TraceGen: A Block-level Storage System Performance Evaluation Tool for Analyzing and Generating I/O Traces

Cheng Li[1], Jiahe Wei[1], Huiru Xie[1], Jinjiang Wang[1], Xiaonan Zhao[1,2,3], Shujie Han[1,2,3], and Xiao Zhang[1,2,3]
[1]*School of Computer Science, Northwestern Polytechnical University*
[2]*MIIT Key Laboratory of Big Data Storage and Management*
[3]*National Engineering Laboratory for Integrated Aero-Space-Ground-Ocean Big Data Application Technology*

***Abstract*—Performance measurement is essential for detecting potential performance issues and guiding optimization efforts. However, acquiring I/O traces of real applications can be costly in production environments. Also, existing performance measurement tools, such as FIO and Iometer, often oversimplify real-world application characteristics. In this paper, we introduce TRACEGEN, a block-level performance measurement tool for storage systems that consists of a trace analyzer and a trace generator. The trace analyzer produces two categories of traces: (i) new traces with specified characteristics designed to accurately simulate a range of applications, and (ii) extended traces that maintain similar workload characteristics to the input traces, thereby improving measurement accuracy during trace replay. We evaluate TRACEGEN using traces from an enterprise production environment and demonstrate its capability to generate new traces with an error margin of less than 1%.**

***Index Terms*—performance measurement tools, simulate workloads, I/O trace analysis and generation**

I. INTRODUCTION

Measuring the I/O performance is crucial for optimizing storage systems. To mirror the workloads of real-world applications, I/O traces are collected from storage systems. Such I/O traces are replayed using simulation tools like btreplay [1] and hfplayer [9] to accurately reproduce system pressures with precise performance measurements [8], [14], [23], [25].

However, capturing I/O traces from real-world applications faces several challenges [23]. First, confidentiality concerns restrict the trace collection from real-world storage systems. Second, gathering traces in systems with complicated architectures imposes significant overhead, potentially degrading system performance during the collection process [13]. Third, accurately and timely collecting a sequence of I/O operations is challenging, especially in high-throughput production environments, where even a small latency of collection can result in substantial discrepancies. Finally, I/O traces captured in one production environment may not accurately represent workloads in another, limiting their generalizability and usefulness for broader performance measurements.

To efficiently measure storage system performance, various tools such as FIO [3], Iometer [4], and IOzone [5] have been developed. These tools can simulate a range of workloads, including random and sequential read/write operations with different request sizes. However, they are limited to generating simple and stable workloads, making it challenging to reproduce the complexity and diversity of real-world scenarios. Additionally, these tools do not support fine-grained I/O construction with granularity finer than one second, which compromises the accuracy of performance measurements.

To address these issues, we design and implement TRACEGEN, a trace generation tool in the block level. TRACEGEN consists of two main components: a trace analyzer and a trace generator. The trace analyzer independently analyzes the workload characteristics of traces, while the trace generator relies on the trace analyzer's output to generate new traces. Based on the trace characterization, TRACEGEN generates I/O traces of two types: (i) new traces with specified characteristics and (ii) long traces with similar characteristics to the input trace. To summarize, the key contributions are as follows:

- We characterize two block-level I/O traces in a production environment from one of the largest cloud companies in China, including the access range, request size distribution, block access popularity, and IOPS across fixed-size time slices. Moreover, we investigate the access patterns of traces including repeated access patterns, sequential access patterns, and random access.
- Based on our trace characterization, we present a complete design of TRACEGEN, a trace generator tool, which comprises the trace analyzer and trace generator. It supports two types of trace generation, including a new trace with specified characteristics and a long trace with similar characteristics to the input trace.
- We implement a complete prototype of TRACEGEN from scratch using Python libraries.
- We evaluate the effectiveness of TRACEGEN for generating a new trace with specified characteristics and a long trace with similar characteristics to the input trace with a small error of less than 1%.

II. DATASET

In this section, we introduce our dataset, including the data collection and trace format.

Data collection. We collected block-level I/O traces in a production environment from one of the largest cloud companies in China. The I/O traces are captured by `blktrace` [1], which is a tool used for tracing and analyzing Linux block device I/O operations. Table I shows an overview of our dataset. Specifically, our dataset covers two traces, denoted by *Trace-A* and *Trace-B*. Both traces include more than 500 million rows with a size of more than 16 GiB. We refer to each row in traces as the *record* and the row index as the *record index*. Trace-B has a longer duration (12.5 hours)

Trace name	Rows	Size	Duration	SSD capacity	Write ratio
Trace-A	540 million	16.5 GiB	3 hours	8 TiB	46.94%
Trace-B	520 million	16.7 GiB	12.5 hours	4 TiB	77.99%

TABLE I: Basic information of traces.

Timestamp	Type	Start offset	Request size	End offset
1813	R	8473419632	144	8473419776
1843	W	7811028032	80	7811028112
1902	R	7058082960	8	7058082968
1934	R	5728177408	8	5728177416
1963	W	6641393664	16	6641393680

TABLE II: An example of trace content.

than Trace-A (3 hours) due to less frequent operations. Both traces are collected from solid-state drives (SSDs) but from different applications. The ratio of write requests for Trace-A is 46.94%, while for Trace-B it is 77.99%.

Trace format. As shown in Table II, Trace-A and Trace-B share the same trace format, including the *timestamp*, *type*, *start offset*, *request size*, and *end offset*. The timestamp indicates the time point (in the microsecond level) that an I/O request is placed into the request queue of the underlying device. For privacy, we anonymize the exact timestamps to relative timestamps from the first record starting from zero. The type indicates the type of an I/O request type, e.g., read or write (denoted by "R" or "W", respectively). The start offset refers to the start of the logical block address (LBA) of an I/O request for addressing a sector with a size of 512 bytes. The request size refers to the number of sectors (with the size of 512 bytes) that are accessed by the I/O request. The end offset refers to the end of LBA of an I/O request, which equals the start offset plus the request size of the I/O request.

III. Characterization of Traces

In this section, we first present the characteristics of traces and then analyze the access patterns of traces.

A. Trace Characteristics

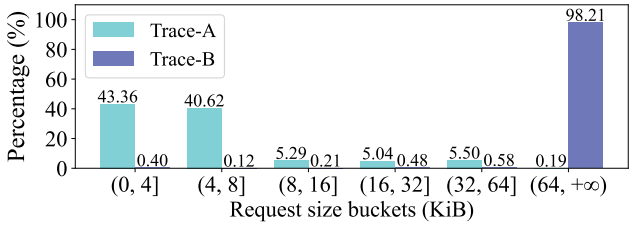
We examine the characteristics based on the entire traces, including the access range, request size distributions, block access popularity, and IOPS.

Access range. The access range refers to the minimum start offset and maximum end offset of a trace. Table III shows the access ranges of Trace-A and Trace-B. The minimum start offset and maximum end offset for Trace-A (Trace-B) are 2048 sectors (0) and around 7.64 TiB (2.91 TiB), respectively. The maximum end offsets are consistently within SSD capacities (Table I). As the maximum end offset for Trace-A is close to its SSD capacity and the data is generally stored contiguously (see Figure 2), it indicates the SSD in Trace-A is almost full.

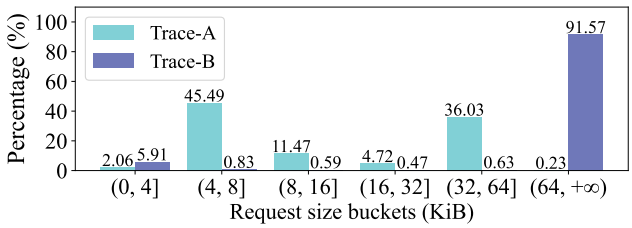
Request size distribution. To examine the request size distribution, we divide request sizes into several buckets, each corresponding to a power of 2 KiB starting from 4 KiB. The last bucket includes all sizes greater than 64 KiB. Figure 1 shows the ratio of request sizes with different buckets for both traces. For read requests, the small requests less than or equal to 8 KiB for Trace-A account for the majority, i.e., 83.98%,

Trace name	Min start offset (sector)	Max end offset (sector)
Trace-A	2048	16499650560 (≈ 7.68 TiB)
Trace-B	0	6250000000 (≈ 2.91 TiB)

TABLE III: The access ranges of Trace-A and Trace-B.



(a) Read request size distribution



(b) Write request size distribution

Fig. 1: The request size distribution of Trace-A and Trace-B.

while the large requests greater than 64 KiB for Trace-B account for the majority, i.e., 98.21%. For write requests, Trace-A includes both small (between 4 KiB and 8 KiB) and large (between 32 KiB and 64 KiB) requests, accounting for 45.49% and 36.03%, respectively, while the large requests greater than 64 KiB for Trace-B still account for a large fraction, i.e., 91.57%. The difference of read request sizes for Trace-A and Trace-B depends on the upper-level applications.

Block access popularity. We divide the disk capacity into a sequence of fixed-sized blocks (e.g., 100 GiB) and count the number of accesses to each block. Figure 2 shows the block access popularity of Trace-A and Trace-B. Recall that the capacities of SSDs for Trace-A and Trace-B are 8 TiB and 4 TiB, respectively. Thus, the access range of Trace-A is larger than that of Trace-B. We observe that the accesses in Trace-A are nearly even but not continuous. In contrast, the accesses in Trace-B are continuous but the number of accesses varies significantly across blocks.

IOPS. To further understand the workload characteristics, we segment the traces into fixed-size time slices (e.g., 10 ms) and examine the number of I/O operations per second (IOPS) over time slices. For the convenience of presentation, we uniformly sample 500 time slices. Figure 3 shows that Trace-A includes a balanced workload of read and write IOPS, while Trace-B records a write-intensive workload. The peak and average read IOPS of Trace-A slightly surpass the write IOPS, while both the peak and average read IOPS of Trace-B are significantly lower than the write IOPS.

B. Access Patterns

We next examine *access patterns*, referring to the sequence of data access. Access patterns are typically divided into two types: (i) *sequential access*, where data blocks are accessed

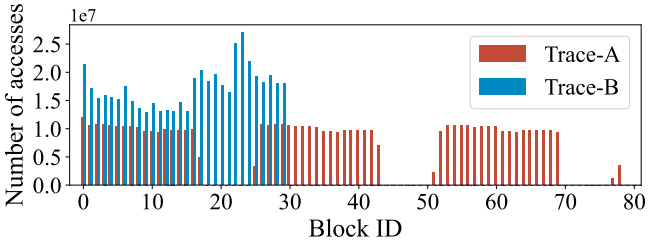
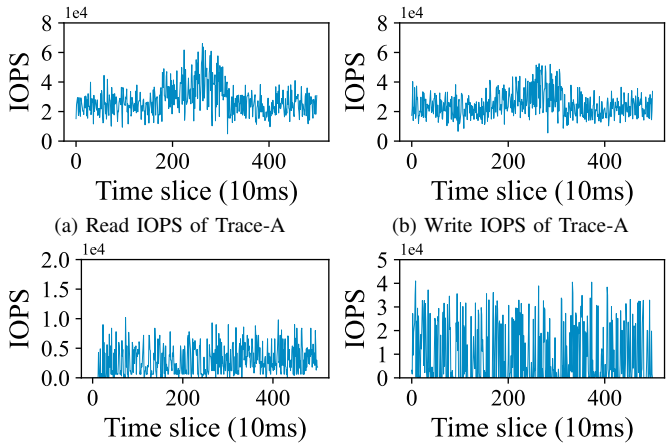


Fig. 2: The block access popularity of Trace-A and Trace-B.



(a) Read IOPS of Trace-A (b) Write IOPS of Trace-A

(c) Read IOPS of Trace-B (d) Write IOPS of Trace-B

Fig. 3: The IOPS sequences of Trace-A and Trace-B.

in a specific order, and (ii) *random access*, where data blocks are accessed without a predictable order. To provide an in-depth analysis of access patterns, we consider the following four categories of access patterns.

- *Repeated access pattern.* It is referred to as a sequence of records with the same start offset and request size.
- *Strict sequential access pattern.* It is referred to as a sequence of records with contiguous accessing addresses, where the start offset of the following record equals the end offset of the preceding record.
- *Jumping sequential access pattern.* It is referred to as a sequence of records with *nearly* contiguous accessing addresses, i.e., a small gap between the end offset of the preceding record and the start offset of the following record within a threshold (denoted by δ).
- *Random access pattern.* Any access that does not fall into the above three categories is considered as random access.

We define some notations to capture the spatial and temporal characteristics of the first three categories of access patterns. Let α denote the minimum threshold for the total amount of accessed data, ensuring that a sufficient volume of data is considered. Also, let β denote the maximum threshold of the difference in record indices between two adjacent records within a specific access[1], ensuring that the accesses occur within a short time frame.

[1]Two records that are accessed consecutively in a specific access pattern may not appear adjacent in the traces, as there may be other intervening requests between them.

Record index	Timestamp	Type	Start offset	Request size	End offset
1	23886837	W	512	8	520
2	23886869	W	524	8	532
6	23886934	W	532	8	540
10	23887081	W	544	8	552
12	23887110	W	560	8	568

TABLE IV: Example of jumping sequential access.

Table IV shows an example of a jumping sequential access by setting α = 40 sectors, β = 5, and δ = 8 sectors. The end offset of the first record is 520 (i.e., start offset + request size = 512 + 8 = 520). The gap between the end offset of the first record and the start offset of the second is 524 − 520 = 4, which is less than δ. The total number of accessed data is 5 records × request size = 5 × 8 = 40, which is equal to α. Also, the differences between each pair of adjacent records are less than β.

Calculation methodology. The repeated access pattern can be easily identified by analyzing the start offset and request size when traversing the traces. In contrast, to detect sequential access patterns (strict or jumping), we formulate the problem as a directed acyclic graph (DAG). At a high level, each vertex represents a unique record, and the edges among vertices describe that such records satisfy the condition of (strict or jumping) sequential access pattern. Let v_i ($i \geq 0$) be a vertex representing the record i, where i is the record index. Let $e_{i,j}$ ($i, j \geq 0$) be a directed edge from v_i to v_j indicating that the end offset of v_i is less than the start offset of v_j within δ, where $\delta = 0$ for the strict sequential access pattern. Also, each edge is associated with a weight (denoted by $w_{i,j}$) to represent the gap between the end offset of v_i to the start offset of v_j, where $0 \leq w_{i,j} \leq \delta$.

Access pattern analysis. Given that our traces include over 500 million records (Section II), analyzing the entire traces is time-consuming. Thus, we select the first 10 million records for each trace. We then divide the traces into non-overlapping blocks, each containing 50,000 records, to enable parallel processing for analyzing I/O patterns. By default, we set α as 1024 sectors, β as 200, and δ as 128 sectors. For each block, we construct a DAG to investigate the access patterns. Note that access patterns at the beginning of subsequent blocks may overlap with those at the end of preceding blocks, as they would belong to the same paths if the blocks were not divided. However, these overlaps are negligible in our analysis.

Table V shows the overall statistics of access patterns in the traces. We analyze the path count, average path length, and total number of records for each type of access pattern, excluding random access patterns. For random access patterns, we only report the total number of records for random access patterns as they do not form distinct paths. There are no repeated access patterns in either trace for both reads and writes. For sequential accesses, Trace-A includes only jumping sequential reads and strict sequential writes, suggesting the presence of fragments after the garbage collection (as known as the fragmentation issue [19]). In contrast, Trace-B, being write-intensive (Section III-A), contains no sequential reads but both strict and jumping sequential writes. Moreover,

Types	Categories	Metrics	Trace-A	Trace-B
Read	Repeated access pattern	# paths	0	0
		Avg. path length	0	0
		# records	0	0
	Strict sequential access pattern	# paths	0	0
		Avg. path length	0	0
		# records	0	0
	Jumping sequential access pattern	# paths	692	0
		Avg. path length	32	0
		# records	21,992	0
	Random access	# records	5,210,423	1,508
Write	Repeated access pattern	# paths	0	0
		Avg. path length	0	0
		# records	0	0
	Strict sequential access pattern	# paths	1,988	43,313
		Avg. path length	22	29
		# records	43,409	1,258,842
	Jumping sequential access pattern	# paths	0	7,139
		Avg. path length	0	51
		# records	0	362,778
	Random access	# records	4,724,176	8,376,872

TABLE V: Overall statistics of access patterns in traces.

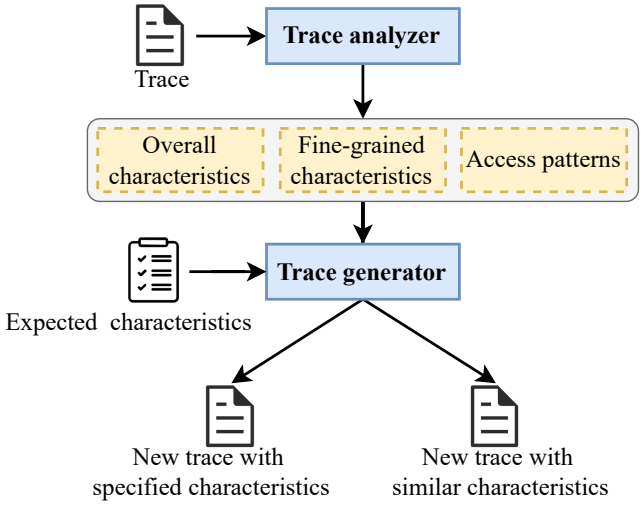


Fig. 4: Architectural overview of TRACEGEN.

strict sequential writes are more prevalent than jumping sequential writes, helping to minimize fragments, while the path length of jumping sequential writes is longer than that of strict sequential writes. Also, random access patterns dominate both traces, maximizing SSD capacity utilization.

IV. TRACE GENERATOR

Based on the trace characterization (Section III), we design a trace generator tool, named TRACEGEN, to generate a new trace with the desired characteristics. Figure 4 provides an architectural overview of TRACEGEN, which consists of two components, *trace analyzer* and *trace generator*. The trace analyzer takes the real-world trace as input and performs functions to analyze trace characteristics and access patterns. The trace generator produces two types of traces: (i) a new trace configured to meet specified characteristics (Section IV-A), and (ii) a new trace that retains similar characteristics to those identified by the trace analyzer (Section IV-B).

A. Generating Traces with Specified Characteristics

To simulate the workload of various real-world applications, TRACEGEN supports three categories of workload characteristic configurations: (i) access range, (ii) the request size distribution, and (iii) IOPS. These configurations are essential for several reasons: varying the access range enables the simulation of application migration from an old storage device to a new one; varying the request size distribution and IOPS allow us to simulate different workloads of upper-level applications and examine their impact on storage system performance. In addition, TRACEGEN maintains the remaining characteristics (i.e., block access popularity and access patterns) similar to the input trace if they are not affected by the above three configurations[2].

To meet the specified characteristic requirements, TRACEGEN first samples records from the input trace and synthesizes a new trace from these sampled records and the input trace. Specifically, TRACEGEN divides the records in the input trace into four sets (for read or write operations) based on the types of access patterns: the *repeated access set*, *strict sequential access set*, *jumping sequential access set*, and *random access set*. It then selects records from each set according to the specified characteristics. Next, we introduce how to generate a new trace for each category of specified characteristics.

Access range. TRACEGEN generates a new trace by specifying a *new access range* through remapping the start offset of each record in the input trace to a new start offset. In particular, it computes the access range of the input trace (referred to as the *old access range*) via the trace analyzer. It takes a new specified minimum start offset and a maximum end offset as inputs, where their difference is the new access range. The old access range is then scaled to the new access range by a factor f, where $f = \frac{\text{length of new access range}}{\text{length of old access range}}$. For the repeated access set and random access set, the new start offset of each record is calculated by $(\mathsf{old\_start\_offset} - \mathsf{old\_min\_start\_offset}) \cdot f + \mathsf{new\_min\_start\_offset}$. In contrast, to preserve the sequential access patterns, TRACEGEN calculates the new start offset for records in the strict or jumping sequential access sets by $\mathsf{old\_start\_offset} - \mathsf{old\_min\_start\_offset} + \mathsf{new\_min\_start\_offset}$ without scaling. Note that we do not modify the request size for each record. If the end offset of a record after remapping exceeds the SSD capacity limit, TRACEGEN adjusts the start offset of the record to ensure it remains within this limit by setting it to $\mathsf{new\_max\_end\_offset}$ minus request size. The impact of such adjustment is negligible on repeated and sequential access patterns.

Request size distribution. TRACEGEN generates a new trace with specifying the request size distribution through modifying the request size of records. Recall in Section III-A that we divide the requests into several buckets. The main idea is to adjust request sizes in buckets so that those

[2]For example, configuring the access range may affect the block access popularity and access patterns, while configuring a new read or write IOPS may change all the remaining characteristics.

exceeding the specified proportion are reduced to sizes that fall within the specified proportion. Specifically, TRACEGEN first calculates the number of records that should be added or removed in each bucket based on the desired request size distribution. To maintain the integrity of repeated and sequential access patterns, TRACEGEN randomly selects the records only from the random access set in the bucket with exceeding the required proportion, where the number of such selected records equals the number of removed records in that bucket. TRACEGEN modifies the request sizes of the selected records to fall within the bucket whose proportion is below the specified limit.

IOPS. TRACEGEN generates a new trace with specifying the IOPS while adhering to two constraints. First, the IOPS in each time slice (i.e., 10 ms) should align with the configured value. Second, the access range and request size distribution should match those in the input trace. Given that a new trace is configured with increasing IOPS, TRACEGEN achieves this by sampling records, appending the sampled records, redirecting timestamps, and adjusting the request size distribution.

- *Sampling records.* TRACEGEN calculates the number of records to be augmented. To maintain the proportion of each category of access patterns, it samples records proportionally from the four sets of access patterns, respectively. For the random access set, TRACEGEN randomly selects the records. For repeated and sequential access sets, it preserves as many complete access patterns as possible. Initially, it retains all records within an access pattern. If the number of retained records exceeds the required number of augmented records, it then retains the initial records up to the number needed for augmentation.
- *Appending sampled records.* TRACEGEN concatenates the four sampled sets of access patterns and resets the index of the selected records, starting from the end of the input trace, without shuffling the selected records across the sampled sets of access patterns. It then appends the sampled records to the end of the input trace.
- *Redirecting timestamps.* TRACEGEN redirects the timestamps for all records to meet the requirements of IOPS. Given that the input trace contains t time slices. The number of records covered in the j-th time slice is denoted by N_j ($j \geq 1$). Suppose that the required IOPS is x times the IOPS of the input trace, where $x \geq 1$. The new trace should include $x \cdot \sum_{j=1}^{t} N_j$ records, where the first $\sum_{j=1}^{t} N_j$ records are from the input trace and the subsequent $(x-1) \cdot \sum_{j=1}^{t} N_j$ are *sampled* from the input trace. TRACEGEN then modifies the timestamps in the j-th time slice to the range between $(x \cdot \sum_{k=1}^{j-1} N_k + 1)$ and $x \cdot \sum_{k=1}^{j} N_k$, where k indexes the previous time slices and $N_0 = 0$ for initialization.
- *Adjusting the request size.* TRACEGEN examines whether the request size distribution changes. If so, TRACEGEN adjusts the request size distribution of the new trace to match that of the input trace.

Suppose that a new trace is configured with decreasing IOPS. Unlike the process for increasing IOPS, TRACEGEN does not perform sampling records and appending sampled records. Instead, the new trace includes the same number of records as the input trace, but decreases the number of IOPS via stretching the timestamps in each time slice of input trace. In particular, it redirects timestamps for all records of the input trace as follows. Still let x be the specified times the IOPS of the input trace, where $0 < x < 1$. TRACEGEN modifies the timestamps in the j-th time slice to the range between (round$(x \cdot \sum_{k=1}^{j-1} N_k) + 1)$ and round$(x \cdot \sum_{k=1}^{j} N_k)$.

By combining the configurations of trace characteristics, TRACEGEN allows for the simultaneous specification of the three categories of characteristics, enabling the simulation of real-world workloads.

B. Generating Long Traces with Similar Characteristics

Replaying a sufficiently long I/O trace can effectively simulate real-world workloads [13]. However, the complexity and scale of production systems make the collection of extensive traces both time-consuming and resource-intensive. To address this, TRACEGEN generates long traces based on the existing traces by configuring the access range and number of records while ensuring that characteristics such as the request size distribution, mean IOPS, and access patterns closely align with those of input trace.

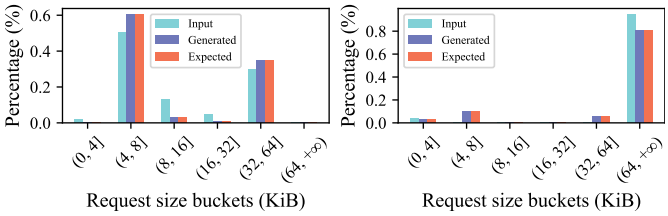
The primary objective of generating extended traces from an input trace is to predict a sequence of IOPS for the near future. TRACEGEN begins by calculating the workload characteristics of the input trace using the trace analyzer. It then formulates the IOPS prediction as a time-series regression problem, employing a long short-term memory (LSTM) model [10] for this task. The LSTM is effective in capturing long-term dependencies within the data and has been successfully applied in various time-series forecasting scenarios [6], [7], [20]. The model takes a sequence of IOPS over time slices from the input trace as input, with the output being a predicted sequence of IOPS for the near future. TRACEGEN utilizes the limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm [26], a quasi-Newton optimization method that uses limited memory to preserve characteristics similar to those of the input trace. It adjusts both the mean and root mean square error (RMSE) of the new IOPS sequence, ensuring that the mean aligns with that of the input trace to maintain consistent workload intensity, while the RMSE remains similar to preserve the fluctuation patterns. By configuring the IOPS based on the new predicted sequence, TRACEGEN generates traces with the specified IOPS (Section III-A).

C. Implementation Details

We implement a TRACEGEN prototype in Python with around 7500 LoC including the trace analyzer and trace generator. We realize the trace analyzer using Python libraries including Numpy [18], Pandas [17] PyQt [22] and Matplotlib [24] to facilitate data analysis and visualization tasks. Also, we realize the prediction model of LSTM [10] in

Trace name	Min start offset (sector)	Max end offset (sector)
Trace-A	2048	16499636272 (≈ 7.68 TiB)
Generated trace	8589934592 (= 4 TiB)	34359737217 (≈ 16 TiB)
Expected trace	8589934592 (= 4 TiB)	34359738368 (= 16 TiB)
Trace-B	0	6249988096 (≈ 2.91 TiB)
Generated trace	8589934592 (= 4 TiB)	34359738318 (≈ 16 TiB)
Expected trace	8589934592 (= 4 TiB)	34359738368 (= 16 TiB)

TABLE VI: Exp#1 (Effectiveness of generating a trace with specified access range).



(a) Trace-A (b) Trace-B

Fig. 5: Exp#2 (Effectiveness of generating a trace with specified write request size distribution).

the trace generator using Darts [2]. In addition, we implement the L-BFGS [26] algorithm using SciPy [12].

V. Evaluation

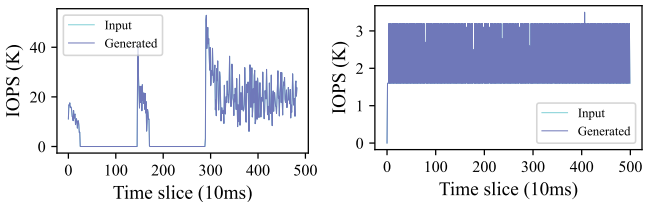
In this section, we present trace-driven evaluation results on the accuracy of trace generation of TraceGen. We summarize our findings as follows.

- TraceGen generates a new trace with specifying the access range (Exp#1), request size distribution (Exp#2), or IOPS (Exp#3) with a small error of less than 1%.
- TraceGen also generates a long trace with similar characteristics, where the error is less than 1% (Exp#4).

A. Methodology

We evaluate the effectiveness of TraceGen by comparing the characteristics of the generated trace against the expected trace. Specifically, we use the trace analyzer to calculate the characteristics of the generated trace and then compare these with those of the expected trace. Let ϵ represent the error of a characteristic, with $\hat{y_j}$ and y_j denoting the j-th generated and expected value of the characteristic, respectively. Let n be the number of values compared, which varies depending on the metric: it equals one for access range, the number of request size buckets for request size distribution, or the number of time slices for IOPS. The error of a characteristic is calculated by $\epsilon = \frac{1}{n}\sum_{j=1}^{n}\frac{|\hat{y_j}-y_j|}{y_j}$.

For our evaluations, we use both traces and select the first 10 million lines by default due to the memory constraints. We set the default length of a time slice to 10 ms and the threshold of ϵ as 1%, meaning that TraceGen is considered effective if ϵ is below 1%. We conduct the evaluation on a machine running Ubuntu 22.04, equipped with an 8-core 3.6 GHz Intel Core i7-11700K processor, 32 GiB of RAM, and a 500 GB Kingston NVMe SSD.



(a) Trace-A (b) Trace-B

Fig. 6: Exp#2 (Effectiveness of generating a trace with specified the request size distribution). The validation of unchanged write IOPS.

Category of access patterns	Trace-A	Generated trace
Read jumping sequential access pattern	692	710
Write strict sequential access pattern	1,988	2,009
Category of access patterns	**Trace-B**	**Generated trace**
Write strict sequential access pattern	43,313	44,522
Write jumping sequential access pattern	7,139	7,268

TABLE VII: Exp#2 (Effectiveness of generating a trace with specified request size distribution). The validation of the number of paths in access patterns.

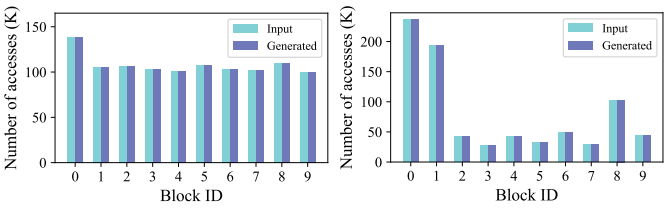
B. Results

Exp#1 (Effectiveness of generating a trace with specified access range). Given that the specified access range is from 4 TiB to 16 TiB, we evaluate the effectiveness of generating a new trace within this range. Table VI shows the access range of input traces, generated traces and expected traces. The minimum start offset and maximum end offset of generated traces are 4 TiB and around 16 TiB, respectively. The difference between the generated trace and expected trace for the minimum start offset is 0. In contrast, the maximum end offset of the generated trace is smaller than that of the expected trace by 50 and 1151 sectors based on Trace-A and Trace-B, respectively, since the maximum end offset is within the capacity limit. Also, other characteristics (i.e., the request size distribution and IOPS) of the new trace remain unchanged.

Exp#2 (Effectiveness of generating a trace with specified request size distribution). We increase the proportion of write requests between 4 KiB and 8 KiB by 10% and between 32 KiB and 64 KiB by 5%. The proportion of other ranges is adjusted proportionally. Figure 5 shows the write request size distribution for the input trace, generated new trace, and the expected request size distribution. The write request size distributions of new traces are closely aligned with the expected distributions, with the error ϵ less than 1%.

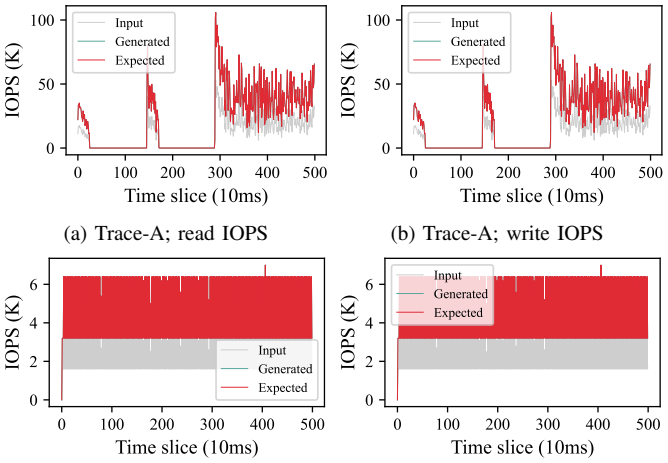
We also verify that other characteristics (i.e., write IOPS, access patterns, and block access popularity) remain consistent. Figure 6 shows that the distribution of write IOPS in the generated traces matches that of the input traces. Table VII shows that the numbers of paths for access patterns are comparable between each pair of the input and generated trace. Figure 7 shows the number of accesses in block access popularity. For clarity, we limit the evaluation to the first 500 GiB of data, with each block sized at 50 GiB. Our observations confirm that the block access popularity distribution of the generated trace aligns with that of the input trace.

Exp#3 (Effectiveness of generating a trace with specified IOPS). Given that the read IOPS of the new trace are set to



(a) Trace-A (b) Trace-B

Fig. 7: Exp#2 (Effectiveness of generating a trace with specified request size distribution). The validation of unchanged block access popularity.



(a) Trace-A; read IOPS (b) Trace-A; write IOPS

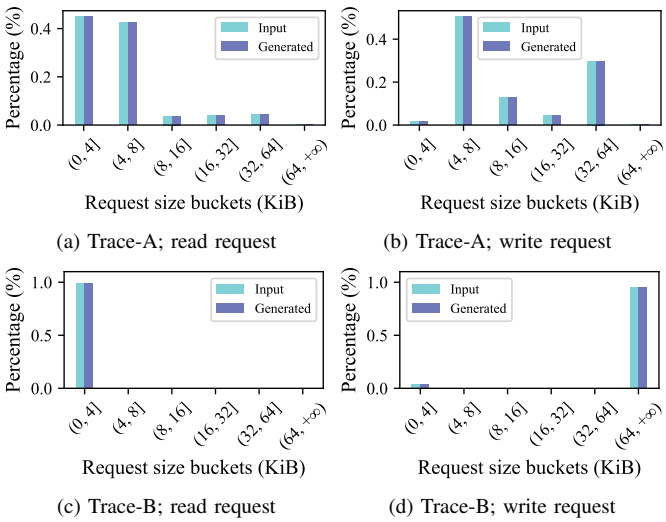
(c) Trace-B; read IOPS (d) Trace-B; write IOPS

Fig. 8: Exp#3 (Effectiveness of generating a trace with specified IOPS). The validation of IOPS alignment between generated and expected traces.

1.5 times that of the input trace and the write IOPS are set to twice that of the input trace, we evaluate the effectiveness of generating the trace under these IOPS conditions. Figure 8 shows that the read and write IOPS of the generated trace closely align with the expected trace for both traces with an error ϵ of nearly 0% (less than 1%). Since the read and write operations of Trace-A are relatively balanced, the distribution of read IOPS for Trace-A is similar to that of write IOPS. For Trace-B, the read IOPS are nearly 0, except for the first time slice, while the write IOPS in the generated trace generally range from 3200 to 6400, which is twice the value of the input trace, thus meeting the specified IOPS requirements.

We also verify that the other characteristics remain unchanged. Figure 9 shows that the read and write request size distributions in the new trace are aligned with those of input traces, with an error ϵ of less than 1%. Since the additional trace records in the new trace are derived from the input trace, the access range of the new trace remains unchanged. In addition, the number of paths of access patterns for the generated trace are comparable with those for the input trace, which we omit in plots due to the space limit.

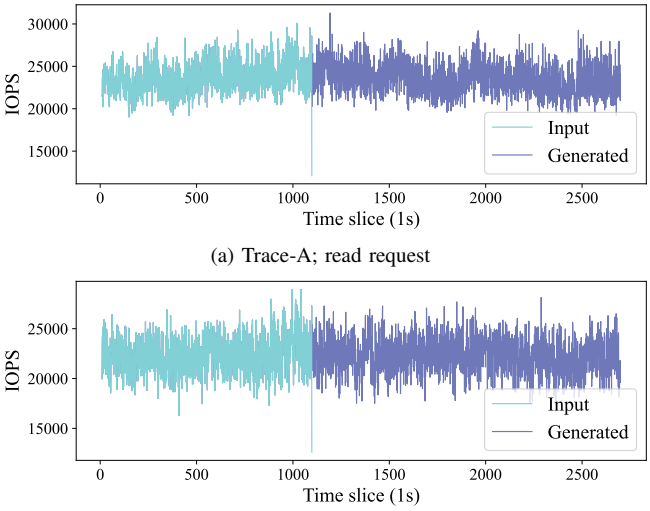
Exp#4 (Effectiveness of generating a long trace with similar characteristics). To evaluate the effectiveness of TRACEGEN for generating long traces with similar characteristics, we select the first 50 million lines of Trace-A as



(a) Trace-A; read request (b) Trace-A; write request

(c) Trace-B; read request (d) Trace-B; write request

Fig. 9: Exp#3 (Effectiveness of generating a trace with specified IOPS). The validation of unchanged request size distributions for specified IOPS.



(a) Trace-A; read request

(b) Trace-A; write request

Fig. 10: Exp#4 (Effectiveness of generating long traces with similar characteristics). The IOPS sequences of input trace and generated trace.

the input trace, which covers the records in the time span of about 18 minutes. We configure TRACEGEN to generate an additional of 27 minutes of trace and extend the input trace to a new trace of 45 minutes. We also adjust the maximum value of access range from 8 TiB to 16 TiB to simulate the change in the capacity of the underlying storage medium. The length of a time slice is set to one second.

We evaluate the effectiveness of generating a long trace by examining its access range, request size distribution and IOPS. First, the access range of the new trace aligns closely with that of the input trace, with an error of nearly 0%. The request size distribution of the new trace is also similar to the input trace, with an error of 0.01% for both read and write request distributions. Additionally, we verify the time span of the new trace for IOPS. Figure 10 shows the read and write

	Average read IOPS	Average write IOPS
Input trace	23654	22410
Generated trace	23595	22374
Error	0.25%	0.16%

TABLE VIII: Exp#4 (Effectiveness of generating long traces with similar characteristics). The average IOPS of input trace and generated trace.

IOPS of both the input and new traces across time slices. The new trace spans 2,700 time slices, equivalent to a 45-minute duration. Furthermore, Table VIII presents the average read and write IOPS for both the input and new traces, revealing a minimal error of less than 1% (specifically, 0.25% for read IOPS and 0.16% for write IOPS).

VI. Related Work

We review previous studies that are related to ours from two aspects: workload measurement and trace generation.

Workload measurement. Existing workload measurement tools mainly characterize the storage system performance. Flexible I/O tester (FIO) [3] measures the performance of file systems and block devices by performing a particular type of I/O operations specified by users. IOZone [5] is a benchmarking tool that measures and generates various file operations. Iometer [4], developed by Intel, serves as an I/O subsystem measurement and characterization tool for both single and clustered systems. IOscope tracer [21] characterizes I/O patterns through filtering-based profiling using fine-grained criteria inside Linux kernel. Such measurement tools generate various workloads, including random and sequential read/write operations with different request sizes. However, they cannot generate the workloads of real applications. In contrast, TraceGen generates new traces based on existing ones, resulting in more complex workloads with similar characteristics to the real-world application workloads.

Trace generation. Prior studies generate traces based on statistical approaches [13], [15], [16] and deep learning algorithms [11]. TraceRAR [13] generates long traces based on the correlation analysis between different characteristics. ScalaIOExtrap [15], [16] extrapolates the trace data to a large-scale cluster based on a mathematical model. Huang *et al.* [11] generate traces with I/O-related characteristics (e.g., request size and offset) and directory/file semantics (e.g., file path and ID) based on Recurrent Neural Networks. The difference between our work and the above studies is that TraceGen generates both new traces with specified characteristics and long traces with similar characteristics to the input traces (instead of only extended traces).

VII. Conclusion

We present TraceGen, an evaluation tool for analyzing and generating I/O traces for block-level storage systems. Based on the trace characterization, it generates new traces with specified characteristics and long traces with similar characteristics. Our evaluation shows that TraceGen accurately generates new traces in various configurations.

Acknowledgements

This work was supported in part by National Key Research and Development Program (No.2022YFB2702101), National Natural Science Foundation of China Major Program (92152301), and Fundamental Research Funds for the Central Universities. Shujie Han and Xiao Zhang are the corresponding authors.

References

[1] Blktrace. git://git.kernel.dk/blktrace.git.
[2] Darts. https://github.com/unit8co/darts.
[3] FIO. https://github.com/axboe/fio.
[4] Iometer. http://www.iometer.org.
[5] IOzone. http://www.iozone.org.
[6] S. Alhirmizy and B. Qader. Multivariate time series forecasting with lstm for madrid, spain pollution. In *Proc. of IEEE ICCISTA*, 2019.
[7] R. Casado-Vara, A. Martin del Rey, D. Pérez-Palau, L. de-la Fuente-Valentín, and J. M. Corchado. Web traffic time series forecasting using lstm neural networks with distributed asynchronous training. *Mathematics*, 9(4):421, 2021.
[8] C. Chung, J. Koo, J. Im, Arvind, and S. Lee. Lightstore: Software-defined network-attached key-value drives. In *Proc. of ACM ASPLOS*, 2019.
[9] A. Haghdoost, W. He, J. Fredin, and D. H. Du. On the Accuracy and Scalability of Intensive I/O Workload Replay. In *Proc. of USENIX FAST*, 2017.
[10] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
[11] K. Huang, X. Li, M. Yuan, J. Zhang, and Z. Shao. Joint Directory, File and IO Trace Feature Extraction and Feature-based Trace Regeneration for Enterprise Storage Systems. In *Proc. of IEEE ICDE*, 2024.
[12] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org.
[13] B. Li, F. Toussi, C. Anderson, D. J. Lilja, and D. H. Du. TraceRAR: An I/O Performance Evaluation Tool for Replaying, Analyzing, and Regenerating Traces. In *Proc. of IEEE NAS*, 2017.
[14] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports. Pegasus: Tolerating skewed workloads in distributed storage with In-Network coherence directories. In *Proc. of USENIX OSDI*, 2020.
[15] X. Luo, F. Mueller, P. Carns, J. Jenkins, R. Latham, R. Ross, and S. Snyder. HPC I/O trace extrapolation. In *Proc. of ACM ESPT*, 2015.
[16] X. Luo, F. Mueller, P. Carns, J. Jenkins, R. Latham, R. Ross, and S. Snyder. ScalaIOExtrap: Elastic I/O tracing and extrapolation. In *Proc. of IEEE IPDPS*, 2017.
[17] W. McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14:1–9, 2011.
[18] T. Oliphant. *Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.
[19] J. Park and Y. I. Eom. Filesystem fragmentation on modern storage systems. *ACM Transactions on Computer Systems*, 41(1-4):1–27, 2023.
[20] A. Sagheer and M. Kotb. Time series forecasting of petroleum production using deep lstm recurrent networks. *Neurocomputing*, 323:203–213, 2019.
[21] A. Saif, L. Nussbaum, and Y.-Q. Song. Ioscope: A flexible i/o tracer for workloads' i/o pattern characterization. In *Proc. of ISC High Performance Workshops*, 2018.
[22] M. Summerfield. *Rapid GUI programming with Python and Qt: the definitive guide to PyQt programming*. Pearson Education, 2007.
[23] R. Talwadker and K. Voruganti. {ParaSwift}: File {I/O} trace modeling for the future. In *Proc. of USENIX LISA*, 2014.
[24] S. Tosi. *Matplotlib for Python developers*. Packt Publishing Ltd, 2009.
[25] S. Wang, Z. Lu, Q. Cao, H. Jiang, J. Yao, Y. Dong, and P. Yang. BCW:Buffer-Controlled writes to HDDs for SSD-HDD hybrid storage server. In *Proc. of USENIX FAST*, 2020.
[26] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on mathematical software (TOMS)*, 23(4):550–560, 1997.