Automatic I/O stream management based on file characteristics

Yuqi Zhang
Samsung R&D Institute China
Xi'an, Samsung Electronics
Xi'an, China
yuqi1.zhang@samsung.com

Ni Xue
Samsung R&D Institute China
Xi'an, Samsung Electronics
Xi'an, China
ni66.xue@samsung.com

Yangxu Zhou
Samsung R&D Institute China
Xi'an, Samsung Electronics
Xi'an, China
yangxu.zhou@samsung.com

ABSTRACT

Data fragmentation exists widely in SSD NAND flash, which greatly increases garbage collection overhead. Dividing data of different lifetimes into different I/O streams can effectively solve the problem of data fragmentation. Therefore, the management of I/O streams is the key to reduce data fragmentation and thereby improve the lifetime and performance of SSD. In this paper, we propose FileStream, a file-based automatic stream management scheme with machine learning, which can be directly applied to multiple workloads. We find that file characteristics are effective in distinguishing data lifetimes, and thus FileStream divides data into different I/O streams based on the file characteristics. The results of six benchmarks on real SSD show that FileStream reduces the write amplification factor by 34.5% and 21.6% on average, compared to the baseline and two advanced automatic stream management methods, respectively.

CCS CONCEPTS

• **Hardware** → *External storage*; • **Computer systems organization** → Secondary storage organization.

KEYWORDS

Data Placement, SSD, Multi-stream

ACM Reference Format:
Yuqi Zhang, Ni Xue, and Yangxu Zhou. 2021. Automatic I/O stream management based on file characteristics. In *13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21), July 27–28, 2021, Virtual, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3465332.3470879

HotStorage '21, July 27–28, 2021, Virtual, USA

ACM ISBN 978-1-4503-8550-3/21/07…$15.00
https://doi.org/10.1145/3465332.3470879

1 INTRODUCTION

Because of high performance and low power consumption, NAND flash based solid-state drives (SSDs) are widely used in data centers. NAND flash does not support in-place update. To reclaim free space, garbage collection (GC) copies the valid data from a victim block to an available block and then erases the victim block. However, the copy of valid data increases the write amplification factor (WAF), which is defined as the ratio of the amount of actual data written to the flash to the amount of data written by the host. High WAF implies that many valid pages are copied during GC, which greatly shortens the SSD's lifetime and degrades its overall performance.

Data fragmentation is an important cause of write amplification. When GC selects a block with data of different lifetimes, some data are likely to be valid and need to be copied, thus resulting in write amplification. Multi-stream technology [11] is a popular mechanism that focuses on data placement to reduce data fragmentation and WAF. It allows the host to pass a hint (stream ID) with the write request, and data with different stream IDs are placed on the SSD separately, i.e., placing them in different blocks. When the host assigns stream IDs to data according to their lifetimes, the data with different lifetimes are separated, which helps to reduce the amount of valid data copied during GC. Due to its advantages, NVM Express (NVMe) specification [6] added multi-stream technology, and Linux kernel 4.13 added write lifetime hints to carry stream IDs. Although multi-stream technology is introduced by multi-streamed SSDs, its idea is also applied to other devices (such as OpenChannel SSD [8] and Zoned Namespace SSD [4]).

There are two main ideas in prior works using multi-stream technology: 1) Manual stream assignment [11, 13, 15]. Programmers fix an assignment strategy in the code of an application to assign streams to files based on the information of its file type and internal mechanism. 2) Automatic stream management [3, 12, 16, 17]. Stream assignment is automatically managed without relying on specific applications. Specifically, AutoStream [16], FIOS [3], and LKStream [17] mainly use the access characteristics of logical block addresses (LBAs) to estimate data lifetimes. However, these LBA-based

methods mainly apply to in-place update workloads and perform poorly on append-only workloads [12]. PCStream [12] estimates data lifetimes with program contexts to assign streams, which reduces GC overhead as much as manual stream assignment does. However, in order to track program contexts, PCStream still needs to modify the code of some applications (such as MySQL [14], PostgreSQL [9], and Java Virtual Machine). In addition, vStream [18] automatically aggregates virtual streams assigned by the host or application to a limited number of physical streams on the SSD.

There are some limitations in stream assignment. First, the existing schemes cannot be effective for multiple workloads. Manual stream assignment cannot cope with the changes and combination of workloads. The existing automatic schemes do not perform well or cannot be directly applied to some common workloads. Moreover, these automatic schemes still cannot outperform the experience-based manual schemes. Second, the cost of deploying code to production environment in existing schemes is relatively high. The existing schemes have to modify some code of applications or Linux kernel, since they need to extract some information.

To address these limitations, we propose FileStream, which automatically assigns streams to files with machine learning based on the file characteristics. FileStream can dynamically adapt to multiple workloads without modifying or relying on them, and is even more effective than manual stream assignment. FileStream is implemented in Linux user mode. Other than passing stream IDs in the Linux kernel, FileStream has no other influence on the applications or the kernel. If the write lifetime hints can be used to pass stream IDs, FileStream even does not require modifying kernel code. The experimental results of six workloads on a real SSD show that FileStream significantly reduces WAF and improves throughput, and outperforms the other methods.

The major contributions of this paper are as follows:

1) We propose a file-based automatic stream management scheme including three modules: mapper, remapper, and divider. It greatly reduces WAF and improves the performance of SSDs on various workloads.

2) In mapper, we propose to estimate the mixing degree of data with different lifetimes to separate the data of newly opened files more thoroughly.

3) In remapper, we design a new feature to reflect the access patterns of long opened files. Based on this, we use machine learning to group files.

2 WORKLOAD FILE ANALYSIS

In Section 1, we mentioned two main types of workloads, dominated by append-only files or by in-place update files. Next, we introduce these two types of files in detail.

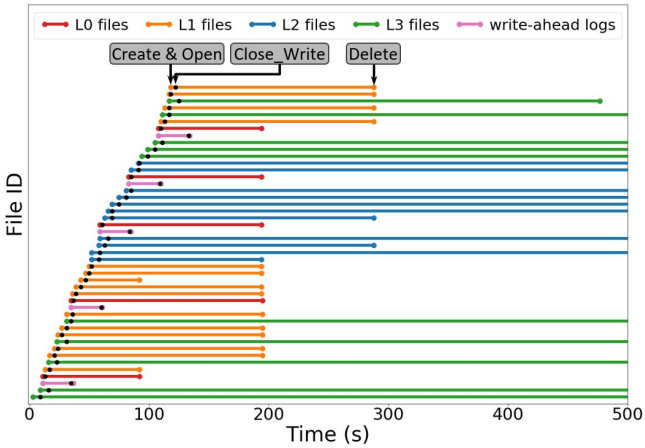


Figure 1: The operations of RocksDB files

2.1 Append-only File

Generally, append-only files can only be written sequentially and data are written by appending new pages. Furthermore, append-only files generally do not allow modifying the old data, and updated data are usually written to a new file. Since the data of one file may not be continuous on LBAs and are usually overwritten by other files, it is difficult to estimate data lifetimes by access characteristics (such as update frequency). Since the file data can be invalid only after the file is deleted, the data lifetimes are the same or similar and can be reflected by the file lifetime.

Most log-structure files are append-only files, such as the write-ahead logs and data files in RocksDB [7] and Cassandra [1], which are based on Log-Structured-Merge-Trees (LSM-Trees). Figure 1 shows the creation, writing, and deletion of RocksDB data files (L0 to L3 files) and write-ahead logs. The data files have multiple levels and low-level files are deleted after being compacted into high-level ones. Figure 1 shows that the lifetimes of low-level files are shorter than those of high-level files, since low-level files are more likely to be compacted. The level information is obtained by modifying the application code, so it can be only used in manual stream assignment [11, 15] rather than automatic stream management.

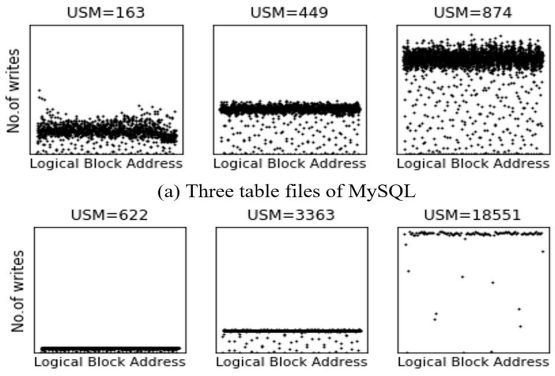
Nevertheless, we can make two significant observations from Figure 1. First, these files are frequently created and deleted. They complete the writing in only one or several seconds and there are a few files written at the same time. Therefore, it is possible to use files as units to distinguish between data lifetimes, since the data lifetimes in one append-only file are similar. Second, the lifetimes of write-ahead logs are obviously shorter than the lifetimes of data files. Different types of append-only files usually have different lifetimes due to different functions, while the same type of files may have similar lifetimes such as write-ahead logs.

2.2 In-place Update File

The write requests of in-place update files can be random and the data are updated inside the file. The data files of relational databases are usually in-place update files (such as the table files in MySQL and PostgreSQL). Generally, the files exist for a long time and they are created or deleted when the tables are created or deleted.

Figure 2 shows the writes of table files in MySQL and PostgreSQL under TPC-C [5]. The more writes to an LBA usually mean that the lifetimes of most data written to this LBA are shorter. Figure 2 shows that the update frequencies of data in the same table file are similar, while those in different table files are quite different. This is due to the different access patterns of table files in relational databases under the influence of business logic. Therefore, for in-place update files, the data in the same file usually have more similar lifetimes than those in different files.

In order to estimate data lifetimes from files' perspective, we propose a new file characteristic, unit-size-modification (USM), the ratio of the number of file modifications to the file size. The file size is part of our metric to correctly reflect the average update frequency of data items within the file. USM works because most of writes to LBAs are caused by the modifications of files. Figure 2 shows that USM has a strong positive correlation with the update frequency of data (i.e., the number of writes), and it thus can reflect the data lifetimes of in-place update files.



(a) Three table files of MySQL

(b) Three table files of PostgreSQL

Figure 2: Table files' writes in MySQL&PostgreSQL

3 SCHEME DESIGN

The overall architecture of FileStream is shown in Figure 3. The file monitor continuously extracts the information of all opened files (except read-only) by the tool (e.g. inotify) and stores them in the file attribute table. The information includes seven attributes: inode number, name, size, the number of modifications, open time, close time, and delete time. The stream manager maps or remaps each file into a stream based on its characteristics. The stream ID corresponding to each file is recorded in the file attribute table and submitted to the virtual file system (VFS). Each write request of this file carries the stream ID to the SSD.

The stream manager is divided into three modules: mapper, remapper and divider. The mapper assigns a stream ID to the file when it is opened with available information. After the file has been written to for a period of time (i.e., T seconds), the remapper uses the extracted access characteristics to reassign another stream ID to the file. The new stream ID will be added to future I/O requests, and does not affect the data already written to the SSD. In addition, the remapper periodically reassigns streams to the files it manages. The divider is responsible for allocating separate sets of stream IDs for the mapper and the remapper, and for determining the size of two sets. Next, we introduce these modules in detail.

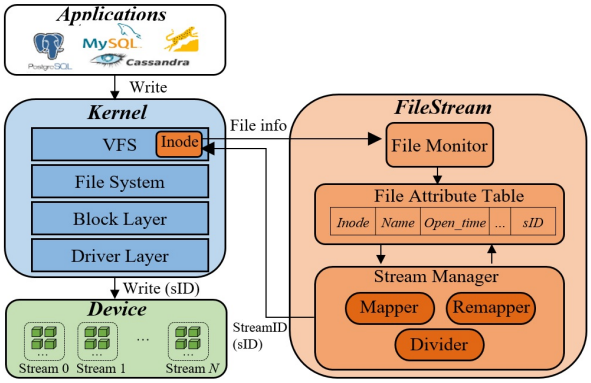


Figure 3: Architecture of FileStream

3.1 Mapper

Some files may be written immediately when they are opened, and the writing may be completed within a few seconds, as in many append-only files. Therefore, it is necessary to assign a stream ID to the file when it is opened. Since the file has not been written, few file characteristics can be extracted. The mapper is responsible for mapping the file to the stream when the file is opened based on its available characteristics and current stream usage.

Our first idea is to minimize the mixing of data between different files, since the data lifetimes of the same file are usually more consistent than those of different files (see Section 2). In practice, most data of files written to the same stream but not at the same time may be written to different blocks (especially data files of big size), and there are only a few newly opened files written at the same time. Hence, it is feasible to reduce the mixing degree of data in different files written simultaneously for separating the data with different lifetimes.

Specifically, When a file f_o is opened, the mapper evaluates the mixing degree of data between f_o and the existing files by estimating the amount of data to be written in existing files (less data to be written means less data will be mixed with f_o). We consider files with the same parent path and extension as the same type of files. Suppose that the mapper currently uses stream IDs 1 to d and the file set mapped to these streams is F_{mapper}. The mapper calculates the following characteristics for each file f_i in F_{mapper}: average number of modifications per second $m(f_i)$, the current opening duration $cd(f_i)$, and the average write duration of files of the same type as f_i (labeled as $wd(f_i)$). If $wd(f_i)$ is greater than T, it defaults to T seconds (the moment the file is remapped to another stream). $wd(f_i)$-$cd(f_i)$ is the estimated remaining duration of f_i to be written before f_i closes writing or is remapped. Then the mapper estimates the amount of data that f_i may write in current stream by estimating the number of future modifications:

$$amount(f_i) = m(f_i) * (wd(f_i) - cd(f_i)) \quad (1)$$

Our second idea is to minimize the lifetime difference between files written to the same stream, since most of the newly opened files are append-only files and their lifetime can represent the lifetime of their data (see Section 2.1). Append-only files are frequently created and deleted, and files of the same type may have very similar lifetimes. Therefore, the mapper estimates the lifetime difference between the new file f_o and an existing file f_i based on the lifetimes of the same type of files:

$$dif(f_i, f_o) = \frac{1}{N_i N_o} \sum_{f_a \epsilon F_i} \sum_{f_b \epsilon F_o} \frac{|l(f_a) - l(f_b)|}{l(f_a) + l(f_b)} \quad (2)$$

where F_i is the set of deleted files with the same type as f_i, N_i is the number of these files, and $l(f_a)$ is the lifetime of the file f_a in F_i. The definition of F_o, N_o, $l(f_b)$ corresponding to f_o is the same. The normalized difference of the lifetimes between f_a in F_i and f_b in F_o is defined by dividing their lifetime difference by their lifetime sum. Equation (2) calculates the average lifetime difference between the files of the same type with f_i and f_o to estimate the lifetime difference between f_i and f_o. It also works, even if f_i and f_o are of the same type. In addition, if there is no historical information of the same type of files with f_i or f_o, $dif(f_i, f_o)$ defaults to the maximum value of 1.

Next, the mapper combines the two factors by multiplying them, and integrates the scores of existing files of stream s (the set of files is F_s) as the score of the stream:

$$score(s) = \sum_{f_i \epsilon F_s} amount(f_i) * dif(f_i, f_o) \quad (3)$$

$score(s)$ represents the mixing degree and lifetime difference between the data of f_o and the existing files in stream s. It defaults to 0, when there is no file in stream s. A smaller score indicates that less data of different lifetimes will be combined within the stream. The mapper therefore selects the stream ID with the lowest score among streams 1 to d and assigns it to the file f_o.

3.2 Remapper

If the files remain open for T seconds, the remapper is responsible for reassigning new stream IDs to the files with the extracted access characteristics. Recall that the updated stream IDs only affect future writes to those files. Most of the files that remain open for a long time are in-place update files, while append-only files usually close writing in a few seconds. The data lifetimes of in-place update files are related to the update frequency, and we propose USM (see Section 2.2). The remapper uses clustering algorithm to group files based on their USM and remap them to streams.

The remapper performs clustering every T seconds, since workloads may change. Our clustering scheme uses Kmeans++ [2] for its high efficiency, with USM as the only feature. Assume that the remapper currently uses stream IDs (d+1) to N (SSD supports N streams) and the file set managed by the remapper is $F_{remapper}$. The remapper calculates the USM of each file f_j in $F_{remapper}$ by dividing the number of modifications in T seconds by file size (i.e. $USM(f_j) = modifications(f_j)/size(f_j)$). The remapper clusters all files in $F_{remapper}$ into k clusters based on their USM where $k = N - d$. The cluster centers (average USM of files in each cluster) are sorted from small to large and the files of each cluster are mapped to the stream (d+1) to N accordingly: files with large USM are mapped to the stream with large ID and those with small USM are mapped to the stream with small ID. In addition to the regular clustering, when the file is remapped for the first time, it is added to the nearest cluster and the corresponding stream ID is assigned to it.

In fact, not every clustering will be executed. If the existing files are still closest to the previous cluster centers and the number of cluster centers k does not change (when the workload is relatively stable, k is also stable), the remapper does not actually perform kmeans++ clustering, but only updates the cluster centers. This has the following two advantages: 1) It reduces the number of remappings of files to streams, thereby reducing the fluctuation of the assigned stream IDs. 2) Skipping clustering operations reduces the remapper's CPU and memory consumption.

3.3 Divider

The divider dynamically allocates different stream IDs for the mapper and remapper. Assume that the stream

IDs used by the mapper and the remapper are 1 to d, and (d+1) to N respectively. The divider adjusts d every T seconds. It uses the number of files managed by the mapper and remapper (labeled as FN_m and FN_r) and the total number of their modifications (labeled as MN_m and MN_r) to estimate the proportion of the streams they should occupy. First, the divider calculates the proportion factor of the mapper:

$$proportion_m = (\frac{FN_m}{FN_m + FN_r} + \frac{MN_m}{MN_m + MN_r})/2 \quad (4)$$

Next, the divider adjusts d based on the proportion factor of the mapper and rounds d (i.e. $d = \lfloor N * proportion_m + \frac{1}{2} \rfloor, 1 \leq d \leq N-1$). When the mapper manages more files and the total number of file modifications is higher, $proportion_m$ is larger and d is larger, that is, the mapper occupies more streams. Conversely, the remapper occupies more streams.

4 EVALUATIONS

In this Section, we evaluate FileStream on a real multi-streamed SSD. The SAMSUNG PM963 SSD supports 8 user-configurable streams (N=8). It does not support the write lifetime hint from kernel 4.13, so we patched the kernel to pass the stream ID in the same way as [11]. We perform experiments on six workloads. The detailed benchmarks are shown in Table 1. In Section 4.1 and 4.2, we compare FileStream with a baseline and with three existing methods whose source code we can obtain: 1) Baseline, the normal mode of PM963 SSD without multi-stream technology. 2) AutoStream [16], an automatic scheme, assigns stream IDs to I/Os in the kernel according to the access characteristics of LBAs, i.e., sequentiality, frequency, and recency. 3) LKStream [17], an automatic scheme, predicts the temperature of data based on the correlative workload features (such as I/O size, I/O count and throughput), and groups data into different streams by the temperature. 4) ManualStream [11], an manual scheme, fixes stream assignment strategy into application code. Specifically, in our experiments, data are separated to different streams by file type first, and then by files' compaction level for RocksDB and Cassandra or tables' update frequency for MySQL, PostgreSQL, and MongoDB. In Section 4.3, we evaluate the effectiveness of each module of FileStream on three representative workloads. The interval T is discussed in Section 4.4 and it is set to 60 seconds in other experiments. In Section 4.5, the resource consumption is evaluated.

4.1 WAF Comparison

Figure 4 shows the WAF values of the five schemes on the six workloads. Due to the effectiveness and flexibility of our stream management algorithm, FileStream achieves the best results across almost all workloads. FileStream reduces WAF by 34.5%, 22.3% and 20.8% on average compared with baseline, AutoStream and LKStream, respectively. On most workloads, it even performs better than ManualStream which needs to modify applications.

Table 1: Benchmarks

Workload	Benchmark tool	Operation
MySQL (v5.17.2)	TPC-C	TPC-C 64 connections
PostgreSQL (v9.6.15)	pgbench	pgbench 64 connections
RocksDB (v5.17.2)	db_bench	Update 50 million records
Cassandra (v3.6.11)	YCSB	Update 170 million records
MongoDB (v3.6.14)	YCSB	Update 70 million records
Docker(2 MySQL+ 2 RocksDB)	TPC-C db_bench	TPC-C 64 connections Update 20 million records

File characteristics are effective in distinguishing data lifetimes of both in-place update files and append-only files, so FileStream performs well across six workloads. Since it is difficult to estimate the data lifetimes of append-only files by the LBA access patterns, AutoStream and LKStream do not perform well on RocksDB, Cassandra, and MongoDB [10] with many append-only files.

Docker is a more complex workload, with many in-place update files and append-only files, and the lifetime distribution of data is more complex. How to effectively separate these data into a limited number of streams is a big challenge. Figure 4 indicates that FileStream can effectively and dynamically group file data through the cooperation of its three modules, and thus works better than other methods.

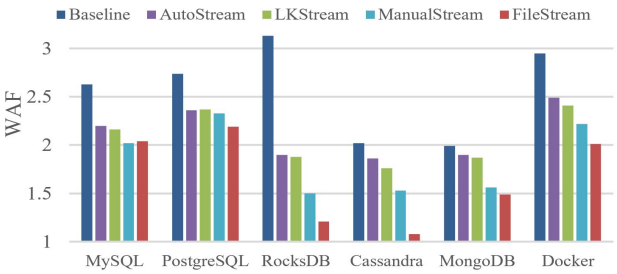


Figure 4: Comparison of WAF

4.2 Throughput Comparison

Figure 5 shows that FileStream significantly improves throughput[1] in comparison to the baseline on most workloads. FileStream improves the throughput by 21.9% and 25.8% on average compared with AutoStream and LKStream respectively. On RocksDB and Cassandra, FileStream even improves the throughput by 29.7% and 25.0% respectively compared to ManualStream. ManualStream separates the data files of RocksDB and Cassandra according to the compaction level, but there are still some lifetime differences between the file data

[1]We measure the throughput with tpmC (transactions per minute) for MySQL and PostgreSQL, ops (operations per second) for RocksDB, Cassandra and MongoDB, and IOPS for docker.

in the same level. The mapper in FileStream separates these data files (append-only files that are frequently created) into different streams according to the lifetimes of the files and the degree of data mixing between the files, thereby separating the data of different lifetimes more thoroughly. As a result, FileStream achieves higher throughput than ManualStream.

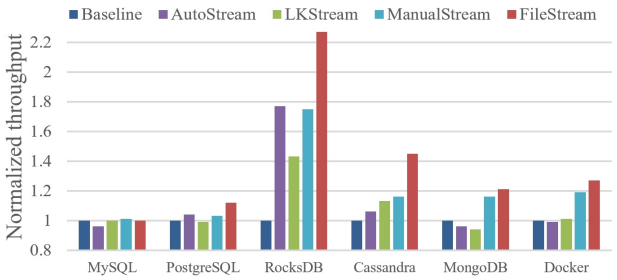


Figure 5: Comparison of normalized throughput

4.3 Discussion of Stream Manager

We verify the effectiveness of each module in FileStream on three workloads, and the WAF is shown in Figure 6. The effect of each module is verified by controlling the number of streams occupied by the mapper and remapper: *mapper*/*remapper* means the mapper/remapper occupies all 8 streams, and *mapper* + *remapper* means the mapper and remapper occupy 4 streams each. In general, each module has its own role. When the three modules work together, FileStream achieves the best results on all three workloads.

The files in RocksDB are mainly append-only files that are frequently created, while those in MySQL are mainly in-place update files that remain open for a long time. The mapper focuses on newly opened files, so it performs better on RocksDB. The remapper focuses on long opened files, so it performs better on MySQL. The mapper and remapper use the same number of streams in *mapper* + *remapper* which is not suitable for all situations, so the divider is significant. The results on Docker further verify that each module is indispensable.

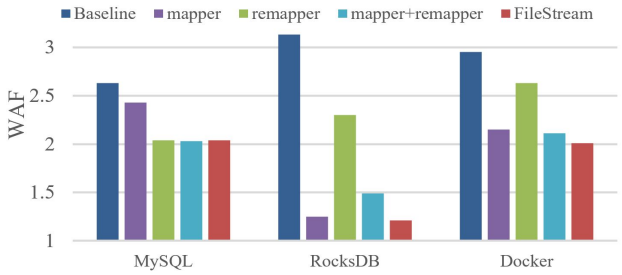


Figure 6: WAF comparison of different modules

4.4 Discussion of Parameter T

The parameter T is an key parameter in FileStream. Figure 7 compares the WAF values of FileStream with different values of T. It shows that the WAF changes with T and the optimal T value for each workload are not the same, which implies further research is required. The GC frequency of the SSD may be helpful for understanding the WAF changes with T, but most SSDs do not expose GC process. However, in general, the effect of T is also not dramatic, implying that FileStream would not be highly sensitive to this parameter.

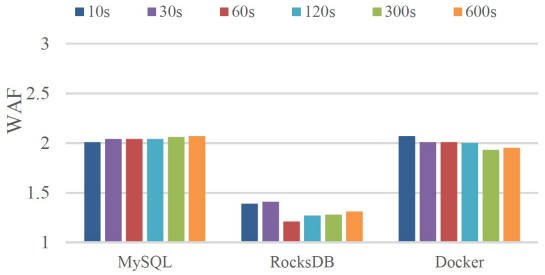


Figure 7: WAF comparison with different T

4.5 Resource Consumption

Table 2 shows the CPU utilization of baseline and FileStream. We record the total utilization of all CPU cores per second and calculate the average and maximum utilization. Compared with the baseline, the increase in average CPU usage is negligible. The increase in peak utilization is also small, because clustering only uses the USM and the number of files in clustering is limited. In addition, FileStream also uses a small amount of memory, no more than 50 MB on these three workloads.

Table 2: Comparison of CPU utilization

Workload	Baseline		FileStream	
	average	max	average	max
MySQL	9.2%	23.9%	9.3%	25.5%
RocksDB	3.8%	9.7%	4.3%	10.5%
Docker	5.9%	15.2%	6.8%	16.1%

5 CONCLUSION

In this paper, we propose FileStream to automatically manage I/O streams on general workloads. First, FileStream separates newly opened files into different streams based on available file characteristics and current stream usage by estimating the mixing degree of data with different lifetimes. Then, based on the extracted file access characteristics, FileStream remaps files to streams by K-Means++ clustering algorithm. Our experimental results show that FileStream can significantly reduce WAF and improve throughput compared with other four schemes across multiple workloads.

In the future, we will apply our scheme to more devices such as Zoned Namespace SSDs. Moreover, in further research, we may combine file information and LBA-related information for situations where the data lifetimes within the file are very different.

REFERENCES

[1] Apache. 2021. Cassandra. http://cassandra.apache.org.

[2] David Arthur and Sergei Vassilvitskii. 2007. K-Means++: The Advantages of Careful Seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*.

[3] Janki Bhimani, Ningfang Mi, Zhengyu Yang, Jingpei Yang, and Vijay Balakrishnan. 2018. FIOS: Feature Based I/O Stream Identification for Improving Endurance of Multi-Stream SSDs. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*.

[4] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, et al. 2020. A New LSM-style Garbage Collection Scheme for ZNS SSDs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[5] Transaction Processing Performance Council. 2021. TPC-C. http://www.tpc.org/tpcc.

[6] NVM Express. 2017. NVM_Express_Revision_1.3. http://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf.

[7] Facebook. 2021. RocksDB. https://github.com/facebook/rocksdb.

[8] Alibaba Group. 2019. Co-Design software and hardware for SSD storage in Alibaba Data Center. https://www.flashmemorysummit.com/Proceedings2019/08-08-Thursday/20190808_SOFT-301-1_Liu-Wu.pdf.

[9] PostgreSQL Global Development Group. 2021. PostgreSQL. https://www.postgresql.org.

[10] MongoDB Inc. 2021. MongoDB. https://www.mongodb.com.

[11] Jeong Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Usenix Conference on Hot Topics in Storage and File Systems (HotStorage)*.

[12] Taejin Kim, Duwon Hong, Sangwookshane Hahn, et al. 2019. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *17th USENIX Conference on File and Storage Technologies (FAST)*. Boston, MA.

[13] Trong Dat Nguyen and Sang Won Lee. 2019. DSM: A Low-Overhead, High-Performance, Dynamic Stream Mapping Approach for MongoDB. *Journal of Information science and Engineering* 35, 2 (2019), 447–469.

[14] Oracle. 2021. MySQL. https://www.mysql.com.

[15] Fei Yang, Kun Dou, Siyu Chen, Mengwei Hou, and Sangyeun Cho. 2016. Optimizing NoSQL DB on Flash: A Case Study of RocksDB. In *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 (UIC-ATC-ScalCom)*.

[16] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: automatic stream management for multi-streamed SSDs. In *the 10th ACM International Systems and Storage Conference (SYSTOR)*.

[17] Pan Yang, Ni Xue, Yuqi Zhang, et al. 2019. Reducing Garbage Collection Overhead in SSD Based on Workload Prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. USENIX Association, Renton, WA.

[18] Hwanjin Yong, Kisik Jeong, Joonwon Lee, and Jin-Soo Kim. 2018. vStream: Virtual Stream Management for Multi-streamed SSDs. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. USENIX Association, Boston, MA.