its value becomes the request `locality_score`.

As requests with the same offset are routed to the same bins by the hash function, hot data continuously update a certain subset of the decaying counters. This results in a skewed distribution of bin values and higher `locality_score` for the popular offsets. While we directly use `locality_score` as a feature, we also generate a feature based on the variance of all the decaying counters. Such a feature is very useful for estimating temporal locality in the workloads as random workloads tend to distribute accesses across all bins, resulting in nearly equal counter values and low variance. The decaying counters used for `locality_score` are order-based with a per-request decay rate of α.

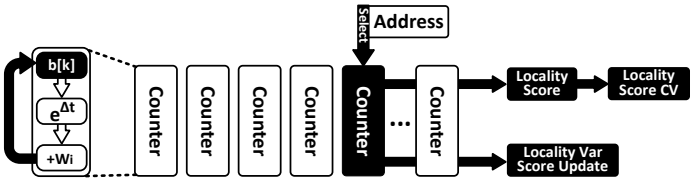


Fig. 5: Calculation of `locality_score` and `locality_score_cv` using request offset.

To calculate variance online, we simplify the online calculation of variance to yield Equation 8[6]. In this formula, i is the request number, σ_i^2 is the variance of decaying counters after request i, N is the number of decaying bins, α is the linear decay, h_i is the bin selected for request i using the hash function, μ_i is the mean of all N counters after request i, and $B_{h_i,i}$ is the value of bin h_i after i where $B_{h_i,i} = \alpha B_{h_i,i-1} + 1$ and $B_{j,i} = \alpha B_{j,i-1}$ for $j \neq h_i$. It must be noted that μ_i itself can be iteratively updated using Equation 9.

$$\sigma_i^2 = \alpha^2 \sigma_{i-1}^2 + \frac{2\alpha(B_{h_i,i-1} - \mu_{i-1})}{N} + \frac{N-1}{N^2} \tag{8}$$

$$\mu_i = \alpha \mu_{i-1} + \frac{1}{N} \tag{9}$$

As can be seen, Equation 8 only relies on bin value $B_{h_i,i-1}$ alongside variables that can be calculated without knowing bin values. To avoid updating every bin on arrival of each request, we use a deferred update method. In this scheme, $B_{h_i,i-1}$ is calculated based on its most recent update (the largest value of $j < i$ where $h_j = h_i$ or $j = 0$ where $B_{h_j,0} = 0$ if i is the first reference to B_{h_i}) using Equation 10, meaning that processing cost of Equation 8 is not affected by N and is $O(1)$. Thus, only a single exponentiation with a base of α is required to calculate `locality_score` variance. Since variance is based on squares and can generate large numerical values, we calculate coefficient of variation (CV) as $CV = \frac{\sigma}{\mu}$.

$$B_{h_i,i-1} = B_{h_i,j} \alpha^{j-i+1} \tag{10}$$

Algorithm 1 describes the implementation used to calculate `locality_score`. After request index is incremented (Line 2), its hash selects the target bin h (L.3). The distance (d) between the last updated to bin h using the lu array is measured (L.4) and lu is updated (L.5). Line 6 calculates the deferred decay based on d and it is applied in the next line.

5. The full list of features will be described in detail in Sec. 2.2

6. Please refer to the appendix for detailed equations.

Online variance and online mean are updated in L.8 and L.9, respectively. `locality_score` and `locality_score_cv` are calculated in the last two lines. Our experiments show negligible difference between our iterative and non-iterative implementations.

Algorithm 1 Calculating `locality_score` and `locality_score_cv`

global variables
N, Number of decaying counters
σ, Online counter variance
μ, Online counter mean
α, Decay rate (constant)
$b[h]$, Bin values $1 \leq h \leq N$
$lu[h]$, last update to bin h, $1 \leq h \leq N$
c, Request counter
LS, Per request `locality_score`
$LSCV$, Per request `locality_score_cv`
end global variables

1: **function** GETLOCALITYSCOREANDCV($offset$)
2: $i \leftarrow i + 1$
3: $h \leftarrow hash(offset_i) \mod N$
4: $d \leftarrow i - lu[h]$
5: $lu[h] \leftarrow i$
6: $b' \leftarrow b[h] * \alpha^{d-1}$
7: $b[h] \leftarrow \alpha b' + 1$
8: $\sigma^2 \leftarrow \alpha^2\sigma^2 + \frac{2\alpha(b'-\mu)}{N} + \frac{N-1}{N^2}$
9: $\mu \leftarrow \mu\alpha + \frac{1}{N}$
10: $LS[i] \leftarrow b[h]$
11: $LSCV[i] \leftarrow \frac{\sqrt{\sigma^2}}{\mu}$
12: **end function**

Here, we perform numerical analysis on the two measures introduced in this section. Although temporal locality and presence of *hot* data are not necessarily the same [35], a simplified form of analysis for temporal locality is working with address access frequency [18]. Here, we calculate expected value of bins based on this form of analysis. It must be noted that both these locality scores *do* record long- and short-term temporal locality and their relative effect on the score varies by the decay rate.

In any workload where *hot* data exist, data access frequency will be uneven. Equation 11 shows the expected value of bin b_k after i accesses based on its access frequency w_k[7]. In other words, assuming a large enough value of i, relative ratio of bins x and y is equal to $\frac{w_x}{w_y}$. In workloads with hot data, accesses to hot data will have a greater `locality_score`. Furthermore, the uneven w_k weights directly increase bin value variance and `locality_score_cv`. A more generalized case of this equation is also covered in the appendix.

$$E(B_{k,i}) = w_k \frac{1-\alpha^i}{1-\alpha} \tag{11}$$

To provide a more intuitive example on the output of `locality_score_cv`, we create a synthetic dataset with the zipf distribution. To this end, nine batches containing 100,000 addresses each are generated with different zipf parameters and are concatenated. We calculate `locality_score_cv` with four different decay rates and graph the results in Figure 6. For each of the nine batches, the zipf parameter a and the resulting distribution are described. **Uniq.%** denotes the number of unique addresses in the batch and a lower percentage of unique values translates into higher temporal locality which is correlated with the rise of the `locality_score_cv`.

7. This equation is obtained through induction and can be seen in the appendix.

2.2 Overview of Features

Table 2 shows a list of the features used in this study. As many of the features used in this work are parameterizable, we refer to each unique combination of feature parameters as a *feature instance*. A list of these parameter values is shown under the third column. The constant memory needed by each feature instance and the cost for processing a single I/O request are shown in the fourth and fifth columns, respectively.

Basic features are directly extracted from request traces, with very little cost. `disk_usage` is based on file-system reported disk utilization, which can be taken as the percentage of logical disk address space in use[8]. `disk_usage_rate` is the rate of change for `disk_usage`. *Decaying* and *Weighted Decaying* features use decaying counters to track the number of requests and the traffic involving *read*, *write*, *sync*, and *discard* commands. For example, `write_score` counts the number of write requests using an exponentially decaying counter with a decay rate of b. In the weighted version `write_score_w`, a weighted exponentially decaying counter is used and request length is used as a weight.

The features in the *Spatial* category rely on the queue-based methodology described in Sec. 2.1.1, which yields minimum spatial distance and request classification. `seq_d_score` is a decaying counter that is incremented whenever a *sequential* request is encountered, serving as an indicator of long term sequentiality. `seq_d_wscore` performs the same but is weighted by request length.

The *Temporal* features estimate temporal locality and are calculated using an array of decaying counters (Fig. 5). `locality_score` estimates the recency of past accesses to the target of the current request while `locality_score_cv` measures lack of uniformity in address accesses. `mlocality_score` and `mlocality_score_cv` are similar but operate on the larger granularity of 4MB blocks.

Both the basic and spatial features were widely used as I/O workload features [8], [14], [15], [30], [37]. Furthermore, empirical observations (see Table 1) reveal the effect of request type and spatial locality on SSD performance. In addition, SSD controllers can benefit from temporal locality for optimizing performance [18]. Therefore, we believe that the presented features can be used to model SSD performance. In Section 3.4, the effect of features on SSD performance models utilizing *permutation importance* is analyzed.

To arrive at the final set of feature parameters, we begin with a large array of feature instances that are created by a combination of the parameters shown in Table 2. The *spatial* feature instances have multiple parameter variables and all possible combinations are instantiated. For example, `seq_d_score` is instantiated with $(RT, Q, b) \in \{(512, 2, 0.9), (512, 2, 0.99), ..., (128KB, 32, 0.9999)\}$. The starting feature instances are then scored and potentially eliminated in a process that will be described in the next section.

8. Modern filesystems declare unused space via *discard* commands. This means that unused file-system capacity is available to SSDs as free blocks.

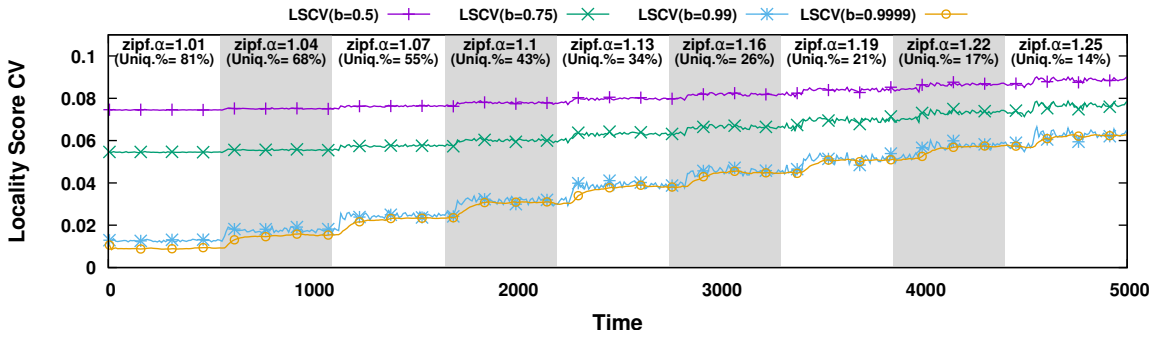


Fig. 6: `locality_score_cv` for a synthetic dataset. The dataset has nine phases with different percentages of address uniqueness (denoted by **U. pct**).

TABLE 2: Overview of Features Declared in This Work

Category	Feature	Feature Parameters				Memory[1]	Cost[2]
		Params.	Values				
Basic	`lengths`	N/A	N/A			N/A	1
	`is_read`[3]						
	`is_write`						
	`is_sync`						
	`is_trim`						
	`disk_usage`						
	`disk_usage_rate`						
Decaying	`sync_score`	b	0.0001, 0.001, 0.01, 0.1, 1			1 Counter	1
	`read_score`						
	`write_score`						
Weighted Decaying	`trim_score_w`		0.0001, 0.001				
	`read_score_w`		0.001, 0.01, 0.1, 1, 10				
	`write_score_w`						
Spatial	`min_distance`	RT, Q, b	RT	Q	b	Q requests	Q
	`is_sequential`		512 4KB 128KB	2, 8, 32	N/A		
	`is_overlapped`						
	`is_random`						
	`is_strided`						
	`seq_d_score`				0.9, 0.99 0.999, 0.9999	1 Counter [4]	1^4
	`seq_d_wscore`						
Temporal	`locality_score`	N, α	N	α		N Counters	**1**
	`mlocality_score`		512	0.5, 0.7, 0.9, 0.99, 0.999, 0.9999			
	`locality_var_score`						
	`mlocality_var_score`						

[1] The constant memory required for each feature instance.
[2] The processing cost for a single I/O request as a function of the supplied parameters.
[3] The boolean `is_*` features are stored as either 1 or 0.
[4] The `seq_d_` are essentially decaying counters for requests classified as sequential (`is_sequential`) and only require an extra decaying counter.

2.3 Implementation of Feature Extraction

As our input workloads contain varying numbers of requests[9], we process the workload traces in batches to avoid running out of memory. We implement a flexible framework illustrated in Fig. 7 to process requests in batches for feature extraction and evaluation of accuracy. Care is taken to preserve state across batches so values are generated as if the workload is generated using a single pass.

Our feature extraction stage uses request address, type, and length, as well as free disk space and its rate of change to calculate features. As our traces periodically record available free space, linear interpolation is used to yield the approximate available free space and its rate of change during each request. The per-request information is then

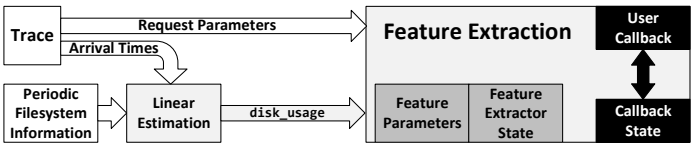


Fig. 7: Feature extraction process, the user supplies the feature extraction parameters as well as a callback routine to run on each batch of generated feature vectors.

processed in batches by feature extractors. We use a batch size of 100K requests for all experiments.

Most of the feature extraction is implemented in Python using Numpy, Scipy, and pybind11. Cython and C++ are also used for performance critical code. After the feature vector for each batch of requests is generated, the framework runs the user supplied callback (shown in Fig. 7) with the feature vector, request response times, and a `dict`

9. Eleven workloads contain over 10M requests including `mysql-tpch-run` with 189M requests.

holding the callback state, which is updated by callbacks and is preserved across calls. We use a single callback to save the training, validation, and testing sets which are used to train, optimize, and evaluate the machine learning models.

3 Model and Feature Selection

Choice of model and features used to train can result in vastly different degrees of accuracy and efficiency. For our particular problem, each workload may contain millions of requests where each request has a rather large number of requests. Although we utilize sampling to reduce the size of training and testing data, to create a practical methodology, it is essential to make use of machine learning models that have low memory and processing requirements. To this end, we evaluate three main categories of models: a) decision trees, which have served as the basis for all past works predicting hardware response time, b) ensemble models of random forest, bagging, and Adaboost based on decision trees, and c) Feedforward Neural Networks (FNNs).

In this section, we begin by describing the methodology used to prepare the input data for generation, optimization, and the final evaluation of the model. After that, we evaluate and choose the machine learning models for the evaluation section, and finally, we perform feature selection to reduce memory requirements, increase feature extraction rate, and drastically improve the learning rate of ensemble and decision tree methods.

3.1 Preparation of Input Data

Our work utilizes SSD traces for model training, feature selection, and evaluation. Very few SSD-based traces are publically available but none of them are suitable for our work. Yadgar *et al.* [24] recorded two billion requests, but their traces lack response time, which is essential for training and evaluating performance prediction models. Zhou *et al.* [25] recorded traces from eMMC devices, which have a much lower performance than standard SSD devices. These traces also lack *sync* and *discard* commands. As an input to the feature extraction process, we implement a custom and cross-platform low-level I/O tracing tool called *AccuTrace*. AccuTrace captures all I/O requests sent to hardware as SCSI [10] commands and records them alongside periodic SMART [11] statistics and high-level system information. AccuTrace utilizes drivers in both Linux and Windows to achieve this purpose. It has been validated to have less than 1% performance variation between Windows and Linux on similar synthetic workloads and has been used to record I/O traces. The benefit of using this tool is the fact that a similar cross-platform representation is being used which is very close to hardware and has fairly accurate measurements.

We use 35 recorded workloads containing 580M read, write, *sync*, and *discard* requests. To our knowledge, this is the most extensive set of SSD workloads in the literature. To be comparable, all traces are recorded on a single Samsung 850 Pro SSD with 128GB of usable capacity[12]. These traces are recorded from a variety of applications including OLTP, benchmark, productivity, and scientific applications. The applications are as follows:

- ATTO and pts-disk[13]: Disk benchmark applications. QD refers to the configurable queue depth in ATTO.
- auctionmark, epinions, tatp, twitter, and wikipedia: Various server workloads simulated using OLTP-Bench [39].
- TPC-C, TPC-E, and TPC-H: Well-known OLTP benchmarks. Trace names include the applications used to run the benchmark. For example, psql, mysql, and mssql refer to the database backend used in the benchmark.
- Jetstress [40]: Benchmarking tool running various mail-server workloads.
- pgbench: Built-in Postgresql benchmarking tool based on TPC-B.
- SpecWPC and pts-workstation: Benchmark tools simulating workloads of a workstation computer.

While a wider variety of hardware (such as NVMe hardware) can also be used to generate more data, we are required to maintain the high variety of workloads for all experiments, which multiplies by the number of hardware. This is challenging as longer experiments are required for larger devices and each experiment may require extra tuning. However, as Samsung 850 Pro belongs to a fairly mature generation of SSD devices and we believe that the findings from these experiments give valuable insights nevertheless.

An important challenge in training machine learning models is the uneven rate and volume of I/O requests, which can cause machine learning bias. Our 35 workloads range from a few hundred thousand to over 180M I/O requests. Uniform random sampling from these workloads to select the training and validation sets will result in a case of imbalanced data where the training process will focus on minimizing error on larger traces while giving much less importance to the rest.

To avoid this issue, we perform undersampling to pick the training and validation data. In the process shown in Fig. 8, 100,000 requests are sampled from each workload and these requests are in turn, split into the training and validation sets. From each trace, the remainder of requests are available for use as the testing set in the final evaluation. Should the number of these requests exceed a million, uniform random sampling is used to pick the testing set representing that particular trace.

Furthermore, up to a million requests are sampled from the requests not selected by undersampling to generate the testing set. It must be noted that as our feature extraction process stores past activity inside decaying counters, sampling I/O requests before feature extraction generates incorrect data. Thus, sampling is performed on the fly in the callbacks described in Sec. 2.3 with a ratio based on trace request count. These three sets are used as follows:

- **Training set**: Used to train machine learning models. This dataset contains ≈2.3M feature vectors.

10. Small Computer System Interface
11. Self-Monitoring, Analysis and Reporting Technology
12. The raw capacity of flash chips used in this device is equal to 192GB.
13. PTS stands for the Phoronix Test Suite [38].

- **Validation set**: Used in model selection as well as feature scoring and elimination. This dataset contains ≈1.1M feature vectors.
- **Testing set**: This dataset is utilized to compare the accuracy of the final models and feature sets. Each trace has its own training set containing up to a million feature vectors. These 35 sets of feature vectors contain a total of ≈30M rows.

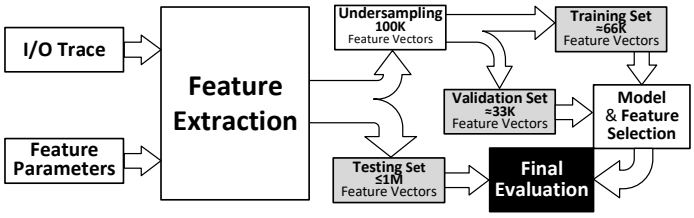


Fig. 8: I/O traces are used to generate feature vectors which are split into the training, validation, and testing sets. The training and validation process are used to select the final model and feature set.

3.2 Choice of machine learning models

As the goal of this research is to provide practical alternatives to real hardware experiments, it is essential to maintain low memory and processing requirements. After a thorough survey of various machine learning models, we arrive at three main categories of machine learning models: a) *decision trees*, which have been used in previous SSD and HDD performance prediction models [13], [14], [15], b) *Feedforward Neural Networks* (FNNs): Also known as *Multi-Layer Perceptrons* (MLPs), and c) *ensemble* methods, which use multiple predictors to reduce bias and variance, and thus increase accuracy. Alternatives such as *Support Vector Regression* (SVR) and *Recurrent Neural Networks* (RNNs) have been ruled out due to their extremely high computational requirements.

Adaboost, bagging, and random forest are the three ensemble methods being evaluated in this research. Adaboost proceeds in multiple steps where at each successive step, the poorly estimated data is weighted higher in order to reduce error. Bagging and random forest train a number of parallel predictors which recieve a subset of the whole data, making them much faster to train than Adaboost. All the ensemble methods in this study utilize decision trees as the base predictor. A comparison is performed between the following models to choose the exact ensemble models:

- A single decision tree, with a maximum depth of 32. The tree is allowed to have a maximum number of 10000 leaf nodes where each leaf must contain a minimum of five samples. These parameters are selected using many exploratory experiments. The rest of parameters are scikit-learn [41] defaults.
- An FNN with a configuration of 256-512-256 nodes with sigmoid activation for internals nodes and a single linear node to generate the response time. To have better generalization and avoid over-fitting, a) we use 20% dropout for the internal layers, b) use Adam optimizer with L1 error (*Mean Absolute Error* or MAE) which is resistant to outliers compared to L2, and c) use early stopping with a *patience* parameter of 10. We implement FNN using PyTorch [42] which utilizes *Graphical Processing Unit* (GPU).
- Ensemble learning utilizing multiple instances of the single decision tree described above with default settings for everything except the number of estimators. Experiments are performed with Adaboost, bagging, and random forest using two, three, five, and ten estimators[14]. We try to use parallel jobs (specified by `n_jobs` in scikit-learn) to make maximum utilization of available computational resources. However, due to the nature of Adaboost, the base estimators cannot be trained in parallel and bagging requires more than 32GB of RAM for any value `n_jobs` that exceeds 2. All three models are implemented using scikit-learn.

The results, comparing the accuracy and training time of each model can be seen in Fig. 9. In this figure, the computational resources used to calculate each model have been specified. The machine used for the purpose of training has 16 cores, 32GB of RAM, and an Nvidia 970 GTX graphics processor. The training time excludes the feature extraction time, which is equal for all models at this stage. Adaboost and the single decision tree model cannot utilize multiple concurrent threads, whereas bagging and random forest utilize the maximum number of available compute resources. However, due to the high memory requirements of bagging, only two threads can be used. Random forest splits the data between estimators and has much lower memory requirements which allows use of parallel job for each estimator. FNN utilizes the GPU through the PyTorch library.

Based on our experiments, FNN model training time and accuracy is greatly affected by the randomly initialized starting state. To represent the FNN model, we train nine independent FNNs, differing only in the starting state. These models have training times ranging from 708s to 1366s with 15% variation in the *Mean Absolute Error* (MAE). We decide to choose the model with median training time and its associated error to represent FNN at each stage of training and feature selection process. As can be seen in Fig. 9, comparison and selection of models is performed using MAE. The reason for use of an L1 error instead of an L2 measure of error such as sum of squared error is the fact that I/O request latency is highly variable and greatly susceptible to outliers. In fact, in our dataset containing 580M I/O requests, response time ranges from 12μs up to 5.6s, varying by five orders of magnitude. L2 measures of error will endanger magnifying the effect of outliers and sacrifice model accuracy in general case to optimize outliers.

We pick random forest with ten estimators and bagging with five estimators alongside the single decision tree and FNN models for the remaining experiments. This selection is made with respect to having a comparable training time across the different models. This also has been the reason for elimination of Adaboost as it is fundamentally unable to train its estimators concurrently using multiple parallel threads. Thus, the range of training time ranges from 601s

14. Our experiments demonstrate diminishing returns from increasing the number of predictors and we do not exceed the number ten.

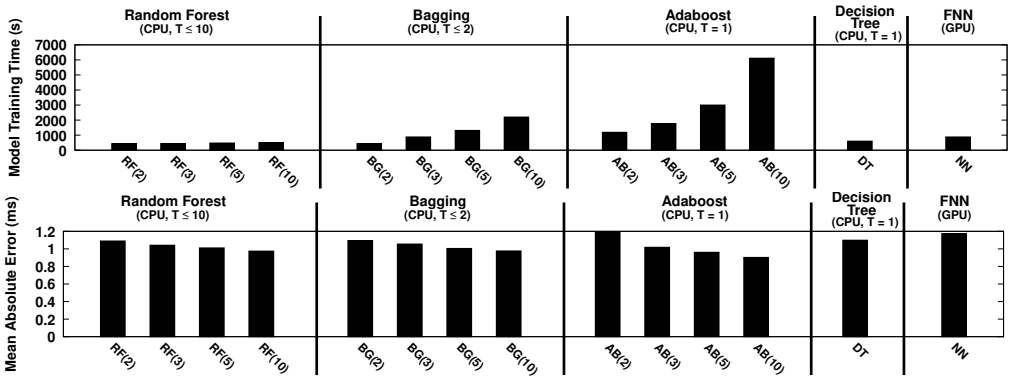


Fig. 9: Initial comparison of the various machine learning models for selection. The top model displays the training time, the bottom figure shows the Mean Absolute Error for the validation set. T for CPU refers to the number of threads.

(decision tree) to 1315s (bagging). As can be seen in the following sections (Figure 10), feature selection greatly speeds up bagging since the reduction of memory requirements allows the number of threads to be increased from two to five while staying within the 32GB limit. Eventually, and at the end of feature selection, FNN remains the model with the lowest training rate.

3.3 Feature Selection

As many of the features described in the earlier sections are parameterized, we calculate each feature with multiple distinct parameter values (as shown in Table 2) and perform feature selection to choose parameter values and eliminate features of negligible utility. In this section, we begin with 174 features used in the previous section and utilize feature selection to optimize key feature values and to eliminate features with little or no positive effect on model accuracy.

Although a wide variety of feature selection methodologies exist, many require $O(N^2)$ memory and are impractical for datasets which have many data points[15]. After implementing and experimenting with many feature selection solutions, we finally arrived at *Permutation Importance*, a wrapper method that uses a trained regression or classification model to evaluate the impact of each individual feature on model accuracy. The benefit of using permutation importance is its ability to work with all kinds of machine learning models.

Permutation importance measures the impact of each individual feature on model accuracy by performing many independent experiments. In each of these experiments, one or more features are selected and their value vector is shuffled. Shuffling completely maintains feature distribution but disrupts its relationship with response time. Thus, if shuffling of a feature reduces model accuracy, it hints at feature importance. After conducting the experiments, each feature is assigned a score which can be used in the selection process.

We use permutation importance in multiple steps using negative MAE as the accuracy metric. At each stage, we perform two actions a) if a specific feature type perform poorly with all the different parameters, it is eliminated, b) two major parameters Q and RT from Table 2 are successively optimized by choosing the parameter value yielding the highest average score for all features calculated by its particular value. The elimination threshold for a feature type is to have score lower than 0.01 in all instances. The reason for limiting parameter optimization to these Q and RT is the sheer number of features (117 out of the original 174) which rely on these two parameters.

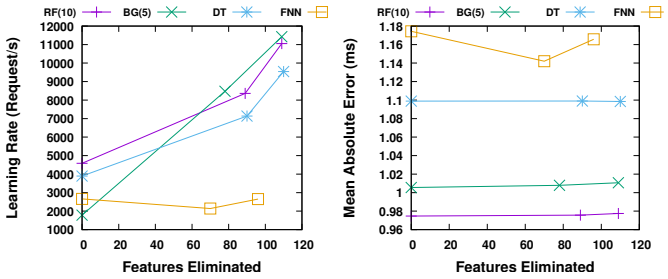


Fig. 10: Comparison of learning rate (left) and Mean Absolute Error (right) against the number of features eliminated by feature selection.

As permutation importance is a wrapper method, its scores are only valid for that particular model and must be performed on each of the four models. Fig. 10 plots learning rate and MAE against the number features that have been eliminated at each stage. Table 3 shows the specific features removed for each model through the stages of feature selection. Table 3 also shows the feature extraction rate with each set of features. The great leap in learning rate for the bagging model is caused by the large reduction of features which in turn, allows us to run five parallel jobs without filling the main memory. Overall, it can be seen that for the ensemble and decision tree models, while feature selection can come with a modest performance decrease, it can greatly boost learning rate, making the methodology more accessible and practical.

It can be seen from Fig. 10 that FNN accuracy and learning rate fluctuates through the feature selection process. This is due to the fact that the initial random weights of the neural network model cause variation in accuracy

15. Such algorithms are suitable for experimental datasets where the number of data point is small but many features are present for each point, e.g. patient data in a clinical trial.

TABLE 3: Extraction rate, feature count, and the eliminate features for each successive stage of feature selection. NF = no selection, F1 and F2 = first and second filtering stages.

Stage	Model	Features	Eliminated Features	Parameters Eliminated	Extr. Rate (Request/s)
NF	All	174	-	-	242K
F1	RF(10)	85	is_overlapped, is_strided, trim_score_w, is_random	Q=2,8	285K
	BG(5)	96	is_strided, trim_score_w, is_overlapped	RT=4KB,128KB	292K
	DT	84	is_sync, is_overlapped, is_strided, is_random trim_score_w	Q=2,8	284K
	FNN	96	-	Q=2,8	282K
F2	RF(10)	65	-	RT=512,4KB	312K
	BG(5)	65	is_random	Q=2, 8	313K
	DT	64	-	RT=512,4KB	312K
	FNN	70	-	RT=512,4KB	311K

and training time. Thus, at each stage of feature selection, the FNN is trained nine times and the model with median training time is selected as the representative.

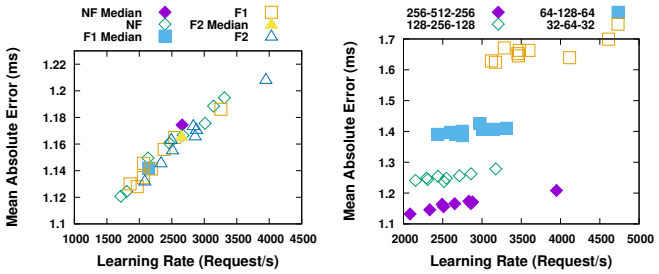


Fig. 11: MAE against learning rate in various feature selection stages (left) and with various node configurations (right) for the FNN model.

Fig. 11 shows the effect of feature selection (left) and scale-down (right) on FNN learning rate and accuracy. As feature selection reduces features from 174 (NF) down to 96 (F1) and finally, 70 (F2), feature extraction rate is greatly boosted (by up to 29% as shown in Table 3) but the learning rate or accuracy is not affected. Thus, significant speed-ups can be gained with feature selection at no cost. As FNN has the slowest learning rate (Figure 10), we do not experiment with wider neural networks which, further reduce learning rates and here attempt to reduce the network size to boost learning rate. As can be seen, while scaling down the model may increase learning rate by over 100%, it does so with a significant increase of error (MAE). Thus we believe that it is better to avoid model scale-downs for the sake of accuracy.

3.4 Inspection of Feature Scores

As stated in the previous section, we perform Permutation Importance to measure feature accuracy. In this section, we look at the final scores yielded by permutation importance for the machine learning models. Table 4 shows a comparison of feature ranks as yielded by permutation importance. All results belong to the last stage of filtering (F2). Features that have been filtered from a specific model are ranked as $N+1$ where N is the last rank for that particular model, e.g. as filtering for decision tree model has removed five of the main feature types and all these features are assigned the last rank in this table (rank 19).

Looking at feature scores, the highest ranked features is `lengths`, which is the current request size. `write_score_w` is concerned with the total amount of recently written data. `sync_score` measures the amount of recent sync requests. `Mlocality_var_score` is concerned with the access randomness at coarse grained address granularity. `seq_d_wscore` and `seq_d_score` measure the recent sequentiality and `disk_usages` measures the current disk utilization. Overall, it can be seen that the top ranked features measure a mixture of recent request types, current request length, recent sequentiality, randomness, and disk space utilization. Looking at the bottom half of the table reveals that properties that only measure the current request behavior (such as the properties prepended with `is_`) have a much lower importance compared to features that track long or short term workload behavior. An important note must be made with respect to `trim_score_w`: while this feature ranks poorly for all models, it must be noted that this feature is in some ways analogous with `disk_usages`, as the modern file systems discard unused data that is no longer part of the free disk space.

TABLE 4: Comparison of feature ranks based on scores yielded by permutation importance.

Feature	RF10	BG5	DT	NN	Mean Rank
lengths	1	1	1	2	1.25
write_score_w	2	3	6	1	3
sync_score	3	4	4	7	4.5
Mlocality_var_score	5	5	9	5	6
seq_d_wscore	7	7	7	4	6.25
seq_d_score	6	2	5	15	7
disk_usages	10	8	8	3	7.25
read_score_w	8	11	2	9	7.5
disk_usage_rates	4	9	10	11	8.5
write_score	9	10	11	13	10.75
min_distance	14	6	3	22	11.25
locality_var_score	13	13	13	8	11.75
is_sync	11	12	19	6	12
Mlocality_score	15	15	14	10	13.5
read_score	12	14	12	16	13.5
is_sequential	17	16	17	14	16
locality_score	16	17	15	17	16.25
is_read	19	19	16	12	16.5
is_write	18	18	18	18	18
trim_score_w	20	20	19	19	19.5
is_strided	20	20	19	20	19.75
is_overlapped	20	20	19	21	20
is_random	20	20	19	23	20.5

3.5 Summary

In this section, we first elaborated the methodology for generation of training, validation, and testing data. Due to the large size of the feature vector, we focused on models that can be trained with an acceptable processing and memory costs and started our experiments with five model types: a) basic decision tree, b) Feedforward Neural Networks (FNN), and c) three ensemble methods of Adaboost, bagging, and random forest using the basic decision tree as the predictor. We refined the hyper-parameters of these methods to arrive at four final machine learning models.

After model selection, we performed feature selection using permutation importance to reduce the initial feature size from 174 to equal or less than 70 features on all four

methods which speed up training of ensemble models by over six times and increase feature extraction rate by up to 29%. Finally, by performing an inspection of the feature scores yielded by the four models, we conclude that features measuring recent workload behavior through decaying counters have a much larger impact on model accuracy compared to per-request features. In the next section, we perform the final evaluation of the four models which have been refined and trained in this section.

4 Evaluation of Prediction Models

As discussed in the previous section, feature selection eliminates low score feature instances, and in cases, complete feature types. In our methodology described in Sec. 3.1, undersampling is performed to choose the training and validation sets and up to one million of the remaining feature vectors are selected as the testing set. In this section, the training set data for 35 traces is used to evaluate the accuracy of the four machine learning methods utilizing two accuracy measures: R^2 score and Mean Absolute Error (MAE).

Fig. 12 shows the mean absolute average (top) and R^2 (bottom) score for each individual trace. As stated earlier, $R^2 \in (-\infty, 1]$ where 1 indicates a perfect fit. As can be seen, while most traces indicate a good quality of fit, a number of outliers exist. For example, `pts-workstation` and `pgbench-init` have high MAE while `psql-tpcc-run`, `mssql-tpcc-run`, and `psql-tpch-run` have a poor R^2 score for one or more of the four models. These examples will be examined with more details in the remainder of this section.

Fig. 13 shows R^2 and MAE percentiles for the four models. For R^2 score, while random forest, bagging, and decision tree models *score below zero in eight instances four of which belong to the single decision tree model (DT)*, FNN exceeds their accuracy at lower percentiles and achieves a higher average R^2 score. This means that while decision trees are much faster and do not require GPUs, they may have poor accuracy in specific instances. For the MAE error, FNN has the worst behavior over the rest, but as can be seen in Fig. 12 and Fig. 13 this is due to the outlier poor MAE of `pts-workstation`. In fact, if this workload is excluded, FNN would have the lowest average MAE.

A key challenge in response time prediction is its extremely high variability. Request response time in `pts-workstation` varies from 18μs up to 553ms with an average response time of 77ms whereas in `psql-tpch-run`, it varies from 18μs to 222ms with an average response time of 502μs. With respect to the high response time variation, we analyze R^2 score and MAE by binning the requests into response time bins and calculating the contribution of each bin to the final error. We examine the accumulative squared error and accumulative absolute error for this purpose. These measures are central to calculation of R^2 and MAE, respectively and are depicted by boxes in Equation 12 and Equation 13. With respect to the analysis of workloads based on their response time bins, we group the six workloads identified in the previous section into three groups:

- `pts-workstation` and `pgbench-init`: These workloads experience the two highest MAE, but have some of the highest (≈0.9) R^2 scores across all four models. As can be seen from comparison of Equation 12 and Equation 13, MAE is greatly affected by magnitude. For example, multiplication of latency by ten increases MAE by ten while not affecting R^2 score at all. These two workloads have the two highest average response times of 77ms and 20ms out of the 35 workloads, which increase MAE. We plot `pts-workstation` in Fig. 14a as a representative of this class.
- `psql-tpcc-run`, `psql-tpch-run`, and `mssql-tpcc-run`: These workloads are OLTP benchmarks which are predicted very poorly by the three decision tree-based models. The FNN performs much better by achieving an R^2 score of 0.66, 0.43, and 0.31, respectively. In contrast, the other three achieve *negative* R^2 scores except for a single experiment achieving an R^2 score of 0.02. In all these workloads, most requests are located in the 0.1s to 1s and as shown in the representative Fig. 14b, the accumulative absolute error and the accumulative squared errors is high even in the 0.1s to 1s range. The accumulative errors for FNN rise after the 10s response time bins and are extremely lower in all three workloads.
- `mysql-tpch-run`: This is the single workload where all models have a poor R^2 score. However, as can be seen in Fig. 12, the MAE for this workload among the smallest of all traces. This is due to the fact that this workload is simultaneously the largest (189M+ requests) and has the second lowest average response time (≈180μs). The error chart also indicates that while almost all the requests fall within the 0.1s-1s range, the bulk of error occurs with a small fraction of requests falling in the higher response time bins. The error caused by these high response times, coupled with the low average response time for these traces, results in low R^2 scores for all four models.

$$R^2 = 1 - \frac{\boxed{\sum_{i=1}^{n} (y_{real_i} - y_{pred_i})^2}}{\sum_{i=1}^{n} (y_{real_i} - \bar{y}_{real})^2} \tag{12}$$

$$MAE = \frac{\boxed{\sum_{i=1}^{n} |y_{real_i} - y_{pred_i}|}}{n} \tag{13}$$

Overall, it can be seen that among the four models, FNN achieves the most stable results with a worst-case R^2 score that is far better than the other three models. The two instances where the FNN R^2 score is lower than 0.4 are the outlier workloads of `mssql-tpcc-run` and `mysql-tpch-run` with the respective R^2 scores of 0.31 and 0.01 and simultaneously the lowest and second lowest average response time of 80μs and 181μs. This low average response time causes the effect of outlier requests to be greatly magnified, reducing the R^2 score considerably. The

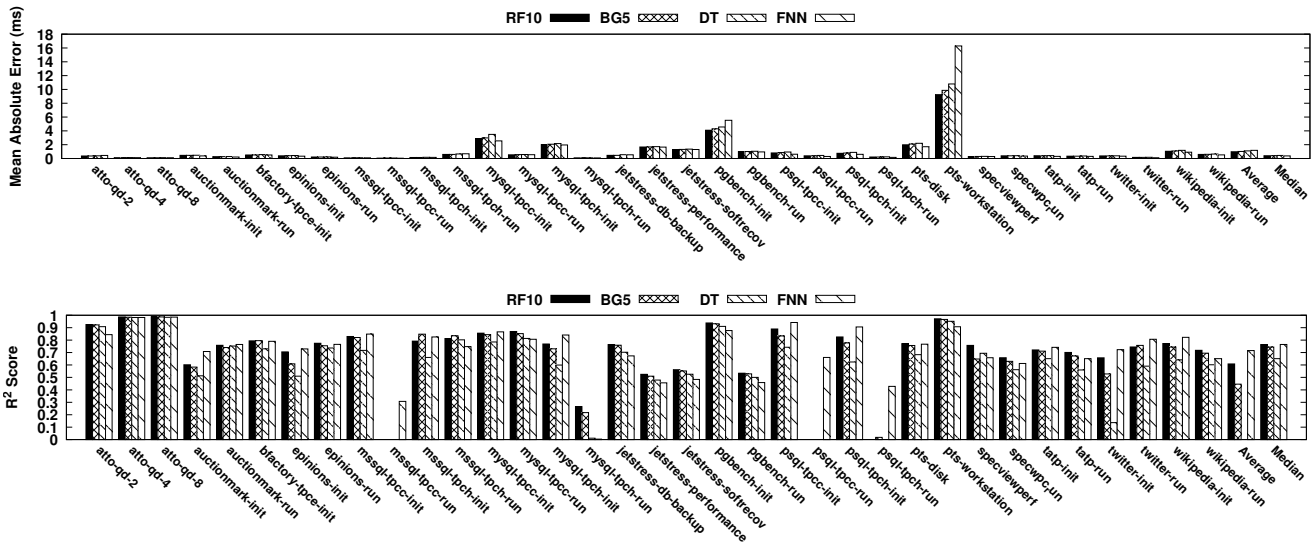


Fig. 12: R^2 Score (top) and Mean Absolute Error (bottom) for the four main models.

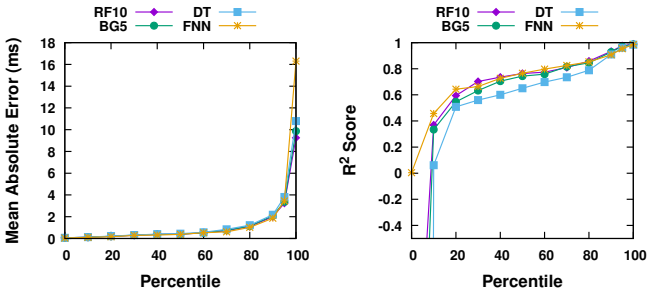


Fig. 13: Mean Absolute Error (left) and R^2 Score (right) percentiles of all evaluated workloads.

two worst-case MAE for this model is also caused by outliers which are `pts-workstation` and `pgbench-init`. These workloads have the highest average response times of 77ms and 20ms.

Although FNN almost requires four times the training time (884 seconds compared to 212 seconds) of random forest, this difference of less than 12 minutes is a negligible cost for having more dependable results. However, random forest has the benefit of not requiring graphical processors, supporting multi-core acceleration, and having much faster training time. This makes random forest a viable choice for scenarios where training is to be performed on a real machine, providing the ability to train specialized models with high accuracy and very low prediction time.

5 Related Work

As this work performs workload characterization and machine learning for prediction of hardware response time, we review related work under two subjects, a) workload characterization and extraction of features, and b) storage performance models.

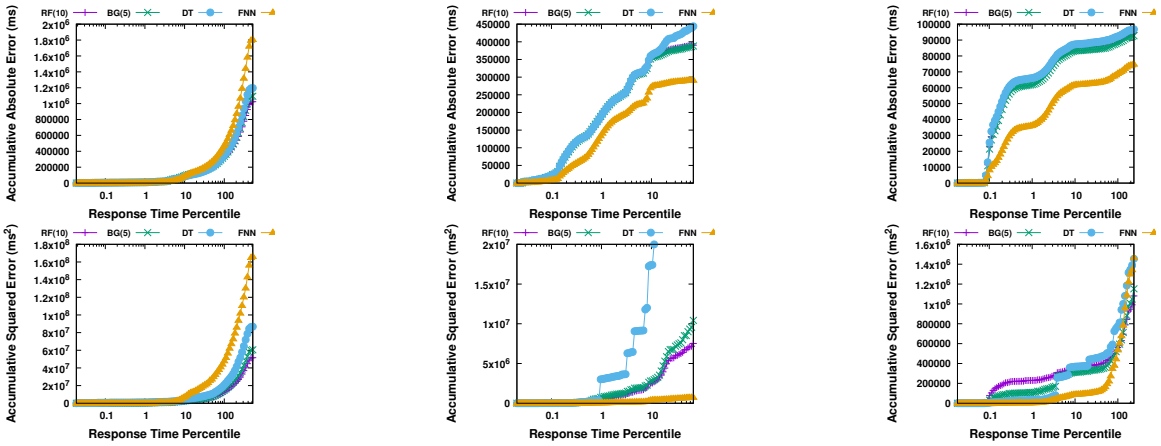
5.1 Workload Characterization and Feature Extraction

Workload characterization is concerned with analysis and study of I/O activities in a workload. To compare and contrast different workloads and to better interpret their behavior, workload properties such as temporal and spatial locality must be quantitatively represented. Riska *et al.* [32] record various storage workloads using specialized hardware and examine properties such as read/write ratio, interface idle time, and self-similarity. Seo *et al.* [37] characterized storage workloads based on a variety of features such as total and non-random request count, request size, sequential run length, address variance, and inter-arrival time. Feature selection was performed to yield a final feature set, which was used to create a decision tree for I/O workload classification.

Anderson *et al.* [1] presented Hippodrome, a backend design and management tool, which automatically characterizes workloads to extract properties such as read/write request rate, mean size, arrival rate and size, sequential run length, device queue depth, and burstiness. DiskAccel [8] suggested to use spatial and temporal locality, read/write ratio, I/O bandwidth, throughput, and other properties to cluster workload intervals and to select per-cluster *representative intervals* for replay on real HDDs. Tarasov *et al.* [7] created a solution based on running benchmark utilities for simulation of real workloads. This solution extracts read/write ratio, inter-arrival time, bandwidth, throughput, and address distribution from the original workload, generating benchmark parameters to replicate workload behavior. Tremblay *et al.* [43] used workload properties such as throughput, average request size, and working set to select an optimal software-defined cloud storage backend. Bi *et al.* [3] offer a solution to predict CPU and memory loads based on past workload behavior. Their solution relies on use of *Long Short-Term Memory* (LSTM) networks to perform predictions. We avoid using LSTM as the huge number of I/O requests make such models very expensive.

5.2 Performance Models for Storage Hardware

There have been multiple past efforts on modeling HDD and SSD performance. Although both devices often have a



(a) `pts-workstation` (b) `psql-tpcc-run` (c) `mysql-tpch-run`

Fig. 14: Accumulative absolute error (top row) and accumulative squared error (bottom row) against response time bins for `pts-workstation`, `psql-hammerdb-tpcc-run`, and `mysql-tpch-run` .

fairly similar host interface[16], their internal construction and operation is vastly different. Here, we will cover works that have offered performance models for HDDs or SSDs.

Li *et al.* [14] created a performance model that predicts response time, bandwidth, and throughput for one-minute request intervals using decision trees based on eight basic workload properties such as device queue depth, read-/write ratio, and per-read and -write inter-request stride, randomness, and request size. This work was extended by Huang *et al.* [15] to include a few non-synthetic workloads obtained by replaying widely-used HDD workloads on SSDs. Both these works analyze I/O workloads as independent one-minute internals. These two works are the closest to our work in terms of comparing SSD performance, however, they predict SSD latency over intervals. Compared to our work, these two works use a vastly simpler feature set with no long term memory. Kim *et al.* [4], [5] presented a simplified performance model for SSDs to predict slowdowns caused by internal SSD activities such as garbage collection and write buffer flush events. This model is used to implement adaptive scheduling [4] and adaptive volume management [5] solutions. In all of these three works, the important effect of past I/O activity on future performance [6], [16], [17], [18] is not taken into account. Dartois *et al.* [2] present a methodology that utilizes multiple different machine learning methodologies such as decision trees and ensemble methods to predict whole workload average throughput when running storage workloads on SSDs in cloud infrastructure.

For HDDs, Wang *et al.* [13] utilize decision trees to predict the performance of individual requests or workload time intervals. Per-interval features include average request arrival rate, read/write ratio, average request size, sequentiality, spatio-temporal burstiness, and attribute correlation over each minute whereas per-request features include spatial and temporal distance with respect to previous requests, sequentiality, size, and type (being read or write). Three of our final machine learning models are based on decision trees which predict per-request response time, however, unlike this work, we predict performance for SSDs, which are much different from HDDs.

16. The main difference is the lack of *discard* commands on normal HDDs, however, SMR HDDs do support *discard* commands.

Work	Drive	Predicted Quantity	Granularity	Aggregate Features	Method	Workload
Li *et al.* [14]	SSD	Response Time &	Interval	Yes	DT[1]	Synth.
Huang *et al.* [15]	SSD	Bandwidth	Interval	Yes	DT	Synth.+ Replay
Wang *et al.* [13][2]	HDD	Response Time	Request	No	DT	HDD Trace
			Interval	Yes		
Kim *et al.* [4]	SSD	*Slowdown*	Request	Yes	Routine	SSD Trace
Kim *et al.* [5]	SSD	*Slowdown*	Request	Yes	Routine	Replay
Dartois *et al.* [2]	SSD	*Throughput*	Workload	Yes	Multiple	SSD Trace
This Work	SSD	Response Time	Request	Yes	FNN[3]+ DT	SSD Trace

[1] Decision Tree [2] This work presents two models.
[3] Feedforward Neural Networks

TABLE 5: Comparison of related works with the current work.

5.3 Summary

Table 5 shows an overall comparison of this work and previous performance modeling research. **Predicted Quantity** is the target quantity being predicted by the model, **Granularity** explains whether the model predictions are per-request, per-time interval, or are calculated over the whole workload, **Aggregate** shows whether the model is utilizing features based on multiple past requests, **Method** is the construct making the prediction, and **Workload** is the type of workloads used in each model. **SSD Trace** and **Replays** differentiate workloads that are recorded from real SSDs or are generated by replaying HDD workloads on SSDs. The latter method requires detailed trace information to infer inter-request dependencies [8], [44], [45] and the HDD traces used by [15] lacks this dependency information. **Synth.** refers to macrobenchmarks that are run on SSDs. In Table 5 while a number of previous methods use aggregate features or interval-based granularity to track past requests in the short term, none of them consider the long term effect of previous activity on future performance. The only exception is Dartois *et al.* [2] which analyzes the workload as a whole. Prediction of request response time while considering long term history is difficult due to the fact that large

and complex SSD state space must be modeled efficiently. Furthermore, we present the only SSD response time model supporting *discard* and *sync* requests.

6 Conclusion and Future Work

In this paper, we introduced a methodology which extracts a variety of I/O workload properties and uses them to create SSD performance models which can be quickly trained. Our features cover a variety of properties such as request type, spatial and temporal locality, and past workload behavior while giving higher importance to recent activity. Through use of efficient data structures such as exponentially decaying counters, these features can be extracted with $O(1)$ memory and processing requirements.

After feature selection, we were able to reduce the feature set by up to 63% and achieve a rate of 313,000 requests per second for feature extraction. Experiments were performed using Feedforward Neural Networks (FNNs), basic decision tree, and two ensemble methods based on decision trees. While a single decision tree performs very poorly, FNN achieves a superior average R^2 score of 0.72, although requiring four times the training time compared to random forest and bagging, which achieve R^2 scores of 0.61 and 0.45, respectively. The high rate of over 1000 requests per second per thread, allows the random forest to be used in an online scenarios where the model is periodically re-trained via sampled feature vectors and then used in performance-aware algorithms in storage servers or backends.

For future works, further tuning and implementation of lightweight feature extraction and machine learning methods can make the methodology easier to deploy in online scenarios. Experiments using a wider variety of devices such as higher-density or higher-bandwidth NVMe devices can further help generalizing our methodology. Furthermore, synthesis of I/O workloads based on workload features is another possible future direction for this research. Upon publication of this work, we release the code and data used in this study for the benefit of the research community.

References

[1] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, "Hippodrome: Running circles around storage administration," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST'02. Berkeley, CA, USA: USENIX Association, 2002, pp. 13–13.

[2] J.-E. Dartois, J. Boukhobza, M.-A. Knefati, and O. Barais, "Investigating machine learning algorithms for modeling ssd i/o performance for container-based virtualization," *IEEE Transactions on Cloud Computing*, vol. PP, pp. 1–1, 02 2019.

[3] J. Bi, S. Li, H. Yuan, and M. Zhou, "Integrated deep learning method for workload and resource prediction in cloud systems," *Neurocomputing*, vol. 424, pp. 35–48, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231220317884

[4] J. Kim, J. Kim, P. Park, J. Kim, and J. Kim, "SSD Performance Modeling Using Bottleneck Analysis," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 80–83, Jan 2018.

[5] J. Kim, P. Park, J. Ahn, J. Kim, J. Kim, and J. Kim, "Ssdcheck: Timely and accurate prediction of irregular behaviors in black-box ssds," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 455–468.

[6] M. Jung and M. Kandemir, "Revisiting widely held ssd expectations and rethinking system-level implications," in *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '13, 2013, pp. 203–216.

[7] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok, "Extracting flexible, replayable models from large block traces," in *Proc. FAST'12*, 2012, pp. 22–22.

[8] M. Tarihi, H. Asadi, and H. Sarbazi-Azad, "Diskaccel: Accelerating disk-based experiments by representative sampling," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '15, 2015, pp. 297–308.

[9] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101)," *Parallel Data Laboratory*, p. 26, 2008.

[10] N. Dayan, M. K. Svendsen, M. Bjørling, P. Bonnet, and L. Bouganim, "EagleTree: exploring the design space of SSD-based algorithms," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, pp. 1290–1293, 2013.

[11] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," *Proceedings of the international conference on Supercomputing*, pp. 96–107, 2011.

[12] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, Oakland, CA, 2018, pp. 49–66.

[13] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. Ganger, "Storage device performance prediction with cart models," in *Proc. MASCOTS'04*, Oct 2004, pp. 588–595.

[14] S. Li and H. H. Huang, "Black-box performance modeling for solid-state drives," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Aug 2010, pp. 391–393.

[15] H. H. Huang, S. Li, A. Szalay, and A. Terzis, "Performance modeling and analysis of flash-based storage devices," *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pp. 1–11, 2011.

[16] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '09. New York, NY, USA: ACM, 2009, pp. 181–192.

[17] N. Jeremic, G. Mühl, A. Busse, and J. Richling, "The pitfalls of deploying solid-state drive raids," in *Proceedings of the 4th Annual International Conference on Systems and Storage*, ser. SYSTOR '11, 2011, pp. 14:1–14:13.

[18] P. Desnoyers, "Analytic Models of SSD Write Performance," *Transactions on Storage*, vol. 10, no. 2, pp. 8:1–8:25, Mar. 2014.

[19] E. Ho, E. Kim, C. Paridon, D. Rollins, and T. West, "Understanding ssd performance using the snia sss performance test specification," Storage Networking Industry Association (SNIA), Tech. Rep., 2012.

[20] K. Smith, "Benchmarking ssds: The devil is in the preconditioning details," *Benchmarking SSDs: The Devil Is in the Preconditioning Details*, vol. 17, 2009.

[21] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: Analysis of tradeoffs," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09, 2009, pp. 145–158.

[22] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009*, ser. SYSTOR '09, 2009, pp. 10:1–10:9.

[23] S. Boboila and P. Desnoyers, "Performance models of flash-based solid-state drives for real workloads," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2011, pp. 1–6.

[24] G. Yadgar, M. Gabel, S. Jaffer, and B. Schroeder, "Ssd-based workload characteristics and their performance implications," *ACM Trans. Storage*, vol. 17, no. 1, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3423137

[25] D. Zhou, W. Pan, W. Wang, and T. Xie, "I / o characteristics of smartphone applications and their implications for emmc design," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 12 – 21.

[26] "StorageReview," http://www.storagereview.com, [Online; accessed 28-February-2021].

[27] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *JILP*, vol. 7, no. 4, pp. 1–28, 2005.

[28] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970.
[29] B. Hong and T. M. Madhyastha, "The relevance of long-range dependence in disk traffic and implications for trace synthesis," in *Proc. MSST'05*, pp. 316–326.
[30] M. Tarihi, H. Asadi, A. Haghdoost, M. Arjomand, and H. Sarbazi-Azad, "A Hybrid Non-Volatile Cache Design for Solid-State Drives Using Comprehensive I/O Characterization," *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1678–1691, June 2016.
[31] H. Hurst, "Long-term storage capacity of reservoirs," *Transactions of the American Society of Civil Engineers*, vol. 116, pp. 770–799, 1951.
[32] A. Riska and E. Riedel, "Disk drive level workload characterization," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06, 2006, pp. 9–9.
[33] I. Ahmad, "Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server," in *2007 IEEE 10th International Symposium on Workload Characterization*, Sept 2007, pp. 149–158.
[34] N. Singh and S. Rao, "Ensemble learning for large-scale workload prediction," *IEEE Transactions on Emerging Topics in Computing (TETC)*, vol. 2, no. 2, pp. 149–165, 2014.
[35] A. Mahanti, D. Eager, and C. Williamson, "Temporal locality and its impact on web proxy cache performance," *Performance Evaluation*, vol. 42, no. 2, pp. 187 – 203, 2000.
[36] L. Cherkasova and M. Gupta, "Characterizing locality, evolution, and life span of accesses in enterprise media server workloads," in *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, ser. NOSSDAV 02. New York, NY, USA: Association for Computing Machinery, 2002, p. 3342.
[37] B. Seo, S. Kang, J. Choi, J. Cha, Y. Won, and S. Yoon, "IO Workload Characterization Revisited: A Data-Mining Approach," *IEEE Transactions on Computers*, vol. 63, no. 12, pp. 3026–3038, Dec 2014.
[38] M. Larabel and M. Tippett, "Phoronix test suite," 2011, [Online; accessed 1-November-2020].
[39] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," vol. 7, no. 4, 2013.
[40] N. Johnson, "Jetstress field guide," Microsoft Services, 2013.
[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.
[43] B. Tremblay, K. Kozubal, W. Li, and C. Padala, "A workload aware storage platform for large scale computing environments: Challenges and proposed directions," in *Proceedings of the ACM 7th Workshop on Scientific Cloud Computing*, ser. ScienceCloud '16. New York, NY, USA: ACM, 2016, pp. 27–33.
[44] Z. Weiss, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "ROOT: replaying multithreaded traces with resource-oriented ordering," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, 2013, pp. 373–387.
[45] T. E. Pereira, F. Brasileiro, and L. Sampaio, "File system trace replay methods through the lens of metrology," in *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, May 2016, pp. 1–15.

Mojtaba Tarihi received his BSc and MSc degrees from the Sharif University of Technology (SUT), Tehran, Iran, in 2011 and 2013, respectively, in computer engineering. He has been a PhD student since 2013 and a PhD candidate since 2015 at SUT. His current research interests include storage workload characterization and device performance modeling.

Soheil Azadvar recieved his BSc in electrical engineering and his MSc degree in computer engineering, from Sharif University of Technology (SUT) in 2014 and 2017, respectively. He was member of Data Storage, Networks, and Processing (DSN) Laboratory from September 2017 to August 2020. His research interests include storage systems and applications of machine learning to storage systems design.

Arash Tavakkol graduated with a Ph.D. from Sharif University of Technology, Tehran, Iran in 2015 and spent two years (2016-2018) as a Postdoc at Systems Group, ETH Zurich. Arash also has more than ten years of experience in high-performance computing and design issues of scale-out data centers. He was one of the founding members and the technical manager of the High-Performance Computing Laboratory at Institute for Research in Fundamental Sciences (IPM), Tehran, Iran, where the most powerful Iranian supercomputer was built on 2008 using modern heterogeneous multicore processors.

Hossein Asadi (M'08, SM'14) received the BSc and MSc degrees in computer engineering from the SUT, Tehran, Iran, in 2000 and 2002, respectively, and the PhD degree in computer engineering from Northeastern University, Boston, MA, USA, in 2007. He is currently a full professor in Department of Computer engineering at SUT. He is the founder and director of the Data Storage, Networks, and Processing (DSN) Laboratory and the director of Sharif HPC Center. His research interests include data storage systems, SSDs, operating systems, and high-performance computing. Dr. Asadi was a recipient of the Distinguished Lecturer Award from SUT in 2010, the Distinguished Researcher Award and the Distinguished Research Institute Award from SUT in 2016, the Distinguished Technology Award from SUT in 2017, and the Distinguished Research Lab Award from SUT in 2019. He also received the Best Paper Award at IEEE/ACM Design, Automation, and Test in Europe (DATE) in 2019.

Hamid Sarbazi-Azad is currently professor of computer science and engineering at Sharif University of Technology, Tehran, Iran. His research interests include high-performance computer/memory architectures, NoCs and SoCs, parallel and distributed systems, social networks, and storage systems, on which he has published over 400 refereed papers. He received Khwarizmi International Award in 2006, TWAS Young Scientist Award in engineering sciences in 2007, and Sharif University Distinguished Researcher awards in years 2004, 2007, 2008, 2010 and 2013. He is now an associate editor of ACM Computing Surveys, IEEE Computer Architecture Letters, and Elseviers Computers and Electrical Engineering.