Auto-Tuning Parameters for Emerging Multi-Stream Flash-Based Storage Drives Through New I/O Pattern Generations

Janki Bhimani[1], Adnan Maruf[1], Ningfang Mi[2], Rajinikanth Pandurangan[3] and Vijay Balakrishnan[3]

[1]School of Computing and Information Sciences, Florida International University
[2]Department of Electrical and Computer Engineering, Northeastern University
[3]Memory Solutions Lab, Samsung Semiconductor Inc. R&D

Abstract—
In the era of big data processing, more and more data centers in cloud storage are now replacing traditional HDDs with enterprise SSDs. Both developers and users of these SSDs require thorough benchmarking to evaluate and configure the variable parameters of emerging technologies. Multi-stream SSD [2], [3] is the recent development of the SSD industry, which assists in placing data on SSDs in a smart way to improve application performance and SSD endurance. The challenging part to use multi-stream SSDs is to assign stream IDs to incoming writes, such that each stream consists of data with a similar lifetime. The benefit of the stream management algorithms varies over different workloads. Thus, first, we propose a new framework, called Pattern I/O generator (`PatIO`), to capture the enterprise storage behavior that is prevailing across various user workloads, virtualization setup, file systems, and volume managers for the database server applications on flash-based storage. Second, using `PatIO`, we study what type of applications may be benefited by which stream assignment algorithm. Third, we design the framework to automatically tune the variable parameters of different stream identification algorithms of the multi-stream SSDs. Our evaluation shows 20% to 110% of the reward function increase, measuring the cumulative impact on application performance and SSD endurance.

Index terms— Flash Memory, I/O Pattern Generator, Benchmarking, Multi-stream SSDs

1 Introduction

Optimizing the operation of modern cloud storage systems for various big data applications is critical. Evaluating the effect of any storage device firmware or hardware amendments using real system deployment requires a lot of resources, time and efforts towards installation and running of different workloads to test. Moreover, many different virtualization and system setup options in a cloud environment also need to be tested for each workload. The research advancement by evaluating a tiny subset of these possible settings and workloads then becomes very limited. Thus, benchmarking is very important for developers and users of evolving cloud storage.

Most traditional I/O benchmarking tools [4], [5] were designed for hard disk drives (HDDs). Hence, when benchmarking storage, the I/O workloads generated by these tools do not resemble the I/O activities of real workloads on flash-based solid-state drives (SSDs). The main problems are that with multiple design choices at the virtualization and system layer, (1) the data generated by traditional synthetic I/O generators might be too simple, or (2) it demands a lot of time and storage space to generate and store different trace logs for each workload with trace-based I/O generators. Figure 1 compares the variance of I/O size for reads and writes over NVMe SSDs (top) and HDDs (bottom). We run the TPC-H decision support benchmark with twenty-two different queries executed on eight different input tables of various sizes with Apache Spark application. We observe that I/O sizes running on NVMe SSD are completely different from those running on HDD. The size of both read and write I/Os exhibits a periodic pattern when using NVMe SSD. Large read/write I/Os are periodically clustered together, with some idle intervals between I/O size spikes. Therefore, new benchmark methods that capture realistic I/O activities and require fewer resources, time, and efforts are needed.

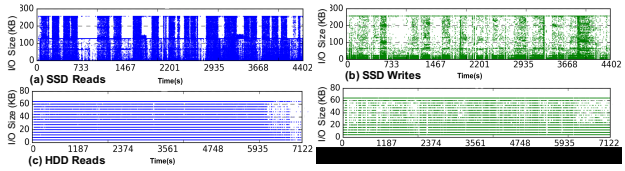


Figure 1: I/O size of TPC-H-Spark with 50G workload comparing (a) SSD reads, (b) SSD writes, (c) HDD reads, and (d) HDD writes.

Motivated by this, first, we propose a new benchmarking framework, called Pattern I/O generator (`PatIO`) to capture the enterprise storage behavior that is prevailing across various user workloads, virtualization setup, file systems, and volume managers for different database server applications on flash-based storage. Second, we integrate `PatIO` with multi-stream SSDs, to study the impact of the various internal stream identification algorithms. Third, we design and integrate the auto-tuning module within multi-stream SSDs to tune the variable parameters of different stream identification algorithms. The main contributions and features of our solution are as follows.

1) Extract and Generate I/O Patterns: An I/O layout pattern is the property of an I/O workload, which is the key to the application performance (efficiency) and storage health (endurance). Multiple dimensions, including disk offset, time, read/write rate (also called data temperature) and I/O size, frame an I/O layout pattern. Each workload may have many different patterns representing different real database activities like compaction and log management. We collect and study I/O patterns of different big data workloads with

This work was initiated during Janki Bhimani's internship at Samsung Semiconductor Inc. [1]. This work was partially supported by the National Science Foundation Awards CNS-2008324 and CNS-2008072 and the National Science Foundation Career Award CNS-1452751.

a different setup for various database server applications using flash-based cloud storage. Specifically, we ran more than 1000 workloads of each database server application with different SSDs of various capacities (from 100GB to 1TB) and different file systems such as xfs and ext4. Our pattern extraction methodology involves a three-step process, i.e., dissect, construct, and integrate. We dissect the overall I/O activities of real workloads to extract distinct repetitive I/O patterns. Then, we identify different input features of an I/O generating engine (e.g., FIO, a popular I/O engine), to construct jobs that generate I/Os resembling different I/O activities of a real workload. We generate unique I/O patterns using various combinations of multiple I/O jobs. We finally construct a pattern warehouse as the collection of these I/O patterns. Different combinations of synthetically generated I/O patterns can reproduce comprehensive characteristics of various real workloads.

2) Ensure Scalability and Usability: The second contribution of our work is to make `PatIO` scalable to generate I/Os over different sizes of storage disks and different storage volumes consisting of multiple SSDs in cloud storage. The user is allowed to specify the storage size and the expected execution time of the desired workload. `PatIO` can then automatically change all I/O jobs at the low level and modify the necessary input options in I/O patterns on-the-fly for all jobs. To provide an easy-to-use experience, we further develop a graphical user interface (GUI) and an automatic plotting wrapper for `PatIO`. It decouples the user from the complexity of underlying code modification, integration, compilation, and execution. The open-source of our benchmarking framework will be available at `GitHub`.

3) Ensure Expandability and Integrability: We aim to capture a variety of different patterns from samples of I/O workloads that we know in a pattern warehouse. It is also easy to expand our pattern warehouse by adding new patterns based on the new knowledge of applications and workloads. In addition, our I/O generator can be integrated into different environments of the storage industry to fasten research, development, and evolution phases. For example, a possible deployment could be to run `PatIO` on FIO using FPGAs for next-generation SSD hardware development (e.g., key-value SSDs) or to use `PatIO` for a firmware configuration such as the proportion of over-provisioning in SSDs.

4) Practical Application: We further enhance our framework to evaluate the efficiency and endurance of multi-stream SSDs. We evaluate the performance of two existing automatic stream assignment algorithms known as auto-stream: SFR and MQ proposed in [6], for different I/O patterns. It helps service providers of cloud storage learn what types of workloads are more benefited by using flash-based multi-stream SSDs. It also helps users of cloud storage to understand if the stream identification is appropriately made, and how their stream assigning algorithms can be further improved to further leverage performance by flash-based SSDs.

5) Auto-tuning Module: Our final contribution is to construct an infrastructure for auto-tuning internal variable parameters of the multi-stream SSDs. In particular, we build a peripheral infrastructure to tune variable parameters for those two existing stream assignment algorithms [6]. In our experiments, we observe that without proper tuning, the benefits of multi-stream technologies may be restrained when some factors like the SSD version, SSD capacity, underlying firmware are changed. Moreover, tuning manually could take a very long time, like a couple of months. Motivated by this, we build an infrastructure on top of `PatIO` to support automatic tuning for different I/O patterns.

We evaluate our framework by using different containerized workloads running using standalone and simultaneous database applications such as MySQL, Cassandra, and ForestDB. Specifically, we compare I/O characteristics (such as arrival address, I/O size, and read over write ratio), and I/O performance (such as throughput, average latency, tail latencies and Write Amplification Factor (WAF)) of generated workloads with those of real-world workloads. Finally, we discuss the scalability of workloads generated by `PatIO` to adapt to the SSDs of different capacities.

The rest of the paper is structured as follows. Section 2, discusses the existing techniques. Section 3 presents the `PatIO` architecture. Section 4 evaluates our technique. The research direction enabled by `PatIO` towards evolving flash based storage devices is explained in Section 5. We describe and evaluate our auto-tuning module in Section 6. Finally, we draw our conclusions in Section 7.

2 Related Works

Most benchmarking techniques [5], [7]–[17] use samples of proprietary data to first record the overall average statistics (e.g., average I/O rate and average read to write ratio) of real workloads and then reproduce I/Os synthetically based on averages. Such benchmarking tools results in a uniform distribution of I/Os on disk and a constant throughput during the execution. Thus, we argue that although these synthetic I/O generators operate with low overhead and negligible resource requirements, they are not sufficient to capture the working of modern cloud workloads on evolving flash technology. Thus, SSDs behave differently on these traditional synthetic workloads, compared with what they do on real-world workloads.

Apart from widely used synthetic I/O generators, another popular benchmarking technique in the storage industry is *workload replay*. The replay tools [4], [18]–[21], record the characteristics of real I/O data for different granularities like blocks, data chunks, and sectors. By using the recorded logs, these tools can almost exactly replicate I/O activities of real workloads. Recent replay tools [19] have enhanced capability to generate additional data dependency graphs and be able to accurately replay the I/O workload. This technique has high precision. However, capturing all possible workload traces to frame a trace repository is challenging. Storing these traces also demands a large amount of storage resources. One way to generate "real" I/Os with a low storage requirement for characterizing data would be to increase the recording granularity of I/O characteristics and get a short trace that only abstracts the characteristics of a real workload. However, it still requires efforts to run different real application workloads to record traces and needs more storage resources when the number of traces increases.

Application level benchmarks [7], [18], [22]–[27] strive to mimic I/O behaviours of specific applications, but require time and efforts for installation, configuration and database

Table 1: PatIO vs existing storage benchmarking tools (Bench. - Benchmarking, Req. - Require, Endu. - Endurance, Cap. - Capture, Var. - Variance, Gen. - Generate, Int. - Interface, Across - Acr.)

Bench. Tool	Flash based Bench.	Cap. Var. Acr. SSDs	Cap. Var. Acr. Time	Cap. Endu. of SSDs	Auto Gen. Output Plots	GUI Int.	Req. Trace Logs
PatIO	✓	✓	✓	✓	✓	✓	
Ezfio [14]	✓				✓	✓	
IOA [11]					✓	✓	
Iom. [8]					✓	✓	
SDG. [18]	✓		✓				
hfp. [19]	✓						✓
blkr. [4]							✓
CH [10]	✓				✓		✓
FB [29]	✓		✓				
DB [33]	✓		✓				

loading before running. YCSB [22] is a framework and common set of workloads to evaluate the performance of different "key-value" and "cloud" serving stores. Another widely used database management system (DBMS) benchmark, DBbench [23], can evaluate the performance of a plurality of DBMS's stores both DBMS independent and DBMS specific files in computer memory.

Filesystem level benchmarks [28], [29] spawn several threads or processes doing a particular type of I/O action as specified by the user. They help to answer the trivial question such as, "Which file system is better." However, our focus is to characterize the performance of SSDs, so it is useful to compare with the benchmarks that report bandwidth and latency when reading from and writing to the disk in various-sized increments without filesystem layer.

Block level benchmarks [19], [28], [30]–[32] provide the ability to record and replay block-level I/Os. However, they have heavy overheads to maintain ordering, CPU mappings, and time-separation of I/Os. BlkTrace [30] provided the ability to collect detailed traces from the kernel for each I/O processed by the block IO layer. HFPlayer [19] used the generated dependency graph and can replay the I/O workload in a scaled environment. Buttress [31] used synchronous I/O to replay block traces.

In contrast, PatIO does not require to store, read, and follow any I/O trace file while regenerating I/Os. Thus, PatIO is more cost-efficient and less time-consuming compared to existing I/O replay techniques, and more importantly, it is much more precise compared to naive I/O generators (i.e., naive FIO [5]). We finally summarize the features of PatIO and different popular existing benchmarking techniques in Table 1. To the best of our knowledge, this is the first attempt to analyze and improve the impact of variable parameters of the internal algorithms of emerging SSDs such as multi-stream SSDs.

3 PatIO Framework

In this section, we discuss the overall architecture and then elaborate on the three components of methodology - dissect, construct, and integrate for PatIO [1]. The driving force of building PatIO is to study the diversities of I/O activities using different data processing workloads on flash-based cloud storage and then regenerate I/O characteristics representing the complex transaction forms like compaction, log management, and key-value store that are performed by different database applications on flash-based SSDs.

In PatIO, we extract the common I/O patterns by observing many different workloads of various applications over different SSDs. We carefully design ready to use I/O workloads to replicate many common I/O patterns observed while running real applications. Particularly, the combination of our I/O patterns replicates the cumulative activities generated by different workloads of any applications. Thus, PatIO strives to capture the common characteristics of a group of similar workloads rather than exactly resembling just one particular workload. PatIO is lightweight, as it does not require to record, store, and retrieve logs of I/O activities. PatIO is also designed to be scalable to generate I/O workloads over different storage sizes. The main contributions and features of our solution are as follows.

3.1 Architecture of PatIO

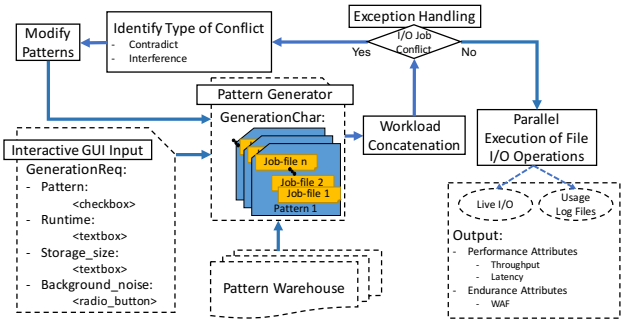


Figure 2: Block diagram: pattern generation

Figure 2 shows the architecture of our framework. First, the front end GUI allows the user to configure a workloads' I/O patterns and its expected execution time and to select the size of the storage disk and the desired level of background noise. The pattern generator then dynamically pulls corresponding files of patterns from the pattern warehouse based on the selected options.

A workload is a combination of single or multiple patterns. Each pattern is a multi-threaded system process and consists of multiple I/O jobs, where a single thread executes each job. Different patterns are executed together to construct a workload. However, before simultaneously executing all jobs of selected patterns, the pattern generator needs to modify the input parameters of these jobs according to the specified execution time and disk size by the user. In addition, the programming commands given by jobs of different processes may have conflicts, e.g., simultaneously writing different values at a specific SSD address. Then, the pattern generator also needs to perform a workload modification to ensure that all job files of the selected workload run correctly during the parallel execution. This whole process of modifying I/O jobs is called workload concatenation.

Figure 2 shows the process of workload concatenation by a loop of events around *Pattern Generator*, which contains exception handling, identifying conflicts, and modifying patterns. Sometimes, job modification to resolve a conflict may cause new conflicts. Thus, workload concatenation is repeated until there are no more conflicts. Later, in Section 3.4, we explain details about job modification and types

1. We use "Disk" interchangeably with "SSD" to represent flash storage throughout this work.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3048303, IEEE Transactions on Computers

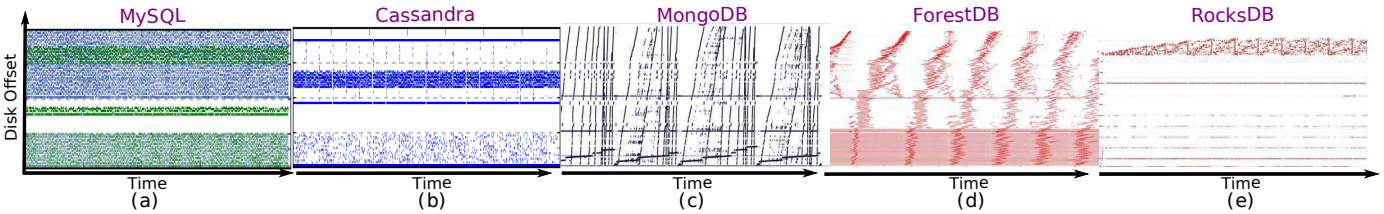


Figure 3: Disk access patterns over time by real data processing applications

of conflicts. Finally, according to the concatenated workload, the I/O engine generates I/Os that will be performed by FIO [5] on back-end SSDs to resemble I/O activities of real parallel applications. `PatIO` also provides detailed reports and graphs to show I/O performance (e.g., throughput, latency, and tail latency) and SSD endurance, such as Write Amplification Factor (WAF). All results can be stored as a backup log for future analysis.

3.2 Study of Real I/O Patterns: Dissect

An I/O pattern is a cluster of I/O activities of a workload that has similar characteristics. Here, we present our observations on I/O patterns of real applications, which inspires our design of `PatIO`. In our study of the I/O activities of various real applications on SSDs, we consider instrumenting general purpose applications such as Kmeans clustering and Pagerank as well as database applications such as MySQL and RocksDB. However, we observed that for generic applications, the processing time is dominated by computation, and intermediate shuffling data generated by them is small and fits in main memory. The data dependencies within such intermediate data that are cached are served from memory. Moreover, these applications perform most of their I/O activities only at the beginning to read the data into memory and at the end to write the final outputs. Thus, we mainly discuss the I/O intensive database workloads whose execution time is dominated by I/O processing.

We study the I/O activities of at least 10 different workloads for each of the 100 different applications on various models and capacities of SSDs. Figure 3 shows the I/O access patterns for some representative real SQL and NoSQL database applications on SSDs. The workload configurations of these real applications are listed in Table 2. We observe that a real application exhibits variance in I/O activities on SSDs (also called disk) over time. For example, some applications perform their I/Os in a uniform horizontal stripe wise fashion, see MySQL and Cassandra in Figures 3 (a) and (b). While some other applications show a periodic pattern of I/O layout over time, such as MongoDB and ForestDB in Figures 3 (c) and (d). Also, there exist applications like RocksDB in Figure 3 (e) that present a horizontal stripe wise pattern. One I/O stripe of RocksDB (see the upper region of Figure 3 (e)) further exhibits a phase-wise pattern of I/O layout where I/O activities slowly start spreading over the disk and then construct a uniform horizontal strip when the workload has run for a prolonged duration. Thus, the diversity in I/O access patterns motivates us to develop a new I/O generator that can capture these I/O behaviors and dynamically generate I/O workloads for different SSD devices.

First, we *dissect* the overall I/O activities of different user workloads, virtualization setup, file systems, and volume managers for various real database server applications on flash-based storage. We identify the prevailing attributes in distinct visual I/O patterns. The I/O characteristics include I/O sizes, I/O densities, ratios of read to write, and I/O inter-arrival time. We analyze the distribution of these I/O characteristics across different address space of flash-based SSDs and the variation of these I/O characteristics over workload execution time. For example, MongoDB (see Figure 3 (c)) comprises of different I/O patterns, such as straight horizontal lines representing overwrites on the same disk offset, and inclined vertical lines across the disk representing a form of sequential writes. To extract different I/O patterns, we perform data classification using the K Nearest Neighbour (K-NN) pattern classifier with different distance measures (such as Euclidean, Manhattan, Chebychev, and Percent disagreement) and then study classification results with different K in K-NN to distinguish various I/O patterns following the majority.

Table 2: Workload configurations of different real applications (KV - key/value, col. - columns)

Application	Workload	Size	Operations	Type
MySQL	TPCC	4200 warehouse size	100 connections	SQL Transactions
Cassandra	Cassandra stress	10 million records	15 minutes	R/W: 70% /30%, fixed(1024) col.
MongoDB	YCSB	220 million records	50 million operations	100% update, 24/1000-KV
ForestDB	DB_bench	200 million records	10 hours	100% update, JSON objects
RocksDB	DB_bench	560 million records	250 million operations	100% update, 24/1000-KV

3.3 Pattern Warehouse: Construct

Pattern warehouse is a collection of I/O patterns used to construct different I/O workloads. Our framework is expandable because we can add new patterns into the pattern warehouse once we obtain the knowledge of other applications and workloads. Our pattern warehouse currently includes 15 different workload patterns. It provides some recommended pattern combinations to resemble real applications, like MySQL, Cassandra, and MongoDB.

Multiple I/O generating jobs constructs each I/O pattern. A real application often exhibits variance in I/O activities across storage space over time. We observe that real workload I/O patterns can be grouped into different categories, such as horizontal stripe wise, periodic, phase-wise, and abrupt. To capture this variance, we develop different I/O jobs. A job is responsible for rendering I/Os for part of the I/O workload pattern to represent some specific I/O layout. Each job is composed of a set of I/O generating features of the FIO engine. Thus, integrating

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3048303, IEEE Transactions on Computers

Table 3: Building I/O Patterns: The input features for jobs of different I/O patterns

I/O Pattern	**Feature_Set**{}
Random I/O with Density Variance (RIDV)	`--rate_iops, --offset, --bsrange, --thinktime, --thinktime_blocks`
Sequential Writes with Multiple Jobs of Different Offset (SWMJDO)	`--rate_iops, --size, --numjobs, --offset, --blocksize_range, --offset_increment`
Bars of R/W (BRW)	`--rate_iops, --numjobs, --offset, --runtime, --size`
Bamboo Sticks Different Slopes (BSDS)	`--rate_iops, --offset, --startdelay, --rw_sequencer`
Fountain Scatter Horizontal (FSH)	`--rw_sequencer, --rate_iops, --numjobs, --offset_increment, --blocksize_range`
Bamboo Sticks Horizontal Density Variance (BSHDV)	`--rate_iops,--bsrange, --startdelay, --rw_sequencer`
Horizontal Overwrites (HO)	`--rw_sequencer, --startdelay, --rate_iops, --random_distribution=zipf`
Raindrops	`--thinktime, --thinktime_blocks, --rw_sequencer, --rate_iops, --numjobs, --offset, --runtime, --size`
Sprinkler	`--rw_sequencer, --rate_iops, --numjobs, --offset, --offset_increment, --blocksize_range`
Bamboo Sticks Vertical Density Variance (BSVDV)	`--rate_iops,--offset,--bsrange, --startdelay, --rw_sequencer, --size`
Backward Steps (BS)	`--rw_sequencer, --rate_iops, --numjobs, --offset, --runtime, --wait_for_previous, --offset_increment, --blocksize_range`
Angular Chopping (AC)	`--rw_sequencer, --rate_iops, --numjobs, --offset, --runtime, --size`
Vertical Chopping (VC)	`--thinktime, --thinktime_blocks, --rw_sequencer, --rate_iops, --offset, --runtime, --size`
Bamboo Different Alignment (BDA)	`--thinktime, --thinktime_blocks, --rw_sequencer, --rate_iops, --numjobs, --runtime, --size`
Horizontal Shower (HS)	`--rw_sequencer, --rate_iops, --numjobs, --offset`

these I/O jobs together can help to capture the diversity of a real I/O workload pattern.

We *construct* 15 different I/O patterns as listed in Table 3. For example, patterns *RIDV* and *SWMJDO* are of horizontal strip-wise fashion. *BRW* is a horizontal stripe wise with alternative read and write intensive phases. *BSDS*, *Sprinkler*, and *FSH* provide periodic I/O patterns. *BSHDV* is a phase-wise I/O pattern. *HO* and *Raindrops* both fall under the abrupt category. Some real I/O workloads were observed to have I/O patterns that are a combination of different categories such as RocksDB 3 as discussed in Section 1. In order to replicate such patterns, we further construct five I/O patterns, namely *BSVDV*, *BS*, *AC*, *VC* and *BRW*, that represent the combination of horizontal stripe wise and periodic I/O fashion. The I/O pattern *BDA* is a combination of periodic and phase-wise types, and *HS* is a combination of horizontal stripe wise and phase-wise categories. Thus, we ensure that pattern warehouse consists of all distinct patterns that we majorly observe in real I/O workloads. We can always add new patterns to our pattern warehouse when required.

One of the challenging problems is identifying features that could be used to construct a particular I/O pattern. We solve this problem by studying and analyzing combinations of different features and then setting appropriate values of features for each I/O pattern. Table 3 lists the obtained combinations of features we also use some other common options, such as `--random_number_generator, --initial_seed, --iodepth, --ioengine, --rw, --device, --if-else, --for_loops, --while_loops, --kill_job` to manage runtime operations of I/O jobs. We name I/O patterns according to their visual appearance like *sprinkler*, *raindrops*, *backward steps*.

Next, we explain some representative I/O characteristics that we identify and use to emulate different I/O patterns.

I/O Holes: We observe that many applications do not perform I/Os during some time intervals or within some disk offset ranges. For example, Figure 3 (b), shows the Cassandra workload that has a horizontal blank space band, where no I/Os are performed to a particular disk offset range. We call such a blank space as *I/O hole*. There could be two types of I/O holes, temporal I/O holes and disk offset I/O holes. A temporal I/O hole may be caused by a system stall for waiting for other resources like CPU, I/O bus, or may happen when the upper layers in I/O stack such as cache or memory are sufficient to serve the desired request. On the other side, a disk offset I/O hole may be caused by wear leveling activities or disk space allocation through application transactions. Modeling such I/O holes is critical to performance because during these I/O holes, overall I/O throughput may fluctuate. Furthermore, a benchmarking tool for flash-based SSDs, that can replicate such I/O holes can better estimate endurance. Thus, we capture I/O holes of different shapes and sizes by setting options like `thinktime, thinkblocks, startdelay, offset increment` for each job.

Byte density: The measure of how many bytes are stored within a particular range of storage addresses is called Byte density. We observe that in many real applications, different disk spaces are accessed with different byte density. For example, when a MySQL database application stores its metadata in some disk space, it might be accessed more frequently than the other disk space. Moreover, we observe that byte density may also vary across different workload execution time. For example, in MongoDB, depending on the keys affected by the "update" operation, it may result in modifying a different number of indexes in the collection. Thus, the number of I/O activities can be sparse or dense depending upon the number of indexes modified during the workload's execution. It is vital to capture byte density because the variation in byte density is the primary source of I/O latency variations and latency tails. Thus, we use various I/O distributions, such as `Zipfian, pareto,`

`uniform` to capture byte density in each I/O job.

I/O Jumps: A pure sequential I/O should span continuously over consecutive disk addresses. However, we observe that real workloads sometimes leave empty disk addresses between small sequential writes, e.g., skipping 16KB after writing every 128KB sequentially. We then say that the I/O patterns of these applications exhibit periodic *I/O jumps* (i.e., addresses left unwritten) while performing sequential reads or writes. Such an I/O jump can allow sequential I/Os to span over a wide range of disk offsets in a short period. I/O jumps result in inclined vertical lines across the disk, as observed in MongoDB, see Figure 3. We use options like `sequencer` and its offset to generate a sequential I/O sequence with I/O jumps.

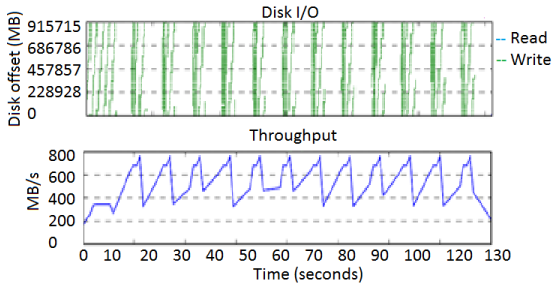


Figure 4: I/O pattern layout generated by `PatIO` for Bamboo Different Allignment (BDA) pattern

Pattern Feature Setting: As mentioned above, Table 3 lists all 15 I/O patterns with corresponding list of features for each. Due to the limited space, we cannot explain the logical derivation for deciding the feature_set of all patterns in Table 3. Here, we use the pattern, called Bamboo Different Alignment (BDA), as a representative to explain our logical process of feature_set derivation. This pattern is inspired by dissecting the MongoDB application when running different YCSB workloads. Figure 3 (c), shows the real I/O layout of one of the workloads. We observe a repetitive pattern with a stretch of partially sequential writes over the whole disk space. We say these I/Os as "partially sequential" because they show an I/O jump after every block of writes to range over the whole disk space in a short period. Apart from that, we consider to use the features `-thinktime` and `-thinktime_blocks` to control periodicity and I/O activities happening in each period. By setting different values for `-rwsequencer` and `-rate_iops`, different slopes of alignment can be achieved. We have preset default values for each of these features within each I/O pattern in the workloads that we design. These values are automatically varied according to the necessary concatenation of multiple I/O patterns, length of the desired workload, and the SSD size.

Figure 4 shows the resultant BDA pattern for 120 seconds on 960GB SSD. As seen from the top plot of the figure, the I/O layout consists of periodic I/Os, where each period has a dense region at the beginning followed by the sparser region. Thus, this I/O pattern is constructed by using two jobs. The corresponding features of each job allow it to generate periodic I/Os with different rates to generate denser and sparser regions. Given these two jobs with different I/O rates, we can observe that the throughput (see the bottom plot in Figure 4) of this generated pattern exhibits variance over time. Such throughput variations well match the throughput variations in real applications.

3.4 Pattern Generator: Integrate

The pattern generator is the central module of `PatIO`, which is responsible for communicating with the interactive GUI input, pattern warehouse, and I/O execution modules. This module *integrate* different combinations of synthetically generated I/O patterns to reproduce the comprehensive characteristics of various real workloads and system setup for the database server applications. Specifically, the pattern generator gets the user input from the interactive GUI input module. It then fetches the corresponding I/O pattern files from the pattern warehouse. These I/O jobs are then adapted according to the user-specified storage disk size and execution time. Among all the features of the jobs, we first identify a subset of features that could be affected by the change in disk size. Then, the features in this subset are modified by a linear scaling, as shown in Equation 1. For example, *I/O range* which is set to 400-500GB for 500GB drive is changed to 800-1000GB for 1TB storage disk size.

$$New\_Feature_i = default\_Feature_i . \frac{New\_Size}{default\_Size} \quad for\ \forall\ Jobs\ i \tag{1}$$

Similarly, the execution time of each job for all the patterns needs to be changed according to the desired execution time given by the user. Finally, we execute jobs of all selected I/O patterns in parallel.

Online Conflict Management: For parallel execution, some jobs may have conflicts with others. As discussed before, a pattern is executed as a multi-threaded system process. When multiple patterns are required to execute simultaneously, programming commands that are given to the I/O generating engine by one process's threads may affect threads of other processes. Thus, before executing all the jobs of selected patterns, the pattern generator performs careful workload concatenation of all the job files. It is essential to identify and handle these conflicts. Our exception handling module identifies conflicts by maintaining a hash table of features and I/O jobs. If there is a conflict, those jobs are modified according to the type of conflict. The modified I/O jobs are then concatenated again until there are no further conflicts. Here, we use two common types of conflicts as examples to show corresponding modifications performed to resolve them.

Contradiction: Jobs of different patterns might set different values of the same feature. We call this type of conflict as *contradiction*. For example, one job might request I/O engine to set I/O size feature to 4K for a particular disk offset. However, at the same time, another job of a different pattern might want to set I/O size of 64KB for the same disk offset. For FIO, we notice that both of these I/O jobs may stall for a long time or be dropped off when such a contradiction occurs. We resolve this contradiction by introducing some time delay between the operation of these two jobs. As a result, in the above example, we allow the first job to perform 4KB I/Os and later let the second job run its 64KB I/Os on the same disk offset.

Interference: Some actions taken by a job of one pattern might unintentionally influence jobs of the other patterns. For example, a `killall` command in a job of pattern_x might also kill jobs of all the other concatenated patterns.

Thus, we need to maintain a list of such features and identify if any jobs are using these features. If yes, then we need to modify these jobs to ensure such an interfering command only affects the jobs of the desired pattern. That is, we would identify the thread ID of the jobs of each pattern and kill only the threads of the concerned pattern rather than using the default *kill all* command. The types of conflicts and their resolutions may vary with different I/O engine. However, it is crucial to observe such behavior as it vastly impacts I/O layout on disk.

Parallel Executor: After resolving all conflicts (i.e., no more conflicts in the concatenated workload set), we execute the generated synthetic I/O workload and measure the performance of I/O activities over the storage space. All the workloads generated by our framework are capable of generating logs during the runtime and record the performance in terms of I/O bandwidth, IOPS, throughput, and latency.

3.5 GUI Interface and Process of Using

In order to provide an easy-to-use experience, we develop a GUI interface for PatIO. We mainly have two use scenarios - 1) if the user wants to generate the I/O activities for one of our pre-defined applications. As of now, we provide a direct option to generate I/Os resembling the five most popular database applications such as MySQL, Cassandra, MongoDB, ForestDB, and RocksDB. The user can use corresponding checkboxes to select one or multiple of these real applications in our GUI. 2) if the user wants to generate the I/O activities for some other applications. Then, we assume that the user should have some idea from their experience or plots of previously collected traces that what type of I/Os are they looking to generate. Depending upon their requirements, they can select one or multiple I/O layout patterns in our GUI, e.g., horizontal overwrites, bars of read/write, and backward steps. We have 15 different patterns with a visual snapshot of the disk layout for each to choose from in our GUI. Then the user defines the desired runtime (i.e., execution time) of an I/O workload and the size of the storage space exposed to I/Os. Additionally, the GUI also allows selecting a different level of background noise, which may be incurred by various background I/O activities of the SSDs such as garbage collection, wear-leveling, etc. Finally, the user clicks run, and PatIO accordingly generates I/Os, instruments the performance, collects traces, and plots various generally used graphs such as average throughput over time, the cumulative distribution of the tail latencies, and instantaneous write amplification factor of SSD. Thus, besides taking the input options of the desired workload, the GUI is also responsible for linking an option in the widget with its corresponding *PatternID* and send this option to the back-end pattern generator module.

4 Evaluation

In this section, we evaluate PatIO by comparing I/O characteristics and performance of generated workloads with real-world workloads of different database applications such as MySQL, Cassandra, and ForestDB. Table 4 gives the detailed hardware configuration of our platform on which we develop PatIO and run real application workloads. PatIO is built using python. It uses inbuilt advance libraries of python like matplotlib, NumPy, and Tkinter. Each pattern in pattern warehouse contains a bash program that can be used to construct I/O jobs. We use the FIO engine to generate I/Os. Figure 5 shows the I/O stack of PatIO in comparison to that of real application. As shown in Figure 5, PatIO generates an I/O pattern that can capture real I/O activities (see the left part in the figure) of different user workloads (e.g., YCSB) running in docker containers on the client-side for various database server applications (e.g., MongoDB) running in the datacenter on the server-side that can use different file systems (e.g., ext4) and volume managers (e.g., LVM). More importantly, the operations of different database server applications using various cloud setup of user workloads and system settings at the file system and volume manager layers are abstracted by PatIO. Thus, PatIO requires less time and resources for benchmarking.

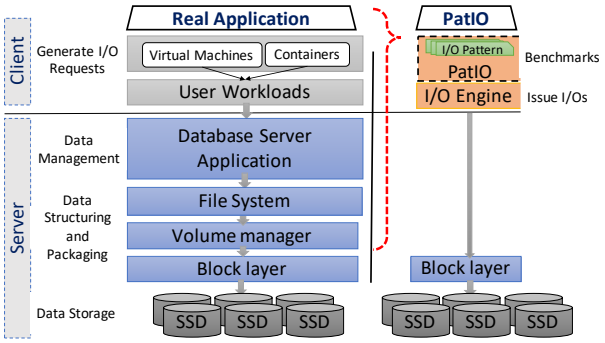


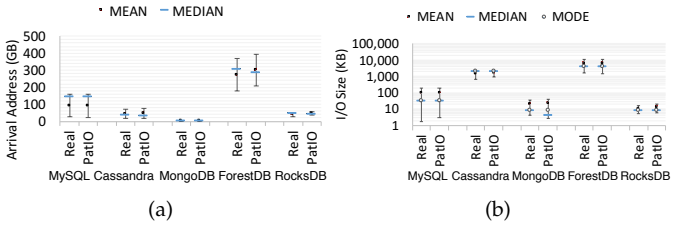
Figure 5: I/O stack of PatIO in comparison to that of real application

Table 4: Hardware configuration

CPU Type	Intel(R) Xeon(R) CPU E5-2640 v3
CPU Speed	2.60 GHz
CPU #Cores	32 hyper-threaded
CPU Cache Size	20480 KB
Main Memory	128 GB
OS	Ubuntu 16.04 LTS
OS Kernel Version	4.4.0-13generic
File System	ext4
Storage	No-stream and Multi-stream NVMe SSD 960GB and 480GB
Docker Version	1.11
VMware Workstation	12.5.0
FIO Version	2.2

We study the I/O activities of different user workloads and applications (e.g., YCSB, Cassandra-stress, and DB_bench) running on parallel virtual machines and containerized infrastructures with different database servers (e.g., MySQL, Cassandra, ForestDB, MongoDB, RocksDB) operating in the data center. As shown in Table 2, each application workload can have its configuration of the number of transactions, compaction rate, and read-write ratio. We also study different combinations of applications operating directly on the local machine and in containerized docker environments, e.g., MySQL+Cassandra with a different number of containers for each application.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3048303, IEEE Transactions on Computers



(a) (b)

Figure 6: (a) Mean, Median, and Standard Deviation of I/O arrivals on disk space over time for real and PatIO workloads, (Note: Mode for Arrival Address is not plotted as I/Os to same block does not necessarily imply I/Os to the same address.) and (b) Mean, Median, Mode, and Standard Deviation of I/O Size for real and PatIO workloads. (Note: y-axis is logarithmic scale, so Standard Deviation does not look to be equally distributed on either side of mean.)

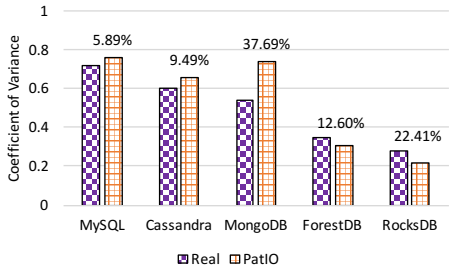


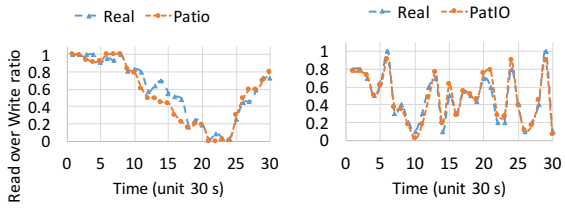
Figure 7: Coefficient of variance of I/O arrivals on disk space over time for real and PatIO workloads with the error (%) mentioned above the bars

4.1 Characteristics Comparison

First, we compare the characteristics of a real workload and a synthetic workload generated by PatIO by measuring their statistical central-tendency like Mean, Median, and Mode. We also compare the spread of data from the central tendency, such as standard deviation and coefficient of variance.

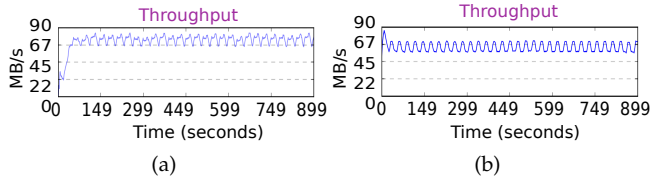
Workload Characteristics: Different characteristics are observed from a workload such as I/O layout on a storage disk, I/O size, and read-write ratio. We perform experiments with 1000+ workloads of different applications. As a representative, we here present results for some of them. The configurations of real applications are given in Table 2. Figure 6(a) compares I/O arrivals on disk space over time for real and PatIO workloads. We see that the statistical results of central tendency (like Mean and Median) for real and PatIO workloads are very similar. Here, we use unit positive and negative standard deviation to measure the spread of data from the central tendency and confidence in statistical conclusions. We also observe that the real and PatIO workloads of all the applications show similar standard deviations. Figure 7 compares the coefficient of variance of I/O arrivals on disk space over time for real and PatIO workloads. We see mostly error between real and PatIO workloads remains small.

I/O Size: I/O size is another important characteristic that affects performance. Because the sizes of I/Os vary over time for a real workload, just reproducing I/Os with the same size (e.g., average I/O size) is not sufficient. Thus, we argue that it is critically important to emulate the variance of I/O size over execution time. Figure 6(b) shows the comparison of the statistics of I/O size over time for real and



(a) (b)

Figure 8: Read to write ratio over runtime of workloads a) MySQL and b) Cassandra



(a) (b)

Figure 9: Throughput variation over time for, a) Real Cassandra workload, and b) Generated Cassandra workload

PatIO workloads. Besides Mean and Median, we further use Mode to represent the size of the majority of I/Os. We can see that the modes of real and PatIO workloads also match well in Figure 6(b). We further observe that PatIO can reconstruct the variance of a real workload as seen from the standard deviation and coefficient of variance. While comparing all different metrics of measurement, the maximum error percentage is less than 25%, which indicates a good resemblance between real workloads and PatIO.

Read-Write Temperature: Besides the above statistics comparisons, we also compare the characteristics over the runtime of real and PatIO workloads. Figure 8 shows the read to write ratio over runtime as a representative by plotting the moving averages taken over every 30 seconds until 15 minutes. Here, we show the results of MySQL and Cassandra. We see that the generated workload can reproduce the actual read/write temperatures.

4.2 Performance Comparison

Throughput: We further compare the throughput (i.e., the number of I/Os performed on disk per second) of the aggregate generated workload using PatIO over variants of 1000 different workloads of Cassandra with the throughput of a sample chosen randomly while running real application with these 1000 different workloads. The goal of our PatIO generated workload is to capture the high-level I/O feature that persists among multiple workloads of an application. The throughput results in Figure 9 shows that the real workload does not have a constant throughput over the execution. Our synthetic workload shows the same wavy nature of the throughput (see Figure 9), as opposed to the constant throughput that is resulted by Naive FIO. We notice that the throughputs in the first few seconds are different because the real workload needs to spend some time to initial cache construction while the generated workload assumes that data required by the application is readily available. Also, for a real application, there are more intermediate layers in I/O stack when compared to PatIO, as seen from Figure 5, which increases the initial latency and decreases the initial throughput. Once the cache is constructed, this

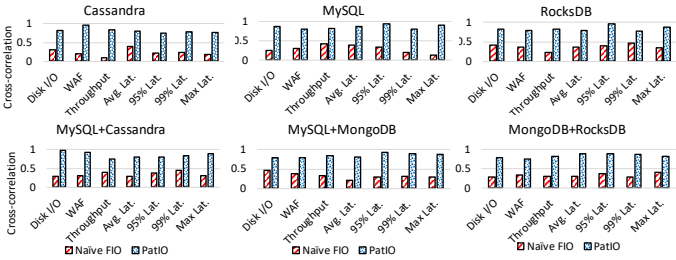


Figure 10: Comparing the cross-correlation between real and synthetic workloads using a lag of number of samples used for training for traditional FIO that uses average statistics (Naive FIO), and PatIO

Table 5: Comparing the operational storage space (MB) consumed and execution time (minutes) for Replay [30], and PatIO

	Cassandra		MySQL		RocksDB	
	(MB)	(min)	(MB)	(min)	(MB)	(min)
Replay	64512	420	48128	360	90112	600
PatIO	13.87	9.04	10.42	6.38	18.11	13.51

latency due to intermediate layers is hidden in parallel tasks. However, the initial performance during this short period is often neglected because it is well known that all storage disks require some initial time for ramp-up. We also observe that the real Cassandra result has varied across time, and PatIO can exhibit similar performance variations.

4.3 Overall Validation

We consider the cross correlation[2] to compare the performance of running workloads generated by PatIO with respect to that of running real workloads. We vary the total number of operations (i.e., transactions) performed to generate 50 different workloads of each application. Figure 10 shows the average of cross-correlations between different real and the corresponding generated 50 workloads while running individual applications (e.g., Cassandra and MySQL) and mixed applications (e.g., MySQL+MongoDB). As a baseline, we also plot the cross-correlation of the workloads generated using naive FIO generator. The naive FIO uses the Mean of different workload characteristics.

Each plot in Figure 10 shows the correlation of the Disk I/O distribution, the endurance SSD measured using Write Amplification Factor (WAF), and the performance measured using throughput, average latency, and tail latency. To measure WAF during runtime, periodically, we instrument physical NAND writes of SSD by using the S.M.A.R.T (Self-Monitoring, Analysis, and Reporting Technology) tool commands such as "nvme smart-log" and "smartctl" [34], and instrument the logical writes to each physical SSD from block layer. We do not propose any SSD firmware amendments in this work. All the other performance matrices are measured at the application layer. We observe that the synthetic workloads generated by our PatIO are highly correlated with real ones, with the cross-correlation large than 0.8 for all the workloads. The naive FIO has a comparatively lower correlation than PatIO because workloads generated by the naive FIO fail to capture the intrinsic diversities and variations of real applications.

2. Cross correlation is a measure of similarity of two series as a function of the displacement of one relative to the other.

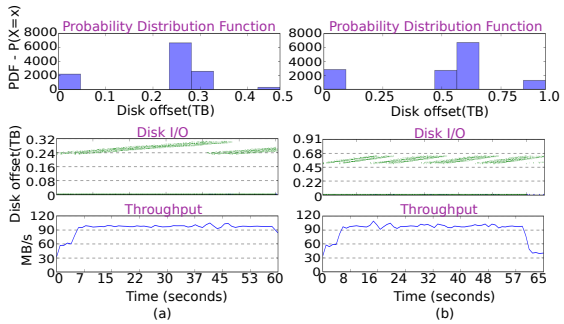


Figure 11: Effect of drive size: a) 480 GB SSD, b) 960 GB SSD

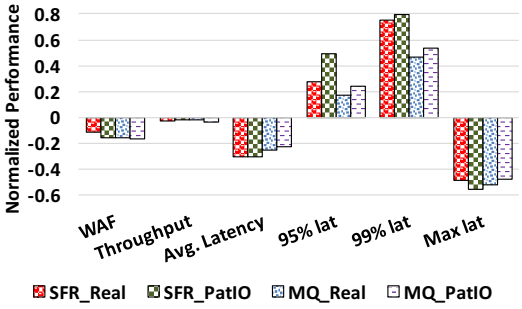


Figure 12: Comparing the application performance while using real and generated workloads running Cassandra application

Table 5 shows that when compared to traditional I/O replay tools [30], PatIO consumes much smaller amount of operational storage space and execution time. This is because PatIO is designed to emulate and regenerate the characteristics of different real workloads rather than storing time logs of I/O activities. Also, the architecture of PatIO bypasses many intermediate I/O stack layers as shown in Figure 5, which allows it to execute much faster.

4.4 Scalability of PatIO

One of our contributions is that PatIO is scalable to generate I/Os over different sizes of storage disks. Thus, we use the Horizontal Shower (HS) I/O pattern (see Table 3) as an example to investigate PatIO's scalability. Figure 11 shows I/O characteristics (e.g., I/O distribution and I/O starting disk offset) and performance (e.g., throughput) when running the generated HS workload in SSDs with different capacities, i.e., (a) 480GB and (b) 960GB. First, we observe that the I/O workloads generated by PatIO scale with the SSD capacity in terms of I/O layout, see the first two rows in Figure 11. We also observe that the throughput remains the same under two different capacities as expected, because the workload does not saturate the I/O bandwidth on both SSDs. Summarizing, workloads generated by PatIO behave similarly as real ones in terms of both I/O characteristics and performance for back-end storage of different sizes.

5 Practical Applications

Multi-stream SSD [2], [3] is the recent development of the SSD industry, which allows us to have multiple append points (erase blocks) at the same time while writing to an SSD. This advancement assists in placing data on SSDs in a smart way such that we have less garbage collection and hence less write amplification factor. Multi-stream

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3048303, IEEE Transactions on Computers

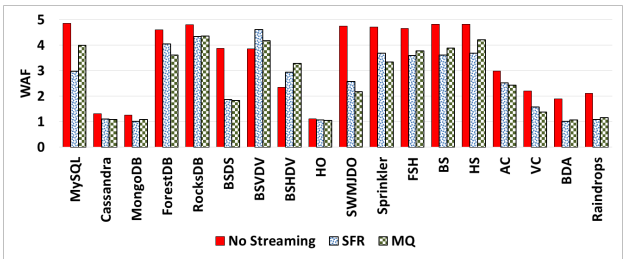


Figure 13: Comparing WAF of legacy drive with a multi-stream drive using SFR and MQ stream assignment algorithms

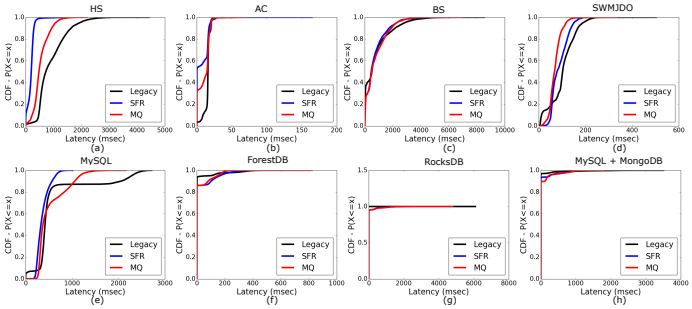


Figure 14: Latency Cumulative Distribution Function (CDF) of different workloads using a legacy SSD and multi-stream SSD with stream assignment by using SFR and MQ algorithms

functionalities are added to Linux mainline kernel in 2017 with the corresponding NVMe drivers available online[3]. To take advantage of the features offered by multi-stream SSDs, it is challenging to identify the stream IDs. For each write/update, a stream ID needs to be assigned such that each stream consists of data with a similar lifetime of being valid. [6] proposed two automatic stream management algorithms (auto-stream), named *SFR* and *MQ*, to assign stream IDs. The SFR algorithm utilizes three attributes (i.e., sequentiality, frequency, and recency) for stream detection. It maintains the rank of data blocks in a table and allocates stream IDs according to the rank. The MQ (Multi-Queue) algorithm utilizes access frequency and recency to maintain multiple queues and uses each queue to represent the rank of data. The calculated rank is used to assign stream-IDs to data. In [6], it is observed that the benefit of these stream management algorithms varies over different workloads. However, it is difficult to study what type of applications may be benefited by which stream assignment algorithm, because there are many varieties of applications and different possible combinations of the variable parameter within each stream assignment algorithm. Moreover, quantifying the aggregated performance benefit over the wide range of the performance parameters, such as high throughput, low latency, high SSD endurance [4], and low write amplification factor may consume a large amount of resources and require a lot of time and efforts. As `PatIO` can mimic the I/O behavior of the real application with negligible spatial and temporal overhead (as discussed in Section 4). Thus, we next discuss how `PatIO` can be integrated with the multi-stream

3. https://elixir.bootlin.com/linux/latest/source/drivers/nvme/host/core.c

4. SSD endurance is the total amount of data that an SSD is guaranteed to be able to write under warranty, and high SSD endurance indicates high device lifetime.

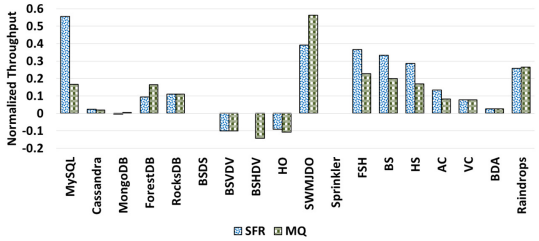


Figure 15: Normalized throughput of multi-stream drive using SFR and MQ stream assignment algorithms with respect to legacy

SSDs to help study the performance variation of each stream identification algorithm with respect to various applications.

5.1 Integrating `PatIO` with Multi-stream SSDs

`PatIO` can be used to study the impact of any particular software, firmware, or hardware improvements over multiple facets such as throughput, latency, and write amplification factor. Particularly, first, we install the new firmware and hardware of multi-stream SSDs in our server. Next, we ensure that the block layer of the Linux OS is able to read and write to the multi-stream SSDs through the NVMe device driver. Finally, we run `PatIO` on the client-side to issue I/Os directly to the block layer of the server, as shown in Figure 5. We use multi-stream NVMe SSD and no-stream legacy NVMe SSD of the same capacity of 480 GB. We measure application performance in terms of average throughput, average latency, and different tail latencies. We also measure SSD endurance in terms of the average of write amplification factor (WAF) calculated using the ratio of the total physical NAND writes to the logical application writes within 5-minute intervals.

5.2 Performance of Auto-Stream

We first compare the performance of the workload generated by `PatIO` to the performance of the real workload. Figure 12 plots the normalized performance results (i.e., (multi-stream - legacy)/legacy) of a multi-stream SSD using SFR and MQ under both real and `PatIO` generated workload of the Cassandra application. The results using legacy SSD without streaming are considered as the baseline. We use the workload configuration of Cassandra as mentioned in Table 2 for the legacy NVMe SSD (no streaming) and multi-stream NVMe SSD to obtain the "real" performance results. The positive bars in Figure 12 reflect that using a multi-stream SSD, the measured performance is higher than that of a legacy SSD and vice-versa for negative bars. Our analysis across different performance metrics helps us examine if the generated workload by `PatIO` shows the same traits as a real workload. We observe that `PatIO` is able to capture the performance trend of either improvement or deterioration exactly. That is, while using a real application, if using multi-stream SSD with SFR or MQ algorithm resulted in performance improvement, then similar performance improvement is also seen with the workload generated by `PatIO`. Figure 12 shows that our generated workload closely mimics all performance metrics. For example, WAF decreases with both the auto-stream algorithms of multi-stream SSDs compared to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3048303, IEEE Transactions on Computers

no-streaming legacy SSDs for the real Casandra workload. We can see the same impact from the PatIO generated Casandra workload. Thus, PatIO *is useful to analyze the impacts of evolving storage technologies such as auto-stream*.

We next use PatIO, for running individual I/O patterns listed in the Table 3 to study in-depth performance benefit of multi-stream SSDs with SFR and MQ algorithms over legacy SSDs (no streaming) for each I/O pattern. We also combine different I/O patterns to emulate various real applications and study the performance impact. Figure 13 shows write amplification factor (WAF) for 5 different workloads, such as MySQL, Cassandra, MongoDB, ForestDB and RocksDB, and 13 other I/O patterns (in Table 3). We observe that multi-stream SSD with both the SFR and MQ algorithms can reduce WAF for most of the workloads compared to legacy. Lower WAF means less internal writes during garbage collection, which leads to better SSD endurance. Thus, we say that multi-stream technology may improve SSDs' lifetime. There are two exceptional I/O patterns, such as BSVDV and BSHDV, under which using multi-stream SSD incurs an increase in WAF. More importantly, our PatIO helps to identify such exceptions. This also shows that PatIO *has great potential to help the specialist in the design of multi-stream SSDs and auto-stream algorithms to improve their algorithm, firmware, or hardware* in order to handle such exceptions.

Figure 14 further indicates that apart from increasing the lifetime of SSD, multi-stream technology may also help reduce the latency of workloads. We see that different I/O patterns have different impacts of streaming on their latency. Both SFR and MQ achieve lower latency for some patterns, like HS and SWMJDO. While for some others like AC and MySQL, SFR does a better job than MQ. In addition, SFR or MQ might be good for some individual workloads, but not be that beneficial when we have multiple simultaneous workloads, like MySQL+MongoDB.

Finally, we analyze application throughput while using legacy (no-streaming) SSD and multi-stream SSD. Figure 15 shows the normalized throughput of SFR and MQ with the throughput of legacy as a baseline. It depicts that among different generated application workloads and I/O patterns, most of them have better throughput while operating on multi-stream SSD than legacy SSD. However, some particular I/O patterns (like BSVDV, BSHDV, and HO) need special attention. The workloads with these patterns may not show a good response to multi-stream technology. They may need some modifications to improve their underlying stream assignment. Note that for I/O patterns BSDS, and Sprinkler the throughput using multi-stream and legacy SSDs is the same, thus the normalized throughput is shown in Figure 15 is null.

5.3 Benefits of Performance Engineering with PatIO

To sum up, PatIO can explore a wide range of different workloads in a short time compared to the time required in real application installation, configuration, and running. PatIO can assist us in comparing different stream assignment algorithms using I/O patterns to evaluate the impacts of any algorithm, firmware, or hardware modifications on various performance metrics. Additionally, PatIO can identify intricate details, e.g., which I/O pattern in a workload which is responsible for performance degradation.

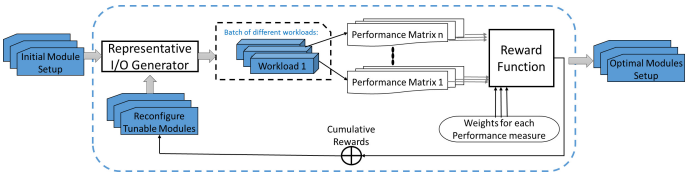


Figure 16: Auto-tuning the variables of stream assignment algorithm

6 Auto-tuning Module

Our evaluation found that the benefits of multi-stream SSDs over legacy varied with the SSD version and the capacity of multi-stream SSDs. Indeed, [6] presents that both two-stream assignment algorithms (i.e., SFR and MQ) have a set of tunable parameters, such as $chunk\_size\_sector$, $decay\_sec$ and $freq\_aging\_sec$ for SFR and $chunk\_size\_sector$ and $adjust\_ref\_cnt$ for MQ. We believe that to achieve optimal performance, these variable parameters need to be properly tuned for a change in the physical firmware like the size or the model of an SSD drive. Furthermore, while identifying the best value for these parameters, different I/O applications should be considered to avoid over-fitting to any particular workload. Additionally, different users may have requirements on different performance parameters, e.g., latency, throughput, or write amplification. Thus, it is also important to take these user requirements into account when tuning variable parameters. Let the parameters for any particular algorithm (e.g., SFR) be α_k, β_k, γ_k, etc. Examples of performance parameters that we desire to measure and improve are 50th, 90th, 99th, and 99.99th percentile of latency (Δl), WAF (ΔW) and throughput (Δt). Weight factor ω_i is further used to differentiate the importance of each performance parameter. Eq. 2 represents the weighted sum of different performance parameters.

$$Wj = \frac{-\omega_1 \Delta l_{50} - \omega_2 \Delta l_{90} - \omega_3 \Delta l_{99} - \omega_4 \Delta l_{99.99} - \omega_5 \Delta W + \omega_6 \Delta t}{\omega_1 + \omega_2 + \omega_3 + \omega_4 + \omega_5 + \omega_6} \tag{2}$$

where, $\forall i \in [0, +\infty) \quad and \quad \omega_i \in [0, +\infty)$.

Thus, Eq. 3 shows that the resultant performance parameters are the function of the internal algorithm parameters of multi-stream SSDs.

$$f(\alpha_k, \beta_k, \gamma_k, ...) = Wj \tag{3}$$

where $\forall\ \alpha_k \in [0,\ Max_\alpha]$, $\beta_k \in [0,\ Max_\beta]$, and $\gamma_k \in [0,\ Max_\gamma]$.

Finally, we want to maximize the weighted "Reward Function" for n workload as shown in Eq. 4.

$$W_{avg} = \frac{1}{n}\sum_{j=1}^{n} Wj \tag{4}$$

Objective:

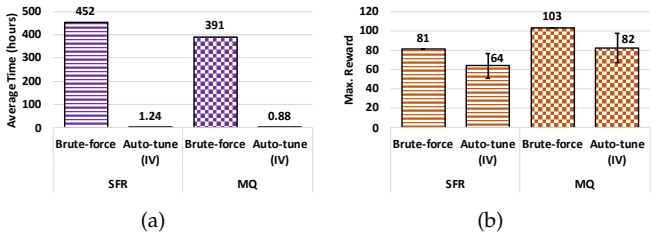
Find the best α_k, β_k, γ_k, etc. such that,

$$g = Max\left\{\frac{1}{n}\sum_{j=1}^{n} Wj = f(\alpha_k, \beta_k, \gamma_k, ...)\right\} \tag{5}$$

6.1 Techniques for Solving Optimization Problem

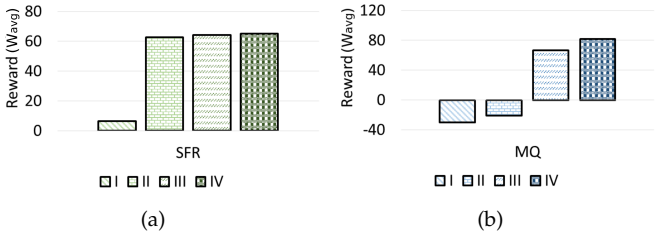
The simplest approach towards solving this convex optimization problem is the brute force evaluation of the objective function by generating all possible α_k, β_k, γ_k, etc. This method gives us the best possible internal parameters, which ensures the best performance and endurance of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3048303, IEEE Transactions on Computers



(a) (b)

Figure 17: Comparing Brute-force and auto-tune converged points (a) Average Time, and (b) Maximum Reward



(a) (b)

Figure 18: Average Rewards while auto-tuning variable parameters of (a) SFR, and (b) MQ

storage device. However, due to the high complexity of the problem, such a brute force approach is time-consuming. Thus, we solve our optimization function by using the multi-armed bandit model of reinforcement learning to auto-tune parameter values that can result in a local maxima[5]. Our *agent* is an internal algorithm (e.g., SFR) of a multi-stream SSD, which needs to take *actions* towards the best performance by either increasing or decreasing the value of each variable parameter. For example, SFR has *three-arms*, as it contains three variables. The *reward* of each action is calculated using the reward function g (see Eq. 5). Our *agent* aims to maximize the reward until it encounters the local maxima.

We build the required peripheral framework shown in Figure 16 with `PatIO` to enable auto-tuning. This framework is constructed to explore the best values of tunable parameters for an auto-stream algorithm, such as SFR and MQ. This framework particularly uses the same SSD in the legacy mode (i.e., without stream) and then runs an auto-stream algorithm in the multi-stream mode to measure the performance changes between legacy and multi-stream. The goal is to identify the values for variable parameters of that auto-stream algorithm in order to obtain the best possible performance using multi-stream SSDs. Our framework uses `PatIO` to generate $\binom{15}{1} + \binom{15}{2} + ... + \binom{15}{15} = 2^{15}$ different workloads with 15 I/O patterns shown in Table 3 to ensure that the parameter tuning is not biased or over-tunned towards any particular workload. The initial value and the minimum and maximum of each variable parameter are defined in the "Initial Module Setup". Then, as shown in Figure 16, `PatIO` reads the constrains and initial values from the initial setup file and generates a set of different workloads. These workloads are run one by one to get their own performance metrics that consist of various performance parameters like throughput, average latency, tail latency, WAF.

6.2 Results

To evaluate the effectiveness of our parameter tuning module, we first compare the time consumed in reaching the convergence point and the maximum reward after convergence while using the brute-force method and our auto-tune module. We run the experiments five times and take the average to present the results. Figure 17(a) shows that the time for auto-tuning to reach local maxima is much shorter than the time required by brute-force to reach global maxima. From Figure 17(b), we see that difference in the achieved maximum reward with brute-force and auto-tuning is not significant. Thus, using our auto-tune module along with PatIO, we can quickly improve the performance and endurance of multi-stream SSDs.

We further analyze the variations of the rewards over runtime when using our auto-tune module. Figure 18 shows the average reward value (W_{avg}) of the initial point (I), two intermediate points (II, III), and the final converged point (IV), for SFR and MQ. The variable parameter values for each of these points are further listed in Table 6. We notice the overall performance reward (W_{avg}) increases 60% for SFR and 110% for MQ from the initial point (I) to the final converged point (IV) in Figure 18.

We note that besides PatIO, our auto-tune module is also complementary to other synthetic workload generators such as FIO and trace-based I/O generators such as block trace replay. However, we observe that the internal parameters selected using simple synthetic workloads are not good enough to obtain better performance when running a real application on these multi-stream SSDs. On the other hand, the storage space and time required to use trace-based I/O generators to configure the internal parameters of multi-stream SSDs are very high and not always feasible. Thus, compared with those existing synthetic workload generators and trace-based I/O generators, our `PatIO` provides a more realistic and highly usable solution. Using our auto-tune module with `PatIO`, our performance engineering team was able to significantly reduce the configuration time of multi-stream SSDs from a couple of months to a couple of hours. Our novel approaches of (a) benchmarking with the combined activities of multiple workloads of any application and (b) tuning the internal variables of the modern SSDs to further improve the performance are very important to the storage community with new emerging storage devices.

Table 6: Intermediate points while auto-tuning

Label	SFR	MQ
I	chunk_size=4096, decay_sec=1200, freq_aging_sec=36000	chunk_size_sector=2048, adjust_ref_cnt=2
II	chunk_size=1024, decay_sec=300, freq_aging_sec=18000	chunk_size_sector=2048, adjust_ref_cnt=4
III	chunk_size=2048, decay_sec=100, freq_aging_sec=10000	chunk_size_sector=1024, adjust_ref_cnt=8
IV	chunk_size=2048, decay_sec=2000, freq_aging_sec=42000	chunk_size_sector=512, adjust_ref_cnt=6

5. A real-valued function f defined on a domain X, is said to have a local (or relative) maximum point at the point x^* if there exists some $\epsilon > 0$ such that $f(x^*) \geq f(x)$ for all x in X within distance ϵ of x^*.

7 Conclusions

To comfort benchmarking in a modern cloud storage with flash-based SSDs, we develop `PatIO`, which can generate I/Os that closely resemble real data processing workloads. `PatIO` captures the common characteristics of a group of similar workloads rather than exactly resembling one particular workload. `PatIO` thus can resemble a wide range of realistic I/O workloads. `PatIO` is also lightweight, as it does not require to record, store, and retrieve logs w.r.t. the timestamp of various I/O activities. We developed a GUI interface for `PatIO` to make it easy to use. We evaluated `PatIO` by comparing workload characteristics and performance of synthetic workloads with real workloads on the same system platform. We currently have 15 different I/O patterns in our pattern warehouse that are capable of reproducing 1000+ real workloads. We also deployed `PatIO` to two auto-stream algorithms and evaluated the current advancement of multi-stream technology in terms of its benefits to application performance and SSD endurance. Finally, we proposed a practical technique to automatically tuning variable parameters of the existing stream assignment algorithms for any change in storage capacity or SSD models. In the future, we plan to extend our PatIO warehouse to add new patterns capturing I/O activities of the other compute intensive workloads, changes in system parameters, such as NVM write buffer size, queue depth, and garbage collection algorithm. We plan to design a module that automatically identify important common characteristics and accordingly self-generates new I/O patterns from a set of given workloads using machine learning techniques in combination with statistical computations. We also plan to explore other global convergence techniques that may incur lower overhead and guarantee better performance.

References

[1] J. S. Bhimani, R. Pandurangan, V. Balakrishnan, and C. Choi, "Methods and systems for testing storage devices via a representative I/O generator," Mar. 21 2019, uS Patent App. 15/853,419.

[2] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The Multi-streamed Solid-State Drive." in *HotStorage*, 2014.

[3] J. Bhimani, N. Mi, Z. Yang, J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "FIOS: Feature Based I/O Stream Identification for Improving Endurance of Multi-Stream SSDs," in *IEEE CLOUD*, 2018.

[4] "Blkreplay." [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/man1/blkreplay.1.html

[5] J. Axboe, "Fio-flexible i/o tester synthetic benchmark," *URL https://github. com/axboe/fio (Accessed: 2015-06-13)*.

[6] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "AutoStream: automatic stream management for multi-streamed SSDs," in *International Systems and Storage Conference*. ACM, 2017, p. 3.

[7] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: a database benchmark based on the Facebook social graph," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 1185–1196.

[8] "I/O Meter." [Online]. Available: http://www.iometer.org

[9] "General Tips on Disk Benchmarking." [Online]. Available: http://beyondtheblocks.reduxio.com/8-incredibly-useful-tools-to-run-storage-benchmarks

[10] "Cloudharmony." [Online]. Available: https://github.com/cloudharmony/block-storage

[11] "I/O Analyzer - open-source by Intel." [Online]. Available: https://labs.vmware.com/flings/i-o-analyzer

[12] D. Vanderkam, J. Allaire, J. Owen, D. Gromer, P. Shevtsov, and B. Thieurmel, "Dygraphs: Interface to 'Dygraphs' Interactive Time Series Charting Library," *R package version 0.5*, 2015.

[13] "Cloud Computing Bare Metal Sorage Testing." [Online]. Available: https://community.oracle.com/community/cloud_computing/bare-metal/blog/2017/05/19/block-volume-performance-analysis

[14] "EzFIO." [Online]. Available: http://www.nvmexpress.org/ezfio-powerful-simple-nvme-ssd-benchmark-tool/

[15] "TKperf." [Online]. Available: https://www.thomas-krenn.com/en/wiki/TKperf

[16] "Automating FIO Tests with Python." [Online]. Available: https://javigon.com/2015/04/28/automating-fio-tests-with-python/

[17] "FIO Tests." [Online]. Available: https://github.com/javigon/fio_tests

[18] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, S. Toledo, and A. Zuck, "SDGen: mimicking datasets for content generation in storage benchmarks," in *USENIX FAST*, vol. 15, 2015, pp. 317–330.

[19] A. Haghdoost, W. He, J. Fredin, and D. H. Du, "On the Accuracy and Scalability of Intensive I/O Workload Replay," in *FAST*, 2017.

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

[21] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Generating realistic impressions for file-system benchmarking," *ACM Transactions on Storage (TOS)*, vol. 5, no. 4, p. 16, 2009.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[23] S. Subramanyam, "System and method for testing multiple database management systems," Dec. 23 1997, uS Patent 5,701,471.

[24] Q. Zheng, H. Chen, Y. Wang, J. Zhang, and J. Duan, "COSBench: cloud object storage benchmark," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 2013, pp. 199–210.

[25] D. R. Jiang, T. V. Pham, W. B. Powell, D. F. Salas, and W. R. Scott, "A comparison of approximate dynamic programming techniques on benchmark energy storage problems: Does anything work?" in *Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2014 IEEE Symposium on*. IEEE, 2014, pp. 1–8.

[26] A. Adir, R. Levy, and T. Salman, "Dynamic test data generation for data intensive applications," in *Haifa Verification Conference*. Springer, 2011, pp. 219–233.

[27] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *ACM Transactions on Storage (TOS)*, vol. 4, no. 2, p. 5, 2008.

[28] N. Zhu, J. Chen, T.-C. Chiueh, and D. Ellard, "TBBT: scalable and accurate trace replay for file server evaluation," in *ACM SIGMETRICS*, vol. 33, no. 1. ACM, 2005, pp. 392–393.

[29] R. McDougall and J. Mauro, "Filebench tutorial," *Sun Microsystems*, 2004.

[30] "btreplay - Block Trace Replay." [Online]. Available: https://linux.die.net/man/8/btreplay

[31] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan, "Buttress: A toolkit for flexible and high fidelity I/O benchmarking," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. USENIX Association, 2004, pp. 4–4.

[32] Z. Liu, F. Wu, X. Qin, C. Xie, J. Zhou, and J. Wang, "TRACER: A trace replay tool to evaluate energy-efficiency of mass storage systems," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*. IEEE, 2010, pp. 68–77.

[33] K. Kanoun, H. Madeira, M. Dalcin, F. Moreira, and J. C. R. Garcia, "Dbench*(dependability benchmarking)," 2005.

[34] "nvme-smart-log." [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/man1/nvme-smart-log.1.html

Janki Bhimani Janki Bhimani is an Assistant Professor at Florida International University, Miami.

Adnan Maruf Adnan Maruf is a Ph.D. candidate in computer science at Florida International University, Miami.

Ningfang Mi Ningfang Mi is an Associate Professor at Northeastern University, Boston.

Rajinikanth Pandurangan Rajinikanth Pandurangan is a senior staff engineer at Samsung Memory Solutions Lab – R&D.

Vijay Balakrishnan Vijay Balakrishnan is the Director of Datacenter Performance and Ecosystem Team at Samsung Memory Solutions Lab.