JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE, VOL.17, NO.5, OCTOBER, 2017 ISSN(Print) 1598-1657
https://doi.org/10.5573/JSTS.2017.17.4.648 ISSN(Online) 2233-4866

# FineDedup: A Fine-grained Deduplication Technique for Extending Lifetime of Flash-based SSDs

**Taejin Kim[1], Sungjin Lee[2*], and Jihong Kim[1]**

*Abstract*—**Data deduplication is an effective solution in improving the lifetime of flash-based solid-state drives (SSDs) by preventing redundant data from being written to flash memory. Existing deduplication techniques for SSDs, however, fail to fully eliminate potential redundant data because of their coarse-grained granularity. In this paper, a fine-grained deduplication technique for SSDs, called FineDedup, is proposed to improve the likelihood of eliminating redundant data. FineDedup also resolves technical difficulties caused by its finer granularity, i.e., increased memory requirement and read response time. The results show that FineDedup reduces the amount of written data by up to 24% over existing techniques with negligible.**

*Index Terms*—**NAND flash memory, solid state disks, data deduplication, lifespan, reliability**

## I. INTRODUCTION

As the price-per-byte of NAND flash memory is rapidly decreasing, NAND flash-based solid-state drives (SSDs) are emerging as attractive solutions for various consumer products such as laptops, smart phones and smart pads. However, as NAND flash memory technology scales down to 20-nm and below, storing data reliably in NAND flash memory is one of key design challenges of NAND flash-based storage systems. For example, the number of program/erase (P/E) cycles allowed for each block is significantly reduced in recent triple-level cell (TLC) NAND technology. While older 5x-nm single-level cell (SLC) NAND flash memory can support about 10 K P/E cycles, recent 2x-nm TLC NAND flash memory can barely support about 1 K P/E cycles [1, 2]. Particularly, the reduction in the number of P/E cycles of NAND flash memory seriously limits the overall lifetime of flash-based SSDs, making it difficult for SSDs to be used in write-intensive applications.

In order to extend the lifetime of flash-based SSDs, data deduplication techniques have been used in recent SSDs because they are effective in reducing the amount of data written to flash memory by preventing duplicate data from being written again [3-6]. As a result, only non-duplicate data, i.e., unique data, are stored in SSDs, thus effectively decreasing the total amount of data written to SSDs. In most deduplication schemes proposed for SSDs, the unit of data deduplication is the same as the flash page size which is usually 4 KB or 8 KB. Using a page as a deduplication unit seems to be reasonable because the unit of a read or write operation of flash memory is also a page. However, this page-based deduplication technique misses many chances of eliminating duplicate data, especially when two pages are almost identical. For example, in the experimental analysis of an existing 4 KB page-based deduplication technique, it is observed that up to 34% mostly identical data. If the unit of deduplication were smaller than 4 KB, about 23% more data could be identified as duplicate data. Furthermore, it is expected that the effectiveness of the page-based deduplication technique would get even worse in future NAND flash memory as the page size of

Manuscript received Dec. 27, 2016; accepted Aug. 27, 2017
[1] Dept. of Computer Science and Engineering, Seoul National Univeristy, 1 Gwanak-ro, Gwanak-gu, Seoul, 08826, Korea
[2] Dept. of Information and Communication Engineering, DGIST, 333 Techno Jungang-daero, Hyeonpung-myeon, Dalseong-gun, Daegu, 42988, Korea
E-mail : sungjin.lee@dgist.ac.kr

flash memory is expected to increase to a bigger size such as 8 KB or 16 KB [1, 2].

This paper proposes a fine-grained deduplication technique for flash-based SSDs, called FineDedup. It is different from other existing deduplication techniques in that it increases the likelihood of finding duplicates by using a finer deduplication unit which is smaller than a single page (e.g., one fourth of a single page). With a smaller deduplication unit, many data segments within a page can be detected as a duplicate one, so the amount of data written to flash memory can be reduced regardless of a physical page size. To the best of our knowledge, this is the first work that uses the fine-grained approach for device-level deduplication.

In order to effectively incorporate fine-grained deduplication into flash-based SSDs, two key technical issues must be addressed properly. First, fine-grained deduplication requires a larger memory space than a coarse-grained one because it needs to keep more metadata in memory to find small-size duplicate data. Second, in fine-grained deduplication, unique data segments from partially duplicated pages can be scattered across several physical pages, which may seriously degrade the overall read performance. The proposed FineDedup technique is designed to take full advantage of fine-grained deduplication with small memory overhead as well as a low read performance penalty. The evaluation results show that FineDedup prolongs the lifetime of SSDs by up to 24% over page-based deduplication while requiring a negligible memory space increase. This improvement comes with a less than 5% read performance penalty over page-based deduplication.

The rest of the paper is organized as follows. Section II briefly reviews the existing deduplication techniques for SSDs. The main motivation of FineDedup is presented in Section III. The proposed FineDedup technique is described in detail in Section IV, and then its effectiveness is evaluated using real-world traces in Section V. Finally, Section VI concludes with a summary.

## II. RELATED WORKS

Because of the "erase-before-write" nature of NAND flash memory, flash storage devices employ a flash translation layer (FTL) that supports address mapping, garbage collection, and wear-leveling algorithms. These firmware algorithms incur a lot of extra write/erase operations, seriously shortening the overall lifetime of a storage device. For this reason, a large number of studies have been focused on reducing such extra operations to improve the storage lifetime. However, considering the decreasing lifetime of recent high-density NAND flash memory such as TLC NAND flash memory [1, 2], more aggressive lifetime management solutions are required.

Data deduplication techniques, which are originally developed for backup systems, are regarded as one of the promising approaches for extending the storage lifetime because of their ability that reduces the amount of write traffic sent to a storage device. In deduplication techniques, a chunk is used as a unit of identification and elimination of duplicate data. Depending on their chunking strategies, deduplication techniques can be categorized into two types, fixed-size deduplication and variable-size deduplication. Fixed-size deduplication divides an input data stream into fixed-size chunks (e.g., pages) [3-6]. Then, it decides if each chunk data is duplicated and prevents duplicate chunks from being rewritten to flash memory. Unlike fixed-size deduplication, the chunk size of variable-size deduplication is not fixed. Instead, it decides a cut point between chunks using a content-defined chunking (CDC) algorithm which divides the data stream according to the contents.

In general, variable-size deduplication techniques can identify more data as duplicate data than the fixed-size deduplication technique. Since variable-size deduplication adaptively changes the size of chunks by analyzing the contents of input stream, duplicate data are more effectively found regardless of their locations. There are several works that exploit variable-size deduplication for system-level research [7, 8]. For SSD-level deduplication, however, fixed-size deduplication is commonly used because of the following practical reasons.

First, the CDC algorithm often requires relatively high computational power and a large amount of memory space. Thus, variable-size deduplication is not appropriate to be employed at the level of storage devices where computing and memory resources are constrained. Second, the size of remaining unique data after deduplication may vary in variable-size deduplication. When writing those data, a complicated scheme for data

size management is required to form sub-page data chunks to fit in a flash page size, preventing an internal fragmentation. For those reasons, most existing deduplication techniques for SSDs employ fixed-size deduplication, which is relatively simple and does not require a significant amount of hardware resources.

There are several existing studies for fixed-size deduplication for SSDs. F. Chen [3] proposed CAFTL to enhance the endurance of SSDs with a set of acceleration techniques to reduce runtime overhead. W. Li [4] also proposed CA-SSD to improve the reliability of SSDs by exploiting the value locality, which implies that certain data items are likely to be accessed preferentially. In these studies, authors focused on the feasibility of deduplication at SSD level and proved its effectiveness rather than improving deduplication itself.

Recently, several deduplication techniques for flash-based storage are proposed. Z. Chen [5] proposed OrderMergeDedup which orders and merges the deduplication metadata with data writes to realize failure-consistent storage with deduplication. G. Narasimhan [6] proposed CacheDedup which integrates deduplication with caching architecture to address limited endurance of flash caching by managing data writes and deduplication metadata together, and proposing duplication-aware cache replacement algorithms. These studies focus on systematic approach such as block layer or flash caching. However, this study improves the effect of deduplication in the device-specific domain, so the approach of this study is quite different.

Similar to the existing deduplication techniques, the proposed FineDedup technique is also based on fixed-size deduplication. Using a smaller deduplication unit, however, FineDedup improves the likelihood of eliminating duplicate data. This approach can complement the limitation of existing fixed-size deduplication techniques, which exhibit a relatively low amount of removed writes in comparison with variable-size deduplication.

## III. MOTIVATIONS

Existing deduplication techniques for SSDs use a single page as a chunk for data deduplication [3-6]. The write-requested page is identified whether the contents of the page have already been written and is written to flash memory only if there is no existing duplicate page. When a write-requested page is the exact duplicate of a previously written page, the requested page is not written to flash memory; only the corresponding entry for a mapping table (between the logical address and physical address) is updated. On the other hand, if there is no existing page duplicate in flash memory whose contents are the same as those of requested one, the requested page has to be written to flash memory. However, even for these unique pages, if their redundancy is checked at a sub-page level, say at a quarter of the page size, many sub-pages of these unique pages can be identified as redundant data. In existing techniques based on page-level deduplication, therefore, many duplicate data are written to flash memory even though the same data chunks have already been written.

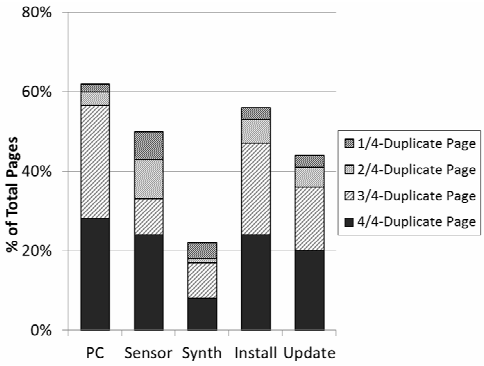


**Fig. 1.** The percentage of pages according to their partial duplicate patterns.

In order to better understand the effect of fine-grained deduplication on the amount of identified duplicate data, how many more chunks can be identified as redundant is analyzed when the chunk size gets smaller than a single page. For the evaluation, five I/O traces, **PC**, **Sensor**, **Synth**, **Install**, and **Update** are used. They are explained in Section V. In the evaluation, the page size was 4 KB and the chunk size was set to 1 KB. Fig. 1 shows the percentage of the page writes from host, classified by their partial duplicate patterns. It is denoted that a page is an n/4-duplicate page when n chunks of the page are duplicate chunks. A 4/4-duplicate page is a duplicate page at the page level. In the existing page-based deduplication, only 4/4-duplicate pages can be identified as a duplicate page. As shown in Fig. 1, 4/4-duplicate pages account for only 8% - 28% of total requested pages. For partially duplicate pages, i.e., 1/4-, 2/4- and 3/4-

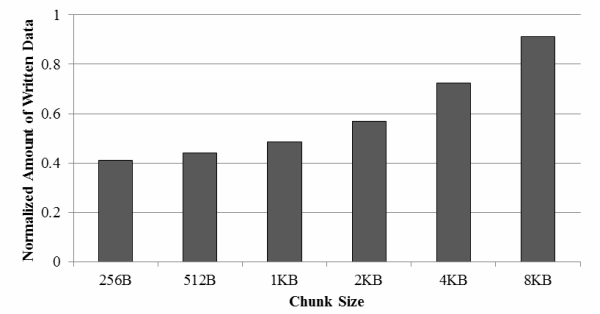


**Fig. 2.** The amount of written data under varying chunk sizes in PC workload.

duplicate pages, the page-based deduplication technique is useless. As shown in Fig. 1, pages with 1-3 duplicate chunks account for 14% - 34%. This means that many duplicate data are unnecessarily written to flash memory due to the large chunk size.

It is also investigated that the amount of data that can be eliminated by data deduplication while varying the chunk sizes from 256 B to 8 KB. As shown in Fig. 2, when the chunk size is 1 KB, the amount of data written to flash memory is reduced by 33% over when the chunk size is 4 KB. In particular, when the size of a chunk is 8 KB (i.e., when the physical page size is assumed to be 8 KB), only 10% of requested data are eliminated by data deduplication. This effectively shows that, as the size of a page increases, the overall deduplication ratio, i.e., the percentage of identified duplicate writes, decreases significantly. Since the physical page size of NAND flash memory is expected to increase as the semiconductor process is scaled down [1, 2], it is expected that the deduplication ratio of the existing deduplication technique will be significantly decreased in near future. In order to resolve this problem, the deduplication chunk size of deduplication techniques needs to be smaller than a page size. As depicted in Fig. 2, the deduplication ratio is saturated when the chunk size is 1 KB. Thus, it is used as a default chunk size in the rest of this paper.

## IV. FINE-GRAINED DEDUPLICATION

In this section, the proposed FineDedup technique is described in detail. The overall architecture of FineDedup is explained first and how FineDedup handles read and write requests is described in Section IV.1. In Section IV.2 and IV.3, it is introduced that a read

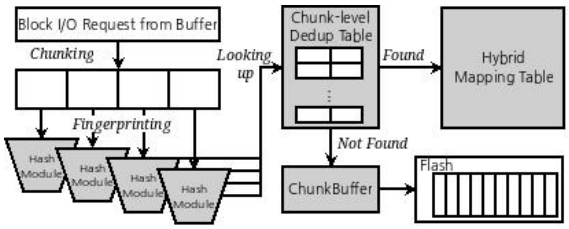


**Fig. 3.** An overview of the proposed FineDedup technique.

performance penalty and memory overheads caused by FineDedup, respectively, and how these problems can be resolved in FineDedup is explained.

### 1. Overall Architecture of FineDedup

Fig. 3 shows an overall architecture of FineDedup with its main components. Upon the arrival of a write request, FineDedup stores requested data temporarily in an on-device buffer. When the requested data are evicted from the buffer, FineDedup divides the data into several chunks which is smaller than a page size.

For each chunk, FineDedup computes a fingerprint, using a collision-resistant hash function. In this work, an MD6 hash function [9], which is one of the well-known cryptographic hash function is used. A fingerprint is used as a unique ID that represents the contents of a chunk. FineDedup has to compute more fingerprints than the existing deduplication schemes because of its small chunk size. To reduce the hash calculation time, FineDedup uses multiple hardware-assisted hash engines for parallel hash calculations. In Fig 3, for example, 4 hardware accelerators are used for fingerprinting. In the FPGA (ML605) implementation of the MD6 hash function, it took about 10 μs to compute a fingerprint using a hardware accelerator. Thanks to the multiple hardware hash functions, the fingerprinting can be executed in parallel, so the elapsed time for calculating fingerprinting is maintained the same even when the number of chunks increases. Furthermore, the execution time and the hardware resources can be reduced further if the hash function is implemented by ASIC logics. Considering that NAND flash memory has a long write latency (e.g., 1.2 ms) and such latency is avoided whenever duplicate data is found, the time overhead of computing fingerprints can be considered negligible. For the read request handling, FTL finds the physical page location of the requested logical address by referencing

the mapping table so the fingerprinting does not affect the read request.

After fingerprinting, each fingerprint is looked up in the dedup table which maintains the fingerprints of the unique chunks previously written to flash memory. Each entry of the dedup table is composed of a key-value pair, (*fingerprint*, *location*), where the location indicates a physical address in which the unique chunk is stored.

If the same fingerprint is found, it is not necessary to write the chunk data because the same chunk is already stored in flash memory. Instead, FineDedup updates the mapping table so that the corresponding mapping entry points to the unique chunk previously written. Note that searching fingerprints quickly is out of focus in this work. If necessary, any optimization process similar to existing deduplication techniques for quick search can be also applied to this work [3].

Unlike existing page-based deduplication techniques, FineDedup handles all the data in the unit of a chunk. For this reason, FineDedup must maintain a chunk-level mapping table that maps a logical chunk address to a physical chunk in flash memory. Because of its finer mapping granularity, the chunk-level mapping table is much larger than the existing page-level mapping table. To reduce the memory space for maintaining the chunk-level mapping table, FineDedup uses a demand-based hybrid mapping strategy, which is described in Section IV.3 in detail.

If there is no matched fingerprint in the dedup table, FineDedup stores the chunk data in a *chunk buffer* temporarily. This temporary buffering is necessary because the unit of I/O operations of flash memory is a single page. The chunk buffer stores the incoming chunk data until there are four chunks, and evicts them to flash memory at once. FineDedup then updates the mapping table so that the corresponding mapping entries indicate newly written chunks. The new fingerprints of the evicted chunks are finally inserted into the dedup table with their physical location.

When a read request arrives, FineDedup reads all the chunks that belong to the requested page from flash memory, and then transfers the read data to the host system. In FineDedup, four chunks in the same logical page can be scattered across different physical pages. In that case, multiple read operations are required to form the original page data, which in turn significantly increases the overall read response time.

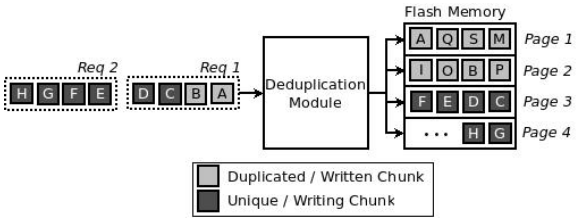


**Fig. 4.** Data fragmentation caused by FineDedup.

### 2. Read Overhead Management

As mentioned previously, FineDedup may increase the read response time significantly. The main cause of the read performance degradation is data fragmentation which occurs when data chunks belonging to the same logical page are broken up into several physical pages.

Fig. 4 illustrates why data fragmentation occurs in FineDedup. There are two page write requests, *Req 1* and *Req 2*, in Fig. 4. *Req 1* consists of four chunks, ‘A’, ‘B’, ‘C’, and ‘D’, and *Req 2* is also composed of four chunks, ‘E’, ‘F’, ‘G’, and ‘H’. Since ‘A’ and ‘B’ of *Req 1* are duplicate chunks, only ‘C’ and ‘D’ need to be written to flash memory. Thus, writes for two chunks ‘A’ and ‘B’ can be reduced. Suppose that there is a read request for the page written by *Req 1* after the chunks of *Req 1* are written to flash memory. In that case, FineDedup has to read three pages, i.e., *page 1*, *page 2*, and *page 3*, from flash memory to form the requested data. The read performance penalty can also occur even when there are no duplicate chunks in the requested page. For example, in Fig. 4, *Req 2* has no duplicate chunks in flash memory, thus all the chunks belonging to *Req 2* being written to flash memory. Because a single page write requires four data chunks, ‘E’ and ‘F’ of *Req 2* are written to *page 3* together with ‘C’ and ‘D’, and ‘G’ and ‘H’ will be written to *page 4* with other data chunks, as shown in Fig. 4. Thus, when the data written by *Req 2* are read later, both *page 3* and *page 4* must be read from the flash memory.

One of the feasible approaches that mitigate the read performance overhead is to employ a chunk read buffer. It is observed that the access frequencies of unique chunks are greatly skewed; that is, a small number of popular chunks account for a large fraction of the total accesses to unique chunks in flash memory. For example, according to the analysis under real-world workloads, top

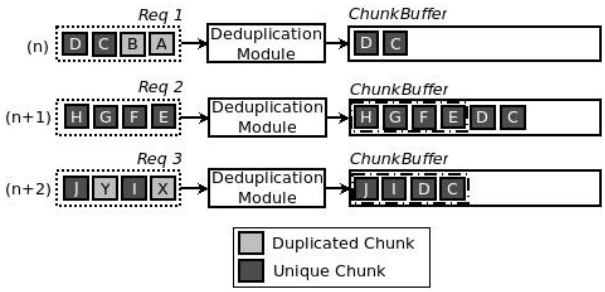


**Fig. 5.** A packing scheme in the chunk buffer.

10% of the unique chunks serve almost 70% of the total data read by a host system. By keeping frequently accessed chunks in a chunk read buffer, therefore, FineDedup can reduce a large number of page read operations sent to flash memory.

On the other hands, it is also observed that about 39% of read requests to unique pages actually require two page read operations. In order to further reduce this read performance penalty, FineDedup uses a chunk packing scheme. The key idea of this scheme is to group chunks belonging to the same logical page in the chunk buffer and then write them to the same physical page together.

Fig. 5 shows an example of the chunk packing scheme when three page write requests, *Req 1*, *Req 2*, and *Req 3*, are consecutively issued from a host system. *Req 1* contains two duplicate chunks 'A' and 'B' and two unique chunks 'C' and 'D'. As expected, only 'C' and 'D' out of four chunks are sent to the chunk buffer. The next request *Req 2* does not have any duplicate chunks, so all of them are moved to the chunk buffer. As depicted in Fig. 5, the chunks 'E', 'F', 'G', and 'H' belong to the same logical page and form single page data. Thus, FineDedup writes them to flash memory together, leaving the chunks 'C' and 'D' in the chunk buffer. When *Req 3* is issued with two more unique chunks 'I' and 'J', 'C' and 'D' along with 'I' and 'J' are written to flash memory. All those chunks can be written to the same physical page together because every chunk of each request is not broken up into two pages.

Note that the main objective of this scheme is to prevent chunks of a unique request to be scattered across multiple pages avoiding unnecessary data fragmentation. In order to directly insert an incoming unique request to chunk buffer, page-sized free space should be managed to be always available in the buffer. When there is no free space for the next request without any suitable chunks of requests to form a single page, the chunks of a partially duplicate request should be broken up into two pages. Most partially duplicated requests, however, are 3/4-Duplicate pages as shown in Fig. 1, which means there are many requests of one unique chunk in the chunk buffer. Therefore, it is expected that most requests will be written to the same page even when the size of the chunk buffer is not large since it is not quite difficult to find an appropriate chunk to fit a flash page. It has been observed that the effectiveness of the chunk buffer does not significantly increase when its size is more than 8 KB which is twice larger than that of a flash page. The size of the chunk buffer was thus set to 8 KB for the evaluation. In the above example, if it is assumed that the chunk buffer can contain 8 chunks and *Req 3* has three unique chunks, only two chunks of *Req 3* will be written along with existing chunks, 'C' and 'D', leaving the other chunk in chunk buffer. A large chunk buffer provides more chance to avoid the request scattering.

Remaining data in the chunk buffer could be lost when a power failure occurs. Recent SSDs have a large DRAM cache (e.g., 256 - 512 MB) and use it as a device buffer. Moreover, they support internal cache power protection through the use of supercapacitors to flush out information in DRAM to flash memory at the event of power failure. In order to keep the reliability in FineDedup, the remaining data in the chunk buffer can be stored to flash memory during power protection procedure as well as the mapping information of the written page. In conclusion for chunk buffer design, there is a trade-off between potential read performance and reliability depending on the chunk buffer size. The size of the chunk buffer, hence, should be determined according to the characteristics of workloads.

## 3. Memory Overhead Management

As mentioned in Section IV.1, FineDedup handles requested data in the unit of a chunk. Therefore, FineDedup must maintain a chunk-level mapping table that maps a logical chunk address to a physical chunk address in flash memory. Since the size of a chunk is smaller than that of a page, a chunk-level mapping table is much larger than the page-level mapping table. For example, if the page size is 4 KB and the chunk size is 1 KB, the size of a chunk-level mapping table is four times

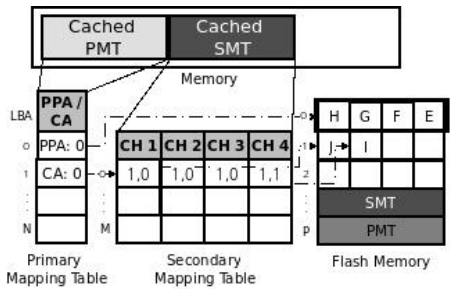


**Fig. 6.** An overview of the demand-based hybrid mapping table.

larger than that of a page-level mapping table.

In order to reduce the amount of memory space required for a mapping table, FineDedup employs a hybrid mapping table which is composed of two types of mapping tables: a page-level mapping table and a chunk-level mapping table. As depicted in Fig. 1, fully duplicate pages (4/4 duplicate) and unique pages still account for a considerable proportion of the total written pages. For these pages, the page-level mapping is more appropriate because they can be directly mapped to corresponding pages in flash memory. The chunk-level mapping is required only for partially duplicate pages.

Fig. 6 shows the overall architecture of the hybrid mapping table used in FineDedup. The primary mapping table (PMT) is maintained in the page level while the secondary mapping table (SMT) is maintained in the chunk level. The entry of the PMT is either a physical page address (PPA in Fig. 6) in flash memory or an index of the SMT (chunk address (CA) in Fig. 6).

If the chunk-level mapping is not necessary for a requested page, for example, unique page or fully duplicate page, the corresponding entry of the PMT directly points to the physical address of the newly written page or existing unique page in flash memory, respectively. On the other hand, if a partially duplicate page is requested for writing, FineDedup allocates a new entry in the SMT. As depicted in Fig. 6, each entry of SMT is composed of four fields, each of which points to the physical chunk address in flash memory. FineDedup then updates the new entry so that each field points to the physical chunk address. The corresponding entry of the PMT indicates the newly allocated entry of the SMT.

Using the hybrid mapping table, FineDedup can reduce the amount of memory space for keeping the mapping table. However, the problem of this hybrid mapping approach is that the size of the mapping table can greatly vary according to the characteristics of workloads. For example, if some workloads have many partially duplicate pages, the size of the SMT gets too big. On the other hand, if workloads mostly have unique pages or duplicate pages, it can be very small. Thus, the hybrid mapping table cannot be directly adopted in real SSD devices whose DRAM size is usually fixed. To overcome such a limitation, FineDedup adopts a demand-based mapping strategy in which the entire chunk-level mapping table is stored in flash memory while caching only a fixed number of popular entries in DRAM memory. The Cached PMT and Cached SMT in Fig. 6 represent the cached versions of the PMT and SMT, respectively.

It has been known that the demand-based mapping requires extra page read and write operations during the address translation [10]. For instance, if a mapping entry for a chunk to be read is not found in the in-memory mapping table, that entry must be read from flash memory while evicting a victim mapping entry to flash memory. The temporal locality present in workload, however, helps keep the number of extra operations small. The mapping information of requests issued in similar times will be stored in the same flash page. Once a mapping page is loaded in memory, hence, most requests issued in similar times are serviced from the mappings in memory.

## V. EXPERIMENTAL RESULTS

In this section, it is described that the experimental settings and explained the benchmarks used for the evaluation in detail. Then the effect of the proposed FineDedup technique is assessed on the SSD lifetime. Finally, it is analyzed that the read performance penalty and the memory overhead caused by FineDedup.

### 1. Experimental Settings

In order to evaluate the effectiveness of FineDedup, the experiments are performed using a trace-driven simulator with the I/O traces collected under various applications. The trace-driven simulator modeled the basic operations of NAND flash memory, such as page read, page write and block erase operations, and included several flash firmware algorithms, such as garbage

**Table 1.** A summary of traces used for the evaluations

| Trace | Description | Writes | Reads |
|---|---|---|---|
| PC | Web surfing, emailing and editing document, etc. | 3.1 GB | 40 MB |
| Sensor | Storing semiconductor fabrication sensor data | 2.6 GB | 66 KB |
| Synth | Synthesizing hardware modules | 2.5 GB | 70 MB |
| Install | Installing & executing programs (office, DB) | 4.9 GB | 119 MB |
| Update | Updating & downloading software packages | 3.5 GB | 103 MB |
| M-media | Downloading & playing multimedia files | 3.2 GB | 36 MB |

collection and wear-leveling. The proposed FineDedup technique and the existing deduplication techniques were also implemented in the simulator. For trace collection, the Linux kernel 2.6.32 is modified and I/O traces are collected at the level of a block device driver. All the I/O traces include detailed information about the I/O commands sent to a storage device (e.g., the type of requests, logical block addresses (LBA), the size of requests, etc.) as well as the contents of the data sent to or read from a storage device. The I/O traces are recoded while running various real-world applications. The detailed descriptions of these I/O traces are summarized in Table 1.

## 2. Effectiveness of FineDedup

As FineDedup exploits the duplicated chunks smaller than a page, the effectiveness of FineDedup is determined by the ratio of duplicated chunks of the workloads. In this section, it is described that how much duplicated data are eliminated by the FineDedup compared to the existing scheme.

Fig. 7 shows the amount of data written to flash memory by FineDedup over the existing scheme. The results shown in Fig. 7 are normalized to *RAW_req*, which represents the total amount of data written to flash memory without data deduplication. It is assumed that the existing page-based deduplication technique as a baseline case. The baseline is denoted by *BL_4KB* for a 4 KB flash page and *BL_8KB* for an 8 KB flash page. FineDedup technique is denoted by *FD_4KB* and *FD_8KB* for a 4 KB flash page and an 8 KB flash page, respectively. The chunk size in FineDedup is set to 1 KB for a 4 KB flash page and 2 KB for an 8 KB flash page.

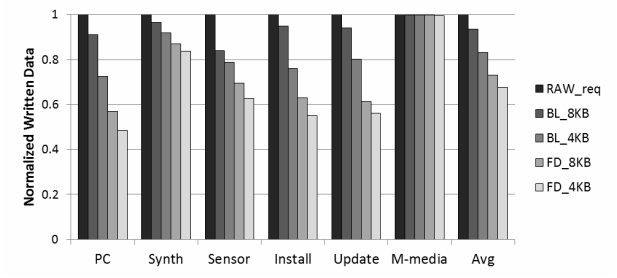


**Fig. 7.** The amount of written data under various schemes.

In order to see the effectiveness of FineDedup with various chunk sizes, the chunk size for an 8 KB flash page is set as 2 KB. In addition, by selecting 1 KB and 2 KB as the chunk size for 4 KB and 8 KB page, respectively, the number of chunks per page is the same for each case, which results the same number of fingerprints calculations.

As shown in Fig. 7, the effectiveness of deduplication techniques is highly workload-dependent. The amount of data eliminated by the deduplication technique notably increases when FineDedup is applied in most of the traces except M-media. When the chunk size is set as one fourth of the flash page size, FineDedup removes on average 16% more duplicate data over *BL_4KB* for a 4 KB flash page. Particularly, FineDedup saves 24% flash writes over *BL_4KB*. For an 8 KB flash page, it removes more duplicate data, on average by 23% over the existing technique. For **PC**, FineDedup saves 37% flash writes over *BL_8K*.

As expected, the benefit of FineDedup mainly derives from the decreased chunk size because it increases the probability of finding and eliminating duplicate data. Especially, **PC** trace shows a large number of update requests with little different data, so FineDedup can effectively identify unchanged data as duplicate while existing deduplication technique regards them as unique data. For the **M-media** trace, it is extremely difficult to find duplicate data because the data were already highly compressed. Thus, both the existing deduplication techniques and FineDedup are not effective in reducing the amount of data written to flash memory.

## 3. Read Overhead Evaluation

As explained in Section IV.2, fine-grained chunking in FineDedup may increase read response time. In this work,

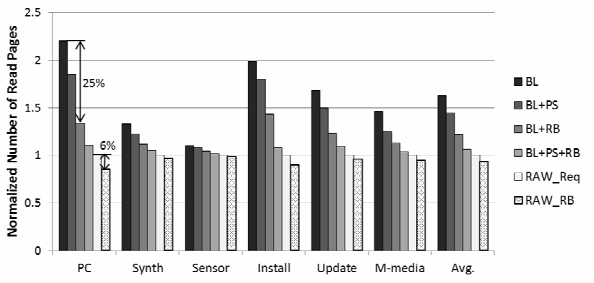


**Fig. 8.** The number of page read operations.

the number of read operations was used as the metric of the read overhead. In terms of the response time, the number of read operations can be regarded as the worst case response time since it cannot reflect parallelism.

Fig. 8 shows the normalized number of page read operations compared with the number of read requests in the workloads. *RAW_Req* indicates the number of original page read requests. *BL* refers to the number of page read operations of the baseline FineDedup without employing proposed optimization schemes. *BL+PS*, *BL+RB* and *BL+PS+RB* indicate the number of page reads of FineDedup with the proposed packing scheme, the chunk read buffer, and both, respectively. The size of the chunk read buffer was set to 8 MB and the chunk buffer size was set to 200 KB. *RAW_RB* indicates the number of page reads when *RAW_Req* has an additional 8 MB read buffer, which is the same size as BL+RB, and it is managed by the LRU policy. As shown in Fig. 8, employing the chunk read buffer is more effective than the packing scheme for reducing additional page read operations in most workloads. This is because the packing scheme is only effective for the requests containing no duplicate chunks whereas the chunk read buffer can absorb most of the read requests to frequently accessed chunks. FineDedup with both the packing scheme and chunk read buffer incurs on average less than 5% of additional read operations over the existing deduplication technique.

Compared with the baseline policy with the 8 MB read buffer (i.e., *RAW_RB*), FineDedup reads about 10% more pages which are still small enough. One interesting observation here is that there is only 6% improvement after adding the 8 MB buffer to *RAW_Req*. This performance gain is quite marginal compared with huge improvement in FineDedup – FineDedup exhibits 25% better performance with the same amount of DRAM. In

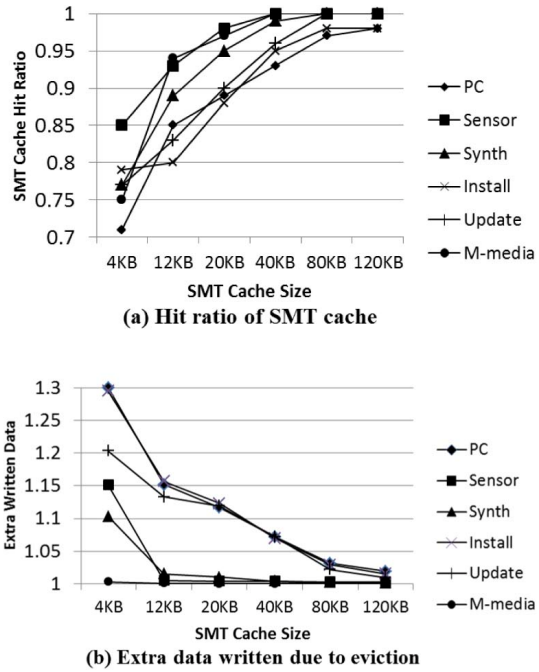


**Fig. 9.** The effectiveness of the demand-based hybrid mapping table in FineDedup with various cache sizes.

our observation, this is because FineDedup deduplicates the contents in DRAM, and thus provides larger effective DRAM capacity. The detailed analysis is given in Section V.6.

## 4. Memory Overhead Evaluation

As explained in Section IV.3, chunk level mapping table requires large memory space to handle partially duplicate pages. In FineDedup, the demand-based hybrid mapping table is proposed to reduce the required memory size for a mapping table without performance degradations.

In Fig. 9, the effectiveness of the proposed mapping table is evaluated in terms of the hit ratio and the amount of additional written data with various memory sizes for the cache. Since the demand-based PMT of the hybrid mapping table in FineDeup is the same approach as the DFTL [10], which is a well-known demand-based scheme to exploit the page-level mapping, the overhead of PMT can be estimated from the overhead of DFTL. Thus, in order to focus on the overhead of the SMT, it is assumed that DFTL is employed as the baseline mapping scheme in the evaluation.

Fig. 9(a) shows the hit ratio of the cached SMT. With a

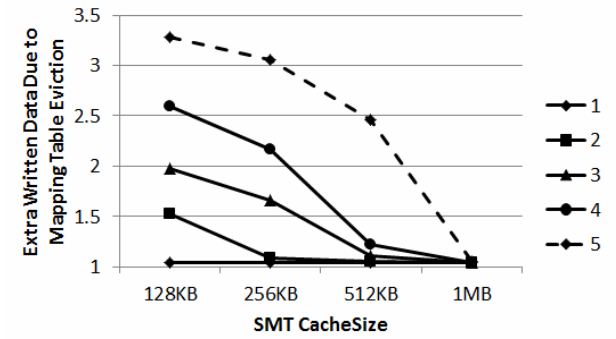


**Fig. 10.** The amount of extra written data due to mapping table eviction for the mixed workload.

120 KB cache, more than 95% of the mapping table accesses are absorbed. In addition, Fig. 9(b) shows extra written data caused by the evicted page entries from the SMT cache. Since mapping table accesses occur in the middle of read/write operations, it is important to reduce the amount of written data from evicted page entries in terms of read/write performance. Similar to the hit ratio, the overhead by the eviction becomes almost negligible when the cache size is set to 120 KB under most workloads.

Note that the memory overhead in the **M-media** trace is not as significant as the other traces when the cache size is very small as shown in Fig. 9, although all of them have a similar number of read requests. It is mainly because the former traces do not benefit from the fine-grained chunking scheme. Since most of data in the **M-media** trace contains unique data, chunk-level mapping table is used only for small amount of data. As a result, FineDedup does not incur a significant memory overhead even when the fine-grained chunking method is not effective.

While achieving the low overheads on read performance and memory space as described in Section V.3 and this section, FineDedup requires only about 10 MB more memory space in total which is 2% of memory space for high-end SSDs, i.e., 512 MB.

## 5. Evaluation of the Effectiveness of Cached Mapping Table for Mixed Workloads

The effectiveness of the cached mapping table is evaluated for mixed workloads. Fig. 10 shows the normalized amount of additionally written data due to the mapping table eviction. The mixed traces are composed

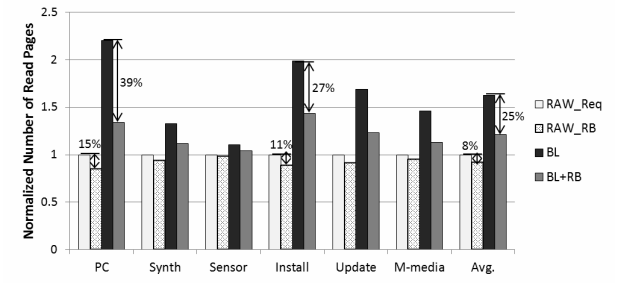


**Fig. 11.** The number of page reads with and without read buffer.

by accumulating individual traces in the order of **PC**, **Synth**, **Sensor**, **Install**, and **Update**. For example, mixed workload 4 is composed with **PC**, **Synth**, **Sensor**, and **Install**.

Although, the cached mapping table is not effective as the number of traces is increased, the performance degradation rate is smaller than the number of workload increasing rate. Moreover, mapping table caching will be effective when the caching memory is big enough to contain the working set of each trace. In the evaluation, the required memory space for the cached mapping table is about 1MB for the mixed workload. Considering that commercial SSDs have hundreds of GBs of DRAM, the memory overhead of a few MB is not large.

## 6. Evaluation of the Effectiveness of the Chunk Read Buffer

The effectiveness of the chunk read buffer between the baseline policy and FineDedup is evaluated. Fig. 11 shows the amount of page reads for the baseline policy and FineDedup with and without the read buffer. The read buffer absorbs about 8% read requests of **RAW_req**, whereas the read requests are reduced by about 25% with the read buffer for FineDedup on average. Based on the analysis, the higher effectiveness of the read buffer in FineDedup is because of the increased memory utilization by deduplication. The read buffer in the baseline policy can absorb read requests for pages that have already been read. Chunk read buffer in FineDedup, however, can also absorb the read requests for deduplicated pages pointed by the hybrid mapping table. Therefore, with the same read buffer size, more read requests can be absorbed in FineDedup.

## VI. CONCLUSIONS

In this paper, a fine-grained deduplication technique for flash-based SSDs is proposed. By using a fine-grained deduplication unit, the proposed FineDedup technique increases the amount of data eliminated by data deduplication by up to 24% over the existing page-based deduplication technique, extending the SSD lifetime by the same amount. FineDedup inevitably increases the overall read response time because of data fragmentation. By employing a chunk read buffer and a chunk packing scheme, however, the read performance overhead is limited to less than 10% in comparison with the existing deduplication technique. To reduce the memory space required for a chunk-level mapping table, FineDedup adopts a hybrid mapping scheme. The evaluation results show that FineDedup is effective in improving the SSD lifetime, requiring only about 2% more memory space of a high-end SSD.

## ACKNOWLEDGMENTS

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2015M3C4A7065645). The ICT at Seoul National University provided research facilities for this study. (Corresponding Author: Sungjin Lee)

## REFERENCES

[1] G. Naso, L. Botticchio, M. Castelli, C. Cerafogli, M. Cichocki, P. Conenna, et al., "A 128Gb 3b/cell NAND flash design using 20nm planar-cell technology," *in Proc. International Solid-State Circuits Conference*, San Francisco, USA, pp. 218-219, Feb. 2013.

[2] M. Sako, Y. Watanabe, T. Nakajima, J. Sato, K. Muraoka, M. Fujiu, et al., "A Low-Power 64Gb MLC NAND-Flash Memory in 15nm CMOS Technology," *in Proc. International Solid-State Circuits Conference*, San Francisco, USA, pp. 1-3, Feb. 2015.

[3] F. Chen, T Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in Proc. 9th USENIX Conference on File and Storage Technologies, San Jose, USA, pp. 1-14, Feb. 2011.

[4] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in Proc. 9th USENIX Conference on File and Storage Technologies, San Jose, USA, pp. 1-13, Feb. 2011.

[5] Z. Chen and K. Shen, "OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash," in Proc. 14th USENIX Conference on File and Storage Technologies, Santa Clara, USA, pp. 1-10, Feb. 2016.

[6] W. Li, G. Jean-Baptise, J. Riveros, and G. Narasimhan, "CacheDedup: In-line Deduplication for Flash Caching," in Proc. 14th USENIX Conference on File and Storage Technologies, Santa Clara, USA, pp. 1-15, Feb. 2016.

[7] D. Meister and A. Brinkmann, "dedupv1: Improving deduplication throughput using solid state drives" in Proc. IEEE Mass Storage Systems and Technologies, Incline Village, USA, pp. 1-6, May 2010.

[8] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters," in Proc. 9th USENIX Conference on File and Storage Technologies, pp. 1-15, San Jose, USA, Feb. 2011.

[9] R. L. Rivest, B. Agre, D. V. Bailey, C. Crutchfield, Y. Dodis, K. E. Fleming, A. Khan, J. Krishnamurthy, Y. Lin, L. Reyzin, E. Shen, J. Sukha, D. Sutherland, E. Tromer, and Y. L. Yin, "The MD6 hash function – a proposal to NIST for SHA-3".

[10] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mapping," in Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 229-240, Washington, USA, Mar. 2009.

**Taejin Kim** received the B.E. degree in computer engineering from Sungkyunkwan University, Korea, in 2010, and the M.E. degree in computer science and engineering from Seoul National University, Korea, in 2012. He is currently working toward Ph.D. degree at Seoul National University. His research interests include storage systems, operating systems, and embedded system software.

**Sungjin Lee** is an assistant professor at the DGIST. He earned the PhD and MS degrees in computer science and engineering from the Seoul National University in 2013 and 2007, respectively, and received the BE degree in electrical engineering from the Korea University in 2005. His current research interests include storage systems, operating systems, and system software.

**Jihong Kim** (M'00) received the B.S. degree in computer science and statistics from Seoul National University, Korea, in 1986, and the M.S. and Ph.D. degrees in computer science and engineering from University of Washington, WA, in 1988 and 1995, respectively. Before joining SNU in 1997, he was a Member of Technical Staff in the DSPS R&D Center of Texas Instruments in Dallas, Texas. He is currently a Professor in the School of Computer Science and Engineering, Seoul National University. His research interests include embedded software, low-power systems, computer architecture, and storage systems.