2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing

Deduplication with Block-Level Content-Aware Chunking for Solid State Drives (SSDs)

Jin-Yong Ha*, Young-Sik Lee†, and Jin-Soo Kim*
*College of Information and Communication Engineering, Sunkyunkwan University, Suwon, South Korea
Email: jinyongha@csl.skku.edu and jinsookim@skku.edu
†Department of Computer Science, KAIST, Daejeon, South Korea
Email: yslee@calab.kaist.ac.kr

***Abstract*—A critical weak point of Solid State Drives (SSDs) is the limited lifespan which stems from the characteristics of NAND flash memory. In this paper, we propose a new deduplication scheme called *block-level content-aware chunking* to extend the lifetime of SSDs. The proposed scheme divides the data within a fixed-size block into a set of variable-sized chunks based on its contents and avoids storing duplicate copies of the same chunk. Our evaluations on a real SSD platform show that the proposed scheme improves the average deduplication rate by 77% compared to the previous block-level fixed-size chunking scheme. Additional optimizations reduce the average memory consumption by 39% with a 1.4% gain in the average deduplication rate.**

I. Introduction

Solid State Drives (SSDs) based on NAND flash memory are quickly gaining popularity. In contrast to hard disk drives (HDDs), SSDs have several advantages such as high performance, shock resistance, and low power consumption. However, one of the critical weak points of SSDs is the limited lifespan. NAND flash memory requires erase operations to overwrite data into an already programmed area. As NAND flash memory cells are gradually worn out by repeated program/erase cycles, the life expectancy of SSDs is limited by *write traffic*, i.e., the amount of data written to NAND flash memory. The lifespan of SSDs is getting worse as the NAND technology moves from MLC (Multi-Level Cell) to TLC (Triple-Level Cell) to reduce the cost per bit [1].

Deduplication is a technique which reduces write traffic by dividing data into small pieces or *chunks* and then storing only the unique chunks. Many data deduplication mechanisms have been proposed to extend the lifetime of SSDs [2], [3]. However, they commonly partition the incoming data into fixed-size chunks. This *block-level fixed-size chunking* suffers from low deduplication rate[1] especially when a portion of a file is inserted or deleted; in this case, all the chunks after the modified position become different from the original versions and thus cannot be deduplicated.

To improve the deduplication rate further, this paper proposes a new deduplication scheme called *block-level content-aware chunking*. The proposed scheme divides the data within a fixed-size block into a set of variable-sized chunks based on its contents and avoids storing duplicate copies of the same chunk. We also explore several optimizations which consider (1) chunks lied at the block boundary, (2) chunks consisting of all zeroes, and (3) chunks of small size. To examine the impact of each optimization on deduplication rate and memory footprint, we have performed comprehensive experiments on a real SSD platform.

Our evaluations with four different workloads show that the proposed scheme improves the average deduplication rate by 77% over the block-level fixed-size chunking scheme. After applying all the optimizations, the average memory consumption is reduced by 39% with an additional gain of 1.4% in the average deduplication rate.

This paper makes following contributions.

- The idea of content-aware chunking is not new in the deduplication domain. However, it is not used for SSDs as the file-level information is not available at the device level. We propose a new approach called *block-level content-aware chunking* which applies the content-aware chunking within each block. Even if the content-aware chunking is performed for each fixed-size block (e.g., 4KB), we find a significant improvement in deduplication rate can be achieved over the traditional block-level fixed-size chunking.
- For block-level content-aware chunking, we suggest several optimizations to further improve deduplication rate and memory footprint.
- We design the firmware architecture and major data structures required to implement the proposed scheme in SSDs. In particular, we have developed a prototype based on a real SSD platform to demonstrate the feasibility of our approach.
- We perform extensive experiments with real workloads to quantitatively evaluate the performance of the block-level content-aware chunking and the effect of each optimization.

The rest of this paper is organized as follows. Section II provides background information. In Section III, the proposed block-level content-aware chunking scheme is presented with several optimizations. Section IV shows our evaluation results and Section V summarizes the related work. Section VI

[1]We define the deduplication rate as the aggregated reduction in storage requirements gained from deduplication technology [4]. Thus, the higher deduplication rate uses less storage space.

978-0-7695-5088-6/13 $31.00 © 2013 IEEE
DOI 10.1109/HPCC.and.EUC.2013.286

concludes the paper.

II. Background

A. Flash Translation Layer (FTL)

NAND flash memory is physically organized in *flash blocks* and each flash block contains 64∼256 *pages*. The read and write operation take place on a page basis, while the erase operation cleans all the pages within a flash block. The internal NAND flash memory of an SSD is managed by special firmware called Flash Translation Layer (FTL). Since in-place update is not allowed, FTL writes each incoming data into clean pages and maintains mapping information between the logical page number (LPN) from the host and the physical page number (PPN) in NAND flash memory. As the new data is written, the previous version becomes obsolete and those obsolete pages are later reclaimed by the garbage collection (GC) procedure. During GC, FTL selects a victim flash block and converts it into a clean flash block. Before erasing the victim flash block, any valid pages still in the victim block should be copied to other clean pages.

B. Extending the Lifespan of SSDs

SSDs have a limited lifespan as NAND flash memory is worn out due to repeated program/erase operations. When the number of erase operations performed on a flash block goes beyond the number guaranteed by the manufacturer, it is not recommended to use the flash block anymore. Even worse, the guaranteed number for each flash block is getting smaller as the manufacturing technology shrinks in size and moves towards more than 2-bit MLC. Thus, extending the lifespan of SSDs is one of the hottest issues in SSDs.

Several different approaches have been studied to extend the lifespan of SSDs. The first approach is to develop sophisticated NAND flash management algorithms for FTL. In many cases, the additional writes caused by GC occupy a large portion of the total write traffic. Therefore, various mapping schemes and GC policies have been proposed to improve the performance and lifetime of SSDs [5], [6]. The second approach is to add a novel wear-leveling algorithm to FTL. Wear-leveling algorithms attempt to balance the erase counts over the entire flash blocks to prolong the time until one of flash blocks fails [7], [8]. The final approach is to reduce the amount of actual data in SSDs via compression [9] or deduplication [2], [3].

C. Deduplication

Deduplication has long been used in backup systems where many copies of very similar or even identical files are stored. In such an environment, a file is divided into a set of chunks and each chunk is compared with already stored chunks. To make chunk comparisons faster, the *fingerprint* of a chunk, generated by a cryptographic hash function such as MD5 [10] and SHA-1 [11], is used. The probability of having an identical fingerprint from two different chunks is very small, even less than hardware bit errors [12]. Although fingerprint collision is not a serious issue in deduplication, the collision can be completely eliminated by byte-to-byte comparison of their contents as is done in [13].

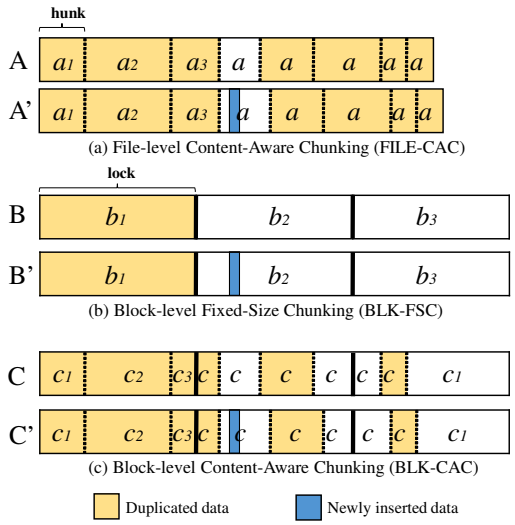

Fig. 1: Examples of deduplication with various chunking schemes.

One of the most important issues in deduplication is the chunking scheme. To save storage space and network bandwidth, many backup systems and network file systems use the chunking scheme such as Rabin fingerprinting [14] where the chunk boundary is determined only when its contents satisfy a specific condition [15], [12]. We call this the *file-level content-aware chunking* (FILE-CAC). An example of FILE-CAC is depicted in Figure 1(a) in which the file A is updated to A' as the new data is inserted in the middle of the file. Since the chunks are determined by their contents, only the chunk a'_4 is affected by this insertion and the remaining chunks are eligible for deduplication.

Unfortunately, FILE-CAC cannot be directly used for SSDs as storage devices have no information on which sequence of data blocks belongs to the same file. Although several deduplication schemes such as CAFTL [2] and CA-SSD [3] have been recently proposed for SSDs and a commercial SSD controller is known to support deduplication [16], this is why they all use the *block-level fixed-size chunking* (BLK-FSC) as shown in Figure 1(b). In the same situation as in Figure 1(a), BLK-FSC is able to deduplicate only the first chunk b_1 since the original data is shifted by the newly inserted data making the chunks b'_2 and b'_3 differ from b_2 and b_3, respectively. Therefore, it is very difficult for BLK-FSC to achieve a high deduplication rate.

To increase the deduplication rate further in SSDs, this paper proposes a novel chunking scheme called *block-level content-aware chunking* (BLK-CAC), as depicted in Figure 1(c). In the following section, we describe the design and implementation of the proposed scheme in detail.

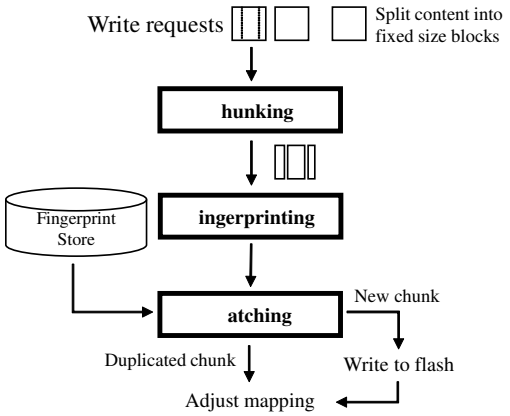

Fig. 2: Deduplication process in BLK-CAC.

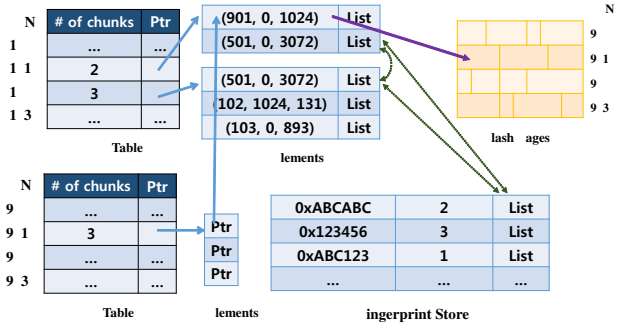

Fig. 3: Mapping and indexing data structures.

III. Design

A. Block-Level Content-Aware Chunking (BLK-CAC)

In the block-level content-aware chunking scheme (BLK-CAC), each block is divided into several chunks according to its contents as in FILE-CAC. However, since each block is treated separately similar to BLK-FSC, a chunk cannot span across two different blocks. Therefore, each chunk ends if its contents satisfy a specific condition *or* it meets the block boundary.

In Figure 1(c), assume that the file C, whose contents are the same as the file A, is stored in three different blocks of the storage device. Note that the chunk a_3 (or a_6) in Figure 1(a) has been split into two chunks c_3 and c_4 (or c_7 and c_8) in Figure 1(c) due to the block boundary. Even if the new data is inserted into c_5, we can see that the chunks c_4, c_6 and c_9 can still be deduplicated after the insertion. In this way, BLK-CAC can deduplicate a more number of chunks than BLK-FSC. Especially, BLK-CAC is more effective when a file size is large as BLK-FSC usually fails to deduplicate those chunks located after the modified region of the file.

One downside of BLK-CAC is that the number of chunks is increased compared to BLK-FSC. The larger the number of chunks, the more memory required as we need to maintain a fingerprint for each chunk. In Section III-D, we describe several optimizations designed for reducing the number of chunks without a significant impact on the deduplication rate.

B. Deduplication Process

Figure 2 illustrates the overall deduplication process in BLK-CAC which consists of three major phases: chunking, fingerprinting, and matching. When a write request is issued from the host, its contents are first divided into fixed-size blocks. We use the block size of 4KB which is identical to the block size of the Ext4 file system. In the chunking phase, each block is split into several variable-sized chunks based on its contents. We use Rabin fingerprinting for the content-aware chunking. If the block boundary is met while applying Rabin fingerprinting, the chunking phase is finished.

In the fingerprinting phase, a 160-bit fingerprint is generated for each chunk using the SHA-1 hash function. The generated fingerprint is compared with other fingerprints stored in the fingerprint store in the matching phase. If there is a matching fingerprint, the physical address of the matching chunk is retrieved from the fingerprint store and the mapping is adjusted accordingly. Otherwise, the chunk is written to NAND flash memory and the corresponding entry is added to the fingerprint store.

As the chunk size is variable and typically smaller than the flash page size, we use a write buffer to coalesce multiple chunks before writing them into NAND flash memory.

C. Data Structures for Deduplication

Figure 3 depicts major data structures needed for implementing deduplication with BLK-CAC. The mapping table for translating logical addresses to physical addresses is composed of the L2P table and L2P elements. When the write buffer is flushed, the number of chunks and the pointer to an L2P element is stored in the corresponding entries in the L2P table. Each L2P element contains the physical flash page number (PPN), the starting offset, the size of each chunk, and pointers to link all the L2P elements that point to the same physical data.

For example, Figure 3 shows that LPN #101 has two chunks. From the associated L2P element, we can see that the first chunk is 1024 bytes in size and it is located in the byte offset 0 of PPN #901. The second chunk of LPN #101 and the first chunk of LPN #102 have been deduplicated, sharing the same data stored in the byte offset 0 of PPN #501.

In our design shown in Figure 3, the memory space for L2P elements must be allocated at runtime since the number of chunks in a block is determined after the chunking phase. In addition, the dynamic memory allocation for variable-sized L2P elements has large overheads to manage the additional metadata such as memory offset and size.

Therefore, we limit the maximum number of chunks in a block to n (cf. Table I) for ease of memory management. When the number of chunks reaches n value while chunking a block, the remaining portion of the block is regarded as one chunk.

We allocate the memory space for L2P elements in advance and eliminate the overhead of dynamic memory management.

The fingerprint store provides the mapping information from fingerprints to their physical addresses. Each entry consists of the fingerprint of a chunk, the number of L2P elements pointing to the chunk, and pointers to link those L2P elements. In Figure 3, the first entry of the fingerprint store has the fingerprint value of 0xABCABC and it is shared by two L2P elements. By following the pointers, we can identify that the corresponding chunk is stored in PPN #501 and its size is 3072 bytes.

The P2L table and the associated P2L elements are used to find the logical address for the given physical address during GC. Figure 3 depicts an example where PPN #901 has three chunks in it and the first chunk is mapped to the first chunk of LPN #101.

D. Optimizations

This subsection presents several optimization techniques for BLK-CAC that can further reduce the number of entries in the fingerprint store and the write traffic to NAND flash memory.

1) Chunks in the Block Boundary: As each unique chunk occupies an entry in the fingerprint store, it is important to reduce the number of entries kept in the fingerprint store to save memory. One possible way is to remove the chunks lied at the block boundary (called *boundary chunks*) from the fingerprint store. This is because those boundary chunks are formed not based on their contents but due to the presence of a block boundary. Hence, they are very likely to be changed when the contents of a file is shifted by the newly inserted or deleted data. For example, we can see that the boundary chunks c'_7 and c'_8 become different from the chunks c_7 and c_8, respectively, due to the inserted data on the chunk c'_5 in Figure 1(c). This optimization is more effective for large files since many boundary chunks are affected by the change in the file contents.

We should be careful so as not to discard the first boundary chunk of a file as this chunk is not affected by the change in the later part of the file. Because the file-level information is not available in the storage device, it is normally impossible to identify whether a boundary chunk is the first chunk of a file. However, there is a way to detect those chunks when the file size is smaller than the block size using the fact that the file system initializes the unused space of a block with zeroes. More specifically, we use a heuristic that the boundary chunk in the front of a block is not discarded if the last chunk consists of all zeroes.

2) Chunks Consisting of All Zeroes: The chunks consisting of all zeroes (called *zero chunks*) need not be written to NAND flash since read operations can be served by filling the buffer with zeroes. There are many such zero chunks because the unused space within a data block after the end of the file is filled with zeroes in most file systems including Ext4. Thus, if there is a region in the block which is composed of successive zeroes, we treat it as a separate zero chunk. The contents of the zero chunk is neither deduplicated nor written to NAND flash memory. Instead, we simply mark a flag in the corresponding entry of the L2P element.

One side effect of having zero chunks is that the number of chunks can be increased. If a chunk had a series of zeroes in the middle of it, the chunk would be divided into three different chunks one of which is the zero chunk. Although the zero chunk itself does not consume any entry in the fingerprint store, the fingerprints of other two chunks should be maintained.

We can reduce the side effect of zero chunks by regulating the minimum size of zero chunks. Because zero chunks of small size do not increase the deduplication rate significantly, we simply discard zero chunks whose sizes are less than a predefined value z (cf. Table I). Due to these special handling of zero chunks, the amount of write traffic to NAND flash memory is reduced noticeably.

3) Chunks of Small Size: When the chunk size is small, the benefit of deduplication is not significant compared to the overhead of keeping the corresponding entry in the fingerprint store. Therefore, we avoid storing the fingerprints of chunks whose sizes are smaller than the predefined threshold x (cf. Table I).

Parameters		Default values
Block size	b	4096 bytes
Average chunk size	c	512 bytes (fixed to b for BLK-FSC)
Min. size of zero chunks	z	512 bytes
Min. size of non-zero chunks	x	128 bytes
Max. number of chunks / block	n	12

TABLE I: Parameters used in experiments

IV. Evaluation

A. Methodology

We perform experiments on a PC with Intel Core i7-3550 3.4GHz CPU and 8GB RAM, which is connected to the Jasmine OpenSSD platform [17] via the SATA2 interface. The Jasmine platform is an SSD development board which consists of Indilinx Barefoot SSD controller, 64MB SDRAM, and two 32GB NAND flash memory modules. For ease of prototyping, we have modified the firmware of the Jasmine platform so that it exposes the native flash read, write, and erase operations, and implemented BLK-FSC and BLK-CAC schemes as a kernel module in the host system running Linux 2.6.32. The deduplication rate of FILE-CAC is measured using a file-level simulator.

The average chunk size in FILE-CAC and BLK-CAC is set to 512 bytes, by default. Note that the larger average chunk size results in less deduplication rate, while the smaller size causes more memory footprint. For BLK-FSC, the average chunk size is set to 4KB which is same as the block size of the Ext4 file system. To accelerate the experiments, the total capacity of SSD is configured to 4GB. Some parameters and their default values used in our experiments are summarized in Table I.

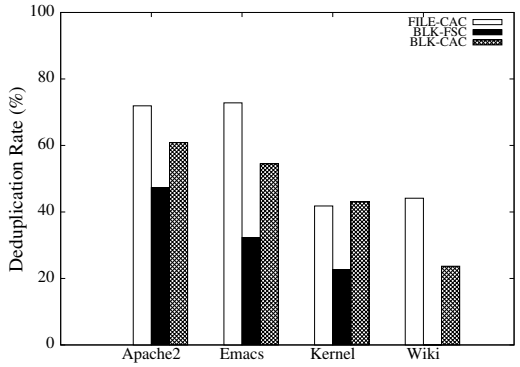

Fig. 4: A comparison of deduplication rates for different chunking schemes

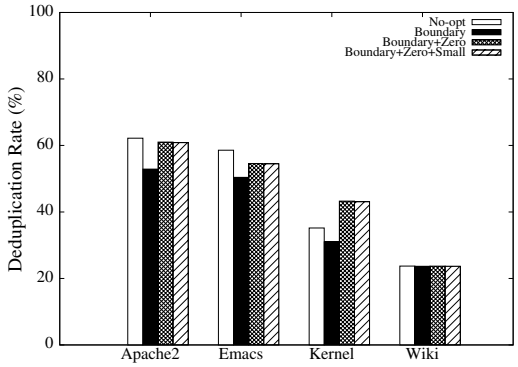

Fig. 5: The impact of various optimizations on the deduplication rate

We use four workloads, APACHE2 [18], EMACS [19], KERNEL [20], and WIKI [21]. APACHE2, EMACS, and KERNEL consist of five (from 2.2.8 to 2.2.22), six (from 23.1 to 24.2), and two (2.6.32.60 and 2.6.34.14) different versions of the source tar files, respectively, which are extracted on the Ext4 file system one by one from the lowest version to the highest version. In WIKI, two large files, which are snapshots of all Wikimedia Wikis taken on Jan. and Feb. 2012, are copied to the file system in chronological order.

B. Deduplication Rate

Figure 4 compares the deduplication rate of each chunking scheme. Note that the theoretical bound for the deduplication rate of KERNEL and WIKI is 50% as only two versions are written into the SSD in these workloads. In Figure 4, we can see that the difference in deduplication rates between FILE-CAC and BLK-CAC is 11% $\sim$ 20% in APACHE2, EMACS, and WIKI. Since FILE-CAC utilizes the file-level information for the content-aware chunking, the deduplication rate of FILE-CAC is normally better than that of BLK-FSC or BLK-CAC. In spite of this, it is surprising that BLK-CAC shows slightly better deduplication rate for KERNEL. This is because there is a large amount of zero chunks in KERNEL and BLK-CAC effectively deduplicates such zero chunks as presented in Section III-D2.

Overall, BLK-CAC reduces the total write traffic by 77% on average compared to BLK-FSC. Although almost 90% of two snapshots of WIKI have the same contents, BLK-FSC fails to deduplicate showing the deduplication rate of only 0.014%. This is because the contents of many fixed-size chunks are affected by small insertions or deletions under BLK-FSC. In the same situation, BLK-CAC achieves the deduplication rate of 23%. Compared to FILE-CAC, the decrease in the deduplication rate of BLK-CAC is due to boundary chunks which are not deduplicated.

The results of BLK-CAC include all the optimizations presented in Section III-D with the default parameter values shown in Table I. The effect of each optimization under different parameter values is discussed in Section IV-E.

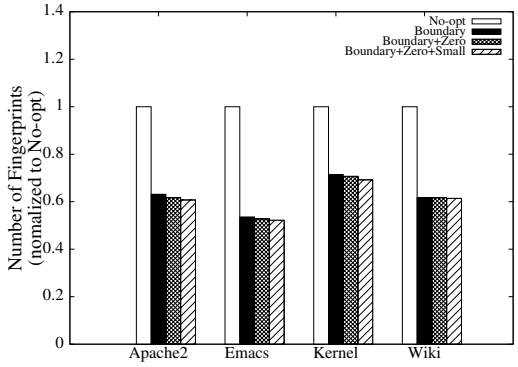

Fig. 6: The number of entries in the fingerprint store after applying various optimizations (normalized to No-opt)

C. Impact of Each Optimization

To investigate the impact of various optimizations on the deduplication rate, we apply each optimization incrementally. In Figure 5, No-opt represents the baseline BLK-CAC scheme where no optimization is performed. In Boundary, we eliminate boundary chunks from the fingerprint store. Additionally, zero chunks are handled as presented in Section III-D2 in Boundary+Zero. Boundary+Zero+Small represents the case where all the optimizations described in Section III-D are applied.

Figure 5 shows that the average deduplication rate drops from 45% to 39% when deduplication is not tried for boundary chunks. However, the special handling of zero chunks recovers the average deduplication rate to 46%. The impact of handling zero chunks is noticeable except for WIKI because APACHE2, EMACS, and KERNEL generate a lot of zero chunks due to small files. Additional elimination of small chunks has hardly affected the deduplication rate. Each optimization has no noticeable impact when the file size is large as in WIKI. By applying all the optimizations, the deduplication rate is increased by 1.4% on average.

	F (MB)	B	S_b (MB)	S_a (MB)	C_b	C_a
APACHE2	152	49,567	194	76	101,140	62,405
EMACS	561	156,266	610	278	394,271	205,097
KERNEL	703	232,584	909	518	849,899	585,723
WIKI	2,144	51964	2,158	1,652	2,836,243	1,734,652

TABLE II: Summary of BLK-CAC results. (F: the total file size, B: the number of 4KB blocks, S_b (or S_a): the total SSD space used before (or after) deduplication, C_b (or C_a): the total number of chunks in the fingerprint store before (or after) optimizations.)

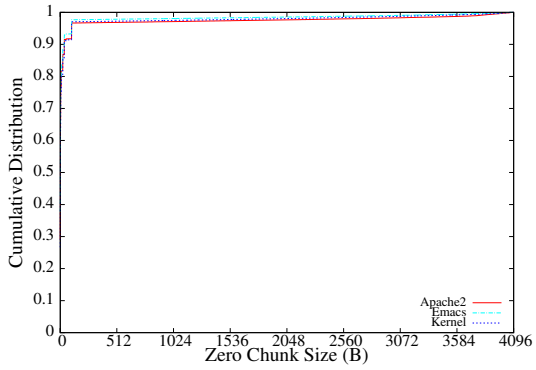

Fig. 7: The cumulative distribution of the number of zero chunks

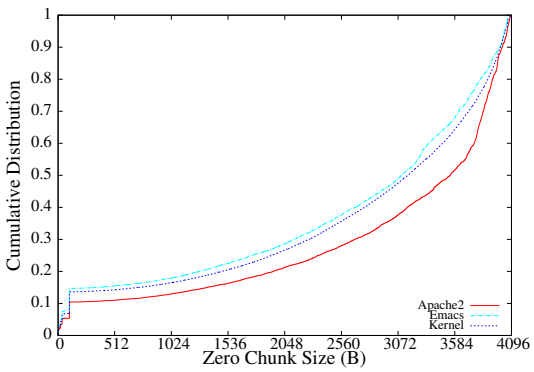

Fig. 8: The cumulative distribution of the amount of space occupied by zero chunks

D. Memory Footprint

Figure 6 depicts the number of entries in the fingerprint store normalized to the value of No-opt. The less number of fingerprints means that the less memory footprint is needed for the fingerprint store. We can see that the impact of removing boundary chunk is quite substantial in reducing the required size of the fingerprint store. Getting rid of zero chunks and small chunks from the fingerprint store also helps to save memory, but the effect is rather limited. The overall memory consumption of the fingerprint store is reduced by 39% on average when we apply all the optimizations. Table II summarizes the results of the proposed BLK-CAC scheme with all optimizations enabled. In Table II, the final deduplication rate is given by $\frac{(S_b - S_a)}{S_b}$.

E. Effect of Parameter Values

The proposed BLK-CAC scheme relies on three parameters, namely, the minimum size of zero chunks (z), the minimum size of non-zero chunks (x), and the maximum number of chunks in a block (n) (cf. Table I)[2]. The optimal values of these parameters are important to get better deduplication rate and lower memory footprint. This subsection investigates the impact of these parameters on the overall performance.

1) Minimum Size of Zero Chunks (z): When handling zero chunks, BLK-CAC regards a series of consecutive zeroes as a zero chunk only if it is equal to or larger than the minimum size of zero chunks, z. The larger value of z results in less number of zero chunks, however it also reduces the deduplication rate. Thus, we examine the relationship between the value of z and the deduplication rate.

Figure 7 and Figure 8 depict the cumulative distributions of the number of chunks and the total amount of space occupied by zero chunks, respectively. We measure the results for APACHE2, EMACS, and KERNEL as WIKI has only few zero chunks. From Figure 7, we can see that 97% of zero chunks are smaller than 105 bytes. However, these small zero chunks take only 13% of the total bytes of zeroes as illustrated in Figure 8. Thus, if we remove small zero chunks, we can reduce a large number of entries in the fingerprint store while minimizing its impact on the deduplication rate.

Figure 9 presents the changes in the deduplication rate when we vary the value of z from 1 byte to 2048 bytes. The results are normalized to the case where the optimization for zero chunks is not enabled. Until the value of z increases to 512 bytes, the deduplication rate is almost same with the case which regards all zeroes as zero chunks (the leftmost point in each line). However, the deduplication rate of APACHE2 starts to fall evidently when the minimum zero chunk size becomes 1024 bytes. From Figure 9, we set the value of z to 512 bytes to avoid a decrease in the deduplication rate. In this case, we can increase the deduplication rate significantly while removing a large number of entries corresponding to zero chunks in the fingerprint store.

[2]The block size (b) is usually set to the file system block size. We fix the average chunk size (c) to 512 bytes which balances deduplication rate and memory footprint.

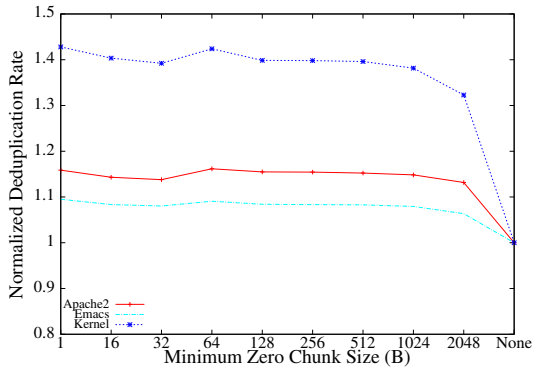

Fig. 9: Changes in the deduplication rate according to the minimum zero chunk size (normalized to the results which do not consider zero chunks)

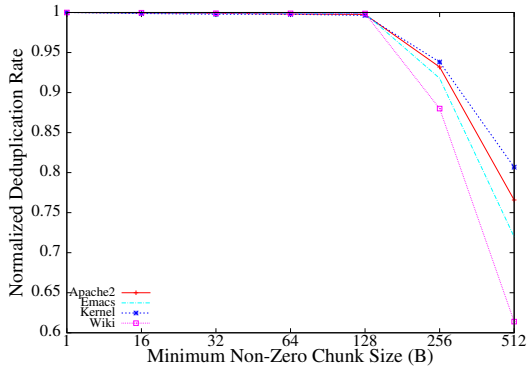

Fig. 10: Changes in the deduplication rate according to the minimum non-zero chunk size (normalized to the results which do not consider small chunks)

2) Minimum Size of Non-Zero Chunks (x): As explained in Section III-D, small chunks make little contribution to the overall deduplication rate. To eliminate those small chunks as many as possible, we measure the deduplication rate while varying the minimum size of non-zero chunks (x) from 1 byte to 512 bytes. Figure 10 depicts the changes in the deduplication rate whose results are normalized to the case where small chunks are not specially handled.

Figure 10 shows that the deduplication rate remains the same until the value of x becomes 128 bytes. When we increase the value of x further beyond 128 bytes, the deduplication rate is decreased sharply. This means that eliminating small chunks whose sizes are less than 128 bytes from the fingerprint store does not harm the deduplication rate. Therefore, we choose 128 bytes as the default value of x.

3) Maximum Number of Chunks in a Block (n): The total size of the mapping table depends on the maximum number of chunks in a block (n). The cumulative distribution of the number of chunks in a block is depicted in Figure 11. We can observe that the contents of most blocks are divided to less than 12 chunks.

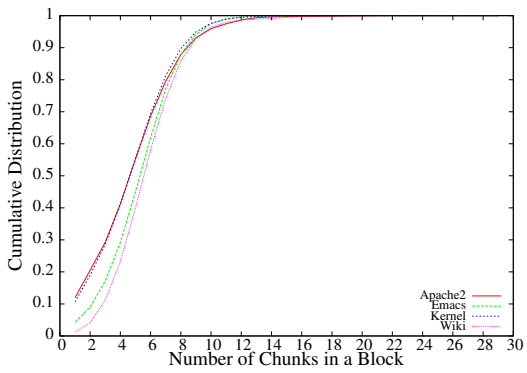

Fig. 11: The cumulative distribution of the number of chunks in a block

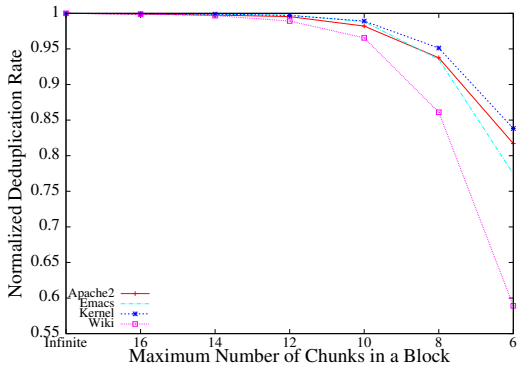

Fig. 12: Changes in the deduplication rate according to the maximum number of chunks in a block (normalized to the results when there is no such limitation)

Figure 12 illustrates the changes in the deduplication rate while we vary the maximum number of chunks in a block from 16 to 6. The results are normalized to the case when there is no such limitation. As the value of n decreases, the deduplication rate also diminishes because the last chunk of a block is not properly deduplicated. There is only about 2% drop in the deduplication rate until the value of n is larger than 12. When we decrease the value of n less than 12, the deduplication rate starts to be affected severely. According to these results, it is reasonable to set the default value of n to 12.

V. Related Work

As deduplication can improve the performance and lifespan of SSDs, there have been several approaches to implement deduplication in SSDs. Berman et al. [22] propose an integrate deduplication and write module in the SSD controller. However, they only present the basic idea and the expected benefit through a simple analytical model.

Chen et al. [2] suggest CAFTL (Content-Aware Flash Translation Layer) to enhance the endurance of SSDs by removing duplicated writes. To reduce the overhead of garbage

collection, they utilize 2-level indirect mapping. They also introduce several techniques to accelerate fingerprinting with small buffer. Gupta et al. [3] propose CA-SSD (Content-Addressable SSD) which employs deduplication on SSDs to exploit value locality. The value locality means that certain contents of data are likely to be accessed preferentially. Thus, CA-SSD improves the performance by caching small number of fingerprints.

Kim et al. [23] develop a mathematical model to calculate the efficiency of deduplication in SSDs and implement the prototype on real SSDs. They analyze the trade offs among design choices to implement deduplication using various combinations of hardware and software techniques. Lee et al. [24] combine several schemes such as deduplication, compression, and performance throttling to maximize the lifetime of SSDs.

However, all the aforementioned previous work employ the block-level fixed-size chunking scheme for deduplication in SSDs. We believe that the use of our proposed block-level content-aware chunking promises higher deduplication rate and longer lifespan of SSDs compared to the convention approach.

VI. Conclusion

We propose a novel deduplication scheme called block-level content-aware chunking to enhance the deduplication rate in SSDs. The baseline version of the proposed scheme enhances the average deduplication rate by 77% compared to the traditional block-level fixed-size chunking, and even shows the performance similar to the file-level content-aware chunking in some cases. We also suggest several optimizations to reduce memory footprint and write traffic. When we apply all the optimizations, the memory consumption required for the fingerprint store is reduced by 39% on average with a slight increase in the average deduplication rate compared to the baseline version.

We have been very successful in reducing the number of entries kept in the fingerprint store through several optimizations. Our future work includes studying of the specialized hardware logic [23] or applying more sophisticated techniques such as pre-hashing and sampling [2] to accelerate the fingerprinting and fingerprint matching process. We also plan to investigate a more efficient mechanism for managing the mapping information and the fingerprint store for the block-level content-aware chunking.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2013R1A2A1A01016441). This work was also supported by the IT R&D program of MKE/KEIT (No.10041244, SmartTV 2.0 Software Platform).

References

[1] L. Grupp, J. Davis, and S. Swanson, "The bleak future of NAND flash memory," in *Proc. USENIX Conference on File and Storage Technologies*, 2012.

[2] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the life span of flash memory based solid state drives," in *Proc. USENIX Conference on File and Storage Technologies*, 2011, pp. 6–6.

[3] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based ssds," in *Proc. USENIX Conference on File and Storage Technologies*, 2011, pp. 7–7.

[4] "Emc glossary: Data deduplication rate," http://www.emc.com/corporate/glossary/data-deduplication-rate.htm.

[5] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 18, July 2007.

[6] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 229–240.

[7] L. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *Proc. ACM Symposium on Applied Computing*, 2007, pp. 1126–1130.

[8] Y. Chang, J. Hsieh, and T. Kuo, "Improving flash wear-leveling by proactively moving static data," *IEEE Transactions on Computers*, vol. 59, January 2010.

[9] Y. Park and J. Kim, "zFTL: Power-efficient data compression support for nand flash-based consumer electronics devices," *Proc. IEEE Transactions on Consumer Electronics*, vol. 57, no. 3, pp. 1148–1156, August 2011.

[10] R. Rivest, "The MD5 message-digest algorithm," in *Proc. Cryptology Conference on Advances in Cryptology*, 1990, pp. 303–311.

[11] "Secure hash standard," Federal Information Processing Standards Publication 180-1, 1995.

[12] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proc. ACM Symposium on Operating Systems Principles*, 2001, pp. 174–187.

[13] L. Freeman, "How safe is deduplication," http://media.netapp.com/documents/tot0608.pdf, NatApp, Tech. Rep., 2008.

[14] M. Rabin, "Fingerprinting by random polynomials," Harvard University, Report TR-15-81, Tech. Rep., 1981.

[15] I. Papapanagiotou, R. Callaway, and M. Devetsikiotis, "Chunk and object level deduplication for web optimization: a hybrid approach," in *Proc. International Communications Conference*, 2012.

[16] "Sandforce SSDs break TPC-C records," http://semiaccurate.com/2010/05/03/sandforce-ssds-break-tpc-c-records/.

[17] "The OpenSSD Project," http://www.openssd-project.org/.

[18] "Ubuntu Source Code Repository for Apache2 (versions 2.2.8. 2.2.14, 2.2.17, 2.2.20, and 2.2.22)," http://ftp.ubuntu.com/ubuntu/pool/main/a/apache2/.

[19] "Ubuntu Source Code Repository for Emacs (versions 23.1, 23.2, 23.3, 23.4, 24.1 and 24.2)," http://ftp.ubuntu.com/ubuntu/pool/main/e/emacs23/, http://ftp.ubuntu.com/ubuntu/pool/main/e/emacs24/.

[20] "Linux Kernel Source Code Repository (versions 2.6.32.60 and 2.6.34.14)," http://www.kernel.org/pub/linux/kernel/.

[21] http://dumps.wikimedia.org/.

[22] A. Berman, "Integrating de-duplication and write for increased performance and endurance of solid-state drives," in *Proc. IEEE Convention of Electrical and Electronics Engineers*, 2010, pp. 821–823.

[23] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H. Lee, S. Kang, Y. Won, and J. Cha, "Deduplication in SSDs: Model and quantitative analysis," in *Proc. International Conference on Mass Storage Systems and Technologies*, 2012, pp. 1–12.

[24] S. Lee, T. Kim, J. Park, and J. Kim, "An integrated approach for managing the lifetime of flash-based SSDs," in *Proc. Conference on Design, Automation and Test*, 2013, pp. 1522–1525.