Emulate Processing of Assorted Database Server Applications on Flash-Based Storage in Datacenter Infrastructures

Janki Bhimani[1], Rajinikanth Pandurangan[3], Ningfang Mi[2] 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 datacenters in cloud storages are now replacing traditional HDDs with enterprise SSDs. Both developers and users of these SSDs require thorough benchmarking to evaluate their performance impacts. I/O performance with synthetic workload or classic benchmark varies drastically from real I/O activities in the datacenter. Thus, we propose a new framework, called Pattern I/O generator (`PatIO`), to collectively capture the enterprise storage behavior that is prevailing across assorted user workloads and system configurations for different database server applications on flash-based storage. `PatIO` is designed to emulate the processing of real-world I/O activities easily with less time and resource requirements. Our methodology comprises three main steps: (1) *dissect* the overall I/O activities of various real workloads and identify the prevailing attributes in distinct visual I/O patterns; (2) *construct* a pattern warehouse as the collection of unique I/O patterns that are generated through various combinations of multiple I/O jobs; and (3) finally *integrate* different combinations of these synthetically generated I/O patterns to reproduce the comprehensive characteristics of various real workloads and system setup for the database server applications. To provide an easy-to-use experience, we develop a graphical user interface (GUI). We evaluate our framework by comparing I/O characteristics and I/O performance of generated workloads with those of real-world workloads for multiple database applications such as MySQL, Cassandra, and ForestDB.

***Index terms*—** SSD, I/O Generator, I/O Pattern, Characteristics, Performance

I. Introduction

Optimizing the operation of modern cloud storage systems for various big data applications is critical. Evaluating the effect of any firmware or hardware amendments using real system deployment requires a lot of resources, time and efforts towards installation and running of different applications such as Cassandra and classic benchmarks such as YCSB to test. Moreover, many different virtualization and system setup options such as file systems also need to be tested for each workload. As the I/O behavior of any single database server application with a particular configuration of classic benchmarking workload running on specific system configuration is not sufficient to capture I/O activities of real datacenter. Assorted workloads each with different client system configurations usually operate in any datacenter. Thus, a thorough evaluation of any new invention such as new firmware for SSDs in the datacenter is required testing if the proposed amendment is beneficial for all different applications, workloads, and system settings which is very time consuming and requires a lot of efforts and computing resources. On the other hand, the research advancement by evaluating a tiny subset of these possible variations may have huge failure risk. Thus, emulating the storage behavior that prevails across assorted workloads and different system configuration is very important for developers and users for evolving datacenter storage. Additionally, most traditional synthetic I/O generation tools [2], [3] 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). Thus, the main challenges are that (1) the data generated by these synthetic I/O generators such as FIO [3] is too simple, (2) the classic benchmarks such as YCSB [4], or I/O replay tools [5] demands a lot of time to setup and execute various workloads with different system configuration for each database application, (3) I/O replay tools also consumes a lot of storage space to generate and store different trace logs for each workload and system setup, (4) Finally, replicating the I/O activities of real applications which resembles to the combination of simultaneously executing multiple classic benchmarks with different database server applications and different system configurations is very difficult. Therefore, new easily operable infrastructure is needed to capture comprehensive behaviors of real applications using different workloads and system setup.

Motivated by this, we propose a new framework, called Pattern I/O generator (`PatIO`) to capture the enterprise storage behavior that is prevailing across various user workloads, and system configurations for different database server applications on flash-based storage. The driving force of building `PatIO` is first 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 I/O activities that are performed by different database applications on flash-based SSDs. `PatIO` captures 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 framework 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 as data temperature) and I/O size, frame an I/O layout pattern. Each workload may have many different patterns to represent different real database activities like compaction, and log management. We identify different I/O patterns and synthetically reproduce them.

2) Ensure Scalability and Usability: `PatIO` is scalable to

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

generate I/Os over different sizes of storage disks for desired execution time. PatIO automatically changes 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.

3) Ensure Integrability and Expandability: PatIO can be integrated with evolving SSD technologies such as multi-stream SSDs [6] to fasten research and development. PatIO can be used to measure and tune SSD firmware automatically in datacenters where hyperscalers deploy their customized SSDs. New patterns can be added to the pattern warehouse based on the evolving knowledge of new applications and new workloads.

We evaluate our framework by using different containerized workloads running using standalone as well as simultaneous database application servers 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 II, discusses the existing techniques. Section III presents the PatIO architecture. Section IV evaluates our technique. Finally, we draw our conclusions in Section V.

II. Related Works

Most benchmarking techniques [3], [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 [2], [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 graph and be able to replay the I/O workload accurately. 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.

Table I: 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. [2]							✓
CH [10]	✓				✓		✓

In contrast, PatIO does not require to store, read and follow any I/O trace files 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 [3]). We finally summarize the features of PatIO and different popular existing benchmarking techniques in Table I.

III. PatIO Framework

In this section, we discuss the overall architecture and then elaborate the three components of methodology such as dissect, construct, and integrate of our PatIO [1]

A. *Architecture of* PatIO

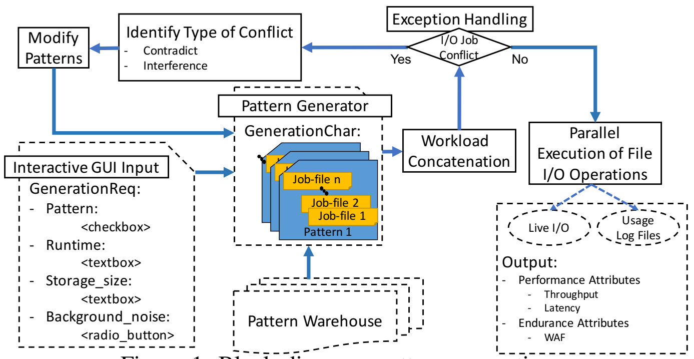

Figure 1: Block diagram: pattern generation

Figure 1 shows the architecture of our framework. First, the front end GUI allows the user to configure a workload's I/O patterns and its expected execution time and to select the size of the storage disk and the desired level of background noise. Based on the selected options, the pattern generator then dynamically pulls corresponding files of patterns from the pattern warehouse.

A workload is a combination of single or multiple patterns, where each pattern is a multi-threaded system process. Also, each pattern 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

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

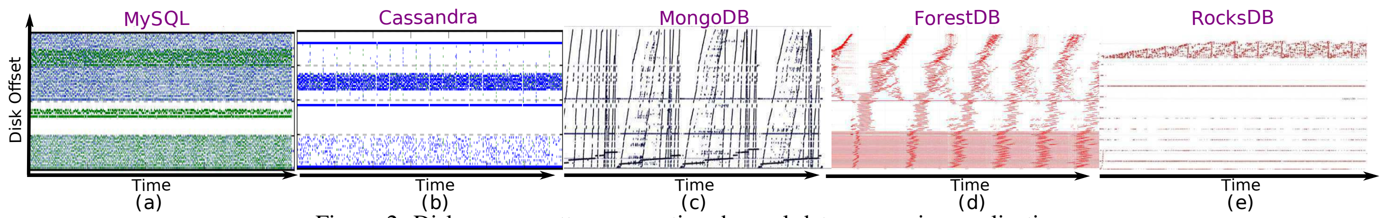

Figure 2: Disk access patterns over time by real data processing applications

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 1 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 arise new conflicts. Thus, workload concatenation is repeated until there are no more conflicts. Later, in Section III-D, we explain details about job modification and types of conflicts Finally, according to the concatenated workload, I/O engine generates I/Os on 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.

B. 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`. Figure 2 shows the I/O access patterns for some representative real SQL and NoSQL database applications. We observe that a real application exhibits variance in I/O activities on disk over time. For example, some applications perform their I/Os in a uniform horizontal stripe wise fashion, see MySQL and Cassandra in Figures 2 (a) and (b). Where MySQL performs SQL transactions on data warehouse tables and Cassandra is a distributed database application working with large amounts of columnar structured data. While some other applications show a periodic pattern of I/O layout over time such as MongoDB and ForestDB in Figures 2 (c) and (d). Where MongoDB performs key value based data model queries, ForestDB is a single-node key value storage engine working with JSON database. Also, there exist applications like RocksDB in Figure 2 (e) that present a horizontal stripe wise pattern. One I/O stripe of RocksDB (see the upper region of Figure 2 (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. Where RocksDB is a high performance embedded database for key-value data. The diversity in I/O access patterns thus 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 2 (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 use different widely used supervised data classification and unsupervised data clustering techniques such as K Nearest Neighbour (K-NN) pattern classifier and K-means clustering. We experiment with different parameter settings such as distance computation matrices (e.g., Euclidean, Manhattan, Chebychev, and Percent disagreement) and number of clusters. We then manually study each result obtained to identify and differentiate various I/O patterns for all the database applications.

C. Pattern Warehouse: Construct

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, each of which is responsible for rendering I/Os for part of the I/O workload pattern to represent some specific I/O layout. Each job is further composed of a set of I/O generating features of FIO engine. Thus, integrating these I/O jobs together can help to capture the diversity of a real I/O workload pattern.

Pattern warehouse is a collection of I/O patterns that can be used to construct different I/O workloads. We *construct* 15 different I/O patterns as listed in Table II and also provide some recommended pattern combinations to resemble real applications, like MySQL, Cassandra, and MongoDB. With different combinations of these 15 patterns we can generate 16384 different workloads. 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 2 as discussed in Section I. In order to replicate such patterns, we

Table II: 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`

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.

To construct a particular I/O pattern, one of the challenging problems is to identify features of I/O generating engine that would issue I/O in a particular fashion to resemble 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 II lists the features that we use to construct each I/O pattern. 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 2 (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. We thus 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, this space might be accessed more frequently compared to 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 that were affected by the "update" operation, 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 execution of workload. 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. We thus use various I/O distributions, such as `zip fin`, `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 into inclined vertical lines across the disk as observed in MongoDB, see Figure 2. We use options like `sequencer` and its offset to generate a sequential I/O sequence with I/O jumps.

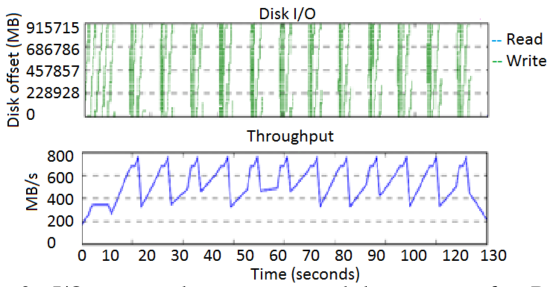

Figure 3: I/O pattern layout generated by `PatIO` for Bamboo Different Allignment (BDA) pattern

Pattern Feature Setting: As mentioned above, Table II 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 II. 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 2 (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.

Figure 3 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 3) of this generated pattern exhibits variance over time. Such throughput variations well match the throughput variations in real applications.

D. 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 and 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 exists a conflict, then according to the type of this conflict, those jobs are modified. 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, but 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 action 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 affect 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, but 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 exe-

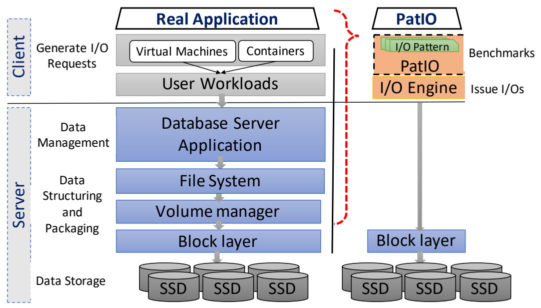

Figure 4: I/O stack of `PatIO` in comparison to that of real application

cute 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.

E. GUI Interface

In order to provide an easy-to-use experience, we develop a GUI interface for `PatIO`. First, this GUI interface allows the user to define the runtime (i.e., execution time) of an I/O workload and the size of the storage space exposed to I/Os. Then, the user can use checkboxes to select one or multiple I/O layout patterns that are close to the I/O patterns of real applications workloads, e.g., MySQL, Cassandra, and ForestDB. Additionally, the GUI allows selecting a different level of background noise which may be incurred by various background I/O activities of the server. 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.

IV. Evaluation

In this section, we evaluate `PatIO` by comparing I/O characteristics and performance of generated workloads with those of real database applications such as MySQL, Cassandra, and ForestDB. Table III gives the detailed hardware configuration of our platform on which we develop `PatIO` and run real application workloads. In `PatIO` we make use of different advance libraries such as pthreads, numpy, and tkinter. We use the FIO engine to generate I/Os. Figure 4 shows the comparison of the I/O stack of `PatIO` (see right side) with that of the real application (see left side). We see that on the client side, `PatIO` abstracts different visualization setup (e.g., Docker containers) and the user workloads (e.g., YCSB, Cassandra-stress, and DB_bench) that are running in the cloud. On the server side, we bypass data management layer of database server applications (e.g., MySQL, Cassandra, ForestDB, MongoDB, RocksDB) and system layers responsible for data structuring and packaging such as file system (e.g., ext4), volume manager (e.g., LVM), and block layer in the datacenters.

To cumulatively capture the NVMe SSDs behavior that is prevailing across various user workloads, virtualization setup, file systems, and volume managers for different database server applications, we setup systems with different server-side configurations and different cloud side virtualizations. Then

Table III: Hardware configuration

CPU	Intel(R) Xeon(R), 32 hyper-threaded CPU E5-2640 v3, 2.60 GHz, Cache Size 20480 KB
Main Memory	128 GB
OS	Ubuntu 16.04 LTS
OS Kernel Version	4.4.0-13generic
File System	ext4
Storage	NVMe SSD 960GB and 480GB
Docker Version	1.11
VMware Workstation	12.5.0
FIO Version	2.2

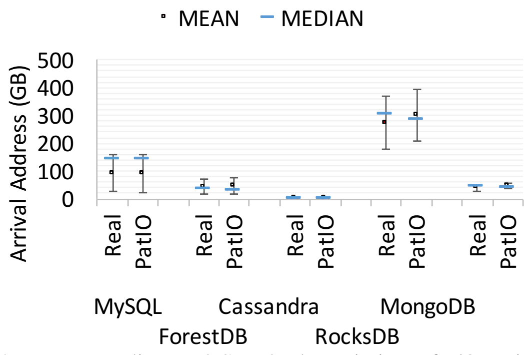

Figure 5: Mean, Median and Standard Deviation of I/O arrivals on disk space over time for real and `PatIO` workloads.

we study the I/O activities by running various user workloads with each database server application. We observe I/O activities of each user workload with different configurations such as the number of transactions, compaction rate, and read-write ratio. We also study the simultaneous operation of the different combinations of applications, e.g., MySQL+Cassandra.

A. Characteristics Comparison

First, we compare the characteristics such as I/O layout on a storage disk, I/O size, and the read-write ratio 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. We perform experiments with 1000+ workloads of different applications. As a representative, we here present results for some of them.

I/O Arrivals: The disk address and the time at which the I/Os arrive is very important I/O characteristic. Figure 5 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.

I/O Size: I/O size is another important characteristic that affects performance. It is critically important to be able to emulate the variance of I/O size over execution time because the sizes of I/Os vary over time for a real workload. It is not sufficient to generate I/Os all with the same size (e.g., average I/O size) as done by most of the prevailing I/O benchmarking tools [14], [15]. Figure 6 shows the comparison of the statistics

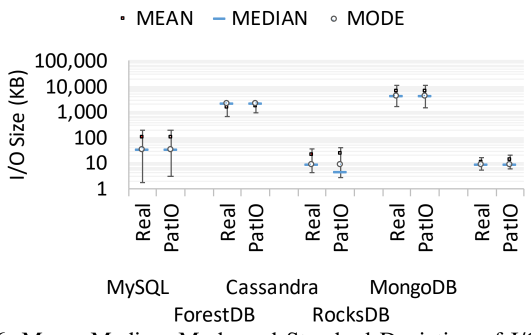

Figure 6: Mean, Median, Mode and Standard Deviation of I/O Size for real and `PatIO` workloads. Note: y-axis is logarithmic scale, so Standard Deviation does not look to be equally distributed on either side of mean.

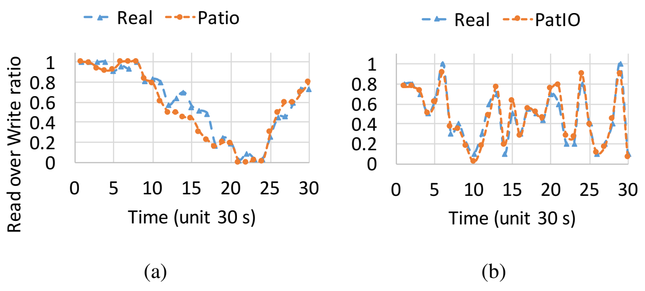

Figure 7: Read to write ratio over runtime of workloads a) MySQL and b) Cassandra

of I/O size over time for real and `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. We further observe that `PatIO` can reconstruct the variance of a real workload as seen from the standard deviation and coefficient of variance. The maximum error percentage while comparing all different metrics of measurement is less than 25%, which indicates a good resemblance between real workloads and `PatIO` ones.

Read-Write Temperature: Besides above statistics comparisons, we also compare the characteristics over runtime of real and `PatIO` workloads. Figure 7 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.

B. Performance Comparison

Throughput: We further run the generated and real workloads on the same system and measure the throughput (i.e., the number of I/Os performed on disk per second) for both. Figure 8 compares the throughput obtained while running one of the real Cassandra workload which was not used in our initial study with the generated workload. Our synthetic workload can precisely capture the throughput range and time. It also resembles the wavy nature of the throughput (see Figure 8), as opposed to the usual constant throughput that is resulted by Naive FIO. We further notice that the throughput of `PatIO` does not exactly mimic the real workload as it is designed to emulate generic behavior prevailing across various user workloads for each database applications, rather than just precisely replicating one particular workload. Also for a real application, there are more intermediate layers in I/O stack when compared to `PatIO` (see Figure 4), which results in initial ramp-up time for real application throughput which is not shown by our generated workloads. Once the cache is constructed, this latency due to intermediate layers is hidden in parallel tasks, so it is custom to neglect the initial performance.

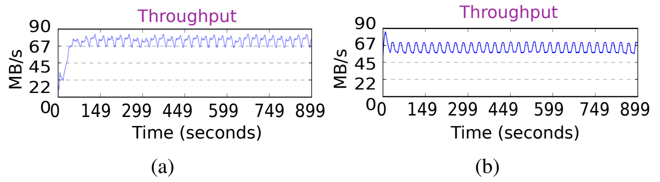

Figure 8: Throughput variation over time for, a) Real Cassandra workload, and b) Generated Cassandra workload

Table IV: Comparing the operational storage space (MB) consumed and execution time (minutes) for (a) Replay [5], (b) `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

C. 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 database application.

Each plot in Figure 9 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, we instrument physical NAND writes of SSD by modifying the `nvme-smart-log tool` [22] for the corresponding firmware of SSDs. 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. While, the naive FIO has comparatively lower correlation than `PatIO`, because workloads generated by the naive FIO fail to capture the intrinsic diversities and variations of real applications.

Table IV shows that when compared to traditional I/O replay tools [5], `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 4, which allows it to execute much faster.

D. Scalability of `PatIO`

`PatIO` is scalable to generate I/Os over different sizes of

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

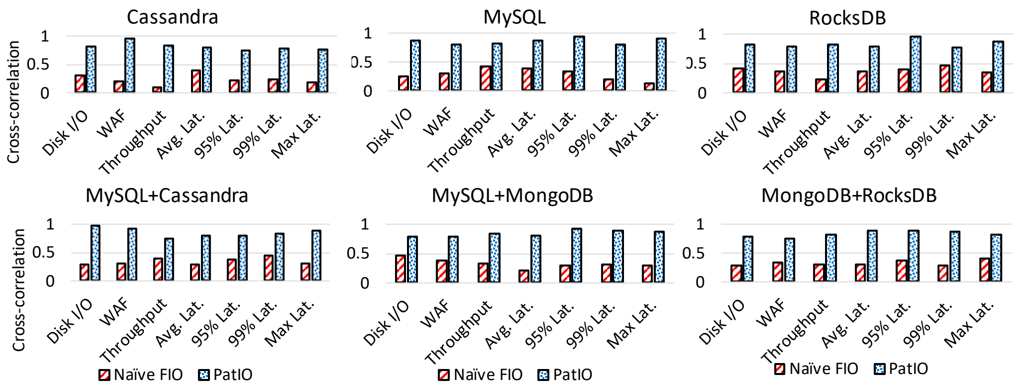

Figure 9: Comparing the cross correlation between real and synthetic workloads using lag of number of samples used for training for (a) traditional FIO that uses average statistics (Naive FIO), and (b) `PatIO`

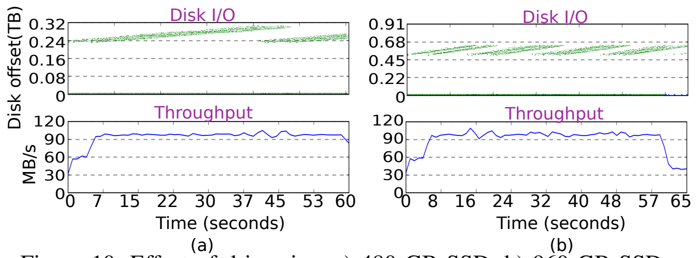

Figure 10: Effect of drive size: a) 480 GB SSD, b) 960 GB SSD

storage disks. We use the `Horizontal Shower (HS)` I/O pattern (see Table II) as an example to investigate `PatIO`'s scalability. Figure 10 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 top row in Figure 10. 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.

V. Conclusions

To comfort benchmarking in a modern cloud storage with flash-based SSDs, we develop `PatIO`, which captures the enterprise storage behavior that is prevailing across various user workloads and system configurations. `PatIO` emulates and allows reproducing I/O activities simultaneous execution of different database server applications on flash-based storage. `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 generated 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. In the future, we plan to add more I/O patterns to our pattern warehouse to capture changes in system parameters, such as NVM write buffer size, queue depth, and garbage collection algorithm.

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] "Blkreplay." [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/man1/blkreplay.1.html
[3] J. Axboe, "Fio-flexible i/o tester synthetic benchmark," *URL https://github. com/axboe/fio (Accessed: 2015-06-13)*.
[4] 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.
[5] "btreplay - Block Trace Replay." [Online]. Available: https://linux.die.net/man/8/btreplay
[6] 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.
[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] "nvme-smart-log." [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/man1/nvme-smart-log.1.html