FineDedup: A Fine-Grained Deduplication Technique for Flash-Based SSDs

Taejin Kim, Sungjin Lee, and Jihong Kim
School of Computer Science and Engineering
Seoul National University, Korea
{*taejin1999, chamdoo, jihong*}*@davinci.snu.ac.kr*

ABSTRACT

In order to improve the lifetime of flash-based solid state drives (SSDs), data deduplication is rapidly adopted in recent SSDs. By preventing redundant data from being written to flash memory, data deduplication extends the overall lifetime of SSDs, avoiding a large number of redundant writes. Existing deduplication techniques for SSDs, however, fail to fully exploit potential benefits of data deduplication because they are typically designed to eliminate redundant data at a coarse-grained level. In this paper, we propose a fine-grained deduplication technique for flash-based SSDs, called *FineDedup*. By using a more fine-grained deduplication unit, FineDedup improves the likelihood of eliminating duplicate data. Using a fine-grained deduplication unit, however, inevitably increases the memory requirement because it needs to keep more information in the memory. Furthermore, because of the data fragmentation problem caused by fine-grained deduplication, the overall read response time is also significantly increased. FineDedup resolves these technical difficulties by employing hybrid mapping and defragmentation techniques. Our evaluation results show that FineDedup reduces the amount of data written to flash memory by up to 32% over the existing deduplication techniques, improving the storage lifetime by the same amount. This improvement comes with less than 5% read response time overhead over the existing techniques, requiring a negligible memory space increase.

1. INTRODUCTION

NAND flash-based solid-state drives (SSDs) have been widely used in many consumer devices like mobile phones, laptops, and desktop PCs due to their low-power consumption, high performance, and high shock resistance. As the price-per-byte of NAND flash memory is rapidly decreasing, NAND flash-based SSDs are considered to be a viable storage solution even for high-performance enterprise systems.

In NAND flash memory, reads and writes are performed in a unit of a page. Because of its 'erase-before-write' nature, a block consisting of multiple pages must be erased before programming (or writing) new data to it. Unfortunately, as the semiconductor process is scaled down and the multi-level cell (MLC) technology is introduced, the number of program/erase (P/E) cycles allowed for each block is significantly reduced. For example, 5x nm single-level cell (SLC) NAND flash memory supports 10K P/E cycles, but in recent 2x nm MLC NAND flash memory, the number of P/E cycles is reduced to 3K [1, 2]. The reduction in the number of P/E cycles seriously limits the overall lifetime of flash-based SSDs, making it difficult for SSDs to be used in various computing environments.

In order to prolong the lifetime of flash-based SSDs, data deduplication techniques have been widely used in recent SSDs because they reduce the amount of data written to flash memory by preventing duplicate data from being written again [8, 9]. As a result, only non-duplicate data, i.e., unique data, are stored in SSDs. In most deduplication schemes for SSDs, the unit of data deduplication is a single page whose size is usually 4K-8K. Using a page as a deduplication unit seems to be reasonable because a page is the unit of a read or write operation as well. However, this page-based deduplication technique loses many chances of eliminating duplicate data. We observed that up to 48% of unique pages actually contain mostly identical data. The existing techniques, however, cannot completely eliminate the unnecessary writes for the duplicate segment in those pages since they are considered as unique pages. Furthermore, it is expected that the advantages of the deduplication technique would significantly diminish as the page size of flash memory increases to 16KB due to the recent technical trend [3, 4].

In this paper, we propose a fine-grained deduplication technique for flash-based SSDs, called *FineDedup*. The proposed FineDedup technique 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 greatly regardless of a physical page size.

To effectively incorporate fine-grained deduplication into flash-based SSDs, the following two types of 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, with fine-grained deduplication, unique data segments from partially duplicate pages can be scattered across several physical pages. This consequently results in data fragmentation which seriously degrades 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. Our evaluation results show that FineDedup prolongs the lifetime of SSDs by up to 32% over page-based deduplication while requiring a negligible memory space increase. This improvement comes with a read performance penalty less than 5% over page-based deduplication.

The rest of the paper is organized as follows. In Section 2, we briefly review the existing deduplication techniques for SSDs. The motivation of the paper is presented in Section 3. We describe the proposed FineDedup technique in detail in Section 4, and then evaluate its effectiveness using real-world traces in Section 5. Finally, Section 6 concludes with a summary.

2. RELATED WORK

The most well-known approach that improves the storage lifetime is to optimize flash firmware algorithms. As mentioned previously, 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 [5, 6, 7]. 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. The firmware-level optimization has been effective in improving the lifetime of flash-based SSDs. However, considering the decreasing lifetime of recent high-density NAND flash memory [1, 2], more advanced solutions that further improve storage lifetime are urgently 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. Data deduplication techniques can be categorized into two types, fixed-size deduplication and variable-size deduplication, depending on their chunking strategies. Fixed-size deduplication divides an input data stream into fixed-size chunks [8, 9, 12]. Then, it prevents those duplicate chunks whose data are already stored in flash memory from being rewritten to flash memory. Unlike fixed-size deduplication, the chunk size of variable-size deduplication is not fixed [10, 11]. 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 exhibits a higher percentage of removed writes than fixed-size deduplication because it adaptively changes the size of chunks by analyzing the contents of an input data stream so that more data can be deduplicated. However, variable-size deduplication works efficiently when it is adopted at the level of an operating system or a file system where the entire data contents as well as the system-level information are available. Furthermore, 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 a storage device where computing and memory resources are severely constrained. For this reason, most existing deduplication techniques for SSDs employ fixed-size deduplication, which is relatively simple and does not require lots of hardware resources.

Similar to the existing deduplication technique, the proposed FineDedup technique is also based on fixed-size deduplication. Using a more fine-grained deduplication unit, however, FineDedup improves the likelihood of eliminating duplicate data, complementing existing fixed-size deduplication techniques which exhibit a relatively low deduplication ratio in comparison with variable-size deduplication.

3. MOTIVATIONS

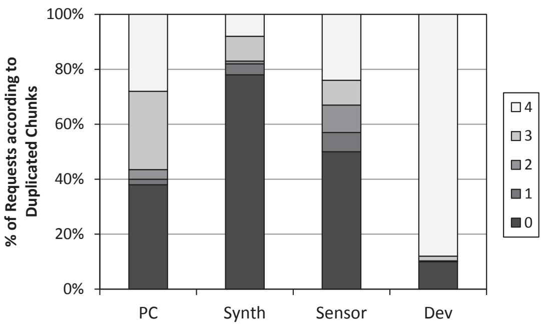


Figure 1: The percentage of the requests according to the number of duplicate chunks.

Existing deduplication techniques for SSDs use a single page as a chunk for data deduplication [8, 9]. If entire contents of a requested page are the same as those of a certain page which has been previously written to flash memory, the requested page is not written to flash memory. On the other hand, if there is no page in flash memory whose contents are the same as those of requested one, it means that the requested page is unique. Therefore, the requested page has to be written to flash memory. The main motivation of this work is the observational result that many pages which are regarded as unique ones actually contain many duplicate data segments that match those of previously written pages. Thus, in existing techniques, many duplicate data are written to flash memory even though they have already been written before.

In order to support our claims, we analyzed how many chunks turn out to be redundant when the chunk size is smaller than a single page. For our evaluation, we used several I/O traces collected from a desktop PC and a number of server systems used for a hardware synthesis, a sensor data analysis, and a software development. In our evaluation, the page size is 4 KB and the chunk size is set to 1 KB. Figure 1 shows the percentage of the requests according to the number of duplicate chunks. If all the chunks within a requested page are duplicate, it means that there exists a page in flash memory whose contents are identical to the requested one. Only in this case, page-based deduplication can eliminate a duplicate write for that page. However, if the number of duplicate chunks within a requested page is between 1 and 3, the page-based deduplication will write that page to flash memory because entire contents are not identical. Note that we call such pages partially duplicate pages. As shown in Figure 1, pages with 4 duplicate chunks account for 8% - 88% of the total requested pages. On the other hand, pages with 1-3 duplicate chunks account for 2% - 34%. This means

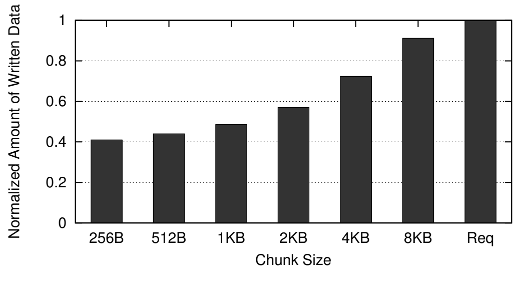


Figure 2: The amount of data written with varying chunk sizes in PC workload.

that many duplicate data are unnecessarily written to flash memory when the size of a chunk is large.

We also investigated the amount of data that can be eliminated by data deduplication while varying the chunk size from 256 B to 8 KB. As shown in Figure 2, when the chunk size is 1 KB, the amount of data written to flash memory is reduced by 33% in comparison with that 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 removed writes, decreases significantly. As pointed out in Section 1, the physical page size of NAND flash memory has been increased as the semiconductor process is scaled down [3, 4]. Therefore, it is expected that the deduplication ratio of the existing deduplication technique will be significantly reduced in the near future. In order to resolve this problem, the chunk size of deduplication techniques needs to be smaller than a page size. As depicted in Figure 2, the deduplication ratio is saturated when the chunk size is 1 KB. Thus, we use it as a default chunk size in the rest of this paper.

4. FINE-GRAINED DEDUPLICATION

In this section, we describe the proposed FineDedup technique in detail. We first explain the overall architecture of FineDedup and describe how FineDedup handles read and write requests in Section 4.1. We introduce a read performance penalty and memory overheads caused by FineDedup in Sections 4.2 and 4.3 and explain how these problems can be resolved.

4.1 Overall Architecture of FineDedup

Figure 3 shows the architecture of FineDedup with its main components and how it handles write requests. Upon the arrival of a write request, FineDedup stores requested data temporarily in an on-device buffer, which is managed by an LRU algorithm. When the requested data are evicted from the buffer, FineDedup divides the data into several chunks. Note that the chunk size is 1 KB in this work, but a different size of a chunk can be used with FineDedup.

For each chunk, FineDedup computes a fingerprint, using a collision-resistant hash function. In this work, we use an MD6 hash function, which is one of the well-known cryptographic hash functions. 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 calculation. In our observation, the time taken to compute a fingerprint using a hardware accelerator is about 10 μs. Considering long write latency (e.g., 1.2 ms) of NAND flash memory, the hash computation overhead is negligible.

After fingerprinting, each fingerprint is looked up in the dedup table which maintains fingerprints of 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. 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 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 hybrid mapping strategy, which is described in Section 4.3 in detail.

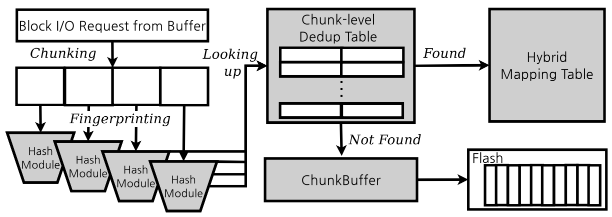


Figure 3: An overview of the proposed FineDedup technique.

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 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 the newly written chunks. The new fingerprints of the evicted chunks are finally inserted into the dedup table with its 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 a host system. The physical addresses of the chunks can be obtained by referring to the mapping table. 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 page data, which in turn significantly increases the overall read response time. We discuss how FineDedup resolves this problem in the following subsection.

4.2 Read Overhead Management

FineDedup effectively reduces the number of pages written to flash memory by using a small-size chunk for deduplication, but it incurs two types of additional overheads, a read performance overhead and a memory space overhead, which are not observed in the existing deduplication techniques. In this subsection, we first introduce why the read performance overhead happens in FineDedup, and then explain how FineDedup resolves this problem. In the following subsection, we describe our memory space overhead reduction technique in detail.

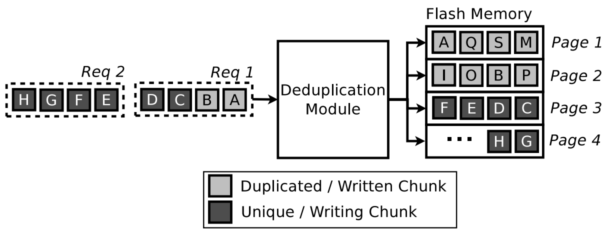


Figure 4: Data fragmentation caused by FineDedup.

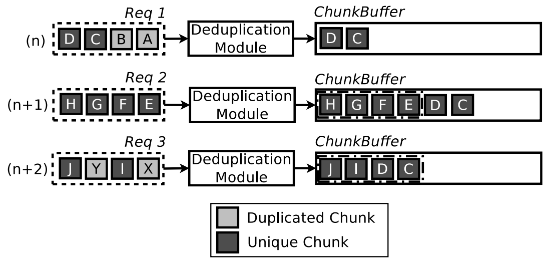


Figure 5: A packing scheme in the *chunk buffer*.

The main cause of the read performance degradation is data fragmentation which occurs when chunks belonging to same logical page are broken up into several physical pages. Figure 4 illustrates why data fragmentation occurs in FineDedup. There are two page write requests *Req1* and *Req2* in Figure 4. *Req1* consists of four chunks ‘A’, ‘B’, ‘C’, and ‘D’, and *Req2* is also composed of four chunks ‘E’, ‘F’, ‘G’, and ‘H’. Since ‘A’ and ‘B’ are already stored in flash memory, only ‘C’ and ‘D’ are needed to be written. Suppose that there is a read request for page data written by *Req1*. In that case, FineDedup has to read three pages *page1*, *page2*, and *page3* from flash memory to form the requested data. The read performance penalty also could occur even when there are no duplicate chunks in the requested page. For example, in Figure 4, *Req2* has no duplicate chunks in flash memory, and thus all the chunks belonging to *Req2* must be written to flash memory. Because of write buffer effect, ‘E’ and ‘F’ of *Req2* are written to *page 3* together with ‘C’ and ‘D’, and ‘G’ and ‘H’ will be written to *page 4* in the future, as shown in Figure 4. Thus, when data written by *Req2* are read later, both *page 3* and *page 4* must be read from flash memory.

One of the feasible approaches that mitigate the read performance overhead is to employ a chunk read buffer. In our observation, the access frequencies of unique chunks are greatly skewed; that is, a small number of popular chunks account for a large fraction of total accesses to unique chunks in flash memory. For example, according to our analysis from real-world workloads, top 10% of the unique chunks serve more than 70% of the total data read by a host system. By keeping frequently accessed chunks in a chunk read buffer, therefore, FineDedup can reduce lots of page read operations to flash memory.

In order to further reduce the read performance penalty, FineDedup uses a chunk packing scheme. The key idea of the chunk packing scheme is to group chunks belonging to the same logical page in a chunk buffer and then write them to the same physical page together. Figure 5 shows the example of our chunk packing scheme when three page write requests *Req1*, *Req2*, and *Req3* are consecutively issued from a host system. *Req1* 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 *Req2* does not have any duplicate chunks, so all of them are moved to the chunk buffer. As depicted in Figure 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. Two unique chunks ‘I’ and ‘J’ are requested for writing by *Req3*, and thus there are four chunks ‘C’, ‘D’, ‘I’, and ‘J’ in the chunk buffer. All those chunks can be written to the same physical page together because every chunk of each request is not broken up into multiple pages.

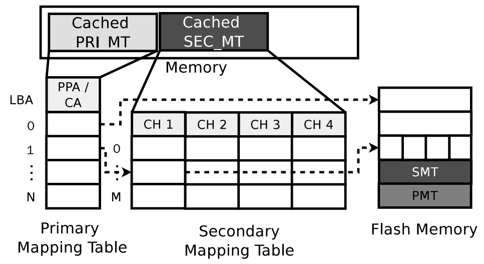


Figure 6: An overview of the demand-based hybrid mapping table.

4.3 Memory Overhead Management

As mentioned in Section 4.1, FineDedup handles requested data in the unit of a chunk. Therefore, FineDedup must maintain the 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 the size of a page, the chunk-level mapping table is much larger than the normal mapping table. For example, suppose that the page size is 4 KB and the chunk size is 1 KB. In that case, the size of chunk-level mapping table is four times larger than that of the page-level mapping table.

In order to reduce the amount of memory space required for the mapping table, FineDedup employs a hybrid mapping table which is composed of two kinds of mapping tables: page-level mapping table and chunk-level mapping table. As depicted in Figure 1, duplicate pages and unique pages still account for a considerable proportion of the total pages requested for writing by a host system. For those pages, the page-level mapping table is more appropriate because those pages can be directly mapped to corresponding pages in flash memory. The chunk-level mapping table is only required for partially duplicate pages.

Figure 6 show the overall architecture of the hybrid mapping table in FineDedup. The primary mapping table is the page-level mapping table and the secondary mapping table is the chunk-level mapping table. The entry of the primary mapping table is either the physical page address in the flash memory or the index of the secondary mapping table. If the requested page is duplicate one, the corresponding entry of the primary mapping table directly points to the physical address of the existing unique page in flash memory. Similarly, if the requested page is unique one, that page is written to flash memory. Then, the corresponding entry fo the primary mapping is updated to point to the newly written unique page. Finally, if partially duplicate page is requested for writing, FineDedup allocates a new entry in the secondary mapping table. As depicted in Figure 6, each entry 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 filed points to the physical chunk address. The corresponding entry of the primary mapping table indicates the newly allocated entry in the secondary mapping table.

Using the hybrid mapping table, FineDedup can reduce the amount of memory space for keeping a mapping table. The problem of this hybrid mapping approach is that the size of the mapping table greatly varies according to the characteristics of workloads. For example, if some workloads have many partially duplicate pages, the secondary mapping table size becomes huge. On the other hand, for workloads with

Trace	Description	Written Data	Read Data
PC	Web surfing, emailing and document editing, etc.	3.1 GB	40 MB
Sensor	A sensor data collected during a semiconductor fabrication process	2.6 GB	660 KB
Synth	Synthesizing hardware modules	2.5 GB	170 KB
Dev	Coding and compiling the Linux kernel source	4.8 GB	86 MB
M-media	Downloading and playing multimedia files	3.2 GB	36 MB

Table 1: A summary of traces for evaluation.

only unique pages or duplicate pages, the secondary mapping table size 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 uses a demand-based mapping strategy; the entire chunk-level mapping table is stored in flash memory while keeping only a fixed number of popular entries in the DRAM memory. Demand-based mapping requires extra page read and write operations. 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 entry to flash memory. As pointed out in Section 4.1, since a relatively small number of popular chunks are frequently accessed, the performance penalty caused by extra reads or writes is not so high even when the in-memory mapping table is small.

5. EXPERIMENTAL RESULTS

In this section, we first describe our experimental settings and explain benchmarks used for the evaluation in detail. We then assess the effect of the proposed FineDedup technique on the SSD lifetime. Finally, we analyze the read performance penalty and the memory overhead caused by FineDedup.

5.1 Experimental Settings

In order to evaluate the effectiveness of FineDedup, we performed our evaluation using a trace-driven simulator with I/O traces collected from various systems. 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 collection and wear-leveling. The proposed FineDedup technique and the existing deduplication techniques were also implemented in our simulator.

For trace collection, we modified Linux kernel 2.6.32 and collected I/O traces at the level of a block device driver. All the I/O traces include the detailed information about I/O commands sent to a storage device (e.g., the type of requests, logical block address (LBA), the size of requests, and etc.) and the contents of data sent to or read from a storage device. We recorded I/O traces while running various real-world applications. The detailed descriptions of the I/O traces are summarized in Table 1.

5.2 Effectiveness of FineDedup

Figure 7 shows the amount of data written to flash memory by FineDedup and the existing scheme. The results shown in Figure 7 are normalized to *Req*, which represents the total amount of data written to flash memory without data deduplication. We assume the page-based deduplication technique as a baseline case. The baseline is denoted by the

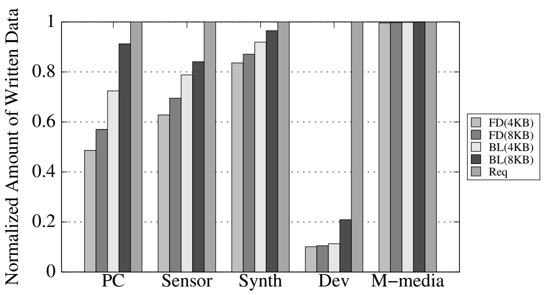


Figure 7: The amount of reduced data with various chunk sizes.

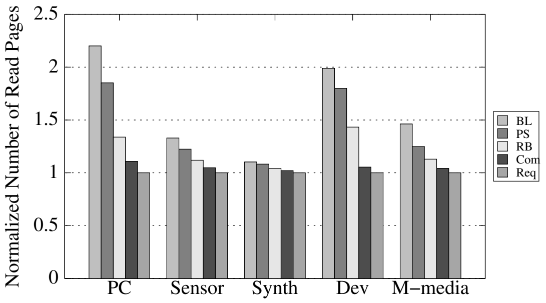


Figure 8: The number of page read operations.

BL(4KB) for 4KB flash page and *BL(8KB)* for 8KB flash page. Our FineDedup technique is denoted by *FD(4KB)* and *FD(8KB)* for 4KB flash page and 8KB flash page, respectively. The chunk size in FineDedup is set to 1KB for 4KB flash page and 2KB for 8KB flash page.

As we can see in Figure 7, duplication of data is highly workload dependent. The amount of data eliminated by the deduplication technique notably increases as the chunk size decreases in three traces, `PC`, `Sensor`, and `Synth`. When we set the chunk size to one fourth of the flash page size, FineDedup removes more duplicate data by 14%, on average, and up to 32% (`PC`) over the existing technique for 4KB flash page. For the 8KB flash page, it also removes more duplicate data by 23%, on average, and up to 50% (`Dev`) over the existing technique. As expected, the benefit of FineDedup mainly comes from the decreased chunk size because it increases the probability of finding and eliminating duplicate data.

In the `Dev` trace, the existing deduplication technique for 4KB flash page can effectively remove duplicate data because the slightly modified source codes are repeatedly compiled. However, when the 8KB flash page is used, the probability of finding duplicate data is highly decreased. In the `M-media` trace, it is extremely difficult to find duplicate data because the data were already compressed. Thus, the existing deduplication techniques and FineDedup are not effective to reduce data to be written.

5.3 Read Overhead Evaluation

As explained in Section 4.2, fine-grained chunking in FineDedup increases the number of page read operations. In FineDedup, we also have proposed the optimization techniques, such as the packing scheme and the chunk read buffer to reduce the additional page read operations.

Figure 8 shows the normalized number of page read operations compared with the number of read requests. The *Req* refers to the number of page read requests and the *BL* refers to the baseline which means the number of page read operations when no optimization technique is applied. Moreover, the rest legends represent the number of read operations when the optimization technique is applied, i.e. *PS* for the packing scheme, *RB* for the chunk read buffer and *Com* for the both of techniques combined. The size of the chunk read buffer is set to the value when the effectiveness of enlarging cache size begins to decrease for each workload. The value for `PC`, `Dev` and `M-media` is 8 MB and for `Sensor` and `Synth` is 64 KB.

As shown in Figure 8, employing the chunk read buffer is much effective than the packing scheme for reducing the additional page read operations in most workloads. This is because the packing scheme can only be effective for the requests containing no duplicate chunks while the chunk read buffer can absorb the read requests to frequently accessed chunks. FineDedup with the combined technique for the read overhead shows only the additional read operations less than 5%, on average, compared to the existing deduplication technique.

5.4 Memory Overhead Evaluation

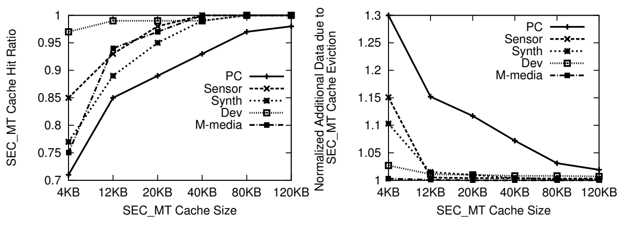
As explained in Section 4.3, large memory space is required to handle the partially duplicate pages. In FineDedup, we have proposed the demand-based hybrid mapping table to reduce and effectively handle the required memory size for the mapping table. In Figure 9, the effectiveness of the proposed mapping table is evaluated in terms of the hit ratio and the additional written data with various memory sizes for the cache. Since the primary mapping table in FineDedup is page-level, we assume that DFTL, which is well known demand-based scheme to exploit the page-level mapping, is used as the baseline mapping scheme. Thus, the target of this evaluation is only the secondary mapping table.

Figure 9(a) shows the hit ratio of the cached SEC_MT. With 120KB-sized cache, more than 95% of the mapping table accesses are absorbed. In addition, Figure 9(b) shows the additional written data caused by the evicted page of entries from the SEC_MT cache. Since the mapping table accesses occur in the middle of the read/write operations, reducing the written data due to the eviction is important in terms of read/write performance. Similar to the hit ratio, the overhead due to the eviction becomes almost negligible when the cache size is set to 120KB-sized cache in most workloads.

Note that the memory overhead in the `Dev` and the `M-media` traces is not as significant as the `PC` trace when the cache size is small although all of them have the similar number of read requests. It is mainly because the former traces do not benefit from the fine-grained chunking. Since most of data in the `Dev` trace are duplicate ones and the `M-media` trace contains unique data. As a result, FineDedup does not incur significant memory overhead when the fine-grained chunking method is not effective.

6. CONCLUSIONS

In this paper, we propose a fine-grained deduplication technique for flash-based SSDs, called FineDedup. By using a fine-grained deduplication unit, the proposed FineDedup technique increases the amount of data eliminated by data deduplication by up to 32% 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, the read performance overhead is limited to less than 5% in comparison with the existing deduplication technique. To reduce the memory space required for a chunk-level mapping table, FineDedup adopts the hybrid mapping scheme. Our evaluation results show that FineDedup achieves a great lifetime improvement, requiring 120 KB more memory space.



(a) Hit ratio of SEC_MT cache (b) Normalized written data due to eviction

Figure 9: The effectiveness of the demand-based hybrid mapping table in FineDedup.

7. ACKNOWLEDGMENTS

This research was supported by WCU(World Class University) program through the National Research Foundation of Korea funded by Ministry of Education, Science and Technology (R33-2012-000-10095-0). This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2012-0006417). The ICT at Seoul National University provided research facilities for this study.

8. REFERENCES

[1] B. You and et. al, "A High Performance Co-design of 26 nm 64 Gb MLC NAND Flash Memory using the Dedicated NAND Flash Controller," *Journal of Semiconductor Technology and Science*, vol. 11, no. 2, pp. 121-129, 2011.

[2] Y. Koh, "NAND Flash Memory Scaling Beyond 20 nm," in *Proceedings of the IEEE International Memory Workshop*, 2009.

[3] H. Kim and et. al, "A 159mm 32nm 32Gb MLC NAND-Flash Memory with 200MB/s Asynchronous DDR Interface," in *International Solid-State Circuits Conference*, 2010.

[4] Y. Li and et. al, "128Gb 3b/Cell NAND Flash Memory in 19nm Technology with 18MB/s Write Rate and 400Mb/s Toggle Mode," in *International Solid-State Circuits Conference*, 2012.

[5] J. Kim and et. al, "A Space-Efficient Flash Translation Layer for Compact Flash Systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366-375, 2002.

[6] S.-W. Lee and et. al, "A Log Buffer Based Flash Translation Layer Using Fully Associative Sector Translation," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, 2007.

[7] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems," *ACM SIGOPS Operating Systems Review*, 2008.

[8] 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 *Proceedings of the USENIX Conference on File and Storage Technologies*, 2011.

[9] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-Based SSDs," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[10] D. Meister and A. Brinkmann, "dedupv1: Improving Deduplication Throughput using Solid State Drives," in *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[11] W. Dong and et al, "Tradeoffs in Scalable Data Routing for Deduplication Clusters," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011

[12] K. Srinivasan and T. Bisson, G. Goodson, K. Voruganti, "iDedup: Latency-aware, Inline Data Deduplication for Primary Storage," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.