StreamFTL: Stream-level Address Translation Scheme for Memory Constrained Flash Storage

Hyukjoong Kim, Kyuhwa Han, and Dongkun Shin
College of Software
Sungkyunkwan University, Suwon, Korea
{wangmir, hgh6877, dongkun.shin}@gmail.com

***Abstract*—Although much research efforts have been devoted to reducing the size of address mapping table which consumes DRAM space in solid state drives (SSDs), most SSDs still use page-level mapping for high performance in their firmware called flash translation layer (FTL). In this paper, we propose a novel FTL scheme, called StreamFTL. In order to reduce the size of the mapping table in SSDs, StreamFTL maintains a mapping entry for each stream, which consists of several logical pages written at contiguous physical pages. Unlike extent, which is used by previous FTL schemes, the logical pages in a stream do not need to be contiguous. We show that StreamFTL can reduce the size of the mapping table by up to 90% compared to page-level mapping scheme.**

I. Introduction

Flash storage systems such as solid state drives (SSDs) and eMMC have been widely used upon various computing systems. The widespread use of SSDs result from the higher performance of SSD compared to hard disk drives and the continuous decreasing in cost-per-bit of SSDs.

In order to lower the price of SSDs, many researchers have tried to reduce the size of the mapping table in the flash translation layer (FTL) [1], [2] which is a special firmware managing flash storage systems. Most recent SSDs use page-level mapping in their FTLs for high performance. However, page-level mapping requires a large amount of DRAM. For example, 8 TB of SSD requires 8 GB of DRAM for its mapping table since page-level mapping maintains 4 byte mapping entry for each 4 KB page. The block-level mapping or hybrid mapping [2], [3] can reduce the size of the mapping table by using a coarser-grained mapping. However, such FTLs have a large amount of write amplification due to their merge operations. We can reduce the required DRAM size of SSD by using a demand-loading scheme such as (DFTL) [1], which loads only a portion of the entire mapping table dynamically on a small DRAM cache. However, DFTL shows poor performance by cache misses. Moreover, even for read request, DFTL can invoke flash write operations for map loading and unloading.

The extent mapping FTL [4] uses a variable-sized mapping unit called *extent*, which has logically contiguous pages written at physically contiguous flash pages. An extent mapping entry consists of the start logical page number (LPN), the start physical page number (PPN), and the extent size. Since a single extent entry can map a large amount of sequentially written pages, the mapping table size can be reduced significantly.

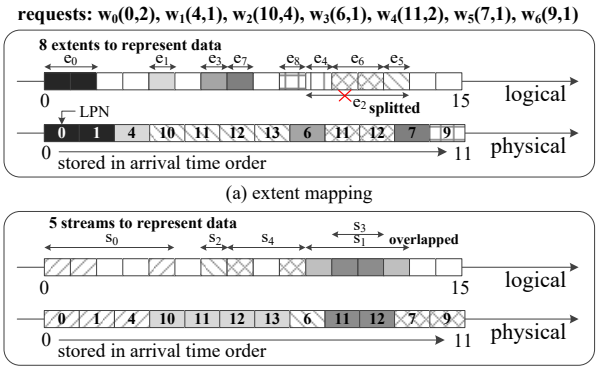

Fig. 1. Comparison between extent and stream (σ=10).

However, if a write request partially updates several pages within an extent, the extent must be split because the updated pages will be written at flash pages. The split operation generates many small-sized extents, and increases the number of mapping entries. Moreover, the maximum number of extent entries cannot be fixed, and thus we cannot reduce the size of DRAM. In order to solve this problem, μ-FTL [5] manages the entire mapping table on NAND flash memory using the μ-tree structure, and loads only a small number of map entries on-demand into DRAM. As a result, μ-FTL suffers from the map loading overhead like DFTL.

In this paper, we propose a novel FTL, called StreamFTL, which can drastically reduce the mapping table size while achieving comparable performance to page-level mapping. The StreamFTL maintains the stream-level mapping entries by exploiting the increasing tendency of address sequences of storage access requests. Whereas an extent used by μ-FTL consists of logically contiguous pages, a *stream* can include *logically non-contiguous* pages, which arrived in address-ascending order and are written at *physically contiguous* flash pages. StreamFTL allows each logical page to be stored at any flash page similarly at page-level mapping. We show that the StreamFTL can reduce the entire mapping table size by up to 90% compared to page-level mapping and thus the entire mapping table can be loaded into a small size of DRAM.

II. Motivation

A. Extent vs. Stream

Fig. 1(a) shows an example of extent mapping. By writing 12 pages, 8 extents are generated. For example, extent 0 (e_0) covers the LPNs from 0 to 1 and its PPNs are from 0 to 1. The

mapping entry of e_0 is (0, 0, 2), which represents start LPN, start PPN and page count, respectively. Notice that the logical address range covered by an extent cannot be overlapped by other extents. If a write request which partially updates an extent is submitted, the extent must be split into multiple extents. At the figure, write request *write*(10,4) makes e_2, but by the following write request *write*(11,2) e_2 was split into three extents e_4, e_5 and e_6. Therefore, the number of extents will be various depending on the write workload, and the maximum number of extents cannot be fixed.

The basic mapping unit in StreamFTL is *stream*, which is more flexible than extent. A stream includes several ascending-ordered logical pages which are written at physically-contiguous pages. In addition, stream satisfies the following condition, when a stream S_i contains pages $p_{i,1}$ to $p_{i,n}$:

$$S_i = \{p_{i,1}, p_{i,2}, ..., p_{i,n}\},\ where,\ L(p_{i,k}) < L(p_{i,k+1}),$$
$$P(p_{i,k}) + 1 = P(p_{i,k+1}),\ L(p_{i,n}) - L(p_{i,1}) \leq \sigma \quad (1)$$

$L(p)$, $P(p)$ are the LPN and the PPN of page p, respectively. σ represents the maximum stream size in the page unit. As the equation shows, the pages written with ascending-ordered LPNs can be represented as a single stream entry if the pages are stored continuously in physical flash pages. Thus, compared to an extent, a stream can cover more logical pages with a single mapping entry. Fig. 1(b) shows an example of stream mapping for the same write requests used in Fig. 1(a). We assumed σ is 10. From the figure, stream 0 (s_0) covers LPNs 0, 1 and 4, and its PPNs are from 0 to 2. The mapping entry will be like (0, 0, 0, 1, 4), start LPN, start PPN, first-third logical pages from start LPN[1], respectively.

The stream mapping can reduce the number of mapping entries compared to the extent mapping. Moreover, since StreamFTL allows overlapping between streams, the split operations are not required. Owing to the flexibility of stream mapping, the maximum number of stream entries can be fixed due to *stream merge* operation which periodically reclaims stream entries. In the case of extent mapping, since an extent covers only contiguous logical pages, it is difficult or might be impossible to reclaim free extents by merging several extents without re-allocating logical addresses of the data. However, StreamFTL can merge multiple streams if their logical address ranges can be covered by a stream. As a result, the mapping table of the StreamFTL can be fixed into a certain size.

B. Current Storage I/O Systems

In real storage systems, the write requests within a short time interval often has an *ascending-ordered* pattern in their logical addresses. In the Linux kernel, since the page cache buffers the file-backed data in DRAM with a page-sorted tree, the addresses of write requests tend to be ascending-ordered. When a legacy filesystem such as EXT4 is fragmented, the filesystem cannot allocate contiguous blocks for a file, thus the write requests will not be contiguous but ascending-ordered. When log-structured filesystems (LFSs) [6] such as F2FS [7] use the threaded logging scheme, the write pattern of LFS also will be similar to the above. Fig. 2 shows the average sizes of extent and stream collected by a simulation with MSR-Cambridge workloads [8]. We assumed the extent or stream size is infinite. As figure shows, stream covers more logical pages. Thus, it can be said that the real storage I/O pattern can benefit from StreamFTL.

[1]These information will be represented as a bitmap, 1 bit per each logical page. Section III-A will explain the details.

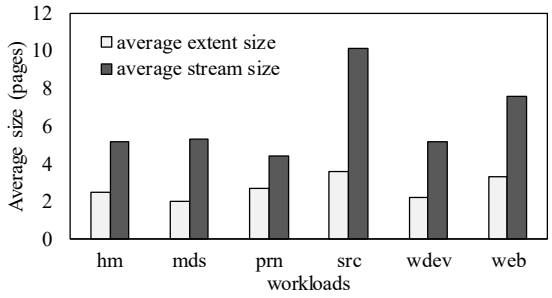

Fig. 2. Average size of extent/stream at MSR trace.

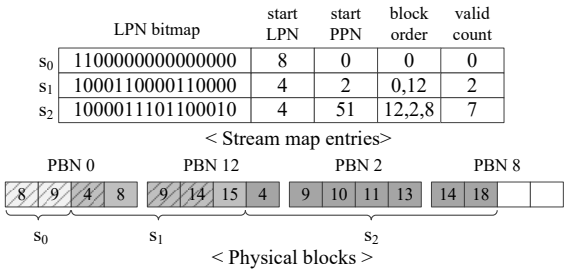

Fig. 3. Mapping entries of StreamFTL. Physical block contains physical pages, and the physical blocks are allocated in out-of-order [1].

III. StreamFTL Scheme

A. Mapping Method & Address Translation

Each stream maintains a mapping entry to map its several LPN elements. Fig. 3 shows an example of the mapping entries at StreamFTL. The bit 1 in the LPN bitmap represents the corresponding LPN which was written in the physical page of the stream. The PPN for an LPN can be calculated by two phase of address translation, `getStream` and `getPPN`. When trying to find the corresponding PPN for an LPN, StreamFTL first finds the stream of the LPN. With the start LPN and the LPN bitmap of a stream map entry, StreamFTL checks whether the target LPN exists in the stream. Since the logical address range of a stream can be overlapped by other streams, the data in most recently allocated stream is the correct data for the LPN. After finding the stream for the LPN, StreamFTL does `getPPN` operation in order to find the target PPN. First, StreamFTL calculates the logical offset within stream with the start LPN of the stream, and the number of bit 1s from the first bit location to logical offset location on the LPN bitmap.

For example, when searching the PPN for LPN 14 from Fig. 3, the first thing to do is finding correct stream for the LPN 14. Since s_2 is the latest stream whose logical offset (14 - 4) of LPN bitmap is set to 1, the data of LPN 14 is in s_2. Second, because the number of bit 1s from first bit to 11-th bit is 6, the physical offset of the LPN 14 in s_2 is 5. Finally, as figure shows, the target PPN for LPN 14 can be found from

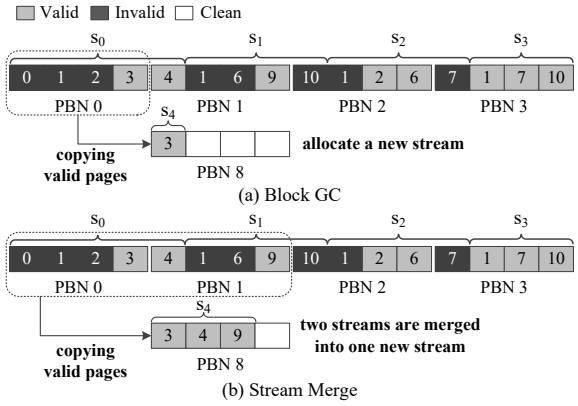

Fig. 4. Block GC and Stream merge.

0-th page of physical block number (PBN) 8. There are many algorithms and hardware instructions to count the number of bit 1s [9], hence StreamFTL can speed up the calculation with using these solutions.

The *block order* field in Fig. 3 stores the allocation order of physical blocks when multiple blocks are associated with a single stream. Since the first PBN can be calculated from start PPN, the stream map entry does not need to store the first one. The *valid page count* of the stream is used at the stream merge operation to find victim streams. In summary, StreamFTL only needs a stream size bitmap, and additional few bytes per each stream map entry compared to the 4 byte for each logical page on page-level mapping. The size of stream is predefined in order to fix the size of LPN bitmap. StreamFTL separates the logical address space into *partitions*, and streams are grouped into a partition based on their LPN. StreamFTL can simplify the stream searching operation at address translation or stream merging using partition.

Since the storage I/O pattern often has multiple working sets, if multiple active streams are managed, the stream utilization can be improved. In the result of Fig. 2, the average size of stream can be increased by about 3 times when there are 4 multiple active streams. Therefore, StreamFTL maintains multiple active blocks to preserve multiple append-in-progress streams. For example, if the number of active stream is 4 at Fig. 1, s_0 can cover the data of s_2 and s_4, thus improving the stream utilization.

StreamFTL allocates a new stream to serve write requests when there are no available active stream for the request, in other words, when new LPNs cannot be appended at the existing streams. When allocating a new stream, one of existing active stream must be closed. There are many ways to choose victim active stream, but currently we simply choose the oldest active stream as a victim stream. After allocating a new stream, the following write requests will be covered by that stream until closing the stream.

B. Garbage Collection & Stream Merge

Fig. 4 shows the behavior of the block garbage collection (bGC) and the stream merge (sMerge). The bGC of StreamFTL is similar to existing FTLs. The bGC selects victim physical blocks with the minimum number of valid pages, copies the valid pages to free blocks, and erase the

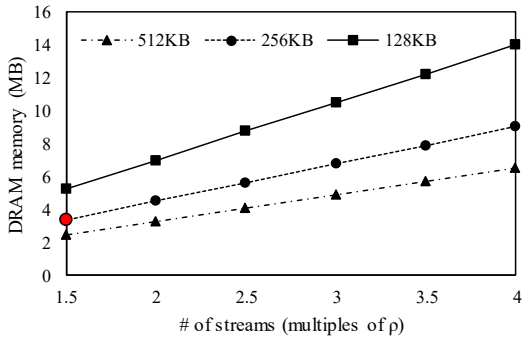

Fig. 5. Mapping table size according to stream number/size.

victim blocks. However, in StreamFTL, the copy operations of valid pages may generate additional streams. At Fig. 4(a), An additional stream s_4 must be generated to write the logical page 3 at flash block PBN 8.

The sMerge operation of StreamFTL selects victim streams, and the valid pages of the victim stream are appended to other streams. sMerge must select victim streams with many overlapping streams and fewer valid pages. Therefore, StreamFTL first selects victim partition by considering the number of streams and the number of valid pages on the partition.

One interesting problem is that there is a correlation between bGC and sMerge. bGC can invoke sMerge because bGC consumes streams, and sMerge also can invoke bGC because sMerge consumes flash blocks. Therefore, we need an integrated operation.

IV. Experimental Evaluation

The total size of the mapping table on StreamFTL is determined by the size of the stream and the maximum number of streams. Fig. 5 shows the size of the mapping table according to those parameters when the storage size is 32 GB. The maximum number of streams is represented as multiples of ρ, and ρ is the number of streams which covers entire storage space, and defined as $storage\ size/stream\ size$. Since each stream map entry consists of LPN bitmap and some additional bytes (startLPN, startPPN, block order, etc.), as the size of stream is growing, the total size of the mapping table will be close to 1/32 of DFTL's mapping table. However, if the size is too large, then the utilization of a stream will be low, hence it will cause more sMerge operations. On the other hand, if the maximum number of stream is too small, the size of the mapping table also will be small while the cost of sMerge will increase because StreamFTL must merge under-utilized streams to make a new stream.

Table I shows the mapping table size of various FTLs. The size of StreamFTL in the table is when the stream size is 256 KB and the maximum number of streams is 1.5 times of the storage capacity. As the table shows, StreamFTL can reduce the mapping table by up to 90% compared to page-level mapping FTL. Although DFTL and μ-FTL also can reduce the size of DRAM due to their demand-loading scheme, they cannot reduce the mapping table itself.

For evaluation, we implemented a trace-driven FTL simulator of StreamFTL and DFTL. We assumed the page size is

TABLE I
THE MAPPING TABLE SIZE OF FTLS.

32 GB storage	DRAM size	Mapping table size
Page-level mapping	32 MB	32 MB
DFTL [1]	configurable	32 MB
μ-FTL [5]	configurable	640 KB - 70 MB
StreamFTL (red dot)	3.25 MB	3.25 MB

4 KB and the number of pages in a physical flash block is 256. We set the size of DRAM cache at DFTL to be same to the required DRAM size of StreamFTL. MSR Cambridge workload is used for input workload.

Fig. 6 (a) compares the amount of operations by DFTL and StreamFTL when there are no GC. DFTL needs map loading/unloading for cache misses and thus invokes additional NAND operations, StreamFTL always outperforms DFTL when there are no GC. Even for a random read-intensive workload such as prn_1, StreamFTL performs well because it does not need demand loading of the mapping table. However, when the storage is utilization is high, StreamFTL may show bad performance due to sMerge overhead. We performed the same experiment with prn_1 at the aged storage, which is initialized with sequential write requests filling 50% of logical storage space, and random writes filling 30% of logical storage space. Since the initialization consumes almost all physical pages, any following writes can invoke GCs. Fig. 6(b) compares the amount of operations by DFTL and StreamFTL at different stream sizes (128 KB, 256 KB, and 512 KB). We did the experiments using different sizes of DRAM configurations. As the DRAM size grows, the maximum number of streams also increases. With the same size of DRAM, StreamFTL using a smaller stream size will have a smaller number of streams. When the number of stream is too small, or the size of stream is too large, the performance of StreamFTL is much worse than DFTL. Because sMerge and bGC occur frequently in StreamFTL. In contrast, as the size or number of stream increases, the StreamFTL performs better than DFTL since the reclamation overhead is reduced, while the size of mapping table is not reduced drastically.

V. DISCUSSION

This is our preliminary work to study the feasibility of the StreamFTL with a simulator. To improve the performance of StreamFTL scheme more works are remaining. First of all, the bGC and sMerge must be improved because they handle similar or duplicated works. Moreover, we need a more intelligent victim stream selection algorithm in sMerge operation. Second, because the LPN bitmap will be sparse with many 0 bits, especially when the stream size is big, we are to find an efficient compression algorithm for LPN bitmap to reduce the size of the mapping table. Lastly, we plan to implement StreamFTL as a host-level FTL for Open-Channel SSD [10]. Since Open-Channel SSD runs the FTL at the host OS, the mapping table must be managed at the host system which consumes much expensive host DRAM. Moreover, the demand loading overhead of DFTL will be worse at Open-Channel SSDs because the demand loading operation incurs more I/O traffic between host system and SSD. Therefore the benefit of StreamFTL will be larger at Open-Channel SSDs.

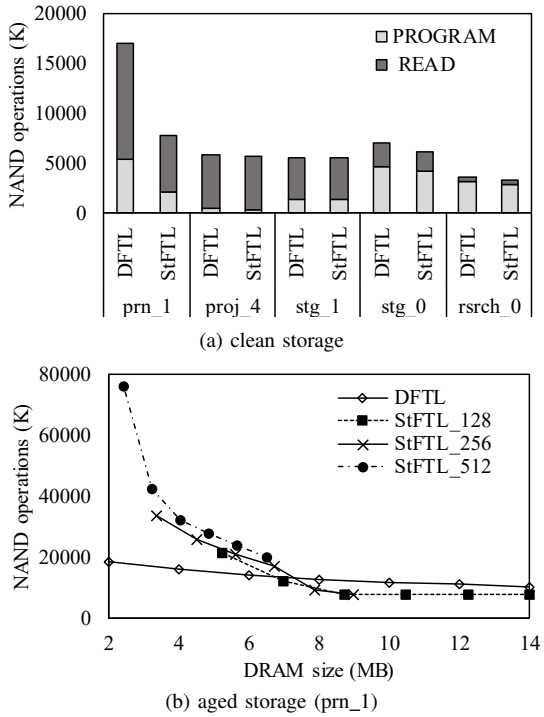

Fig. 6. Amount of NAND operations on clean/aged storage.

VI. ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP). (No. 2016R1A2B2008672)

REFERENCES

[1] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. of the 14th international Conference on Architectural Support For Programming Languages and Operating Systems*, ser. ASPLOS '09, 2009, pp. 229–240.

[2] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compact flash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, 2002.

[3] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, 2007.

[4] L.-P. Chang and T.-W. Kuo, "Efficient management for large-scale flash-memory storage systems with resource conservation," *ACM Transactions on Storage (TOS)*, vol. 1, no. 4, pp. 381–418, 2005.

[5] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim, "μ-ftl: a memory-efficient flash translation layer supporting multiple mapping granularities," in *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 2008, pp. 21–30.

[6] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[7] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST '15.

[8] "MSR Cambridge Traces," http://iotta.snia.org/traces/388.

[9] P. Wegner, "A technique for counting ones in a binary computer," *Communications of the ACM*, vol. 3, no. 5, p. 322, 1960.

[10] M. Bjørling, J. González, and P. Bonnet, "Lightnvm: The linux open-channel ssd subsystem." in *FAST*, 2017, pp. 359–374.