Automatic Stream Identification to Improve Flash Endurance in Data Centers

JANKI BHIMANI, Florida International University
ZHENGYU YANG and JINGPEI YANG, Samsung Semiconductor Inc.
ADNAN MARUF, Florida International University
NINGFANG MI, Northeastern University
RAJINIKANTH PANDURANGAN, CHANGHO CHOI, and VIJAY BALAKRISHNAN, Samsung Semiconductor Inc.

The demand for high performance I/O in Storage-as-a-Service (SaaS) is increasing day by day. To address this demand, NAND Flash-based Solid-state Drives (SSDs) are commonly used in data centers as cache- or top-tiers in the storage rack ascribe to their superior performance compared to traditional hard disk drives (HDDs). Meanwhile, with the capital expenditure of SSDs declining and the storage capacity of SSDs increasing, all-flash data centers are evolving to serve cloud services better than SSD-HDD hybrid data centers. During this transition, the biggest challenge is how to reduce the Write Amplification Factor (WAF) as well as to improve the endurance of SSD since this device has a limited program/erase cycles. A specified case is that storing data with different lifetimes (i.e., I/O streams with similar temporal fetching patterns such as reaccess frequency) in one single SSD can cause high WAF, reduce the endurance, and downgrade the performance of SSDs. Motivated by this, *multi-stream* SSDs have been developed to enable data with a different lifetime to be stored in different SSD regions. The logic behind this is to reduce the internal movement of data—when garbage collection is triggered, there are high chances of having data blocks with either all the pages being invalid or valid. However, the limitation of this technology is that the system needs to manually assign the same *streamID* to data with a similar lifetime. Unfortunately, when data arrives, it is not known how important this data is and how long this data will stay unmodified. Moreover, according to our observation, with different definitions of a lifetime (i.e., different calculation formulas based on selected features previously exhibited by data, such as sequentiality, and frequency), *streamID* identification may have varying impacts on the final WAF of multi-stream SSDs. Thus, in this article, we first develop a portable and adaptable framework to study the impacts of different workload features and their combinations on write amplification. We then propose a feature-based stream identification approach, which automatically co-relates the measurable workload attributes (such as I/O size, I/O rate, and so on.) with high-level workload features (such as frequency, sequentiality, and so on.) and determines a right combination of workload features for assigning *streamIDs*. Finally, we develop an adaptable stream assignment technique to assign *streamID* for changing

This work was partially supported by National Science Foundation Awards CNS-2008324 and CNS-2008072, the National Science Foundation Career Award CNS-1452751, and the Samsung Semiconductor Inc. Research Grant.
Authors' addresses: J. Bhimani and A. Maruf, Florida International University, 11200 SW 8th St, CASE 238B, Miami, FL 33199; emails: jbhimani@fiu.edu, amaruf@fiu.edu; Z. Yang, 10100 Venice Boulevard Culver City, CA 90232; email: Yangzy1988@gmail.com; J. Yang, 1600 Amphitheatre Parkway, Mountain View, CA 94043; email: jingpei@google.com; N. Mi, 409 Dana Research Building, 360 Huntington Avenue, Boston, MA 02115; email: ningfang@ece.neu.edu; R. Pandurangan, Rajinikanth Pandurangan, 110 Holger Way, San Jose, CA 95134; email: rajinikanth.p@gmail.com; C. Choi, 3655 N First St, San Jose, CA 95134; email: changho.c@samsung.com.

1553-3077/2022/04-ART17 $15.00
https://doi.org/10.1145/3470007

workloads dynamically. Our evaluation results show that our automation approach of stream detection and separation can effectively reduce the WAF by using appropriate features for stream assignment with minimal implementation overhead.

CCS Concepts: • **Computer systems organization → Multiple instruction, multiple data;**

Additional Key Words and Phrases: Solid state drives, multi-streaming, write amplification factor, I/O stream detection, coherency, I/O workload characterization, NAND flash endurance

ACM Reference format:
Janki Bhimani, Zhengyu Yang, Jingpei Yang, Adnan Maruf, Ningfang Mi, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2022. Automatic Stream Identification to Improve Flash Endurance in Data Centers. *ACM Trans. Storage* 18, 2, Article 17 (April 2022), 29 pages.
https://doi.org/10.1145/3470007

1 INTRODUCTION

The emergence in I/O intensive applications requires more fast and more reliable storage pools. Flash-based storage devices such as **Solid-state Drives (SSDs)** thus becoming the mainstay due to their better performance compared to traditional **hard disk drives (HDDs)**, and as a result, today's computing devices widely adopt SSDs. Moreover, the decrease in NAND flash cost-per-gigabyte further accelerates the adoption of SSDs to support enterprise datacenters and virtualized cloud environments.

Although with many advantages of NAND flash technology, one major drawback lies in its limited number of *Write Cycles*. In SSD, there are three main operations, i.e., *Read*, *Program* (write) and *Erase* (delete). Among these operations, Read and Program can be performed at the page level. However, Erase can only be performed at a larger unit of erasure block level that consists of multiple pages. In order to reclaim free space to write new data, the SSD device needs to erase the whole block. The valid pages in the selected block are copied to a new block before erasing. This process of selecting the erase block and moving its valid pages before erasing is called ***Garbage Collection*** (**GC**). Consequently, actual physical writes performed on the SSD device internally are more than the total logical writes performed by user applications running on the host. The ratio of the amount of data physically written to the NAND flash, including additional internal writes, to the amount of data logically written by user applications running on the host, is known as ***Write Amplification Factor*** (**WAF**). The lower WAF indicates better endurance of an SSD device. This is because the flash cells in SSD have a fixed number of write cycles (also called as the **program and erase** (**PE**) cycles), and once the limit is reached, the corresponding cell is worn out and cannot be used anymore (i.e., died). Therefore, it is essential to have an efficient data placement on SSDs for reducing the WAF and enhancing the endurance of SSDs. There are numerous of technologies proposed to address this WAF issue, and one of them is multi-stream SSD. The motivation is to enable data with a different lifetime to be stored in different SSD regions so that the internal movement of data can be reduced.

Multi-stream SSDs: In data centers, the period data resides is often highly variable. Upon storing such data in SSDs, it often inevitably causes a large degree of data fragmentation due to data invalidation and then dramatically increases the WAF when garbage collection is triggered. To address this issue, the storage industry recently developed a new multi-streaming technology [23] that allows a host system to explicitly open different streams in SSD devices and allocate write requests to these streams according to the expected lifetime of data. Traditional SSDs have only one active append point where new data is written. Now, multi-stream technology enables the device to maintain more than one open erase block to append data writes in different physical locations of an SSD. In a multi-stream SSD [1], *streamIDs* are assigned to data according to their

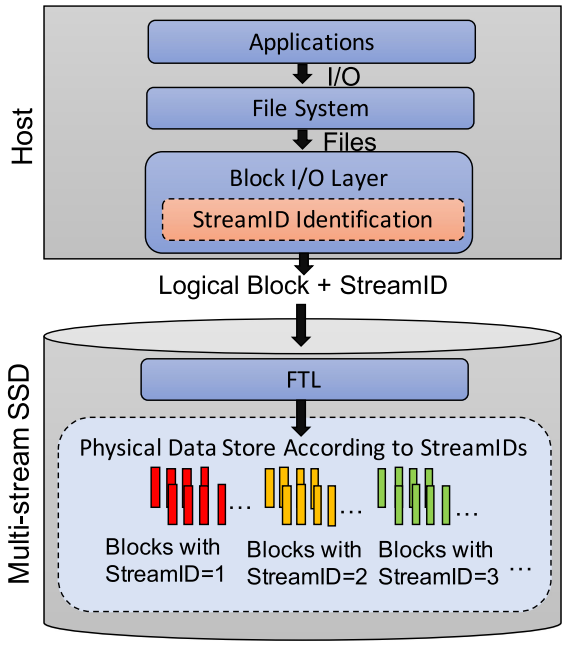


Fig. 1. Operation of Multi-stream SSDs with respect to I/O stack.

lifetime. The assignment of streamIDs can be done at any layer, such as the application layer, the file system layer, or the block layer. Figure 1 shows the I/O stack of a multi-stream SSD where streamID identification is made at the block layer. Once each logical block is assigned a streamID, a list of logical blocks with their corresponding streamIDs will be sent as an input to the **Flash Translation Layer** (**FTL**) in the multi-stream SSD device. The FTL then stores data blocks with the same streamID to the same physical blocks. This ensures that data with the same lifetime (i.e., the same streamID) can be invalidated together, which thus reduces the garbage collection overhead and results in low write amplification by avoiding extra internal data movements.

Challenges of Using Multi-Stream Technology: An efficient streamID assignment in multi-stream flash drives can reduce write amplification and improve the endurance of SSDs. To achieve the best possible benefit of multi-stream SSDs, it is critical to construct good streams, i.e., data with similar lifetime should be assigned the same streamIDs. However, it is challenging to predict the data lifetime. Although historical information of different features (such as frequency, sequentiality, and so on.) for past data accesses can be used to predict the expected lifetime of data, we found that stream identification using different features may have different impacts on the WAF of multi-stream SSDs. Moreover, feature sets that can accurately capture the expected lifetime of data may vary with different applications and workloads. Using a combination of not useful features cannot offer good performance improvement of multi-stream SSDs over no-streaming legacy SSDs. Therefore, a good streamID assignment technique to quantify the impact of different features and their combinations on the endurance of SSDs is essential.

Novel StreamID Assignment Approach: We extract various I/O workload features (e.g., frequency, adjacent access, and sequentiality) to study the impacts of these features and their combinations on write amplification of multi-stream SSDs. However, we observed that these features are not sufficient to capture the lifetime of data when using them individually. We thus propose a new feature, named *coherency*, to capture the friendship between logical blocks. Coherency can be more closely related to the lifetime of data. Unfortunately, by investigating all these different

features, we found that none of the features (e.g., frequency, adjacent access, sequentiality, and even coherency) can be claimed as the best for all I/O workloads, different features have varying impacts on WAF, and the benefit derived by using the combination of multiple features is not additive. Thus, another big challenge in developing this multi-stream framework is *how to determine a combination of workload features that are best for assigning appropriate streamIDs under a given I/O workload.*

To address this issue, we build an analytical *correlation model* to capture the co-relation between easily obtained workload attributes (such as I/O size, random write ratio, reuse ratio, and autocorrelation of write rates) with high-level workload features (such as frequency and coherency) and develop a novel **F**eature-based **I/O** **S**tream identification (**FIOS**) method to identify the best set (or combination) of features suitable to a workload for streamID assignment. FIOS can obtain a good feature combination automatically rather than experimenting with all possible combinations in a brute-force manner. we believe in the domain of multi-stream SSDs, this work is very important, as this is the first work that systematically measure, quantify, and understand the performance of multi-stream SSDs while running simultaneous instances of various containerized applications.

We modify the SSD module of DiskSim[1] [7] to simulate the multi-stream SSDs. First, we implement FIOS in modified DiskSim and then spend a lot of efforts and time towards designing Linux kernel Daemons for the real system implementation of OFIOS. We compare our performance with the state-of-the-art algorithms [39, 40, 42]. We consider the legacy SSDs that do not use multi-streaming technology as the baseline. Our experimental results show that our techniques can always identify a good combination of appropriate features to decide streamIDs and thus be able to improve the lifetime of SSD devices by reducing WAF.

Summarizing, this article provides the following major contributions

- Investigate the correlation between workload attributes (e.g., reuse ratio, write rate, and so on.) and high-level application features (e.g., coherency, sequentiality, and so on.), and analyze the impact of different features and their possible combinations on SSD **write amplification** (**WA**).
- Propose a new feature, *coherency*, which represents the friendship between write operations with respect to their update time.
- Design a dynamic streamID detection scheme, which can determine a good combination of multiple features for different workloads.
- Develop and implement in Linux kernel, a parallel background learning mechanism that does not interfere with foreground stream identification.
- Design a novel auto-tuning module to dynamically identify and set the best feature combination while running simultaneous parallel instances of diverse containerized applications.

Section 2 presents the preliminary results to motivate this research. Section 3 presents the framework design of our proposed FIOS and OFIOS. Section 4 describes the architecture and data structures used and explains the system modifications required to deploy our framework. Section 5 describes the details of our platform setup to perform the evaluation. In Section 6, we evaluate FIOS and OFIOS. We discuss the implementation choices and overheads in Section 7. Section 8 presents related works. Lastly, we summarize our conclusions and future plan in Section 9.

2 MOTIVATION

The technology of multi-stream SSDs [1, 15] is newly invented to allow to explicitly write different data that are associated with each other or have a similar lifetime into a single stream. That

[1] https://github.com/benh/disksim/tree/master/ssdmodel.

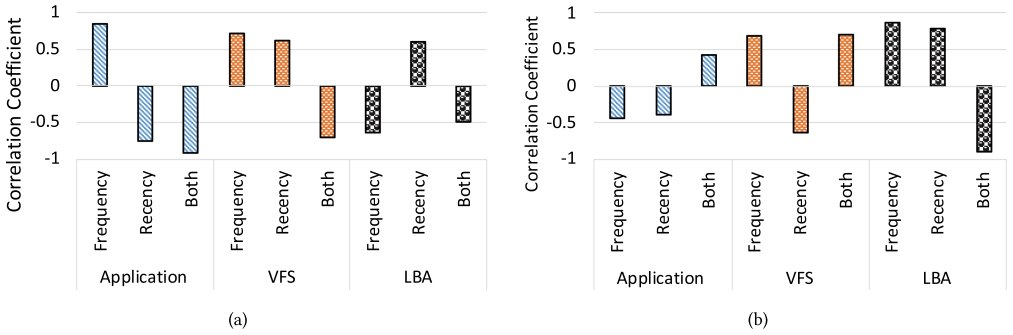


Fig. 2. Correlation Coefficient in RocksDB: (a) Standalone, (b) 16 Parallel Containers.

is, a group of data writes may be a part of a stream, and each stream is identified by a unique streamID that can be assigned either by the operating system or by the corresponding application. Identifying streamID that captures the lifetime of data is very important.

As discussed in the earlier works [39, 42], it is challenging to choose an appropriate layer of instrumentation to accurately identify streamID that captures the lifetime of pages on SSDs. A straightforward approach is to study the types of operations in the write path of an application and assign streamID based on file types [2, 15, 38]. For example, an application like Cassandra [31] mainly consists of three different types of operations in its write path—logging data to the commit log, flushing data from memtable to sstable, and performing background compaction. Thus, depending upon such information from each application, we can modify applications like Cassandra [31] to assign a different streamID to commit logs, compressed metadata, sstable indices, and sstable data. This approach results in reducing WAF up to 65% [2]. However, first, *it is not adaptable as this requires studying and individually modifying each application to make them capable of assigning streamIDs.* Second, in data centers and virtualized cloud infrastructures where multiple applications run simultaneously, the combined characteristics of the write path may vary significantly compared to that of each individual application.

On the distributed and parallel platforms, the details instrumented at higher levels of the OS stack such as application layer, file system, or container data volume may not correlate well with the actual data layout on storage devices, especially when we have multiple instances of different applications running simultaneously in parallel using multiple storage devices. To further analyze the above hypothesis, we instrument characteristics at three different layers of I/O stack, (1) application—by grouping data using the application-level information of SSTable levels following the approach discussed in previous works such as VStream [37, 42], (2) **virtual file system (VFS)**—by grouping data according to the **program context (PC)** at VFS layer following PCStream [25], and (3) block layer—by grouping data according to the characteristics observed by analyzing logical block addresses as done in AutoStream [39]. We use the RocksDB database with a YCSB workload of 10,000 records. We operated standalone instances and 16 simultaneous instances, each running in a separate docker container. We choose two features that are most commonly used in all the prior works [25, 39, 42] as frequency also known as temperature and recency. We rank the importance of the data according to frequency, recency, and both combined. Figure 2, shows the **Pearson correlation coefficient (PCC)** [9] of features with the data pages lifetime on SSD. The closer the value is to 1 or −1, the more relevant it is to the data page lifetime. For this experiment, we compute the lifetime of the data pages by performing post hoc analysis on the data placements while replaying workload traces on the DiskSim simulator.

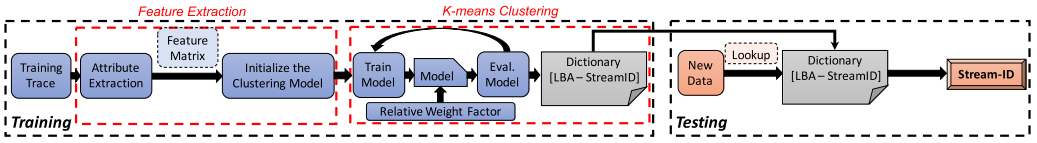


Fig. 3. Block diagram of our FIOS framework.

		Feature_1	Feature_2		Feature_j		Feature_m
1	Sector_chunk_1	0	1		1		0
2	Sector_chunk_2	1	0		0		0
.							
i	Sector_chunk_i	0	1		0		1
.							
n	Sector_chunk_n	1	0		0		1

Fig. 4. Feature matrix data structure.

From Figure 2, we see that for standalone deployment, the application level data co-relates strongly with the data lifetime. However, while multiple instances of applications are running in parallel, then the data instrumented at lower levels such as block layer strongly co-relates with the data lifetime. This is because, while running parallel application containers, the farther layer of instrumentation and decision making means higher chances of dependencies on design choices used for the intermediate layer, which thus reduces the co-relation of the observations from instrumented data to the lifetime of data stored on SSDs. As a result, the prior techniques such as "VStream," "Optimizing NoSQL DB on flash," "PCStream," and "AutoStream," that are tightly coupled with higher layers of the host to identify streamIDs, are not much useful in heterogeneous parallel platforms. From Figure 2, we also see that the correlation to the data lifetime while using information from multiple features can be higher than just using individual features. Thus, motivated by the two major abovementioned observations, we propose a new feature-based approach that strives to determine and assign streamID at block layer, based on the best combination of multiple features.

3 FRAMEWORK DESIGN

In this section, we explain the design and the main components of our framework.

3.1 `FIOS` Design

The block diagram of our `FIOS` technique is shown in Figure 3, which consists of two main phases: *training* and *testing*. The training phase is the pre-processing step for streamID detection, which needs to be performed only once. The testing phase represents the actual runtime phase of applications, during which streamID assignment is performed. Training and testing phases are both performed in a cyclic pattern to capture the runtime workload changes. To better understand how `FIOS` operates, we give an example of single cycle of these two phases. `FIOS` first uses `blktrace` and `blkparse` commands to obtain a real I/O block trace from the application platform as a training trace. This trace is then used by the training phase to extract the required features such as frequency, adjacent access, sequentiality, and coherency. Later, we describe the details of how each feature is extracted from an I/O trace in Section 3.2.

The captured features are enclosed in the form of a feature matrix, such as the one shown in Figure 4. Accordingly, the feature matrix consists of $n \times m$ cells, where m is the number of features analyzed for streamID detection, and n is the total number of sector_chunks in the storage

volume. The storage volume may include a single SSD or multiple SSDs. Each sector_chunk comprises of several sectors. The sector_chunks in rows are arranged in an incremental numerological order such that the product of row number and each sector_chunk's size gives the sector_chunk address. Each cell in the feature matrix gives data quanta, which reflects the importance of the *i*th sector_chunk with respect to the *j*th feature. Notice that if a sector_chunk consists of only a single block address, then that sector_chunk would be the same as a **logical block address** (**LBA**). Thus, we use sector_chunk and logical block address interchangeably in this article.

Through feature extraction, `FIOS` creates a feature matrix, which is then used for clustering write/update sector_chunks into different streams. To obtain high computational efficiency, we improved a multi-threaded K-means clustering algorithm [10] to cluster sector_chunks into K streams in parallel. In particular, each feature makes one dimension of clustering inputs, and a relative weight factor can be used as an optional input to emphasize the relative importance of each feature in deciding streamIDs. By default, all features are considered to be equally important with the same weights. The number of clusters (i.e., K) of the K-means algorithm maps to the number of streams supported by the SSD drive (e.g., 16 streams in the latest SSD drives). `FIOS` uses the K-means algorithm to group all data points in the feature matrix into the given number of streams. The clustering results are stored in the `LBA-StreamID` dictionary that consists of the pairs of sector_chunks and streamIDs.

In the testing phase, we have actual I/O operations (e.g., writes/updates) performed on storage devices. For a read, the operation of legacy and multi-stream SSDs remains the same. For a write or an update, a multi-stream SSD allows multiple append points. Thus, to decide data placement on a multi-stream SSD, `FIOS` assigns a streamID to each sector_chunk through a quick lookup in the `LBA-StreamID` dictionary. Thereafter, the assigned streamID is penetrated through the I/O stack until the data is actually written to the physical address space of that streamID. For an erase, both legacy and multi-stream SSDs work in the same way by searching all finalized blocks for a GC candidate and then copying out valid pages from the candidate.

3.2 Feature Extraction

Now, we turn to present how `FIOS` extracts workload features from the collected I/O traces and completes a feature matrix. As introduced above, the feature matrix comprises of the importance factor for each sector_chunk with respect to different features. In order to keep low instrumentation overhead, we decide to represent the importance factor for a sector_chunk as a binary datum. Each entry (i,j) in the feature matrix can then be represented by a single binary bit (e.g., 1 or 0) that indicates whether *j*th feature is considered to be important ("1") or not ("0"). We use a vector $\overrightarrow{\delta}$, to contain the thresholds for each feature, as criteria to determine the values (i.e., 0 or 1) for each entry.

For instance, the feature of frequency indicates how often a particular sector_chunk is accessed. If the number of accesses of that sector_chunk is greater than the predefined threshold (e.g., $\delta[frequency] = 4$ times), then the frequency entry of that particular address is set as 1, otherwise 0. We set $\delta[frequency]$ to the median value of the frequencies of sector_chunks, assuming that the frequencies of sector_chunks follow a Gaussian distribution. The feature of the adjacent access indicates that sector_chunks that are adjacently accessed during a time window are more likely to be accessed together again. We set $\delta[adjacent\_access]$ to construct multiple groups according to the access time of sector_chunks. We can also have the feature of sequentiality to capture if an incoming I/O access is sequential to the previous one. We calibrate the threshold vector using an ensemble of piecewise linear regression models [33].

		Feature_1	Feature_2		Feature_j		Feature_m
1	Sector_chunk_1	0.3	0.6		0		0.3
2	Sector_chunk_2	0.9	1		0.7		0.6
.							
i	Sector_chunk_i	1	0.1		0.4		0.2
.							
n	Sector_chunk_n	0.5	0.2		1		0.8

Fig. 5. Decimal fraction feature matrix.

3.3 Binary to Decimal Extension of Algorithm

As explained above, one of the limitations of FIOS is that it is a binary model. The binary version of FIOS only considers a single binary bit as an importance factor for each feature. This may easily become a limitation to capture complex workloads. For simple features like sequentiality, it is straightforward that any sector_chunks can either be sequential or not. However, for more advanced features like coherency, a particular sector_chunk may be more coherent with some groups than others. In order to capture such various levels of similarities, we extend FIOS to consider the decimal feature matrix instead of binary. The decimal extension gives a better resolution to the feature matrix. We choose decimal over hexadecimal just because the number with base ten is well understood and can be easily represented as percentages. It is important to note that the number of partitions to separate data into different streams for multi-stream SSD is not limited to 2 for binary and 10 for the decimal extension. But, being independent on the resolution of the feature matrix, FIOS as explained in Section 3.1 can be used to construct a dictionary with any number of streams supported by multi-stream SSDs.

Mainly, we modify our feature extraction technique to label all sector_chunks with a decimal between 0 and 1 that indicates internal grouping with respect to each feature. Figure 5 shows an example of the feature matrix with decimal fractions. Particularly, for features, such as frequency and adjacent_access, we set the threshold (δ) as a stepwise function. This stepwise function partitions the sector_chunks into ten different categories. The feature of sequentiality is straightforward and binary in nature, so it remains the same. Finally, while extracting coherency, we maintain a counter to obtain the number of friendly groups of each sector chunk. Then, we perform linear range shifting to obtain a value of coherency feature between 0 and 1 for all sector_chunks.

Once the decimal feature matrix is generated, the K-means algorithm is run to cluster features to build a dictionary. After that, during the runtime for the testing phase, the operation remains the same as in the case of a binary algorithm. It is interesting to analyze the endurance gain and additional overhead introduced by generating such a decimal feature matrix when compared to its binary counterpart. We next present our evaluation using the decimal feature matrix. We integrate the binary and decimal feature matrix as a plugin with our FIOS framework that allows user you change easily upon restart. Also, note that changing feature extraction from binary to decimal and vice-versa upon reboot will not impend any operation on existing data as the only data we persist from prior mode is Dictionary and its format remains the same for both binary and decimal feature extraction.

3.4 Coherency

Unfortunately, we observe that existing well-known features like frequency and sequentiality are not sufficient to capture the lifetime of data when using them individually. We thus propose a new feature, named *Coherency*, to capture the friendship between sector_chunks. Coherency can be more closely related to the lifetime of data. We refer to two addresses (i.e., sector_chunks) as friends if we observe that they are mostly updated together in multiple time windows. Intuitively,

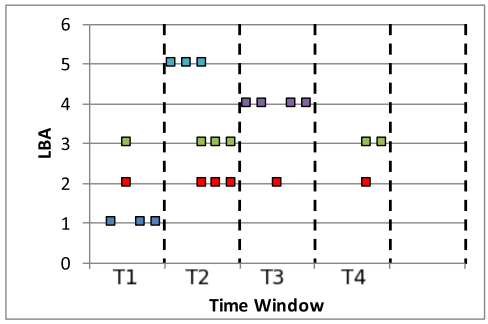


Fig. 6. A sampling graph for describing a novel feature of coherency.

grouping coherent addresses in the same stream can be beneficial because all sector_chunks in that stream will have a similar update pattern.

Figure 6 illustrates the general idea of the coherency feature, where the x-axis corresponds to time windows, and the y-axis corresponds to LBAs or sector_chunks. If a particular group of LBAs are mostly updated together in multiple time windows, then these LBAs may be referred to as being coherent with each other. For example, if we snoop at a specific time interval, which is shown by a vertical dashed line in Figure 6, we can say that LBA2 and LBA3 are coherent because they are concurrently updated in three of the four-time windows, i.e., T1, T2, and T4. This indicates that these two sector_chunks (i.e., LBA2 and LBA3) have a tendency to be updated at the same time, such that grouping them together into a single stream can help to reduce WAF.

In particular, we maintain lists of unique sector_chunks that have been accessed in each time window. We compare the list of each time window to identify the common sector_chunks that appear in at least two lists. We then mark these common sector_chunks as coherent (i.e., 1). The process is finished when all unique sector_chunks of every time window are allocated their coherency values. In our current model, because we consider the datum of our feature matrix as either 0 or 1, it is a limitation that we cannot differentiate different groups of friends while capturing coherency. As of now, our model only partitions sector_chuncks into two groups, i.e., one consisting of LBAs which are "friendly to somebody" and the other consisting of LBAs, which are "friendly to nobody."

Additionally, as many of our workloads and file systems are append-only, it is interesting to understand how coherency can be useful for them. The SSD address space is fixed depending upon the capacity of the SSD. So, eventually, after SSD reaches its steady-state, even if the workload is append-only, physical pages in that SSD that stores "friendly" LBAs according to our feature of coherency will be invalidated at nearly the same time. It is also important to clarify that even though an append-only application maintains a never-ending log, this log is eventually mapped to the finite LBA space. Updates to a set of "friendly" LBAs are not necessarily caused by the updates of the same application data. Although we do not perform in-place updates for append-only workloads, but rather perform compaction such as for levels in RockDB. We want to claim that because of data mappings done at the intermediate operating system layers (such as file system) and SSD-related management policies (e.g., FTL mapping, wear-leveling, garbage collection), a group of sector chunks can be recurrently accessed together. The coherency feature we consider and instrument at the block layer aims at capturing the relationship of these sector chunks.

Regarding the bursty and not heavy writes, there are some different scenarios. First, if there are very sparse and eventual writes that have arriving intervals considerably longer than the time interval we compute coherency, then we label these sparse writes as "coherent to nobody". Second,

Table 1. Summary of the Relation between Features and Workload Characteristics

Feature	*ACF*	*RW*	*R*	*S*	λ
Adjacent Access (A)	High	-	-	-	High
Coherency (C)	High	High	-	-	-
Sequentiality (S)	-	-	-	High	-
Frequency (F)	-	-	High	-	-

(*ACF* - Auto-Correlation of Inter-arrival Time, *RW* -Random Write Ratio, *R* - Reuse Ratio, *S* - Estimated Size of each Writes in KB, λ - Write Rate in GB/Day).

if these writes are sparse but have intervals shorter than the time interval we compute coherency, then we mark these sector chunks as coherent because they are updated together even though their traffic is not heavy. On the other hand, if writes are bursty but these simultaneously accessed sector chunks are not re-accessed together later, then the proportion of coherent sector chunks will remain small.

3.5 Combination of Multiple Features

As we discussed, an I/O workload has different features, such as frequency, coherency, and so on. How to use one or multiple features to cluster data points into the desired number of streams is not trivial. We found that using multiple features might give better performance than using a single feature. However, we also found that using unsuitable features to assign streamIDs may not be able to help to improve the performance and even may cause performance degradation. Thus, a critical issue is how to find out an optimal combination of features that can enable `FIOS` to achieve the high quality of streamID packetization with the minimum overhead. Given n features, we can have $2^n - 1$ possible combinations. We investigate and evaluate the impacts of these feature combinations on the write amplification of multi-stream SSDs in Section 6.1.

In summary, our results indicate that (1) none of the features (such as frequency, adjacent access, sequentiality, and coherency) can be claimed as the best for all I/O workloads, (2) different features have varying impacts on WAF, and (3) the benefit derived by using the combination of multiple features is not additive. Therefore, a big challenge in developing this multi-stream framework is *how to determine a combination of workload features that are best for assigning appropriate streamIDs under a given I/O workload.* Investigating the results for many workloads, we further propose a new approach to determine a good feature combination.

3.6 Automatic Feature Selection

To address the above issue, we build an analytical *correlation model* to capture the co-relation between easily obtained workload characteristics, such as I/O **size** (**S**), **random write ratio** (**RW**), **reuse ratio** (**R**), and **autocorrelation** (**ACF**) of write rates) with high-level workload features (such as frequency and coherency) and identify a good set (or combination) of features suitable to a workload for streamID identification. Our goal is to obtain such a good feature combination automatically, rather than experimenting with all possible combinations.

The initial step in our approach is to determine the workload characteristics that can mainly affect the selection of features. Table 1 summarizes the implications that we develop regarding the relationship between workload characteristics and features used for streamID identification. For example, as shown in the first row of the Table 1, if a workload has high ACF and high λ, then **adjacent access** (**A**) is one of the good features to select.

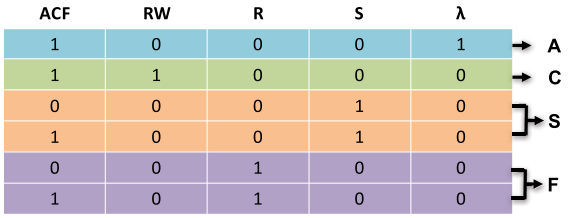


Fig. 7. Bit mapping log of base matrix $\mathbb{B}$.

Based on the information in Table 1, we construct a base matrix $\mathbb{B}$ as shown in Figure 7, where "1" corresponds to the "High" impact in Table 1 and "0" corresponds to the "Low" impact. In $\mathbb{B}$, some features (e.g., **sequentiality** (**S**) and **frequency** (**F**)) need to have 2 rows each. For example, no matter ACF of inter-arrival time is high or low, the feature of frequency is a good choice as long as the R is high. If we can map a workload's characteristics to a row in the base matrix, then we can choose the corresponding feature. However, we notice that the characteristics of a workload may not exactly map to one of the 6 possible rows of base matrix $\mathbb{B}$, and thus it is not straightforward to determine the best combination of features. Given this, we have the following problem objective and solutions.

Objective: Assume a map function f_{map} which determines the best possible combination of *features* ($\overrightarrow{\psi} \rightarrow F, A, S, C$) based on the workload characteristics ($\overrightarrow{\theta} \rightarrow ACF, RW, R, S, \lambda$), i.e.,

$$\overrightarrow{\psi} = f_{map}(\overrightarrow{\theta}). \tag{1}$$

Thus, our objective is to determine the above defined function f_{map}.

Solution: The attribute vector $\overrightarrow{\theta}$ is constructed based on users input of workload characteristics, where an attribute is assigned 1 if the given workload exhibits such a characteristic. For instance, after analyzing a given workload's characteristics (consider *MSR-prn0*), we can get its attribute vector as $\overrightarrow{\theta}$ *as* [1 1 0 0 1]. It represents that *MSR-prn0* has high *ACF*, *RW*, and λ, and low *R* and *S*. If the attribute vector $\overrightarrow{\theta}$ is given by the user has "0" for a particular attribute, then any feature that has a high impact of this attribute is definitely not a good candidate. For example, for the above considered $\overrightarrow{\theta}$, attributes such as R and I/O S are "0". Thus, the features of sequentiality and frequency which have a high impact on I/O size and reuse ratio (as seen from Figure 7) are not good candidates. We construct the factorization vector $\overrightarrow{\alpha}$ that indicates the above information of whether or not each row in base matrix $\mathbb{B}$ is a good candidate for given $\overrightarrow{\theta}$. Such a factorization vector $\overrightarrow{\alpha}$ can be represented as below,

$$\overrightarrow{\alpha} = [\alpha_1,\ \alpha_2,\ \alpha_3,\ \alpha_4,\ \alpha_5,\ \alpha_6]^T, \tag{2}$$

where $\alpha_i \in \{0, 1\}$. If α_i is 0, then the i^{th} row of base matrix $\mathbb{B}$ is not a good candidate and vise-verse. Moreover, as atleast one of the features must be selected, we have $\sum_{i=1}^{6} \alpha_i > 0$. Thus, we obtain α by the following elimination on base ($\mathbb{B}$) with respect to careful examination of input attribute vector $\overrightarrow{\theta}$. This elimination is represented by a factorization function f_{fact} as,

$$\overrightarrow{\alpha} = f_{fact}(\overrightarrow{\theta}). \tag{3}$$

Finally, the function f_{map} can be expressed by some other function f^{-1}_{deriv}. Function f^{-1}_{deriv} is just the replacement of f_{map} such that $f_{map}(\overrightarrow{\theta}) = f^{-1}_{deriv}(\overrightarrow{\theta}) \quad \forall \overrightarrow{\theta}$. This is done to make the rest of the process easier. Further, $f^{-1}_{deriv}(\overrightarrow{\theta})$ gives all possible combinations of the rows of base

ALGORITHM 1: Workload_Run_Daemon (WRD)

Parameter: $T_{RunWindow}$, $T_{TrainWindow}$
Initialize : $TimeStamp = 0$,
$Workload\_Set[W_1, W_2, \ldots, W_n]$,
$curr \rightarrow BD = NULL$,
$curr \rightarrow SID = NULL$,
$\overrightarrow{W_{\overrightarrow{\theta}}} = [W_{1\overrightarrow{\theta}}, W_{2\overrightarrow{\theta}}, \ldots, W_{n\overrightarrow{\theta}}]$,
$\overrightarrow{W_{\overrightarrow{\psi}}} = f_{mapFeat}(\overrightarrow{W_{\overrightarrow{\theta}}})$
Input: *Sector_Chunk*
Result: *StreamID*

while $TimeStamp < Workload\_Time$ **do**
  while $curr \rightarrow time \bmod T_{RunWindow} == 0$ **do**
    $start\_time \leftarrow \overrightarrow{curr \rightarrow time}$
    $curr \rightarrow W \in \overrightarrow{Workload\_Set}$
    set $W_{\overrightarrow{\theta}} \leftarrow \overrightarrow{W}_{\overrightarrow{\theta}}[curr \rightarrow W]$
    set $W_{\overrightarrow{\psi}} \leftarrow f_{mapFeat}(W_{\overrightarrow{\theta}})$
    while $0 \leqslant \frac{(T_{TrainWindow} - start\_time)}{T_{TrainWindow}} \leqslant 1$ **do**
      $SID \leftarrow \text{WTD}(W_{\overrightarrow{\psi}}, T_{TrainWindow})$
  if $SID \neq \emptyset$ **then**
    $StreamID \leftarrow hash\_get(SID(Sector\_Chunck))$
  else
    $curr \rightarrow time = TimeStamp++$

B for which elements in $\overrightarrow{\alpha}$ is equal to 1, i.e., $(\overrightarrow{\theta} = f_{deriv}(\bigvee_{i=0}^{6}(\overrightarrow{\alpha_i} \times B_i))$. We solve equation $\overrightarrow{\theta} = f_{deriv}(\bigvee_{i=0}^{6}(\overrightarrow{\alpha_i} \times B_i)$ for the only unknown "f_{deriv}" in it. Recall that the attribute vector $\overrightarrow{\theta}$ is constructed based on users input of workload characteristics, where an attribute is assigned 1 if the given workload exhibits such a characteristic. We construct the factorization vector $\overrightarrow{\alpha}$ that indicates the above information of whether or not each row in base matrix B (i.e., shown in Figure 7) is a good candidate for given $\overrightarrow{\theta}$. Finally, in the last step just by computing the inverse of this function f_{deriv}, we can achieve our objective of finding ψ as $f^{-1}_{deriv}(\theta) = f_{map}\overrightarrow{(\theta)} = \overrightarrow{\psi}$.

3.7 OFIOS: Online FIOS

The workloads being executed in data centers change over time. Also, in a virtual machine and containerized environment, the number of active instances of the workloads varies. In order to predict the best feature combinations and maintain a dictionary that automatically adapts to the changing workloads, we design and develop the **Online `FIOS` (`OFIOS`)** technique. To dynamically adapt the dictionary constructed using `FIOS`, we run two parallel bunches of processes. One bunch of processes undertake the operations of `FIOS` as explained in Sections 3, 4, and 3.3. To manage these processes, we design a new **`Workload_Run_Daemon` (`WRD`)** and implement it using `systemd` [3]. Algorithm 1 describes the runtime process of `OFIOS`. Upon a write/update to any particular sector_chunk, `WRD` performs a lookup in the **`Simplified Index Dictionary` (SID)** as explained in Section 4 and returns the StreamID to place that data on the flash-based storage. To initialize, we capture the characteristics (θ) of different workloads and multiple instances of parallel workloads (i.e., $W_1, W_2, \ldots, W_n$) into the feature matrix ψ as described in Sections 3 and 3.3. During the workload execution, within the current time window $T_{RunWindow}$, the latest patch of `SID` with respect

ALGORITHM 2: Workload_Training_Daemon (WTD)

Initialize: $TimeStamp = 0$
Input: $\overrightarrow{W_{\psi}}, T_{TraceWindow}, T_{TrainWindow}$
Result: SID
while $TimeStamp < T_{TraceWindow}$ **do**
  $label\_blk \leftarrow random\_sampling[/dev/disk]$
  $trace\_exe \leftarrow blktrace[label\_blk]$
  set $TrainTrace \leftarrow blkparse[trace\_exe]$
  $TimeStamp++$
while $TimeStamp < T_{TrainWindow}$ **do**
  set $BD \leftarrow$ Run FIOS on $TrainTrace$ with feature $\overrightarrow{W_{\psi}}$
  $hashset\ SID \leftarrow Reduction(BD)$
  Update $curr \rightarrow BD = BD$
  Update $curr \rightarrow SID = SID$

to that of the last time window is pulled from the **`Workload_Training_Daemon (WTD)`** (`WTD` is explained next). Finally, OFIOS looks up `SID` for the StreamID of any particular sector chunk.

The second set of processes operates simultaneously in the background during the operation of `WRD` by the first set of processes. The processes in this second set use non-blocking I/Os [13] to construct a new dictionary. This dictionary under construction ensures to incorporate the recent and ongoing changes in the workloads and their active instances. Our `WTD` manages the second set of processes. Algorithm 2 describes the operations of `WTD`. For each time window ($T_{TrainWindow}$), `WTD` takes the input of the feature matrix ($\overrightarrow{W_{\psi}}$) generated in the last window and produces a new `SID`. In the first $T_{TrainWindow}$ time period of each window which is $T_{TraceWindow}$ long, we do the random sampling of LBAs from the storage address space. Then, we run `blktrace` and `blkparse` to instrument the characteristics of the running workload as $TrainTrace$. Finally, we run `FIOS` with $TrainTrace$ and $\overrightarrow{W_{\psi}}$ to update the dictionary, which is used to assign streamID in the `WRD` launched after that. We evaluate OFIOS and compare our performance with the state-of-the-art existing techniques in Section 6.4.

4 MAIN ARCHITECTURE

In this section, we present the I/O stack architecture overview of our `FIOS` implementation and introduce the basic data structures used in the implementation. Following that, we discuss the modifications on an existing SSD simulator, i.e., Disksim [7] to enable the multi-stream interface.

4.1 FIOS: Architecture Overview

Our prototype of `FIOS` can be implemented at any levels, such as the file system layer on the host side or the FTL layer inside the device. In our implementation, we choose to deploy our prototype at the block layer. Later in Section 7, we will discuss the benefits of this design choice. Figure 8 shows the I/O stack of `FIOS` architecture, which consists of two main components to assist in streamID assignment, i.e., (1) **Base Dictionary** (**BD**) and (2) SID. As shown in Figure 8, BD is persisted in flash memory and SID is stored within the memory management subsystem in DRAM. On a system failure like a power outage, SID can be rebuilt from BD on rebooting. At the software level, host applications may run directly in the system, or containers like Docker and LxC. The application layer performs read and write I/Os, which are passed to the underlying file system and flash memory through system calls. For every I/O write or update, a lookup operation is performed to SID by calling `hash_get` to get streamID for the corresponding block addresses. The

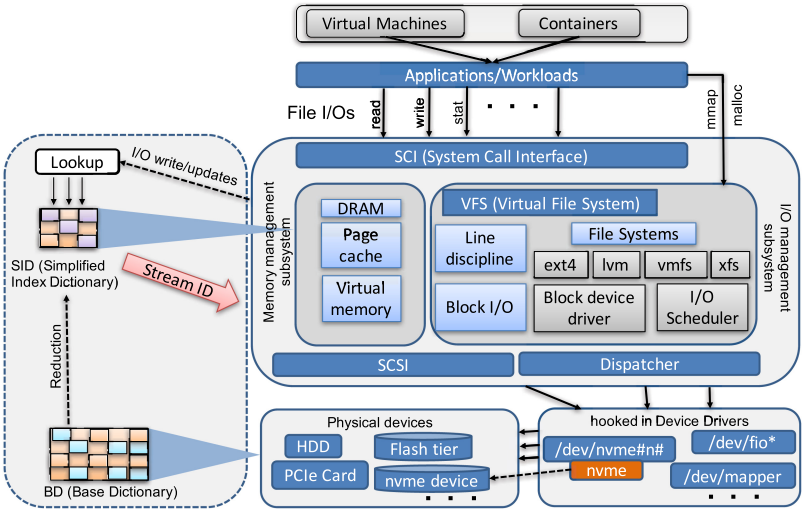


Fig. 8. The I/O stack of FIOS architecture.

obtained streamID is piggybacked on a reserved field of regular and queued write commands as specified in the ATA command set. Along with every write operation, the corresponding streamID is penetrated through all the below layers until that I/O is written or updated on the SSD.

4.2 Basic Data Structures

Base Dictionary: The BD is the result of the training phase (see Figure 3) of our FIOS, which contains the mapping between streamIDs and sector_chunks. Recall that there are 64 sectors in one sector_chunk. Each sector_chunk has a streamID, as metadata associated with it, which is stored in BD on flash memory. Such a metadata uses 0.5 bytes to keep the streamID. In our implementation, we consider a logical volume of flash of a 3 TB, which is stripped over the physical volume of three 1 TB SSDs. Thus, for 3 TB flash volume, we require less than 50 MB in total to store all streamID metadata, which is only 0.001% of total flash disk space.

Simplified Index Dictionary: The SID is used to perform a lookup for streamIDs at run time (i.e., the testing phase). SID is the compressed version of BD, where we reduce the size of base dictionary by combining all consecutive sector_chunks that have the same streamID and replacing multiple lines with one line. Following that, instead of sector_chunk number, we have the range of sector chunks for each row in SID. We can then store SID in DRAM memory for fast `hash_get` lookup. We observe that SID is 50% more efficient in terms of space when compared to BD. Once SID is created and stored, we can move BD to the back-end storage until we have a new dictionary from another round of the training phase. In our experiments, we have 128 GB DRAM in the server. The SID footprint for different workloads consumes less than 25 MB of main memory. Therefore, the SID footprint overhead is only about 0.02% of the size of the main memory.

4.3 Multi-stream SSD Architecture

In an SSD, a single flash internal consists of multiple connected dies through a serial I/O bus and common control signals. Each die has its own chip enable and ready/busy signals. Thus, one of the dies can accept commands and data while the others are carrying out other operations. Furthermore, a die consists of multiple planes, and each plane is framed by multiple blocks containing pages where data is actually stored. Additionally, each plane has some register space to

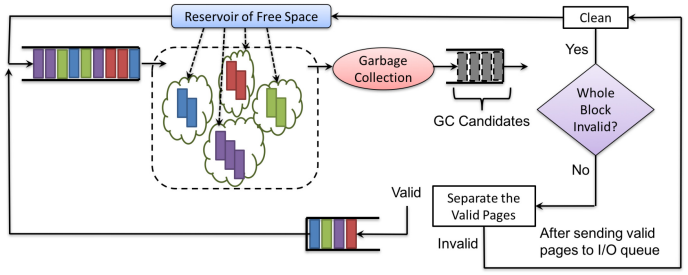


Fig. 9. Modification to garbage collection in order to enable multi-stream.

store data allocation and each block has one page reserved for metadata. We implement our `FIOS` in DiskSim's SSDSim plugin [7], which has been widely used to simulate SSDs. However, DiskSim does not support the multi-streaming technology. Thus, we modified this existing SSD simulator to allow us to simulate multi-stream SSDs and evaluate our new `FIOS` method. The overhead of our implementation will be discussed in Section 7.

In particular, we mainly modify two modules, i.e., `ssd_init` and `ssd_clean`. The `ssd_init` module is modified to maintain a streamID attached to the hierarchy of active_page, active_block, and active_plane. We ensure that for all dies, the total number of active_planes is always equal to the number of streams. Each active_plane has only one active_block and each active_block has only one active_page. Thus, the main difference between a legacy SSD and a multi-stream SSD is that we have multiple simultaneous open erase blocks (one for each stream) in the multi-stream SSD, but only a single open erase block in the legacy SSD.

If all pages in active_block are filled, then `block_alloc_pos` will take care of assigning a new block, which then becomes active_block with the same streamID as the previous active_block. Blocks are not pre-allocated to each stream. Instead, they are allocated on-demand to each stream after the previous one of the same stream is finalized. In Disksim, the `ssd_clean` module performs GC, which is a global component and not aware of the streams. Thus, when garbage collection starts, `ssd_clean` should search all the blocks to find one or multiple candidates to clean. On the other hand, there are still blocks with some valid pages whose streamIDs need to be preserved. The modified `ssd_clean` module of Disksim thus assigns these pages to a new physical plane that is ensured to belong to the preserved streamID. Figure 9 illustrates how the modified DiskSim works for supporting multi-streaming when garbage collection happens.

5 PLATFORM SETUP

Our evaluation environment of Disksim is calibrated based on the real testbed specs, summarized in Table 2. We adopt the similar flash volume structure developed in our previous work [11], which consists of a logical volume of 3 TB with full stripping of 128 KB. We configure the parameter file of DiskSim [7] to support the **On-Stack Replacement** (**OSR**) write policy [30] and the wear-aware garbage collection cleaning policy [27]. We modify the SSDSim module of DiskSim to enable simulating operations of multi-stream SSDs to validate our results. To calibrate DiskSim, we use the prototype of multi-stream SSDs made by using Samsung's NVMe PM953 SSD with M.2 form factor. The Ubuntu kernel patches to use this multi-stream SSD prototype is available at https://github.com/multi-stream/multi-stream. Here, we set 64 sectors on a disk (i.e., 64 × 512 Bytes on SSDs) as one sector_chunk by default for most of our experiments considering the empirically observed sensitivity towards storage overhead of streamID computation.

Table 2. Testbed Specs

CPU Type	Intel(R) Xeon(R) CPU E5-2640 v3
CPU Speed	2.60 GHz
Number of CPU Cores	32 Hyper-Threaded
CPU Cache Size	20480 KB
CPU Memory	128 GB
OS Type	Linux
Kernel Version	4.2.0-37-Generic
Operating System	Ubuntu 16.04 LTS
File System	ext4
Flash Storage	960 GB
Page Size	8 KB
Pages Per Block	64
Blocks Per Plane	351,562
Planes Per Die	8
Dies Per Element	2
Elements Per Gang	1
Flash Over-Provisioning	11%
Metadata Storage Reservation	1 Page Per Block

We conduct a trace-replay simulation in modified DiskSim by using 100+ enterprise workloads from the**University of Massachusetts (UMass)** [4], **Microsoft Research—Cambridge (MSR)** [34], and **Florida International University (FIU)** [5] trace repositories. We use Linux default file system ext4, which is a journaling file system. We understand that using different file systems may yield different performance, but this is outside the scope of this work. We analyze the performance using a wide variety of workloads with different characteristics. Table 3 shows some workload characteristics of the selected workloads. Brief descriptions about workload characteristics shown in Table 3 are as follows,

— *ACF*—the auto-correlation factor of inter-arrival time between two write/update requests;
— *RW*—the percentage of random writes among the total write I/Os;
— *S*—the estimated average size of each write;
— *WV*—the working volume that represents the spatial capacity demand;
— *R*—the reuse ratio of the total amount of data (in bytes) accessed in the disk (i.e., WV) to the total address range (in bytes) of accessed data, which is the unique set of WV. A large R ensures that there is some unique data that is reused heavily;
— λ—the daily logical write rate (GB/day);
— *Peak rate*—the peak throughput demand with the 5 minutes statistical analyses window;
— *Throughput*—the overall operation rate.

The primary goal of multi-streaming technology is to provide better data placement which can reduce the extra writes during garbage collection. Thus, the major evaluation metric is WAF, which is the ratio of physical writes (in bytes) on a device to logical writes (in bytes) by applications running on a host. The lower the WAF is, the better the lifetime and performance of the flash device are.

To evaluate our `OFIOS`, we setup a total of eight parallel docker containers to simultaneously execute four different SQL and NoSQL database applications—MySQL, Cassandra, RocksDB, and MongoDB for 1 hour. We have two containers for each application. Each database application

Table 3. Statistics for Postmark, IOzone and Selected FIU, MSR-Cambridge and UMASS Workloads

Workload	ACF	RW (%)	R	S (KB)	WV (KB)	λ (GB/Day)	Peak rate (IOPS)	Throughput (IOPS)
Postmark	0.99	98.29	37.61	26	10244024	43.87	112.32	18.45
IOzone	0.98	49.45	1.21	197	5094328	28.08	109.53	14.08
FIU-1	0.78	75	157.35	9	1065000000	139.4	271.65	6.6
FIU-2	0.87	82	32.89	8	1084000000	114.72	156.79	1.02
FIU-3	0.97	57	182.04	8	1084000000	185.09	207.02	2
MSR-hm0	0.92	66	4.48	15	28000000	58.51	254.55	9.24
MSR-hm1	0.95	58	157.37	31	51000000	1.57	156.13	6.98
MSR-mds0	0.98	69	32.47	19	67000000	21.04	298.33	23.38
MSR-prn0	0.98	61	4.72	25	132000000	131.33	409.66	17.72
MSR-proj0	1	27	6.27	29	32000000	412.19	484.82	28.95
UMASS-1	0.21	64	1242.93	8	1289000000	575.94	218.59	122.05
UMASS-2	0.02	76	1.13	5	1156000	76.6	159.94	90.25

(ACF - Auto-Correlation of Inter-arrival Time, RW - Random Write, R - Reuse Ratio, S - Estimated Size of each Write, WV - Working Volume Size, λ - Write Rate, BFC - Best Feature Combination).

Table 4. Workload Configurations of Different Real Applications (KV - key/value, col. - columns, conn. - connections, R - reads, W - writes, U - updates, Bench. - benchmark, W1 to W6 - workloads)

Application	Bench.	Size	W1	W2	W3	W4	W5	W6
MySQL-1	TPCC	4200 warehouse	100 conn.	200 conn.	300 conn.	400 conn.	500 conn.	600 conn.
MySQL-2	TPCC	200 warehouse	1000 conn.	2000 conn.	3000 conn.	4000 conn.	5000 conn.	6000 conn.
Cassandra-1	Cassandra stress	10 million records	R/W 70%/30%	R/W 60%/40%	R/W 50%/50%	R/W 40%/60%	R/W 30%/70%	R/W 20%/80%
Cassandra-2	Cassandra stress	1 million records	R/W 80%/20%	R/W 70%/30%	R/W 60%/40%	R/W 50%/50%	R/W 40%/60%	R/W 30%/70%
RocksDB-1	DB_bench	560 million records	U/R 50%/50%	U/R 60%/40%	U/R 70%/30%	U/R 80%/20%	U/R 90%/10%	U/R 100%/0%
RocksDB-2	DB_bench	10 million records	U/R 100%/0%	U/R 90%/10%	U/R 80%/20%	U/R 70%/30%	U/R 60%/40%	U/R 50%/50%
MongoDB-1	YCSB	220 million records	U/R 50%/50%	U/R 60%/40%	U/R 70%/30%	U/R 80%/20%	U/R 90%/10%	U/R 100%/0%
MongoDB-2	YCSB	10 million records	U/R 100%/0%	U/R 90%/10%	U/R 80%/20%	U/R 70%/30%	U/R 60%/40%	U/R 50%/50%

runs 6 different streams of workloads. Each workload runs for 10 minutes. Table 4 shows the configuration of our 8 containers and 6 workloads of each container.

6 EVALUATION

In this section, we first study the impact of streamID identification using different features, such as frequency, sequentiality, on the WAF of multi-stream SSDs, and overall throughput of workloads. Then, we analyze characteristics of different workloads and derive some implications for

determining which feature or a combination of features can obtain the minimum WAF for a given workload. Then, we validate the FIOS automatic feature selection model to determine the best features for streamID identification. We also evaluate the performance impact of FIOS on the overall throughput of applications, and the sensitivity of FIOS with different sector chunk sizes. Finally, we discuss our evaluations of OFIOS, by comparing its performance with the state-of-the-art techniques. We also analyze the sensitivity of OFIOS to its variable parameters.

6.1 Impacts of Workload Features

We inspect the impact of streamID identification using different features on the WAF. The baseline of our comparison is the WAF of no streaming legacy SSDs with the same capacity and configurations. We also compare the results of our feature-based streamID identification with one straightforward way of partitioning data into different streams. In this straightforward approach, the total address space of SSDs is equally partitioned into the number of supported streams, and streamID is assigned according to I/O address.

As discussed in Section 3, our FIOS framework extracts various workload features and considers different feature combinations to cluster logical blocks into streams. Figure 10 shows the results of relative WAFs normalized by the WAF of no streaming legacy SSD for binary feature matrix. Later in Section 3.3, we discuss our advancement regarding the decimal feature matrix. The blue horizontal line at 1 represents the relative WAF of legacy. Thus, the smaller the relative WAF is, the better the endurance of multi-stream SSDs can be obtained. Specifically, 15 bars represent some possible feature combinations used by FIOS. For example, *FSAC* stands for a combination of *Frequency*, *Sequentiality*, *Adjacent access,* and *Coherency*.

We conduct our experiments with a totally of 100+ workloads and summarize our observations with 12 representative workloads in Figure 10. Overall, we observe that the multi-streaming technology offers a good opportunity to reduce WAF, e.g., at least a 20% reduction in WAF for all workloads. For some write-intensive workloads e.g., Postmark (see Figure 10(a)) that have the majority of random writes, the WAF can be reduced by more than 70%. On the other hand, we also notice that none of these features can be always the best for different types of applications. For example, *Coherency* gives the best WAF reduction for MSR-hm0 (see Figure 10(f)), while a combination of ***Frequency* and *Coherency* (*FC*)** is much better for MSR-hm1 (see Figure 10(g)). This is because different workloads have different characteristics. For example, by looking closely to the workloads we find that both MSR-hm0 and MSR-hm1 have more random writes and a high auto-correlation (i.e., ACF) of update time, see Table 3. As a result, the feature of coherency that groups randomly occurred friendly sector_chunks into the same stream becomes a good choice. Additionally, MSR-hm1 has a high reuse ratio of updates. Thus, combining features of frequency can capture the multi-touch reuse count. Moreover, from Figure 10(g), we also find that further adding other features (e.g., sequentiality), FSC does not further reduce the WAF, which implies that the combination of *more features does not guarantee a better reduction in WAF*.

Furthermore, Figure 10 shows the results of relative WAFs normalized by the WAF of no streaming legacy SSD using the decimal feature matrix when compared with the binary feature matrix. Overall, we observe that for different workloads, the WAF is reduced by using the decimal feature matrix when compared to the binary feature matrix (see Figure 10). This is because, with the decimal feature matrix, we can capture granular properties to cluster data more accurately into streams. Furthermore, we also notice that the best feature combination with which we obtain the maximum WAF reduction for a workload remains the same irrespective of the granularity of the feature matrix. For example, *Coherency* gives the best WAF reduction for MSR-hm0 with both the binary and decimal feature matrices. This validates that the best feature combination is

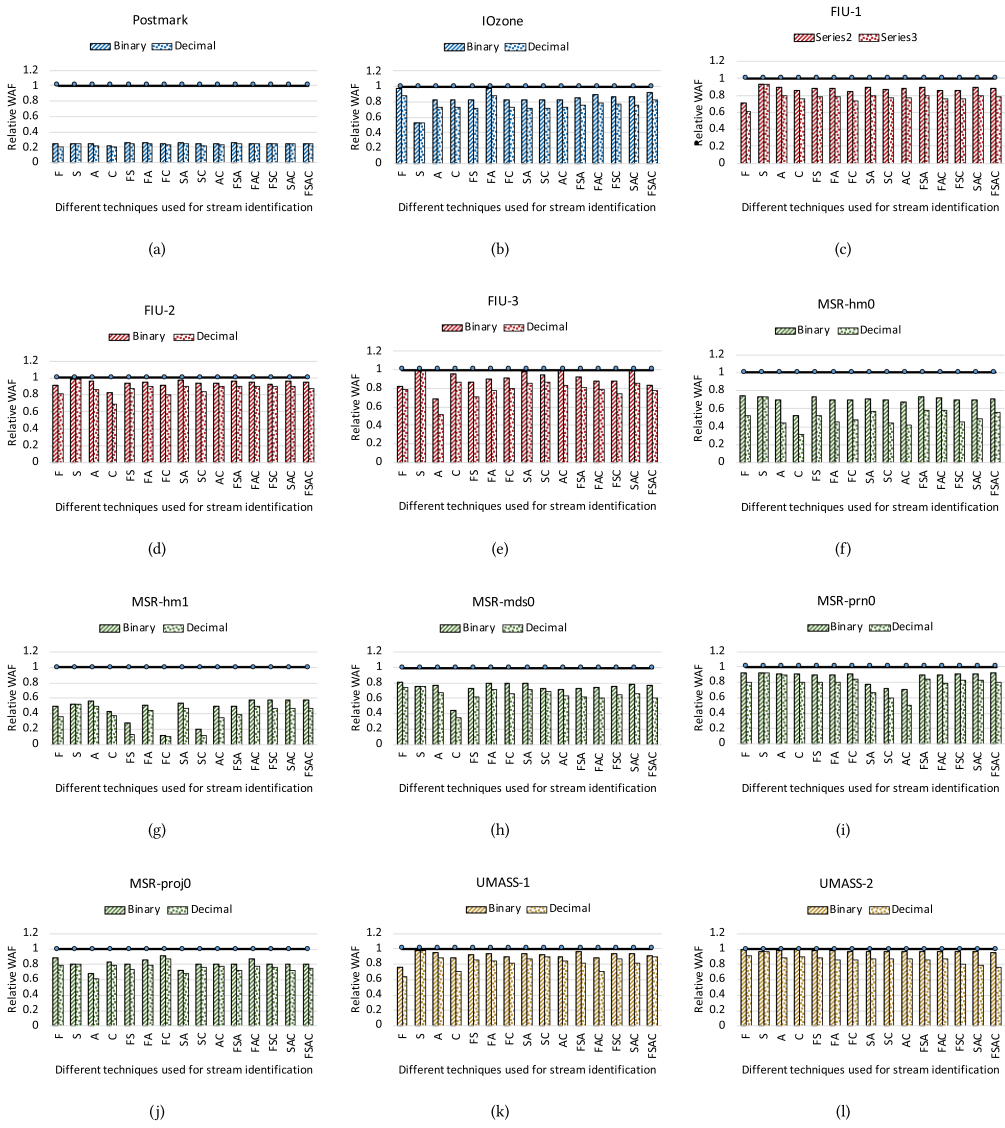


Fig. 10. Relative WAF w.r.t. a legacy device for Postmark, IOzone and selected FIU, MSR-Cambridge, and UMASS workloads by using our framework with dictionary framed from different features and their combinations for streamID identification. (F - Frequency, S - Sequentiality, A - Adjacent access).

closely related to workload characteristics, and we can accurately predict the best feature combination by understanding the relation between features and workload characteristics.

6.2 Validation of Feature Selection by FIOS

Now, we turn to validate our approach for a given workload choosing the best combination of features under various workloads. We use WAF as the metric to evaluate our FIOS under six different I/O workloads, as shown in Figure 11. The best feature combination derived by FIOS are also mentioned on the top of the FIOS bars in Figure 11. For comparison, we plot the results of

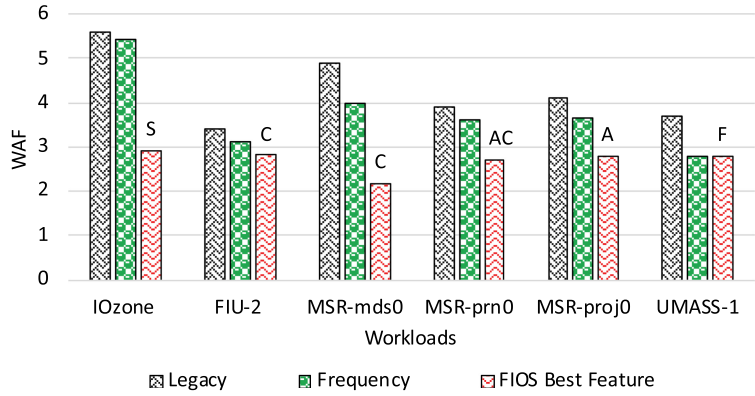


Fig. 11. Write amplification factor using the best feature selected using FIOS when compared to legacy no streaming, streamID identification with equal partition, and with a usually used feature of frequency. The best feature selected by FIOS is mentioned on top of the bar for each workload.

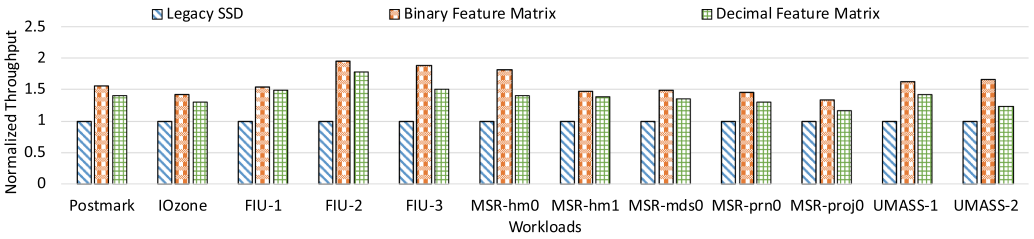


Fig. 12. Throughput of workloads using the binary feature matrix and the decimal feature matrix for normalized to throughput obtained using non-streaming legacy SSD.

legacy and multi-stream SSDs under the fixed feature (e.g., frequency) selection approaches. These results imply that the existing methods that use the fixed features (such as frequency) for streamID assignment cannot effectively utilize the benefits of multi-stream SSDs. In contrast, for all these workloads FIOS is able to obtain the lowest WAF by using appropriate features for streamID assignment.

6.3 Performance and Sensitivity of FIOS

Furthermore, for the best feature combination, we compare the workload throughput of multi-stream SSDs with non-streaming SSDs, while using the binary feature matrix and the decimal feature matrix in Figure 12. We observe that by using a multi-stream SSD with accurate data streams, the throughput of workloads is improved by at least 1.17 times when compared with a legacy SSD with no streaming capability. This is because by better organizing the data within SSDs, the overhead for internal operations such as GC is reduced, and hence more device bandwidth becomes available for the application data. We also see that the throughput of using the binary feature matrix is better when compared to the decimal feature matrix. This is because computing the decimal feature matrix for each feature requires more time, as it is more complicated than the binary feature matrix. Thus, using the decimal feature matrix instead gives better endurance, but consumes some additional performance overhead. However, the overall throughput using the decimal feature matrix still remains better when compared to the legacy SSD. *In summary, our results show*

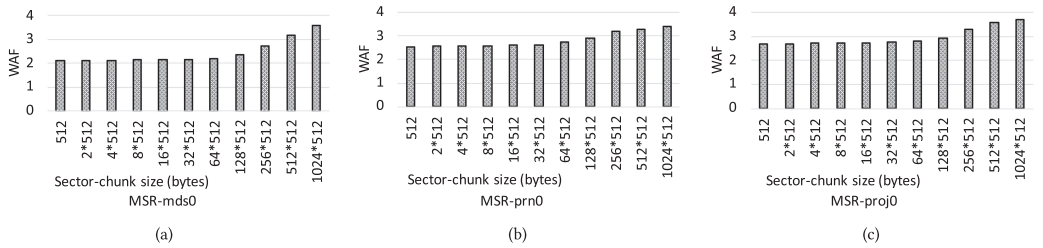


Fig. 13. Sensitivity analysis by changing sector chunk size (a) MSR-mds0, (b) MSR-prn0, and (c) MSR-proj0.

that by allowing us to better organize the data within multi-stream SSDs, we can reduce WAF as well as improve throughput.

Finally, in Figure 13, we observe the sensitivity of FIOS with different sector chunk sizes. We see that with a smaller sector chunk size, better (i.e., lower) WAF can be obtained. However, the rate of the increase in WAF with the increase in the size of sector chunk is initially very low, but WAF increases rapidly for sector chunks with larger sizes. Very small size of sector chunks (e.g., 512 bytes) will lead to high management overhead in terms of memory required to store dictionaries as well as CPU cycles required to compute for many chunks. While, very large sector chunk size (e.g., 1024 × 512 bytes) reduces the WAF reduction benefit of FIOS. Therefore, after conducting such empirical analysis on many different workloads, we chose to set chunk size as 64 × 512 bytes because for most of the workloads, we observed this is a knee point until which the rate of increase of WAF with the increase in the sector chunk size is low and after that, it increases rapidly (see Figure 13).

6.4 OFIOS Results

To evaluate our OFIOS, we setup a total of eight parallel docker containers to simultaneously execute four different SQL and NoSQL database applications as described earlier in Section 5. We empirically configure $T_{RunWindow}$ and $T_{TrainWindow}$ to have an equal length of 100 seconds for optimally capturing the dynamics of workloads. However, to reduce the overhead, depending upon the number of cores in the system, the length of the $T_{TrainWindow}$ can be reduced in OFIOS. The length of the $T_{TraceWindow}$ needs to be less than the length of $T_{TrainWindow}$. Empirically observing the time taken to construct new dictionaries by FIOS, we set $T_{TraceWindow}$ as 75% of $T_{TrainWindow}$. Later, in Section 7 we will further explore the time consumed by FIOS to construct new dictionaries.

Figure 14 shows the runtime WAF under both FIOS and OFIOS, where the green line with a circle marker shows the WAF observed using FIOS with Frequency as a feature and the red line with a triangle marker shows the WAF observed using OFIOS. We can see that the overall WAF can be further reduced by using OFIOS. As we know, FIOS cannot dynamically adapt the dictionary during the runtime. For example, after the initial training in the first 100 seconds, Frequency is identified as the best feature and then used for the streamID assignment throughout the entire execution time by FIOS. Whereas, OFIOS automatically adapts the dictionary during the runtime of the workload. Thus, according to the workload, the best feature combination to assign streamIDs is periodically adapted. In Figure 14, the feature combinations of OFIOS are shown along the x-axis for every 100 seconds. We see that the adaptive OFIOS can further improve the endurance of SSDs.

Further, in Figure 15, we compare the performance of three existing techniques with OFIOS. Figure 15(a) shows that the WAF of our scheme is reduced by 47.1% and 2.6% compared with baseline and LSTM+KM [40], respectively. OFIOS also results in lower average WAF compared to the other two techniques of multi-queue using AutoStream [39] and vStream [42]. All these three

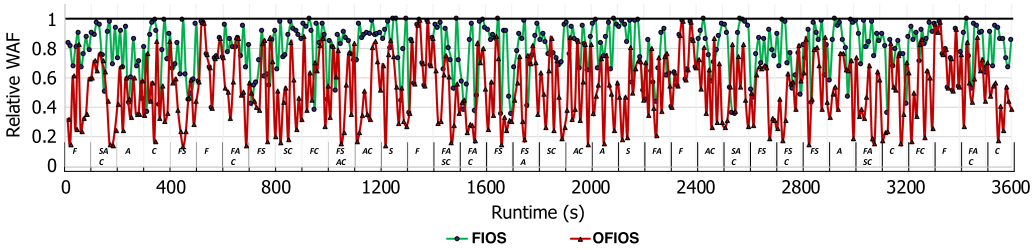


Fig. 14. Reduction in relative WAF w.r.t. legacy device using OFIOS compared to FIOS .

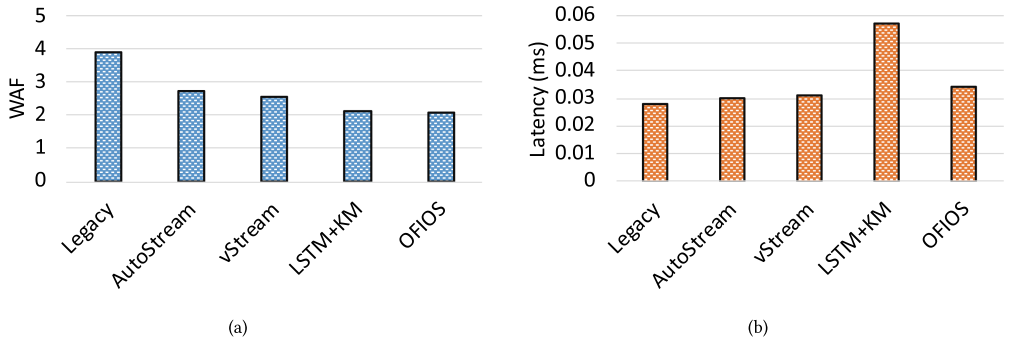


Fig. 15. Comparison with state-of-the-art techniques for: (a) Write Amplification Factor, and (b) Average I/O Latency.

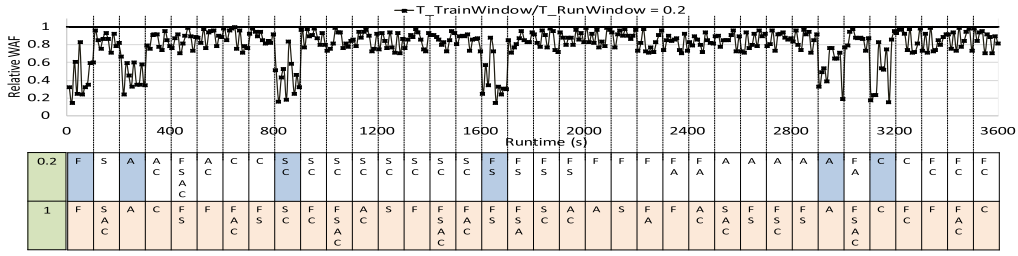
previous techniques uses update frequency as its feature. We implement a multi-queue technique proposed in AutoStream [39] that uses multiple software queues to maintain data in sorted order w.r.t. its update frequency. The multi-queue technique implements promote and demote operations within these queues to track the changes in the workloads. vStream [42] identifies virtual streams by post processing the dead pages, and then uses k-means clustering to group any number of virtual streams to an available number of physical streams. LSTM+KM [40] predicts the future frequency using a neural network called **Long Short-Term Memory** (**LSTM**) and then uses k-means clustering to group writes into an available number of physical streams.

For an efficient streamID assignment, it is very important that it runs with minimum critical I/O path latency. Usually millions of I/Os are performed every second on high performance storage server, so even a small additional latency to each I/O can result in very drastic application throughput degradation. Thus, apart from just evaluating the reduction in WAF, we also evaluate and compare the additional I/O latency overhead in Figure 15(b). Figure 15(b) shows the average combined read/write latency while running the above eight simultaneous application containers. We see that OFIOS introduces very minimal additional latency compared to legacy (non-streaming) SSD. Other techniques of AutoStream [39] and vStream [42] also introduce small additional latencies due to their simplistic algorithms. However, their WAF benefit is not as good as that of OFIOS (see Figure 15(a)). LSTM+KM [40] introduces comparatively the highest additional latency due to its use of the neural network. LSTM neural network require a lot of resources and time to get trained and become ready to provide accurate prediction. Moreover, at runtime recurrent training is required to adopt to the changing workloads. Thus, LSTMs can yield high prediction accuracy, but become quite inefficient for latency sensitive applications. This is also acknowledged by the authors of LSTM+KM in their article. With reduced GC overhead in OFIOS, more device

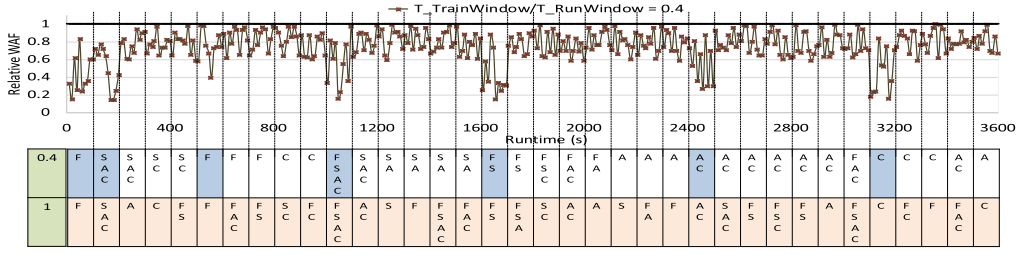
bandwidth is released to serve both read and write I/Os. So, the additional performance overhead of our framework is mostly compensated with such performance improvement, resulting into a minimal overhead. Finally, the above results indicate that our scheme can obviously reduce the WAF by introducing very low additional read/write latency to capture the impact of dynamically changing parallel applications. Additionally, notice that OFIOS performs well even while using append-only applications such as RocksDB, because the application's append-only pages that stores the always increasing log is a notion of virtual addressing which is different from the physical pages within SSDs. Thus, although the application's append-only pages (e.g., SST files) keep increasing instead of overwriting, as a fixed size SSD can only have fixed number of LBAs, so the same LBAs needs to be reused. To perform good stream identification, OFIOS do not need to relate the lifetime of the application's append-only pages with the LBAs. Rather, the feature that we discuss in our work are directly captured at the block layer.

Finally, we analyze the sensitivity of OFIOS to the duration of $T_{RunWindow}$ and $T_{TrainWindow}$. Configuring $T_{TrainWindow}$ less than the duration of $T_{RunWindow}$ ensures that the additional resources consumed by background operations to train our model are only occupied during the fraction of the runtime window. Note that configuring $T_{RunWindow}$ smaller than $T_{TrainWindow}$ just leads to additional overhead without any benefits. Thus, we conducted experiments with the training for a different proportion of the run window. Particularly, we train for the last "n" portion of each run window. We set "n" as 20%, 40%, 60%, 80% and compare the results with n = 100% (i.e., $T_{TrainWindow} = T_{RunWindow}$). n = 100% means that we always train in the background. The results in Figure 14 show the WAF of OFIOS with n = 100%. The length of the training window that is equal to that of the run window (i.e., $T_{TrainWindow}/T_{RunWindow} = 1$) signifies that we capture the activities of the workloads in the background to dynamically train OFIOS to adapt quickly to workload changes. Decreasing the ratio $T_{TrainWindow}/T_{RunWindow}$ means we train only for the portion of each run window. For example, $T_{TrainWindow}/T_{RunWindow} = 0.2$ means we train for last 20% of each run window. Then, based upon the observations in this last 20% of the run window, we decide the feature to use in the next time window.

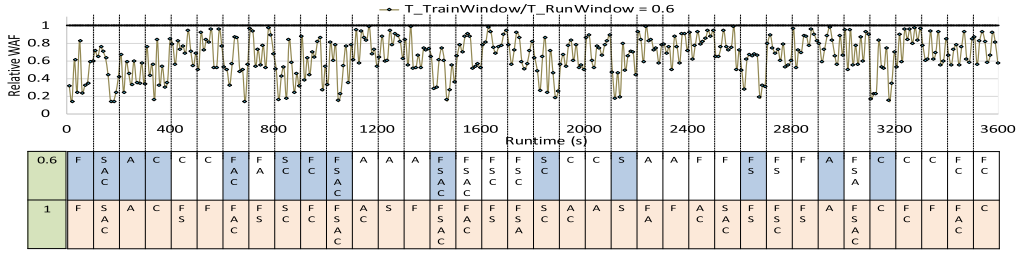
Figure 16 shows the runtime WAF for four different settings of the training window time (i.e., $T_{TrainWindow}/T_{RunWindow}$ = 0.2, 0.4, 0.6, and 0.8). Figure 16 also shows the feature combinations used by OFIOS to identify streams under each of these settings while executing multiple parallel containerized workloads. Figure 17 shows the average WAF for the same experiments discussed in Figure 16. Figures 16 and 17 show that higher WAF reduction can be achieved with a larger training window as feature identification is done better according to the dynamics of the workloads. Upon in-depth analysis of feature combinations used by OFIOS during run time for different intermediate time intervals, we see that the larger the training window, more number of windows used same as that of features used with $T_{TrainWindow}/T_{RunWindow} = 1$. The blue shaded features represent that for those time windows, the same feature as that of $T_{TrainWindow}/T_{RunWindow} = 1$ is used. More number of blue-shaded time windows represent that OFIOS can perform well using more appropriate feature combinations resulting in lower WAF. OFIOS will not work well if the LBAs are managed in an append-only fashion. This is the limitation of our proposed technique. However, we would like to emphasize that as the file system organizes and maintains the mapping of application data into logical block addresses. Hence, OFIOS works fine with append-only applications if the underlying file system is not append-only. This can be observed from our results in this section in which we validate OFIOS using append-only applications such as RocksDB. We use Linux default file system ext4, which is a journaling file system. We understand that using append-only file systems such as **log-structured file system (LFS)** may raise new challenges. We will pay attention to design new methods for dealing with append-only file systems in our future study.



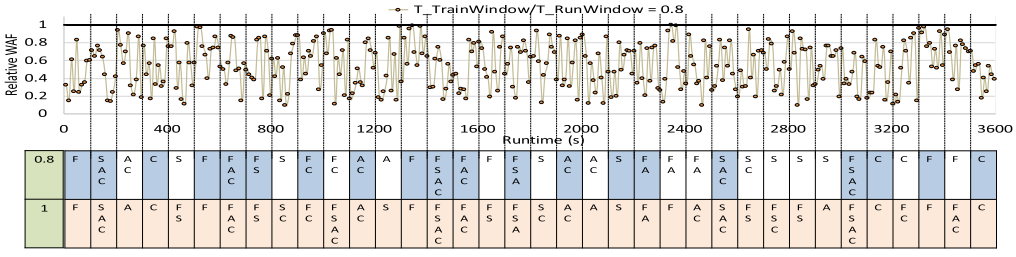
(a)



(b)



(c)



(d)

Fig. 16. Relative WAF w.r.t. legacy device, training and running for different size of time windows (i.e., $T_{TrainWindow}/T_{RunWindow}$ and the features used by OFIOS executing dynamically changing workload for different size of time windows (i.e., $T_{TrainWindow}/T_{RunWindow}$: (a) $T_{TrainWindow}/T_{RunWindow}$ = 0.2, (b) $T_{TrainWindow}/T_{RunWindow}$ = 0.4, (c) $T_{TrainWindow}/T_{RunWindow}$ = 0.6, and (d) $T_{TrainWindow}/T_{RunWindow}$ = 0.8

7 DISCUSSION

7.1 Implementation Layer

The identification of streamIDs can be done at any layer, such as the application layer, the file system layer, the block layer, or the FTL layer. Stream identification at the application layer is to let

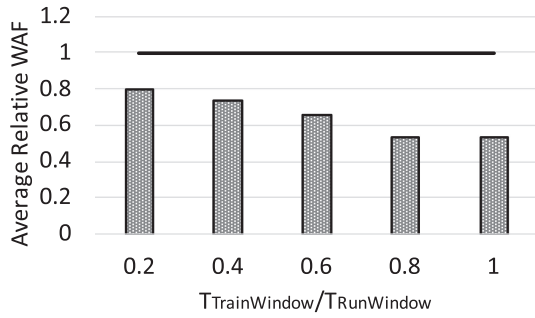


Fig. 17. Average relative WAF w.r.t. legacy device, training and running for different size of time windows (i.e., $T_{TrainWindow}/T_{RunWindow}$.

applications be responsible to identify different streams (for e.g., commit logs, metadata, indices, and so on.) in their workloads. However, when multiple applications are running simultaneously on the same host, stream management becomes more complex. Another option is to perform stream identification at the file system layer that appends each file with a corresponding streamID. This is portable to different applications without modifying them. But, this approach sometimes does not work when some applications bypass the file system to speed up data access. In addition, as the upper layer application changes, the file system may not be able to dynamically adjust the streamID assignment. On the other hand, streamID identification at the FTL layer is limited by the computing and buffering capability of an SSD that is comparatively slower and smaller when compared to the computing speed and buffering capability of the host (i.e., CPU and main memory). We find that the block layer implementation can avoid the above issues, i.e., being portable to different applications and multi-tenant environments and being able to utilize the host resources for stream management. Thus, we choose to implement our stream identification framework at the block layer in this article.

7.2 Implementation Overhead

We use BD and SID, as described in Section 4, to maintain additional data of our framework. For a 3 TB flash volume SSD storage, we require less than 50 MB in total to store all streamID metadata, which is only 0.001% of the total flash disk space. In our experiments, we have 128 GB DRAM in our server. The SID footprint for different workloads consumes less than 25 MB of main memory. Therefore, the SID footprint overhead is only about 0.02% of the size of the main memory. We mainly modified two modules, i.e., `ssd_init` and `ssd_clean` in DiskSim. In order to pertain the modification throughout, we had to make subsequent changes in the `ssd_timing`, `ssd_gang`, and `ssd_utils` modules. In total, we modified approximately 560 lines of codes to enable the operation of a general multi-stream SSD excludeing streamID identification.

The streamID identification time under `FIOS` is considered as the total additional operation time, including the time required for feature extraction, K-means clustering, construction of aBD and SID. Figure 18 shows the average time for streamID identification by `FIOS` using four features (i.e., frequency, adjacent access, sequentiality, and coherency) with respect to a number of blocks in the workload. We can see that `FIOS` does add extra latency in order to identify the best feature combinations for improving SSD endurance. With an increase in dataset size (i.e., number of blocks), the time to identify and maintain streamIDs increases as well. In our experiment, a 1.5 TB SSD was required to run a workload with 3 billion LBAs (blocks). Thus, we believe it is acceptable for `FIOS` to take around 600 seconds for streamID identification. `FIOS` is greatly advantageous to be used for all the applications and workloads that are not very latency sensitive such as email-server, but even

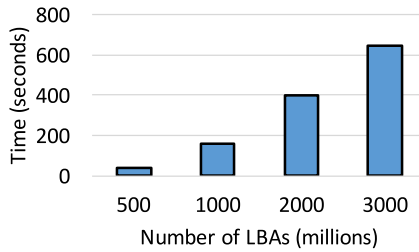


Fig. 18. Average time for streamID identification using four features (frequency, adjacent access, sequentiality, and coherency) and the binary feature matrix as a function of the number of blocks in the workload.

for latency-sensitive applications (such as online marketplace) the additional delay introduced is manageable. The OFIOS simultaneously queries the streamID using SID, while working to identify data streams in the background. As DRAM access latency is uniform and very low, so we notice that on an average each query takes around 20 ns. Thus, collectively querying for 3 billion LBAs only consumes 6μs additional to the time taken by background operations to identify streams. Finally, we note that there exists a trade-off between time efficiency and endurance improvement. To take advantage of OFIOS, configuring appropriate window size is necessary considering the abovementioned overheads and the estimated blocks accessed per second by the workloads. Since a multi-stream SSD is developed to enhance the lifetime of SSDs, we believe that the main goal of a streamID identification algorithm should be to reduce write amplifications with the minimal time overhead.

8 RELATED WORK

The high performance (compared to HDDs) and recently reduced cost (e.g., $/GB) of SSDs make them perfect persistent storage devices. However, currently, the major issue with the flash technology is the limited lifetime of flash cells due to limited PE cycles. In order to balance the wear out of flash cells in the device and enable them to use it as plug and play, efficient FTL algorithms [8, 14, 24, 28, 29, 36, 43] have been developed. The FTL plays an important role in filtering out sequential streams from the mixed (sequential + random) streams. The work in [7] observed that SSD's performance and lifetime are highly sensitive to I/O workloads. The previous work in [32] designed new file systems, specifically for enhancing the lifetime of the SSDs. These file systems concentrated on transforming all random writes at the file system level to sequential ones at the SSD level and considered a new data grouping strategy on writing by putting data blocks with similar update likelihood into the same segment. It was noticed that the reduction of the internal write amplification of SSDs might increase the lifetime of SSDs. Thus the research work started exploring all features that are related to the write amplification factor. Reference [16] proved that the write amplification factor of SSDs depends on over-provisioning, cleaning policy, and workload pattern. Based on the measurement on NVMe disks, the study in [41] further revealed the relationship between write amplification factor and the write I/O sequential ratio.

The recent technology evolves a new product of multi-stream SSDs, which offers an intuitive storage interface to inform the host system about the expected lifetime of the data [1, 18, 20, 21, 23, 35]. Experimentation in [23] uses a real multi-stream SSD prototype to show that the worst-case update throughput of a Cassandra NoSQL database system can be improved by nearly 56%. The work in [15, 23, 38] did modification in certain applications (i.e., FIO [6], Cassandra [31], and RocksDB [19]) to enable application assigned streamID. The study shows that with multi-streaming, SSDs can be more efficiently used for achieving consistently better performance and endurance. These existing techniques to assign streamIDs for a multi-stream SSD are too

specific (such as application assigned streamIDs which requires to modify the application). Recently, LSTM+KM [40] proposes a scheme to place data according to its predicted future temperature using a neural network called LSTM is in temporal and spatial dimensions. Ref. [42] proposes a new concept of **virtual streams** (**vStreams**) for the multi-streamed SSDs. vStreams provide application developers to a large number of virtual streams regardless of the number of physical streams supported by the device. However, the mapping of these virtual streams to the physical streams needs to be managed by applications. In [25, 26], stream management technique called PCStream the stream allocation decisions are also made at a higher abstraction level. Thus, PCStream does not support widely used applications like MySQL [17] that rely on a write buffer. Moreover, applications MonetDB [22] that heavily access files with mmap related functions such as mmap() and msync() are not supported. As a result, in our research, we present a more portable and adaptable method with minimalist overhead for feature-based streamID assignment in multi-stream SSDs at the LBA level. Additionally, the innovation of our work lies in the scalable nature of our method, which considers multiple I/O features simultaneously to group data into any number of streams required by the SSD firmware.

9 CONCLUSION

In this article, we first presented the difference between multi-stream storage and legacy storage and explored the benefits of enhancing SSDs with multi-stream, i.e., increasing the lifetime and performance of SSDs. We then investigated the impact of different workload features on write amplification to enable better utilization of multi-stream SSDs. We proposed a feature-based streamID assignment technique (`FIOS`), which is capable of extracting features and assigning streamIDs with low overhead. `FIOS` is also scalable to incorporate different numbers of features and construct multiple streams to support the framework. To the best of our knowledge, this is the first streamID assignment technique for multi-stream SSDs, which does not require any modification to applications for using multi-stream SSDs. We also proposed a new feature to better capture a lifetime, called coherency, which represents the friendship between write operations with respect to their update time. We further investigated the correlation between workload attributes and workload features to automatically determine a combination of features that can offer the most WAF reduction. Finally, we designed the `OFIOS` technique to learn along with the changing-workload in the background and automatically adapt the best feature combination to assign streamIDs. Our experimental results show that `FIOS` achieves at least a 20% decrease in write amplification across different workloads, which is a good qualitative improvement. We also found that different features have varying impacts on WAF, which indicates the importance of identifying a good combination of features. Our evaluation also shows that our automation approach `OFIOS` of stream detection can effectively further reduce the WAF by dynamically changing the feature combination w.r.t. changes in workloads. In our future work, we plan to improve our methods to deal with the append-only file systems and explore different machine learning techniques to learn the optimal value for the parameters such as window size of background training, trace collection time, and sampling size.

ACKNOWLEDGMENTS

This work was initiated during Janki Bhimani's internship at Samsung Semiconductor Inc. [12].

REFERENCES

[1] Multi-Stream Technology. 2020. Retrieved 15 March, 2020 from http://www.samsung.com/semiconductor/insights/article/25465/multistream.

[2] Performance and Endurance Enhancements with Multi-stream SSDs on Apache Cassandra. 2020. Retrieved 27 Jan., 2020 from https://www.samsung.com/semiconductor/global.semi.static/Multi-stream_Cassandra_Whitepaper_Final-0.pdf.

[3] systemd. 2020. Retrieved 18 Dec., 2020 from http://manpages.ubuntu.com/manpages/bionic/man1/systemd.1.html.
[4] UMass Trace Repository. 2020. Retrieved 18 Dec., 2020 from http://traces.cs.umass.edu/index.php/Storage/Storage.
[5] (accessed January 13, 2017). *SNIA Iotta Repository*. Retrieved from http://iotta.snia.org/historical_section.
[6] (accessed Septenber 7, 2016). *FIO - flexible I/O benchmark*. Retrieved from http://linux.die.net/man/1/fio.
[7] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference*. 57–70.
[8] Jinwook Bae, Hanbyeol Kim, Junsu Im, and Sungjin Lee. 2019. Demand-based FTL cache partitioning for large capacity SSDs. *IEMEK Journal of Embedded Systems and Applications* 14, 2 (2019), 71–78.
[9] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In *Proceedings of the Noise Reduction in Speech Processing*. Springer, 1–4.
[10] Janki Bhimani, Miriam Leeser, and Ningfang Mi. 2015. Accelerating k-means clustering with parallel implementations and GPU computing. In *Proceedings of the High Performance Extreme Computing Conference*. IEEE, 1–6.
[11] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R Pandurangan, and V. Balakrishnan. 2016. Understanding performance of I/O intensive containerized applications for NVMe SSDs. In *Proceedings of the 35th IEEE International Performance Computing and Communications Conference*. IEEE.
[12] Janki S. Bhimani, Jingpei Yang, Changho Choi, and Jianjian Huo. 2017. Smart I/O stream detection based on multiple attributes. (March 16 2017). US Patent App. 15/344,422.
[13] Daniel Campello, Hector Lopez, Ricardo Koller, Raju Rangaswami, and Luis Useche. 2015. Non-blocking writes to files. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. 151–165.
[14] Hyunjin Cho, Dongkun Shin, and Young Ik Eom. 2009. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. European Design and Automation Association, 507–512.
[15] Changho Choi. Multi-Stream Write SSD: Increasing SSD Performance and Lifetime with Multi-Stream Write Technology. 2020. Retrieved 18 Dec., 2020 from http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160809_FC12_Choi.pdf.
[16] Peter Desnoyers. 2012. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 12.
[17] Paul DuBois and Michael Foreword By-Widenius. 1999. *MySQL*. New riders publishing.
[18] Stephen G Fischer, Changho Choi, Jason Martineau, and Rajinikanth Pandurangan. 2018. Methods for multi-stream garbage collection. (October 25 2018). US Patent App. 15/821,708.
[19] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. 2017. NoSQL database systems: A survey and decision guidance. *Computer Science-Research and Development* 32, 3–4 (2017), 353–365.
[20] HUEN Hingkwan and Changho Choi. 2019. Method of consolidate data streams for multi-stream enabled ssds. (May 2 2019). US Patent App. 16/219,936.
[21] HUEN Hingkwan, Changho Choi, Derrick Tseng, and Jianjian Huo. 2020. Multi-stream SSD QoS management. (March 17 2020). US Patent 10,592,171.
[22] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. 2012. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.
[23] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX Association, Philadelphia, PA. Retrieved from https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang.
[24] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. 2002. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics* 48, 2 (2002), 366–375.
[25] Taejin Kim, Sangwook Shane Hahn, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2018. PCStream: Automatic stream allocation using program contexts. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems*.
[26] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2019. Fully automatic stream management for multi-streamed ssds using program contexts. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. 295–308.
[27] Andrey V. Kuzmin and James G. Wayda. 2016. Multi-array operation support and related devices, systems and software. (January 5 2016). US Patent 9,229,854.
[28] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review* 42, 6 (2008), 36–42.
[29] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems* 6, 3 (2007), 18.

[30] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. 2015. WARM: Improving NAND flash memory lifetime with write-hotness aware retention management. In *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies*. IEEE, 1–14.

[31] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2014. CaSSanDra: An SSD boosted key-value store. In *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering*. IEEE, 1162–1167.

[32] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*. 12.

[33] Mahdi Pakdaman Naeini and Gregory F. Cooper. 2018. Binary classifier calibration using an ensemble of piecewise linear regression models. *Knowledge and Information Systems* 54, 1 (2018), 151–170.

[34] D. Narayanan, A. Donnelly, and A. Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage* 4, 3 (2008), 10:1–10:23.

[35] Hyun-Woo Park, Soyee Choi, Mijin An, and Sang-Won Lee. 2019. Freezing frozen pages with multi-stream SSDs. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–3.

[36] Wei Xie, Yong Chen, and Philip C. Roth. 2016. Parallel-DFTL: A flash translation layer that exploits internal parallelism in solid state drives. In *Proceedings of the 2016 IEEE International Conference on Networking, Architecture and Storage*. IEEE, 1–10.

[37] Fei Yang, Kun Dou, Siyu Chen, Mengwei Hou, Jeong-Uk Kang, and Sangyeun Cho. 2015. Optimizing nosql db on flash: A case study of rocksdb. In *Proceedings of the 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops*. IEEE, 1062–1069.

[38] Fei Yang, Kun Dou, Siyu Chen, Jeong-Uk Kang, and Sangyeun Cho. 2015. Multi-streaming RocksDB. In *Proceedings of the Non-Volatile Memories Workshop*.

[39] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: Automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference*. 1–11.

[40] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. 2019. Reducing garbage collection overhead in {SSD} based on workload prediction. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems*.

[41] Zhengyu Yang, Manu Awasthi, Mrinmoy Ghosh, and Ningfang Mi. 2016. A fresh perspective on total cost of ownership models for flash storage. In *Proceedings of the 2016 IEEE 8th International Conference on Cloud Computing Technology and Science*. IEEE.

[42] Hwanjin Yong, Kisik Jeong, Joonwon Lee, and Jin-Soo Kim. 2018. vStream: Virtual stream management for multi-streamed SSDs. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems*.

[43] Peiyong Zhang and Huanjie Tang. 2020. High-efficient superblock flash translation layer for NAND flash controller. *Electronics Letters* 56, 6 (2020), 278–280.

Received July 2020; revised March 2021; accepted June 2021