Design and Implementation of Virtual Stream Management for NAND Flash-Based Storage

Hwanjin Yong, Joonwon Lee, and Jin-Soo Kim, *Member, IEEE*

***Abstract*—NAND flash memory is being widely used as data storage in consumer electronics devices such as tablet computers and smartphones. However, due to the inherent nature of NAND flash memory where in-place update is not supported, NAND flash-based SSDs (Solid-State Drives) suffer from severe performance degradation as they need to move valid data during garbage collection (GC). Recently, multi-streamed SSDs have been proposed to reduce the cost of GC in the SSDs. However, commercial SSDs used in consumer electronics devices support only a small number of streams due to the device's limitation in hardware resources. This makes it difficult to fully utilize the benefits of the multi-streamed SSDs. In this article, we propose a new concept of *virtual streams* (*vStreams*) that are independent of the number of available streams within the multi-streamed SSDs. We present the design and implementation of virtual stream management architecture, called vStream-FTL, for efficient stream management in the SSD. Specifically, we present novel mechanisms to monitor the lifetime of each stream with a negligible memory overhead and map one or more vStreams into a physical stream at run time according to the lifetime of vStream. In addition, we implement the enhanced garbage collection scheme, called *vStream-aware GC* that increases the benefits of multi-streamed SSDs further. Our vStream-FTL allows embedded system developers to manage a sufficient number of streams regardless of the physical streams supported by the device. The evaluation results with smartphone workload show that the proposed vStream-FTL improves throughput by 48% compared to the Legacy-FTL with no stream support.**

***Index Terms*—Solid-state drives (SSD), NAND flash memory, flash translation layer (FTL), garbage collection.**

I. Introduction

NAND flash-based solid-state drives (SSDs) are becoming increasingly popular in consumer electronics devices thanks to high performance, low power consumption, and enhanced reliability. However, unlike the conventional hard disk drives (HDDs), NAND flash does not support in-place update. For this reason, SSDs employ a software layer called flash translation layer (FTL) that provides the traditional block device interface by redirecting new write requests to unprogrammed pages.

FTLs need to perform garbage collection (GC) periodically to reclaim the NAND flash space occupied by obsolete data. This is done by copying valid data in a victim block to a new flash block and then erasing the victim block. The data movement during GC activity, however, substantially increases write amplification inside the SSD. The increased write amplification [13] results in the reduced lifespan of SSDs because each NAND flash memory cell can endure a limited number of program/erase (P/E) cycles before being worn out. Thus, the GC operation has a significant impact not only on the SSD's performance but also on its lifespan.

Recently, the market for ball-grid array (BGA) form factor SSD with NVMe PCIe interface [9], [10] is growing thanks to the high storage performance. The BGA NVMe SSD is being widely adopted as a storage device for the high-end smartphones, tablets, and laptops. Based on the NVMe standard, the multi-stream interface [2] has been proposed to reduce the cost of GC and improve its performance by grouping the data having the same stream ID in the same flash block. With the use of the multi-stream interface, the host can control data placement in the flash media by tagging a stream ID along with each WRITE command. The multi-stream interface presents new opportunities for system designers to place the data in an effective way to mitigate the cost of GC. In addition, high-performance mobile storage devices are carefully reviewing the adoption of multi-stream technology for reducing GC cost [18].

Regarding the multi-stream interface, the NVMe specification allows up to 65535 streams per device. However, the actual number of streams that each device can support (we call these *physical streams or pStreams*) is restricted to 3–16 [2], [5] in commercial multi-streamed SSDs. This is because it is difficult for the SSD to support a large number of streams due to the device's limitation in hardware resources. In other words, increasing the number of physical streams cannot be achieved without extra hardware cost and/or a loss in performance.

When SSDs support only a limited number of physical streams, embedded system developers are forced to carefully assign a specific stream ID according to the *hotness* of the data. However, this is possible only when system developers have detailed knowledge about the characteristics (i.e., lifetime) of their data. Actually, it gets more complicated when the characteristics of the data dynamically change over time. Furthermore, when the underlying SSDs are either replaced

Manuscript received November 18, 2020; revised February 13, 2021; accepted March 7, 2021. Date of publication March 17, 2021; date of current version May 25, 2021. This work was supported in part by the Research Resettlement Fund for the New Faculty of Seoul National University; and in part by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MSIT) under Grant 2019R1A2C2089773. *(Corresponding author: Jin-Soo Kim.)*

Hwanjin Yong and Joonwon Lee are with the Department of Semiconductor and Display Engineering, Sungkyunkwan University, Suwon 16419, Republic of Korea (e-mail: hwanjin.yong@csl.skku.edu; joonwon@skku.edu).

Jin-Soo Kim is with the Department of Computer Science and Engineering, Seoul National University, Seoul 08826, Republic of Korea (e-mail: jinsoo.kim@snu.ac.kr).

Digital Object Identifier 10.1109/TCE.2021.3066524

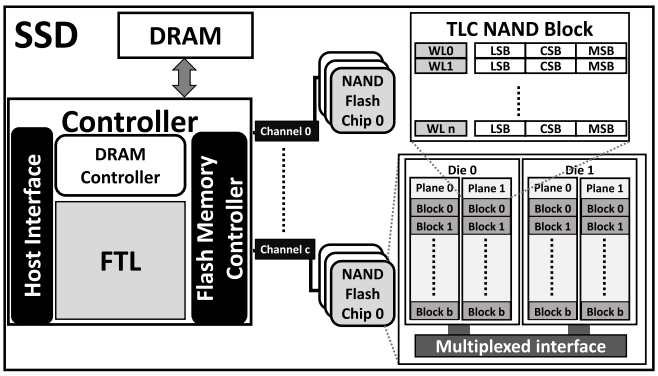

Fig. 1. General NAND Flash-based storage organization.

with or upgraded to other SSDs having a different number of physical streams, the embedded system software has to be modified to match the new physical stream count. This leads to severe degradation of portability and compatibility from an embedded system development point of view. Consequently, the limited number of physical streams which is also different from device to device is a big obstacle to fully utilize the benefits of multi-streamed SSDs. To address these limitations, we propose a new concept of *virtual streams* (or *vStreams*) that are independent of the physical streams available in the device. The goal of this article is to provide a large number of virtual streams to embedded system developers without adding any extra hardware resource inside the storage device nor experiencing any performance degradation while taking full advantage of the multi-streamed SSD.

In this article, we present the design and implementation of a virtual stream management architecture, called *vStream-FTL*. The vStream-FTL groups multiple virtual streams into a single physical stream dynamically according to the lifetime of each stream, while offering a sufficient number of (virtual) streams for application developers. In addition, the vStream-FTL incorporates a novel mechanism for NAND flash-based consumer electronics devices to monitor the lifetime of each stream with a negligible memory overhead. Furthermore, we implement a stream-aware GC mechanism in the vStream-FTL (we refer to this as *vStream-aware GC*). We demonstrate the superior performance of vStream-FTL in comparison to the Legacy-FTL and other data separation techniques with synthetic and real-world smartphone workload.

The remainder of this article is organized as follows. Section II reviews the technical background of SSDs. Section III presents the motivation of this article and Section IV presents the related work. In Section V, we describe the design and implementation of the proposed virtual stream management scheme in detail. Section VI shows the evaluation results and Section VII concludes this article.

II. Background

A. Solid-State Drive (SSD)

As illustrated in Figure 1, a typical SSD is composed of an SSD controller, DRAM, and an array of NAND flash chips. The SSD controller runs embedded software modules (i.e., FTL) that translate host I/O requests into flash memory operation and manage the underlying NAND flash memory. In general, the FTL internally maintains some number of pre-erased flash blocks, called update blocks [5] to process incoming host write requests. The update blocks are often organized using the notion of *superblocks* [6] that groups flash blocks across multiple channels, chips, dies, and planes.

A host system communicates with the SSD through a standardized host interface such as SATA, eMMC [11], UFS [12], or NVMe. The SSD exploits internal parallelism at various levels to achieve higher performance. Since each channel I/O bus can be independently and simultaneously operated, the SSD controller is able to improve SSD performance by exploiting *channel-level parallelism*. Moreover, each chip connected to the same channel can be also operated independently. Thus, the SSD controller can interleave requests among multiple chips as well, which is referred to as *chip-level parallelism*. A flash memory chip is composed of multiple dies. Each die is capable of performing I/O requests concurrently independent of the other dies in the same chip. This is called *die-level parallelism*. Each die, in turn, has multiple planes and supports multi-plane operations which can perform the same operation on multiple planes simultaneously, which is referred to as *plane-level parallelism*.

Each plane is composed of multiple blocks and each flash block, in turn, contains multiple rows (i.e., wordlines) of flash cells. In the original SLC NAND flash memory, all the flash cells belonging to the same wordline are logically combined to form a *page*. With the advent of multi-level cell technology, each cell becomes capable of storing multiple bits of data. For example, in the TLC NAND flash memory, the least significant bit, the central significant bit, and the most significant bit of each cell are grouped to form an LSB, CSB, and MSB page, respectively. Furthermore, the 3D NAND technology has been proposed to provide an even higher storage capacity, performance, and endurance. The 3D NAND stacks its memory cells vertically so as to improve scalability and chip capacity. This 3D NAND structure can virtually eliminate cell-to-cell interference [3], leading to better write performance than the planar (2D) NAND flash.

More specifically, the traditional 2D TLC NAND flash has to separate the program steps within the LSB, CSB, and MSB pages in order to mitigate the program interference on neighboring cells. On the contrary, the 3D NAND flash is capable of programming multiple bits at once via the *one-shot programming* technique [7] due to the virtually coupling-free structure which reduces the cell-to-cell program interference. Thanks to the one-shot programming, the multiple pages (LSB, CSB, and MSB) that share the same wordline can be programmed simultaneously. This results in faster programming speed and lower power consumption when compared to the 2D TLC NAND technology.

In NAND flash memory, each page can be logically divided into a user data area and a spare area. The user data area contains the actual data written by the user. In contrast, the spare area stores the management information such as ECCs (error correction codes) and the internally generated metadata for the corresponding user data.

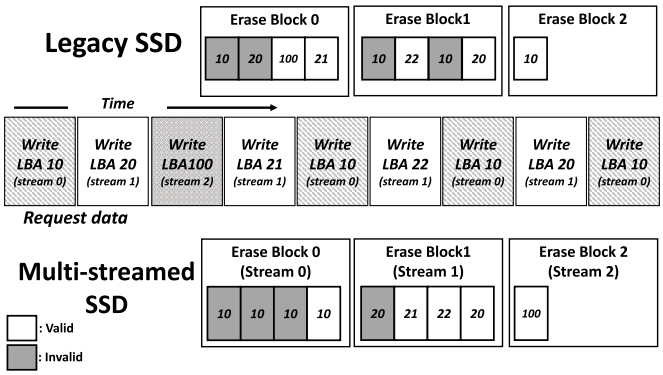

Fig. 2. Comparison of data placement with or without multi-stream support.

B. Multi-Streamed SSD

With the multi-stream interface, the SSD can store incoming data from host into separate flash blocks according to the stream ID. Figure 2 shows a simplified example of how a multi-streamed SSD works. The legacy SSD (i.e., the SSD without stream support) simply places data in their original request order. In contrast, the multi-streamed SSD places incoming data into different flash blocks according to the stream ID.

If the host system assigns the same stream ID to the data with a similar lifetime, then those data will be stored in the same flash block. This helps to eliminate or significantly reduce the data movement during GC because all the data belonging to the flash block are likely to be invalidated at the same time.

Embedded system developers in consumer electronics devices can take advantage of multi-streamed SSDs by carefully allocating stream IDs based on data lifetime. For instance, they can assign different stream IDs according to various data types such as temporary data, metadata, log data, user data, etc. Previous work showed that the application performance is notably improved by leveraging these characteristics with the multi-streamed SSD [2], [5]. Consequently, the multi-stream interface presents new opportunities for embedded system designers to control data placement in an effective way while reducing the cost of GC significantly.

C. Flash Friendly File System (F2FS)

Flash-Friendly File System (F2FS) [1], the standard file system for mobile smartphones, is a log-structured, flash-optimized file system that takes into account the characteristics of NAND flash memory-based storage devices.

F2FS employs a multi-head logging scheme that writes metadata and data into separate logs based on their hotness. For the multi-head logging scheme, F2FS concurrently manages six active logs for storing each of Hot/Warm/Cold node and data. The criteria for defining data separation in the F2FS file system are as follows: (1) Hot node contains direct node blocks for directories. (2) Warm node contains the rest of direct node blocks. (3) Cold node contains indirect node blocks. (4) Hot data contains directory blocks. (5) Warm data contains data blocks except Hot and Cold data blocks. (6) Cold data contains multimedia data or migrated data blocks.

F2FS achieves high performance by reducing GC overhead with their hot/cold data separation mechanism. Furthermore, the benefits of hot/cold data separation can be more effective in flash-based storage using the multi-stream interface. With the use of the multi-stream interface, flash-based storage can also store logical pages into separate flash blocks internally according to hotness information at the file system level.

III. MOTIVATION

In multi-streamed SSDs, there is a limitation in the maximum number of concurrently active open streams. In case of NVMe SSDs, if the host system tries to use a new stream beyond this limitation, an arbitrary stream is released by the device controller to allocate the stream resources associated with that stream ID to the new stream [10]. Therefore, the host system should explicitly close one or more open streams for the multi-streamed SSDs when the available streams are exhausted for processing new stream requests.

In any case, the small number of physical streams leads to a situation where the data which has a different lifetime is mixed together in a flash block due to frequent release and reuse of stream IDs, thereby reducing the benefits of the multi-streamed SSD. For this reason, embedded system developers ask SSD manufacturers to support a large number of physical streams, but it is not easy because of the device's restricted hardware resources. Therefore, it is a new challenge for system designers to control a limited number of physical streams effectively to fully take advantage of multi-streamed SSDs.

Our approach is to provide a large number of *virtual* streams (vStreams) by virtualizing the notion of streams, instead of increasing the number of physical streams within the storage device. It is motivated by the observation that if the data belonging to different stream IDs have similar lifetimes, they can be grouped together to share the same physical stream resources inside the SSD. This allows embedded system developers to manage a sufficient number of streams regardless of the number of physical streams, making storage applications more portable and easier to be interoperable across different models of multi-stream SSDs from different vendors.

IV. RELATED WORK

Much work has been done to reduce the cost of GC by classifying data according to their *hotness* in flash storage, both at the file system level and at the device driver level. A common approach is to monitor various characteristics of the data such as update frequency, update interval, etc. and then group them into two (e.g., hot and cold) or more levels of temperature [13]. Since the hot data is likely to be invalidated or updated soon, the data movement overhead during GC can be significantly alleviated if the hot data is grouped together in the same flash block.

DAC (Dynamic dAta Clustering) [16] maintains a number of logical regions and assigns a specific region number to each data so that the data with similar temperature is stored

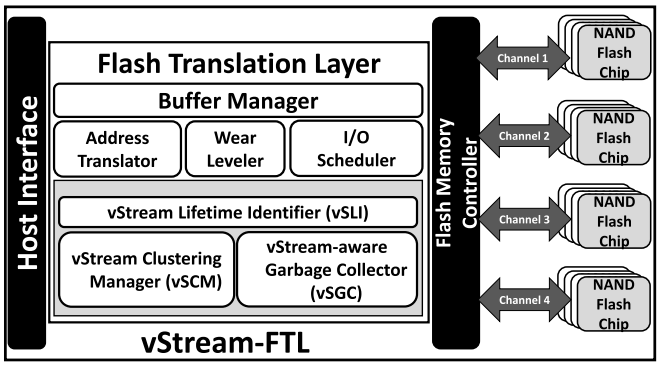

Fig. 3. The overall architecture of vStream-FTL.

in the same flash block. In DAC, the region number associated with the data changes dynamically; if some data are overwritten, they are migrated to the upper region, while the data copied to other flash blocks during GC is moved down to the lower region. Combo-FTL [15] classifies the hotness of the data based on the I/O request size. This is based on the observation that the small-sized data tends to be updated more frequently than the large-sized data. However, this approach is not effective when applications generate I/O with a uniform request size. The recently proposed AutoStream [5] implemented an automatic stream detection mechanism based on data temperature. AutoStream automatically assigns stream IDs by monitoring the update frequency of the incoming write requests.

Note that all the aforementioned techniques work at the level of each logical page. In other words, their goal is to find a set of logical pages that have the similar update frequency and group them together in the same flash block without using any hint from the host. However, our approach works at the stream level. We leverage the virtual stream ID assigned by embedded system developers and focus on identifying a set of virtual streams that have a similar lifetime. Compared with previous work, our approach is more effective with much less overhead.

This article enhances the previous version of our work [17] in terms of the efficiency of GC by implementing an enhanced garbage collection scheme described in Section V-D. In addition, this article includes evaluation results using real-world smartphone workload.

V. vSTREAM-FTL

In this section, we introduce a new FTL architecture, called vStream-FTL, that supports the notion of virtual streams (vStreams).

A. Overall Architecture

Figure 3 illustrates the overall architecture of vStream-FTL. The vStream-FTL is similar to the traditional page-level FTL, which supports the multi-stream interface.

The buffer manager controls both the write buffer and the internal copy buffer. The write buffer is used for processing incoming write requests from the host, while the internal copy buffer for the internal data movement during GC. The wear leveler attempts to extend the lifespan of the device by ensuring that all the blocks within the SSD are worn out evenly. The I/O scheduler in the SSD controller manages a request queue for each channel of the device.

As shown in the gray box in Figure 3, three new components, the vStream lifetime identifier (vSLI), the vStream Clustering Manager (vSCM), and the vStream-aware GC (vSGC) have been added for efficient virtual stream management within the SSD. The vStream Lifetime Identifier (vSLI) calculates the lifetime of each virtual stream and then the vStream Clustering Manager (vSCM) maps one or more virtual streams to a physical stream according to their lifetimes. In addition, the existing GC has been extended to improve the benefit of the multi-streamed SSD further, which we refer to as the vStream-aware GC (vSGC). In the following sections, we describe the role of each component in more detail.

B. vStream Lifetime Identifier (vSLI)

To determine the lifetime of each vStream, we implement a vStream Lifetime Identifier (vSLI) that monitors the lifetime of dead logical pages (*dead pages*) in each vStream. The age of a dead page can be simply calculated by measuring the time when the page is written and the time it is overwritten by the WRITE command or it is invalidated through the DISCARD (or TRIM) command [10].

More specifically, we define $L(v)$, the lifetime of each vStream v, as the average age of dead pages in the vStream as shown in Eq. (1). In Eq. (1), $D(v)$ denotes the set of dead pages for the given vStream v and $S(v)$ indicates the sum of the ages of all the dead pages in $D(v)$. The age of a particular dead page p, is calculated using $T_w(p)$ and $T_u(p)$, which represent the time when the dead page p is written and the time when it becomes obsolete due to subsequent overwrite or TRIM command, respectively.

$$L(v) = \frac{S(v)}{|D(v)|} = \sum_{p \in D(v)} \frac{T_u(p) - T_w(p)}{|D(v)|} \tag{1}$$

Whenever one or more live logical pages are invalidated or discarded, we update $S(v)$ and $D(v)$. After that, these parameters are used when the vStream Clustering Manager (vSCM) calculates the lifetime of each vStream.

As expected, we need to require a massive amount of memory to keep track of the written time $T_w(p)$ for each logical page p in order to calculate the lifetime of a page. For example, a 1TB SSD based on the full 4KB page-mapping has 256M logical pages. If we use a 4-byte timestamp per logical page, approximately 1GB of memory is required to maintain $T_w(p)$. This makes it difficult to achieve high scalability for large-capacity SSD used in consumer electronics devices.

In order to mitigate this memory pressure, we present a novel scheme which leverages the following characteristics of modern SSD architecture. First, the basic unit of NAND flash program operation is the flash page size which is larger than the logical page size. Second, multiple flash pages are programmed simultaneously due to the one-shot programming of the state-of-the-art 3D NAND flash devices. Third, two or

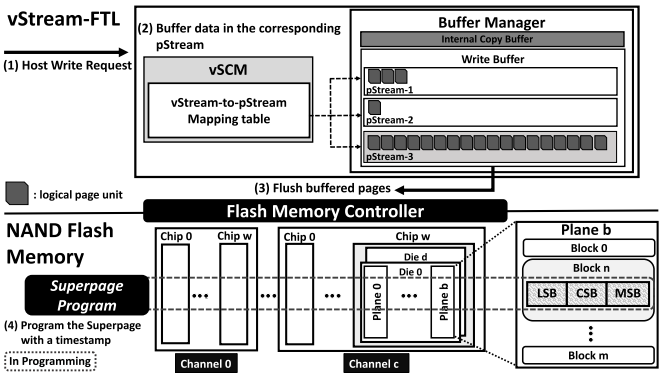

Fig. 4. The write procedure in vStream-FTL.

more planes can be programmed at the same time by using a multi-plane operation. Finally, modern SSDs exploit channel-level, chip-level, and die-level parallelism to achieve higher throughput [6]. As a large number of logical pages are written into the NAND flash memory at once in modern SSDs, we can make them share a single timestamp for the written time.

For example, we assume a realistic organization of a 1TB SSD based on TLC NAND flash memory where the number of channels is four ($c = 4$) and a channel is composed of 4 chips ($w = 4$). Each chip internally has two dies ($d = 2$) and flash blocks in each die are organized as two planes ($b = 2$). A flash block is composed of 256 wordlines and each wordline holds the data for LSB, CSB, and MSB pages ($p = 3$). Finally, the flash page size is 16KB that can contain the data of four 4KB logical pages ($l = 4$). In this case, we can reduce the memory overhead by a factor of 768 ($= c \times w \times d \times b \times p \times l$) by sharing the same timestamp among the logical pages written together (i.e., a superpage unit), requiring only 1.3MB of memory to maintain the written time for all logical pages. In this way, the FTL maintains write timestamps in units of superpages within each superblock as metadata. This metadata is stored and updated in DRAM when handling the host write requests. Thus, we can support a large number of vStreams with a reasonable resource overhead.

In Figure 4, we assume that the target SSD supports only three physical streams internally. This means that vStream-FTL opens three update superblocks corresponding to each pStream at a time for processing of the host write operations. When the controller receives a host write command tagged with a vStream ID, vStream-FTL places data in the superblock of one of the pStreams with the assistance of the vStream Clustering Manager (vSCM).

Figure 4 illustrates the write procedure inside the SSD in more detail, which explains how each logical page can share the same written time $T_w(p)$. (1) When a host write request arrives at the SSD, the FTL divides the newly-arrived write request into logical page units. (2) Then, the FTL temporarily stores the incoming logical pages into the write buffer of a certain physical stream (*pStream-n*) by referring to the virtual-to-physical stream mapping table maintained by vSCM. (3) Later, the FTL flushes the buffered logical pages (*pStream*-3) into the flash memory chips by striping I/O requests across all of the flash parallel units to maximize write performance when the amount of buffered data exceeds the superpage size. (4) At the same time, the FTL assigns the same timestamp $T_w(p)$ for all the logical pages programmed together. As a result, the FTL needs to maintain the timestamps not for each logical page, but for each physical *superpage* whose size is larger than the logical page size by a factor of more than several hundreds.

Algorithm 1 Updating the Lifetime of a vStream

```
input : N(v): set of newly dead pages in vStream v
output: D(v): new set of dead pages in vStream v
        S(v): new age sum of dead pages in vStream v
1 T_curr ← Get_CurrentTime();
2 foreach l ∈ N(v) do
3 |   p ← TranslateL2P(l);
4 |   T_w(p) ← Get_WrittenTime(p);
5 |   S(v) ← S(v) + T_curr − T_w(p);
6 |   D(v) ← D(v) ∪ {p};
7 end
```

Algorithm 1 shows the detailed steps of how the parameters related to the lifetime of a vStream is updated when some of live pages are turned into dead pages via the overwrite or the TRIM command. First, the vStream Lifetime Identifier (vSLI) obtains the current time (line 1). After that, for each newly dead page l, vSLI obtains the physical superpage number p through the L2P mapping table (line 3). Using p, vSLI gets the written time $T_w(p)$ by looking up the FTL metadata (line 4). Finally, vSLI updates $S(v)$ and $D(v)$, which represent the sum of the ages of all the dead pages and the set of dead pages in vStream v, respectively (lines 5–6). Both $S(v)$ and $D(v)$ are used to re-calculate the lifetime of vStreams using Eq. (1).

C. vStream Clustering Manager (vSCM)

To classify vStreams according to their lifetimes, we use the K-means clustering algorithm [4] which is one of the most widely used data clustering algorithms. We let K represent the number of physical streams supported by the SSD. Our statistical classification scheme measures the Euclidean distance between the average lifetime of each vStream and the average lifetime of a certain physical stream to map vStreams closer to one of the physical streams with a similar lifetime. We periodically (e.g., every 10 minutes in our evaluation) update the mapping from vStreams to physical streams in order to adapt to the changes in the workloads' access patterns.

Algorithm 2 illustrates how each vStream is grouped based on its lifetime. Every predefined period of time, the vStream Clustering Manager (vSCM) calculates the average lifetime of each vStream, $L(v)$, using $S(v)$ and $D(v)$ maintained by vSLI (lines 1–3). Then, vSCM re-calculates $L_{avg}(p)$, the average lifetime of the given physical stream p, using the updated values of $L(v)$ for the vStreams currently belonging to the physical stream p (lines 4–11). Now for each vStream v, vSCM attempts to determine the physical stream p' where the vStream v can be co-located by measuring the distance between its lifetime and the centroid lifetime value of each physical stream (lines 13–15). Finally, vSCM updates the virtual-to-physical stream mapping table that maps one or

Algorithm 2 Clustering vStreams

input : N: number of virtual streams
K: number of physical streams
$V(p)$: current set of vStreams mapped to pStream p
output: $V_n(p)$: new set of vStreams mapped to pStream p

1 **for** $v \leftarrow 1$ **to** N **do**
2 $\quad L(v) \leftarrow S(v)$ / $|D(v)|$;
3 **end**
4 **for** $p \leftarrow 1$ **to** K **do**
5 $\quad L_{sum}(p) \leftarrow 0$;
6 $\quad V_n(p) \leftarrow \emptyset$;
7 $\quad$ **foreach** $v \in V(p)$ **do**
8 $\quad\quad L_{sum}(p) \leftarrow L_{sum}(p) + L(v)$;
9 $\quad$ **end**
10 $\quad L_{avg}(p) \leftarrow L_{sum}(p)$ / $|V(p)|$;
11 **end**
12 **for** $v \leftarrow 1$ **to** N **do**
13 $\quad$ **for** $p \leftarrow 1$ **to** K **do**
14 $\quad\quad$ Find p' such that $L_{avg}(p')$ is closest to $L(v)$;
15 $\quad$ **end**
16 $\quad V_n(p') \leftarrow V_n(p') \cup \{v\}$;
17 **end**

more vStreams into a physical stream (line 16). When the vStream-FTL processes the incoming write requests or internal data movement requests during GC, the FTL translates the vStream ID into the corresponding pStream number by looking up the virtual-to-physical stream table which vSCM updates periodically.

When it comes to the time complexity, our vStream clustering mechanism executes in $O(N \times K)$, where N is the number of active virtual streams and K is the number of physical streams that the SSD can support. Note that the time complexity of our clustering algorithm does not contain any parameter related to the storage capacity, which makes it extremely scalable beyond multi-terabyte SSDs.

D. vStream-Aware Garbage Collector (vSGC)

We develop a vStream-aware garbage collector (vSGC) to increase the benefits of multi-streamed SSDs further. The goal of vSGC is to achieve a more sophisticated data placement by distributing valid pages from a victim block into separate flash blocks according to the lifetime of each stream.

The general garbage collection (GC) process is performed as follows: (1) the FTL selects a victim block, (2) it copies valid pages within the victim block to a new flash block, and (3) it erases the victim block to reclaim free space. The FTL usually assigns a separate update block for storing valid pages generated during GC to prevent those pages from being mixed with the incoming data from host in the same update block. This helps to separate data with different lifetimes since the newly-arrived data is considered as *hot*, while the valid pages copied during GC are *cold* in the sense that they are not updated for a long time.

However, dedicating a separate update block for GC'ed data is not enough for multi-streamed SSDs. The ultimate goal of multi-streamed SSDs is to classify data and put them into separate physical flash blocks according to their (estimated) lifetimes. Although we have succeeded in classifying data when they are written, it becomes meaningless if we store them back together in a single flash block during GC.

Algorithm 3 vStream-Aware Garbage Collection

input : $U(p)$: set of GC update blocks for pStream p

1 Choose a victim flash block B;
2 **foreach** *valid page p in block B* **do**
3 $\quad$ *copybuf* $\leftarrow$ *ReadPage*(p);
4 $\quad$ *vsid* $\leftarrow$ *GetVSID*(p);
5 $\quad$ *psid* $\leftarrow$ *TranslateV2P*(*vsid*);
6 $\quad$ Store *copybuf* to U(*psid*);
7 **end**

Instead, our vSGC maintains a set of update blocks, one per each physical stream. During the step (2) shown above, vSGC checks the matching physical stream for each logical page within the victim block and copies it to the corresponding physical block for that physical stream. This helps valid pages to be classified in a more fine-grained way, leading to much better data separation during GC.

In order to regroup valid pages during GC, we need to identify the virtual stream number associated with each valid page quickly without affecting the overall performance of GC. To this end, we have re-designed the NAND page layout where the *virtual stream ID* (*VSID*) corresponding to the user data is stored in the spare area. Because the spare area is also read along with the user data area by the FTL, this approach allows the FTL to identify the VSIDs of valid pages without any additional flash operations.

Consider a situation that there are two valid pages p_a and p_b in the victim block B whose VSIDs are 1 and 2, respectively. The fact that p_a an p_b were written into the same block B implies that, at the time of writing, vSCM decided vStream 1 and 2 had a similar lifetime and mapped them to the same physical stream. However, at the time when the block B is reclaimed by vSGC, the things may have changed, i.e., the vStream 1 may become *colder* than the vStream 2. Even in such cases where the lifetime characteristics of virtual streams change over time, vSGC can adapt well because it re-evaluates the virtual-to-physical stream mapping of all the valid pages at the time of GC.

Regrouping data during GC is also effective in handling data belonging to a new virtual stream. When there are write requests for a new virtual stream, it is very difficult to estimate its lifetime due to the lack of any history for the given virtual stream. In this case, the vStream-FTL temporarily maps them to the *default pStream*. Once a sufficient amount of history has been accumulated for the new virtual stream, vSCM can map it to another physical stream. The data written into the default pStream will be gradually migrated to the new physical stream as they are GC'ed later.

Algorithm 3 illustrates how vSGC can reclassify valid pages according to their lifetimes. When invoked, vSGC first selects a victim block by using a well-known policy such as greedy algorithm [14] (line 1). For each valid page in the victim block, vSGC reads the page (including the spare area) into the internal copy buffer (line 3). Then, vSGC extracts the VSID from the spare area and then obtains the

TABLE I
SMARTPHONE I/O TRACE CHARACTERISTICS

Number of I/Os (Millions)			Total request size (GiB)		
Write	Read	Discard	Write	Read	Discard
26.19	23.45	5.56	390.74	794.43	692.85

corresponding *physical stream ID (PSID)* by referring to the virtual-to-physical stream mapping table (lines 4–5). Finally, vSGC copies the valid page into the update block of the physical stream that corresponds to PSID (line 6).

VI. EVALUATION

In this section, we present the evaluation environment and various experimental results.

A. Experimental Setup

The proposed virtual stream management scheme has been implemented on an open-source FTL simulator [8]. The simulator is configured to emulate 128Gb TLC 3D NAND flash memory. The simulator provides the total capacity of 256GB with a 7% over-provisioning area, running a page-mapping FTL where the logical page size is set to 4KB. In our simulation, the flash storage is organized to contain 4 channels, each of which contains 4 chips each with 2 dies. Each die has 2 planes. Each plane contains 360 flash blocks, each of which consists of 256 wordlines. The flash page size is set as 16KB. For timing parameters, the latency for page read, high speed program, and block erase is 80us, 2ms, 4ms, respectively. If the number of free flash blocks available in the FTL becomes lower than 3% of the total number of flash blocks, the garbage collection (GC) process is invoked. In this case, the FTL chooses a victim block based on the greedy policy that selects a block with the smallest number of valid pages. In addition, we have extended the simulator to support the multi-stream interface with up to three physical streams.

We use both synthetic and real-world workload to evaluate the vStream-FTL architecture. To evaluate the performance of the proposed approach, we collect the I/O traces on a real smartphone with the F2FS file system by letting consumers use the smartphone in a usual way for two months (e.g., Map, Messenger, SNS, Mail, Browser, Game, and Streaming Services). The characteristics of the real-world smartphone workload are summarized in Table I.

In our evaluations with the smartphone I/O workload, we use the total seven virtual streams by assigning six virtual stream IDs to each of Hot/Warm/Cold node and data, and one for file system metadata.

In order to compare the performance gains of vStream-FTL over the other FTLs, we compare the proposed scheme (vStream-FTL) with other alternatives: Legacy-FTL, Auto-FTL, and Manual-FTL. Legacy-FTL represents the traditional page-mapping FTL with no multi-stream support. Auto-FTL indicates the FTL that automatically assigns a stream ID to each logical page based on the AutoStream technique [5]. Finally, Manual-FTL shows an example of what embedded system developers can manually map seven virtual streams into three physical streams. For Manual-FTL, we consider the static stream ID assignment as shown in Table II. Before running each experiment, we warm up the device by writing user data sequentially until it reaches 90% of the total SSD capacity.

TABLE II
STATIC STREAM ID ASSIGNMENT IN MANUAL-FTL

Stream ID	1	2	3
Data Type	Hot node Hot data	Warm node Warm data System metadata	Cold data Cold node

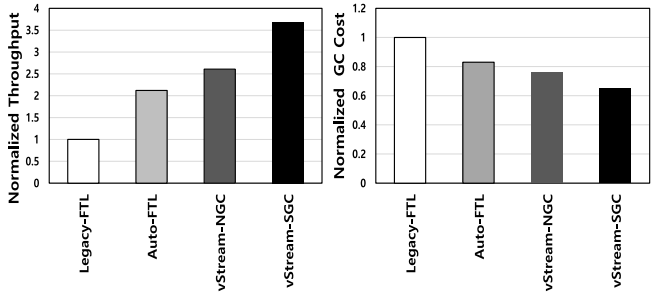

Fig. 5. Comparisons of the normalized throughput and the cost of GC with synthetic benchmark.

B. Evaluation With Synthetic Benchmark

To evaluate the performance of various data separation schemes in a controlled environment, we have developed a synthetic benchmark that is similar to the one used in the previous study [5]. In our model, we divide the total 256GB of storage capacity into 64 partitions: 48 hot partitions (each 2GB in size), 8 warm partitions (each 4GB in size), and 8 cold partitions (each 16GB in size). We loop over the 64 partitions one by one, issuing a single 128KB write request at a time in each partition. Within a partition, each 128KB region is written sequentially to the end of the partition and then the write starts over from the beginning again. In this experiment, we assign a different stream ID (i.e., 64-virtual streams) to each partition and see if the vStream-FTL can cluster those hot, warm, and cold streams effectively. This experiment shows that our vStream-FTL aims at addressing the chaos caused by mapping a large number of virtual streams to the limited number of physical streams. Since the size of the hot partition is much smaller than that of the cold partition, logical pages in a hot partition will receive the overwrite more frequently.

Figure 5 illustrates the overall throughput and the GC cost in Legacy-FTL, Auto-FTL, and vStream-FTL. We measure the GC cost as the number of valid pages copied during GC operations. Note that all the results are normalized to those of Legacy-FTL for comparison. In Figure 5, vStream-NGC represents the vStream-FTL architecture with the normal garbage collection, while vStream-SGC implements the vStream-aware garbage collection described in Section V-D.

We observe that the use of data separation schemes is very effective; it improves the overall throughput by 2–4x compared to Legacy-FTL that has no data separation scheme. In Figure 5, we can see that vStream-FTL performs best among the evaluated schemes. In particular, vStream-SGC significantly

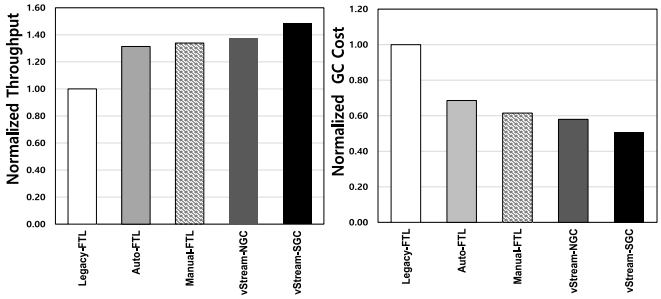

Fig. 6. Comparisons of the normalized throughput and the cost of GC with smartphone workload.

outperforms vStream-NGC due to the efficient virtual stream management with the vStream-aware data separation even for the data copied during GC operations. When comparing to Legacy-FTL, our vStream-SGC improves the throughput by 3.7x and reduces the GC cost by 35%. In this experiment, we observe that vStream-FTL can successfully separate the data after the initial clustering is completed because the lifetime of each stream does not change dynamically.

C. Evaluation With Real-World Smartphone Workload

We compare the throughput and the cost of GC in Legacy-FTL, Auto-FTL, Manual-FTL, vStream-NGC (normal GC), and vStream-SGC (vStream-aware GC) with the real-world smartphone workload on top of the F2FS file system. As in Figure 5, the results are normalized to those of Legacy-FTL. In Figure 6, we observe that Auto-FTL, Manual-FTL, and vStream-SGC (including vStream-NGC) achieve much higher performance and lower GC cost than that of Legacy-FTL which does not support the multi-stream interface. As expected, vStream-SGC achieves much higher throughput than vStream-NGC. From this experiment, we can confirm that vStream-SGC is very effective even in real-world smartphone workload where the lifetime of the data changes dynamically over time by achieving efficient data movement during GC. When comparing to Legacy-FTL, we find that the vStream-SGC improves performance by 48% while reducing the GC cost by 51%.

We can observe that Auto-FTL achieves good performance close to Manual-FTL without using the stream information (i.e., hotness) provided by the F2FS file system. Despite this, Auto-FTL is not feasible in cost-sensitive consumer storage systems. This is because Auto-FTL maintains the update frequency information based on the chunk unit (1MB in size by default) for data classification and monitoring. For example, a 1TB SSD has 1M logical chunk units and the size of each chunk is about 32 Bytes for monitoring the update frequency and access time of each chunk in the AutoStream technique [5]. As a result, the total memory requirement of the Auto-FTL can be calculated as 32 MB. As expected, vStream-SGC achieves much higher performance and lower GC overhead than that of other approaches. Note that vStream-SGC achieves a significantly higher throughput than Manual-FTL which models the situation where several virtual streams are forced to combine statically due to the limited number of the physical streams inside the storage. This means that the static approach used in Manual-FTL cannot cope well with the dynamic changes in realistic workloads' characteristics over time.

Furthermore, combining virtual streams based solely on the definition of data hotness provided by F2FS (such as Table II) can negatively impact storage performance. This is because each of Hot/Warm/Cold data and node is not guaranteed to have a similar lifetime, leading to the situation where the data with different lifetimes share the same stream ID. On the contrary, vStream-SGC is capable of grouping virtual streams more precisely by updating the virtual-to-physical stream mapping table at run time according to the change in the lifetimes of virtual streams. Finally, our vStream-FTL architecture makes embedded system developers do not have to worry about how to combine several virtual streams when the underlying storage supports only a limited number of physical streams.

D. Overhead Analysis

In this section, we measure the overhead of vStream-FTL algorithm on a real device. In the vStream-FTL design, vStream Clustering Manager (vSCM) is the most time-consuming operation due to the K-means clustering algorithm. To analyze the overhead of processing the K-means algorithm inside the flash-based storage device, we have implemented the vSCM by modifying the firmware in the state-of-the-art commercial SSD product which is a representative consumer electronics device using a high-speed microprocessors running at 750MHz clock speed.

We measured the overhead of the K-means clustering using the synthetic workload (64 virtual streams and 3 physical streams) used in Section VI-A. For precise analysis, we have utilized the embedded performance monitoring unit (or PMU) that is available in modern processing units. According to our measurement results, we observe that the K-means algorithm takes 1.8 microseconds in the real SSD. This cost is not a big burden in the modern flash storage architecture because the flash storage controllers are equipped with multi-core processors. Moreover, performing the K-means algorithm is not on the critical I/O path, as it can be run in the background periodically. We believe the overhead is negligible by exploiting the underutilized processing resources.

VII. Conclusion

In this article, we propose a new concept of virtual streams (vStreams) that are independent of the number of available streams within the multi-streamed SSD for NAND flash-based consumer electronics. We also present the design and implementation of a virtual stream management architecture (vStream-FTL) to remove the restriction on the number of physical streams in the existing multi-streamed SSD. In our vStream-FTL, we implement novel mechanisms to monitor the lifetime of each stream with a negligible memory overhead and map one or more vStreams into a physical stream at run time according to their lifetime. In addition, we implement the enhanced garbage collection (called as vStream-aware GC)

that attempts to copy valid pages while considering their lifetime during GC operations. Consequently, the proposed vStream-FTL provides the embedded system developers with a sufficient number of virtual streams so that they can take full advantage of the multi-streamed SSD regardless of the amount of device's hardware resources. Experimental results with real smartphone workload show that our vStream-FTL improves the overall throughput by 48% and reduces the average GC cost by 51% compared to the Legacy-FTL with no stream support.

REFERENCES

[1] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, Santa Clara, CA, USA, Feb. 2015, pp. 273–286.

[2] J. U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proc. 6th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2014, p. 13.

[3] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, "Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques," in *Proc. 23th IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 49–60.

[4] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-means clustering algorithm," *J. Royal Stat. Soc. C, Appl. Stat.*, vol. 28, no. 1, pp. 100–108, 1979.

[5] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "AutoStream: Automatic stream management for multi-streamed SSDs," in *Proc. 10th ACM Int. Syst. Storage Conf. (Systor)*, Haifa, Israel, 2017, pp. 1–11.

[6] J. Zhang, J. Shu, and Y. Lu, "ParaFS: A log-structured file system to exploit the internal parallelism of flash devices," in *Proc. USENIX Annu. Tech. Conf. (ATC).*, Denver, CO, USA, 2016, pp. 87–100.

[7] J. W. Im *et al.*, "7.2 A 128 Gb 3b/cell V-NAND flash memory with 1 Gb/s I/O rate," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2015, pp. 1–3.

[8] *Droll Simulator: Open Source SATA SSD Simulator*. Accessed: May 27, 2019. [Online]. Available: https://github.com/essencloud/droll

[9] H. J. Kim and J. S Kim, "A user-space storage I/O framework for NVMe SSDs in mobile smart devices," *IEEE Trans. Consum. Electron.*, vol. 63, no. 1, pp. 28–35, Feb. 2017.

[10] NVM Express Workgroup. *NVM Express Revision 1.3*. Accessed: Jan. 15, 2020. [Online]. Available: https://nvmexpress.org/resources/specifications/

[11] *Embedded MultiMediaCard (e.MMC)*. Accessed: Jan. 15, 2020. [Online]. Available: http://www.jedec.org/standards-documents/technology-focus-areas/flash-memory-ssds-ufs-emmc/e-mmc

[12] *Universal Flash Storage (UFS)*. Accessed: Jan. 15, 2020. [Online]. Available: http://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs

[13] J. Lee and J.-S. Kim, "An empirical study of hot/cold data separation policies in solid state drives," in *Proc. 6th Int. Syst. Storage Conf. (SYSTOR)*, 2013, p. 12.

[14] J. H. Kim, S. H. Kim, and J. S. Kim, "Utilizing subpage programming to prolong the lifetime of embedded NAND flash-based storage," *IEEE Trans. Consum. Electron.*, vol. 64, no. 1, pp. 101–109. Feb. 2018.

[15] S. Im and D. Shin, "ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer," *J. Syst. Archit.*, vol. 56, no. 12, pp. 641–653, Dec. 2010.

[16] M. L. Chiang, P. C. Lee, and R. C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Softw. Pract. Experience*, vol. 29, no. 3, pp. 267–290, 1999.

[17] H. Yong, K. Jeong, J. Lee, and J. S. Kim, "vStream: Virtual stream management for multi-streamed SSDs," in *Proc. 10th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2018, pp. 1–6.

[18] S. Vishwakarma, A. Sharma, and S. Kodase, "Enhancing eMMC using multi-stream technique," *EAI Endorsed Trans. Cloud Syst.*, vol. 5, no. 14, p. 7e, 2019.

Hwanjin Yong received the B.S. and M.S. degrees in computer science and engineering from Hanyang University, Seoul, South Korea, in 2006 and 2008, respectively. He is currently pursuing the Ph.D. degree with the Department of Semiconductor and Display Engineering, Sungkyunkwan University, Suwon, South Korea. Since 2008, he has been working as an Engineer with the Software Development Team, Memory Division, Samsung Electronics, Suwon. His current research interests include embedded systems and storage systems.

Joonwon Lee received the B.S. degree in computer science from Seoul National University, Republic of Korea, in 1983, and the M.S. and Ph.D. degrees from the Georgia Institute of Technology, USA, in 1990 and 1991, respectively. He is currently a Professor with Sungkyunkwan University. Before assuming that role, he was a Professor with the Korea Advanced Institute of Science and Technology from 1992 to 2008. His current research interests include low-power embedded systems, systems software, and virtual machines.

Jin-Soo Kim (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer engineering from Seoul National University, Republic of Korea, in 1991, 1993, and 1999, respectively. He is currently a Professor with the Department of Computer Science and Engineering, Seoul National University (SNU), South Korea. Before joining SNU, he was a Professor with Sungkyunkwan University from 2008 to 2018, and an Associate Professor with the Korea Advanced Institute of Science and Technology from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute from 1999 to 2002 as a Senior Member of the research staff, and with the IBM T. J. Watson Research Center as an Academic Visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.