Reducing Solid-State Storage Device Write Stress through Opportunistic In-place Delta Compression

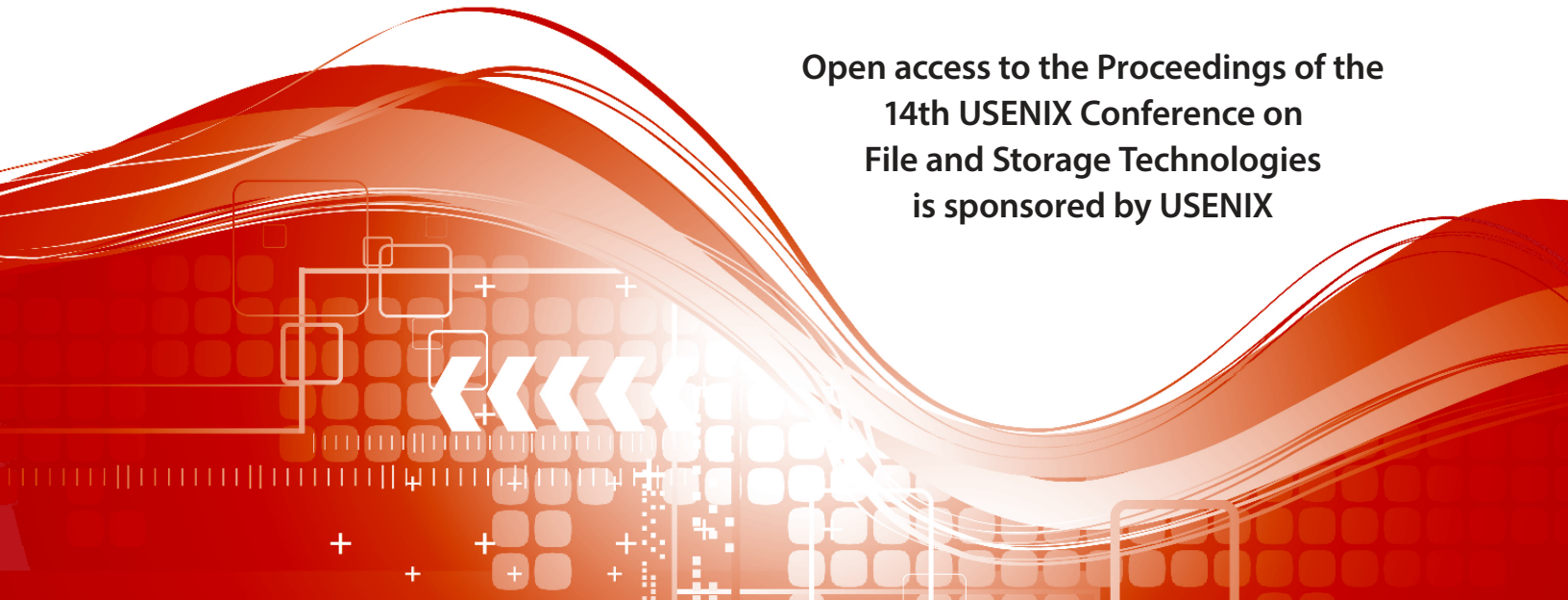
Xuebin Zhang, Jiangpeng Li, and Hao Wang, ***Rensselaer Polytechnic Institute;***
Kai Zhao, ***SanDisk Corporation;*** **Tong Zhang,** ***Rensselaer Polytechnic Institute***

https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-xuebin

This paper is included in the Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16).

February 22–25, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-28-7



Open access to the Proceedings of the 14th USENIX Conference on File and Storage Technologies is sponsored by USENIX

Reducing Solid-State Storage Device Write Stress Through Opportunistic In-Place Delta Compression

Xuebin Zhang*, Jiangpeng Li*, Hao Wang*, Kai Zhao† and Tong Zhang*
*ECSE Department, Rensselaer Polytechnic Institute, USA
†SanDisk Corporation, USA
{xuebinzhang.rpi@gmail.com, tzhang@ecse.rpi.edu}

Abstract

Inside modern SSDs, a small portion of MLC/TLC NAND flash memory blocks operate in SLC-mode to serve as write buffer/cache and/or store hot data. These SLC-mode blocks absorb a large percentage of write operations. To balance memory wear-out, such MLC/TLC-to-SLC configuration rotates among all the memory blocks inside SSDs. This paper presents a simple yet effective design approach to reduce write stress on SLC-mode flash blocks and hence improve the overall SSD lifetime. The key is to implement well-known delta compression without being subject to the read latency and data management complexity penalties inherent to conventional practice. The underlying theme is to leverage the partial programmability of SLC-mode flash memory pages to ensure that the original data and all the subsequent deltas always reside in the same memory physical page. To avoid the storage capacity overhead, we further propose to combine intra-sector lossless data compression with intra-page delta compression, leading to opportunistic in-place delta compression. This paper presents specific techniques to address important issues for its practical implementation, including data error correction, and intra-page data placement and management. We carried out comprehensive experiments, simulations, and ASIC (application-specific integrated circuit) design. The results show that the proposed design solution can largely reduce the write stress on SLC-mode flash memory pages without significant latency overhead and meanwhile incurs relatively small silicon implementation cost.

1 Introduction

Solid-state data storage built upon NAND flash memory is fundamentally changing the storage hierarchy for information technology infrastructure. Unfortunately, technology scaling inevitably brings the continuous degradation of flash memory endurance and write speed. Motivated by data access locality and heterogeneity in real-world applications, researchers have well demonstrated the effectiveness of complementing bulk MLC/TLC NAND flash memory with small-capacity SLC NAND flash memory to improve the endurance and write speed (e.g., see [1–3]). The key is to use SLC memory blocks serve as write buffer/cache and/or store relatively hot data. Such a design strategy has been widely adopted in commercial solid-state drives (SSDs) [4–6], where SSD controllers dynamically configure a small portion of MLC/TLC flash memory blocks to operate in SLC mode. The MLC/TLC-to-SLC configuration rotates throughout all the MLC/TLC flash memory blocks in order to balance the flash memory wear-out.

This paper is concerned with reducing the write stress on those SLC-mode flash memory blocks in SSDs. Aiming to serve as write buffer/cache and/or store hot data, SLC-mode flash memory blocks account for a large percentage of overall data write traffic [7]. Reducing their write stress can directly reduce the flash memory wear-out. Hence, when these SLC-mode memory blocks are configured back to operate as normal MLC/TLC memory blocks, they could have a long cycling endurance. Since a specific location tends to be repeatedly visited/updated within a short time (like consecutive metadata updates or in-place minor revisions of file content), it is not uncommon that data written into this SLC-mode flash based cache have abundant temporal redundancy. Intuitively, this feature makes the *delta compression* an appealing option to reduce the write stress. In fact, the abundance of data temporal redundancy in real systems has inspired many researchers to investigate the practical implementation of delta compression at different levels, such as filesystems [8, 9], block device [10–13] and FTL (Flash Translation Layer) [14]. Existing solutions store the original data and all the subsequent compressed deltas separately at different physical pages of the

storage devices. As a result, to serve a read request, they must fetch the original data and all the subsequent deltas from different physical pages, leading to inherent read amplification, particularly for small read request or largely accumulated delta compression. In addition, the system needs to keep the mapping information for the original data and all the compressed deltas, leading to a sophisticated data structure in the filesystem and/or firmware. These issues inevitably lead to significant read latency and hence a system performance penalty.

This paper aims to implement delta compression for SLC-mode flash memory blocks with small read latency penalty and very simple data management. First, we note that the read latency penalty inherent to existing delta compression design solutions is fundamentally due to the per-sector/page *atomic write* inside storage devices, which forces us to store the original data and all the subsequent deltas across different sectors/pages. Although per-sector atomic write is essential in hard disk drives (i.e., hard disk drives cannot perform partial write/update within one 4kB sector), per-page atomic write is not absolutely necessary in NAND flash memory. Through experiments with 20nm MLC NAND flash memory chips, we observed that SLC-mode pages can support partial programming, i.e., different portions of the same SLC-mode page can be programmed at different times. For example, given a 16kB flash memory page size, we do not have to write one entire 16kB page at once, and instead we can write one portion (e.g., 4kB or even a few bytes) at a time and finish writing the entire 16kB page over a period of time. This clearly warrants re-thinking the implementation of delta compression.

Leveraging the per-page partial-programming support of SLC-mode flash memory, we propose a solution to implement delta compression without incurring significant read latency penalty and complicating data management. The key idea is simple and can be described as follows. When a 4kB sector is being written the first time, we always try to compress it before writing to an SLC-mode flash memory page. Assume the flash memory page size is 16kB, we store four 4kB sectors in each page as normal practice. The use of per-sector lossless compression leaves some memory cells unused in the flash memory page. Taking advantage of the per-page partial-programming support of SLC-mode flash memory, we can directly use those unused memory cells to store subsequent deltas later on. As a result, the original data and all its subsequent deltas reside in the same SLC flash memory physical page. Since the runtime compression/decompression can be carried out by SSD controllers much faster than a flash memory page read, this can largely reduce the data access latency overhead in the realization of delta compression. In addition, it can clearly simplify data management since everything we need to re-construct the latest data is stored in a single flash page. This design strategy is referred to as *opportunistic in-place delta compression*.

For the practical implementation of the proposed design strategy, this paper presents two different approaches to layout the data within each SLC-mode flash memory page, aiming at different trade-offs between write stress reduction and flash-to-controller data transfer latency. We further develop a hybrid error correction coding (ECC) design scheme to cope with the significantly different data size among original data and compressed deltas. We carried out experiments and simulations to evaluate the effectiveness of proposed design solutions. First, we verified the feasibility of SLC-mode flash memory page partial programming using a PCIe FPGA-based flash memory characterization hardware prototype with 20nm MLC NAND flash memory chips. For the two different data layout approaches, we evaluated the write stress reduction under a variety of delta compression values, and quantitatively studied their overall latency comparison. To estimate the silicon cost induced by the hybrid ECC design scheme and on-the-fly compression/decompression, we further carried out ASIC (application-specific integrated circuit) design, and the results show that the silicon cost is not significant. In summary, the contributions of this paper include:

- We for the first time propose to cohesively integrate SLC-mode flash memory partial programmability, data compressibility and delta compressibility to reduce write stress on SLC-mode pages in SSDs without incurring significant read latency and storage capacity penalty;
- We develop specific solutions to address the data error correction and data management design issues in the proposed opportunistic delta-compression design strategy;
- We carried out comprehensive experiments to demonstrate its effectiveness on reducing write stress at small read latency overhead and show its practical silicon implementation feasibility.

2 Background and Motivation

2.1 Write Locality

The content temporal locality in storage system implies that one specific page could be visited for multiple times within a short time period. To quantitatively investigate this phenomenon, we analyzed several typical traces including Finance-1, Finance-2 [15], Homes [16] and Webmail Server traces [16], and their information is listed in Table 1. We analyzed the percentage of repeated LBA (logical block address) in the collected traces. Figure 1 shows the distribution of repeated

overwrite times within one hour. In the legend, '1' means a specific LBA is only visited once while '2-10' means an LBA is visited more than twice and less than 10 times. We can find more than 90% logical blocks are updated more than once in Finance-1 and Finance-2.

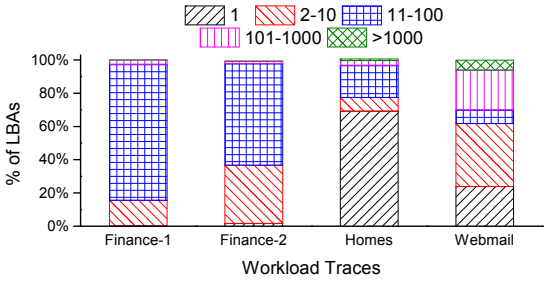


Figure 1: Percentage of repeated overwrite times of several typical workload traces.

Table 1: Disk traces information

Name	duration	# of unique LBAs	# of total LBAs
Finance-1	1h	109,177	3,051,388
Finance-2	1h	31,625	571,529
Homes	24h	20,730	28,947
Webmail	24h	6,853	16,514

Another noticeable characteristic in most applications is the partial page content overwrite or update. Authors in [17] revealed that more than 60% of write operations involve partial page overwrites and some write operations even only update less than 10 bytes. This implies a significant content similarity (or temporal redundancy) among consecutive data writes to the same LBA. However, due to the page-based data write in flash memory, such content temporal redundancy is however left unexplored in current conventional practice.

2.2 Delta Compression

Although delta compression can be realized at different levels spanning filesystems [8, 9], block device [10–13] and FTL [14], their basic strategy is very similar and can be illustrated in Figure 2. For the sake of simplicity, we consider the case of applying delta compression to the 4kB content at the LBA of L_a. Let C_0 denote the original content at the LBA of L_a, which is stored in one flash memory physical page P_0. At time T_1, we update the 4kB content at LBA of L_a with C_1. Under delta compression, we obtain the compressed delta between C_0 and C_1, denoted as d_1, and store in another flash memory physical page P_1. At time T_2, we update the content again with C_2. To maximize the delta compression efficiency, we obtain and store the compressed delta between C_2 and C_1, denoted as d_2. The process continues as we keep updating the content at the LBA of L_a, for which we need to keep the original content C_0 and all the subsequent deltas (i.e., $d_1, d_2, \cdots$).

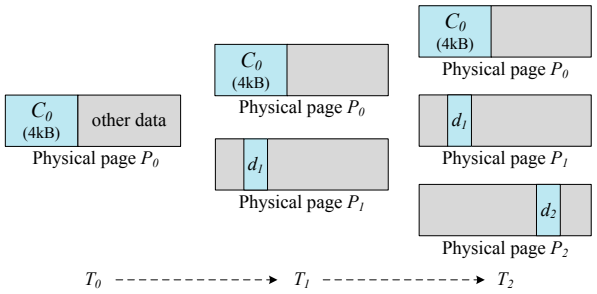


Figure 2: Illustration of conventional method for realizing temporal redundancy data compression.

Clearly, conventional practice could result in noticeable read latency penalty. In particular, to serve each read request, we must fetch the original data and all the deltas in order to re-construct the current content, leading to read amplification and hence latency penalty. In addition, it comes with sophisticated data structure and hence complicates data management, which could further complicate flash memory garbage collection. As a result, although delta compression can very naturally exploit abundant temporal redundancy inherent in many applications, it has not been widely deployed in practice.

2.3 Partial Programming

Through experiments with flash memory chips, we observed that SLC-mode NAND flash memory can readily support partial programming, i.e., different portions in an SLC flash memory page can be programmed at different time. This feature can be explained as follows. Each SLC flash memory cell can operate in either *erased* state or *programmed* state, corresponding to the storage of '1' and '0', respectively. At the beginning, all the memory cells within the same flash memory page are erased simultaneously, i.e., the storage of each memory cell is reset to be '1'. During runtime, if we write a '1' to one memory cell, memory chip internal circuits simply apply a prohibitive bit-line voltages to prevent this cell from being programmed; if we write a '0' to one memory cell, memory chip internal circuits apply a programming bit-line voltage to program this cell (i.e., move from erased state to programmed state). Meanwhile, a series of high voltage are applied to the word-line to enable programming. This can directly enable partial programming as illustrated in Figure 3: At the beginning of T_0, all the four memory cells m_1, m_2,

m_3 and m_4 are in the erased state, and we write '0' to memory cell m_3 and write '1' to the others. Internally, the memory chip applies programming bit-line voltage to m_3 and prohibitive bit-line voltage to the others, hence the storage content becomes {'1', '1', '0', '1'}. Later at time T_1, if we want to switch memory cell m_1 from '1' to '0', we write '0' to memory cell m_1 and '1' to the others. Accordingly, memory chip applies prohibitive bit-line voltage to the other three cells so that their states remain unchanged. As a result, the storage content becomes {'0', '1', '0', '1'}.

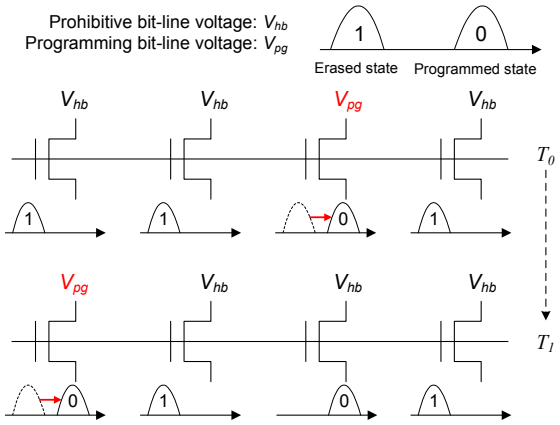


Figure 3: Illustration of the underlying physics enabling SLC-mode flash memory partial programming.

Therefore, we can carry out partial programming to SLC-mode flash memory pages as illustrated in Figure 4. Let $\mathbf{I}_s$ denote an all-one bit vector with the length of s. Given an erased SLC flash memory page with the size of L, we first write $[d_1, \cdots, d_n, \mathbf{I}_{L-n}]$ to partially program the first n memory cells and leave the rest $L-n$ memory cells intact. Later on, we can write $[\mathbf{I}_n, c_1, \cdots, c_m, \mathbf{I}_{L-n-m}]$ to partially program the next m memory cells and leave all the other memory cells intact. The same process can continue until the entire page has been programmed.

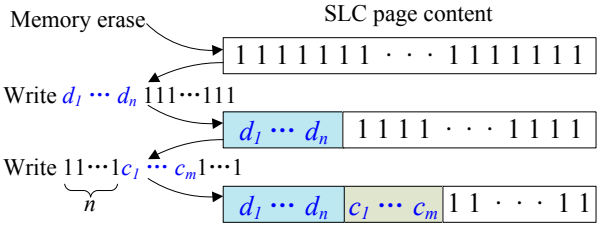


Figure 4: Illustration of SLC-mode flash memory partial programming.

Using 20nm MLC NAND flash memory chips, we carried out experiments and the results verify that the chips can support the partial programming when being operated in the SLC mode. In our experiments, we define "one cycle" as progressively applying partial programming for 8 times before one entire page is filled up and then being erased. In contrast, the conventional "one cycle" is to fully erase before each programming. Figure 5 demonstrates the bit error rate comparison of these two schemes. The flash memory can be used for 8000 cycles with the conventional way. The progressive partial programming can work for more than 7100 cycles. And this modest endurance reduction indicates that the partial programming mechanism does not bring noticeable extra physical damage to flash memory cells.

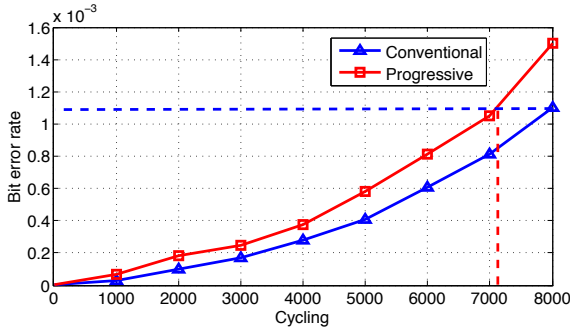


Figure 5: Comparison of the bit error rate of conventional programming and progressive partial programming.

3 Proposed Design Solution

Leveraging the partial programmability of SLC-mode flash memory, very intuitively we can deploy in-place delta compression, as illustrated in Figure 6, to eliminate the read latency penalty inherent to conventional design practice as described in Section 2.2. As shown in Figure 6, the original data content C_0 and all the subsequent deltas d_i's are progressively programmed into a single physical page. Once the physical page is full after the k-th update, or the number of deltas reaches a threshold T (we don't expect to accumulate too many deltas in case of a larger retrieval latency), we allocate a new physical page, write the latest version data C_{k+1} to the new physical page, and reset the delta compression for subsequent updates. This mechanism can guarantee that we only need to read a single flash memory page to retrieve the current data content.

In spite of the very simple basic concept, its practical implementation is subject to several non-trivial issues: (i) *Storage capacity utilization*: Suppose each flash memory page can store m (e.g., 4 or 8) 4kB sectors. The straightforward implementation of in-place delta compression explicitly reserves certain storage capacity within each SLC flash memory page for storing deltas. As a result, we can only store at most $m-1$ 4kB sectors per page at the very beginning. Due to the runtime

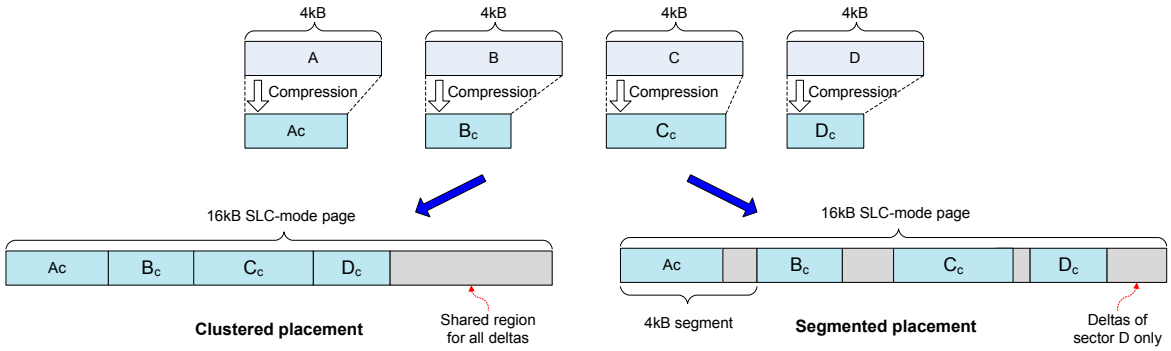


Figure 7: Illustration of opportunistic in-place delta compression and two different data placement strategies.

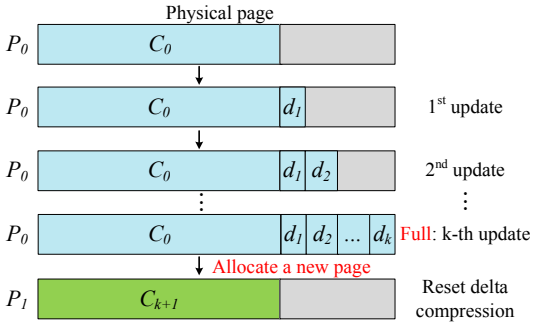


Figure 6: Illustration of the basic concept of in-place delta compression.

variation of the delta compressibility among all the data, these explicitly reserved storage space may not be highly utilized. This clearly results in storage capacity penalty. In addition, by changing the number of 4kB sectors per page, it may complicate the design of FTL. (ii) *Error correction*: All the data in flash memory must be protected by ECC. Due to the largely different size among the original data and all the deltas, the ECC must be devised differently. In particular, the widely used low-density parity-check (LDPC) codes are only suitable for protecting large data chunk size (e.g., 2kB or 4kB), while each delta can only be a few tens of bytes. In the remainder of this section, we present design techniques to address these issues and discuss the involved trade-offs.

3.1 Opportunistic In-place Delta Compression

To eliminate the storage capacity penalty, we propose to complement delta compression with intra-sector lossless data compression. In particular, we apply lossless data compression to each individual 4kB sector being written to an SLC-mode flash memory page, and opportunistically utilize the storage space left by compression for storing subsequent deltas. This is referred to as opportunistic in-place delta compression. This is illustrated in Figure 7, where we assume the flash memory page size is 16kB. Given four 4kB sectors denoted as A, B, C, and D, we first apply lossless data compression to each sector individually and obtain A_c, B_c, C_c, and D_c. As shown in Figure 7, we can place these four compressed sectors into a 16kB SLC-mode flash memory page in two different ways:

1. *Clustered placement*: All the four compressed sectors are stored consecutively, and the remaining space within the 16kB page can store any deltas associated with these four sectors.
2. *Segmented placement*: Each 16kB SLC-mode flash memory page is partitioned into four 4kB segments, and each segment is dedicated for storing one compressed sector and its subsequent deltas.

These two different placement strategies have different trade-offs between delta compression efficiency and read latency. For the clustered placement, the four sectors share a relatively large residual storage space for storing subsequent deltas. Hence, we may expect that more deltas can be accumulated within the same physical page, leading to a higher delta compression efficiency. However, since the storage of original content and deltas of all the four sectors are mixed together, we have to transfer the entire 16kB from flash memory to SSD controller in order to reconstruct the current version of any one sector, leading to a longer flash-to-controller data transfer latency. On the other hand, in the case of segmented placement, we only need to transfer a 4kB segment from flash memory to SSD controller to serve one read request. Meanwhile, since the deltas associated with each sector can only stored within one 4kB segment, leading to lower delta compression efficiency compared with the case of clustered placement. In addition, segmented placement tends to have lower computational complexity than clustered placement, which will be further elaborated later in Section 3.3.

3.2 Hybrid ECC and Data Structure

The above opportunistic in-place delta compression demands a careful design of data error correction and overall data structure. As illustrated in Figure 8, we must store three types of data elements: (1) compressed sector, (2) delta, and (3) header. Each compressed sector and delta follows one header that contains all the necessary metadata (e.g., element length and ECC configuration). Each element must be protected individually by one ECC codeword. In addition, each header should contain an unique marker to identify a valid header. Since all the unwritten memory cells have the value of 1, we can use an all-zero bit vector as the header marker.

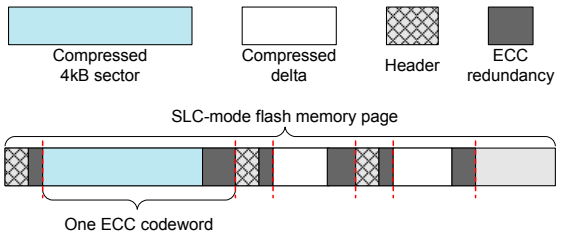


Figure 8: Illustration of three types of data elements, all of which must be protected by ECC.

Since all the elements have different different size, the ECC coding must natively support variable ECC codeword length, for which we can use the codeword puncturing [18]. Given an (n,k) ECC that protects k-bit user data with $(n-k)$-bit redundancy. If we want to use this ECC to protect m-bit user data $\mathbf{u}_m$ (where $m<k$), we first pad $(k-m)$-bit all-zero vector $\mathbf{O}_{k-m}$ to form a k-bit vector $[\mathbf{u}_m, \mathbf{O}_{k-m}]$. We encode the k-bit vector to generate $(n-k)$-bit $\mathbf{r}_{n-k}$ of redundancy, leading to an n-bit codeword $[\mathbf{u}_m, \mathbf{O}_{k-m}, \mathbf{r}_{n-k}]$. Then we remove the $(k-m)$-bit all-zero vector $\mathbf{O}_{k-m}$ from the codeword to form an $(n+m-k)$-bit *punctured* ECC codeword $[\mathbf{u}_m, \mathbf{r}_{n-k}]$, which is stored into flash memory. To read the data, we retrieve the noisy version of the codeword, denoted as $[\tilde{\mathbf{u}}_m, \tilde{\mathbf{r}}_{n-k}]$, and insert $(k-m)$-bit all-zero vector $\mathbf{O}_{k-m}$ back to form an n-bit vector $[\tilde{\mathbf{u}}_m, \mathbf{O}_{k-m}, \tilde{\mathbf{r}}_{n-k}]$, to which we apply ECC decoding to recover the user data $\mathbf{u}_m$.

In order to avoid wasting too much coding redundancy, the ratio of m/k in ECC puncturing should not be too small (i.e., we should not puncture too many bits). Hence, instead of using a single ECC, we should use multiple ECCs with different codeword length to accommodate the large variation of data element length. To protect relatively long data elements (in particular the compressed 4kB sectors), we can use three LDPC codes with different codeword length, denoted as $LDPC_{4kB}$, $LDPC_{2kB}$, and $LDPC_{1kB}$. The code $LDPC_{4kB}$ protects all the elements with the length bigger than 2kB, the code $LDPC_{2kB}$ protects all the elements with the length within 1kB and 2kB, and the code $LDPC_{1kB}$ protects all the elements with the length within 512B and 1kB. Thanks to recent work on versatile LDPC coding system design [19, 20], all the three LDPC codes can share the same silicon encoder and decoder, leading to negligible silicon penalty in support of multiple LDPC codes. Since LDPC codes can only work with relatively large codeword length (i.e., 1kB and beyond) due to the error floor issue [21], we have to use a set of BCH codes to protect all the elements with the length less than 512B. BCH codes with different codeword length are constructed under different Galois Fields, hence cannot share the same silicon encoder and decoder. In this work, we propose to use three different BCH codes, denoted as BCH_{4B}, BCH_{128B}, and BCH_{512B}, which can protect 4B, 128B, and 512B, respectively. We fix the size of element header as 4B, and the BCH_{4B} aims to protect each element header. The code BCH_{512B} protects all the elements with the length within 128B and 512B, and the code BCH_{128B} protects all the non-header elements with the length of less than 128B.

3.3 Overall Implementation

Based upon the above discussions, this subsection presents the overall implementation flow of the proposed opportunistic in-place delta compression design framework. Figure 9 shows the flow diagram for realizing delta compression to reduce write stress. Upon a request of writing 4kB sector C_k at a given LBA within the SLC-mode flash memory region, we retrieve and re-construct the current version of the data C_{k-1} from an SLC-mode physical page. Then we obtain the compressed delta between C_k and C_{k-1}, denoted as d_k. Accordingly we generate its header and apply ECC encoding to both the header and compressed delta d_k, which altogether form a bit-vector denoted as p_k. If there is enough space in this SLC-mode page and the number of existing deltas is smaller than the threshold T, we write p_k into the page through partial programming; otherwise we allocate a new physical page, compress the current version C_k and write it to this new page to reset the delta compression. In addition, if the original sector is not compressible, like video or photos, we simply write the original content to flash memory without adding a header. Meanwhile, we write a special marker bit to the reserved flash page metadata area [22]. During the read operation, if the controller detected the marker, it will know that this sector is written uncompressed.

The key operation in the process shown in Figure 9 is the data retrieval and reconstruction. As discussed in Section 3.1, we can use two different intra-page data placement strategies, i.e., clustered placement and segmented placement, for which the data retrieval

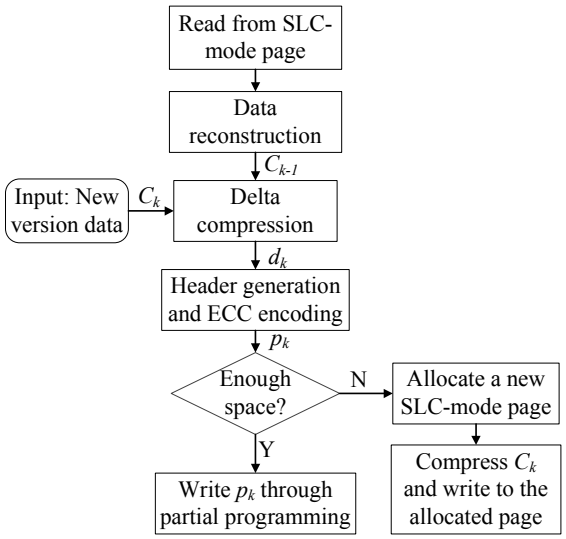


Figure 9: Flow diagram for realizing delta compression.

and reconstruction operation involves different latency overhead and computational complexity. In short, compared with clustered placement, segmented placement has shorter latency and less computational complexity. This can be illustrated through the following example. Suppose a single 16kB flash page contains four compressed 4kB sectors, A_c, B_c, C_c, and D_c. Associated with each sector, there is one compressed delta, $d_{A,1}$, $d_{B,1}$, $d_{C,1}$, and $d_{D,1}$. Each of these eight data elements follows a header, hence we have total eight headers. Suppose we need to read the current content of sector B, the data retrieval and reconstruction process can be described as follows:

- In the case of clustered placement, the SSD controller must retrieve and scan the entire 16kB flash memory page. It must decode and analyze all the eight headers to determine whether to decode or skip the next data element (compressed sector or delta). During the process, it carries out further ECC decoding to obtain B_c and $d_{B,1}$, based upon which it performs decompression and accordingly reconstruct the current content of sector B.
- In the case of segmented placement, the SSD controller only retrieves and scans the second 4kB from from the 16kB flash memory page. As a result, it only decodes and analyzes two headers, and accordingly decodes and decompresses B_c and $d_{B,1}$, and finally reconstructs the current content of sector B.

From above simple example, it is clear that, compared with clustered placement, segmented placement largely reduces the amount of data being transferred from flash memory chips to SSD controller, and involves a fewer number of header ECC decoding. This leads to lower latency and less computation. On the other hand, clustered placement tends to have a better storage efficiency by allowing different sectors to share the same storage region for storing deltas.

Thus the proposed design solution essentially eliminates read amplification and filesystem/firmware design overhead, which are two fundamental drawbacks inherent to conventional practice. Meanwhile, by opportunistically exploiting lossless compressibility inherent to data content itself, this design solution does not incur a storage capacity penalty on the SLC-mode flash memory region in SSDs.

Based upon the above discussions, we may find that a noticeable write traffic reduction could be expected with a good compression efficiency and delta compression efficiency. So if the data content is not compressible (like multimedia data or encrypted data), the reduction would be limited. In addition, another application condition is that the proposed design solution favors update-in-place file system because only the write requests to the same LBA have a chance to be combined to the same physical page. Therefore, the proposed technique could not be very conveniently applied to some log-structured file system like F2FS, LFS because the in-place update is not inherently supported in the logging area of these file systems. And besides, the proposed design solution can be integrated with other appearing features of SSD such as encryption. SSDs are using high performance hardware modules to implement encryption. And the data/delta compression will not be affected if the encryption module is placed after compression.

4 Evaluations

This section presents our experimental and simulation results to quantitatively demonstrate the effectiveness and involved trade-offs of our proposed design solution.

4.1 Per-sector Compressibility

To evaluate the potential of compressing each original 4kB sector to opportunistically create space for deltas, we measured the per-4kB-sector data compressibility on different data types. We collected a large amount of 4kB data sectors from various database files, document files, and filesystem metadata. These types of data tend to be relatively hot and frequently updated, hence more likely reside in the SLC-mode region in SSDs.

We use the sample databases from [23, 24] to test the compressibility of MySQL database files. MySQL database uses pre-allocated data file, hence we ignored the unfilled data segments when we measured the compression ratio distribution. The Excel/Text datasets were collected from an internal experiment lab server. We used Linux Kernel 3.11.10 source [25] as the source code dataset. We collected the metadata (more than

34MB) of files in an ext4 partition as the metadata dataset. Figure 10 shows the compressibility of different data types with LZ77 compression algorithm. The *compression ratio* is defined as the ratio of the size after compression to before compression, thus a smaller ratio means a better compressibility. As shown in Figure 10, data compression ratio tends to follow a Gaussian-like distribution, while different datasets have largely different mean and variation. Because each delta tends to be much smaller than 4kB, the results show that the simple LZ77 compression is sufficient to leave enough storage space for storing multiple deltas.

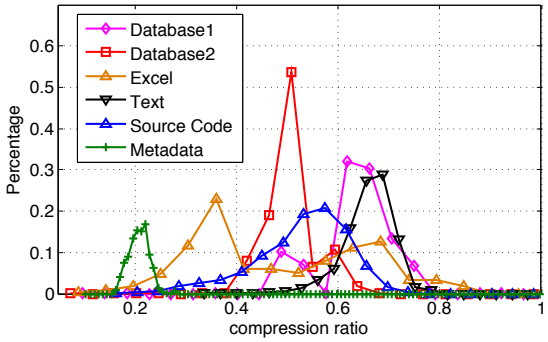


Figure 10: Compression ratio distribution of different data types with LZ77 compression.

4.2 Write Stress Reduction

We further evaluated the effectiveness of using the proposed opportunistic in-place delta compression to reduce the flash memory write stress. Clearly, the effectiveness heavily depends on the per-sector data compressibility and delta compressibility. Although per-sector data compressibility can be relatively easily obtained as shown in Section 4.1, empirical measurement of the delta compressibility is non-trivial. Due to the relative update regularity and controllability of filesystem metadata, we empirically measured the delta compressibility of metadata, based upon which we analyzed the write stress reduction for metadata. To cover the other types of data, we carried out analysis by assuming a range of Gaussian-like distributions of delta compressibility following prior work [10, 13].

4.2.1 A Special Case Study: Filesystem Metadata

To measure the metadata delta compressibility, we modified Mobibench [26] to make it work as the I/O workload benchmark under Linux Ubuntu 14.04 Desktop. We use a large set of SQLite workloads (`create, insert, update, delete`) and general filesystem tasks (file `read, update, append`) to trigger a large amount of file metadata updates. To monitor the characteristics of metadata, based upon the existing tool `debugfs` [27], we implemented a metadata analyzer tool [28] to track, extract, and analyze the filesystem metadata. We use an ext4 filesystem as the experimental environment and set the system page cache write back period as 500ms. Every time before we collect the file metadata, we wait for 1s to ensure that file metadata are flushed back to the storage device. For each workload, we collected 1000 consecutive versions of metadata.

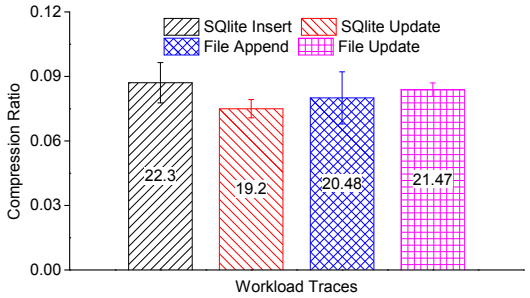


Figure 11: Delta compression ratio of consecutive versions of metadata for different workloads.

Based on the collected consecutive versions of metadata, we measured the delta compressibility as shown in Figure 11. The number inside the bar indicates the average number of bytes needed to store the difference between two consecutive versions of metadata, while the complete size of ext4 file metadata is 256 byte. The average delta compression ratio is 1:0.087 with the standard deviation of 0.0096. The results indicate that the delta compression ratio is quite stable with a very small deviation.

Figure 12: Number of flash memory pages being programmed for storing 1000 consecutive versions of metadata. (In comparison with conventional practice, we need at most 1000 pages to store these versions.)

The results in Figure 11 clearly suggest the significant data volume reduction potential by applying delta compression for metadata. To estimate the corresponding

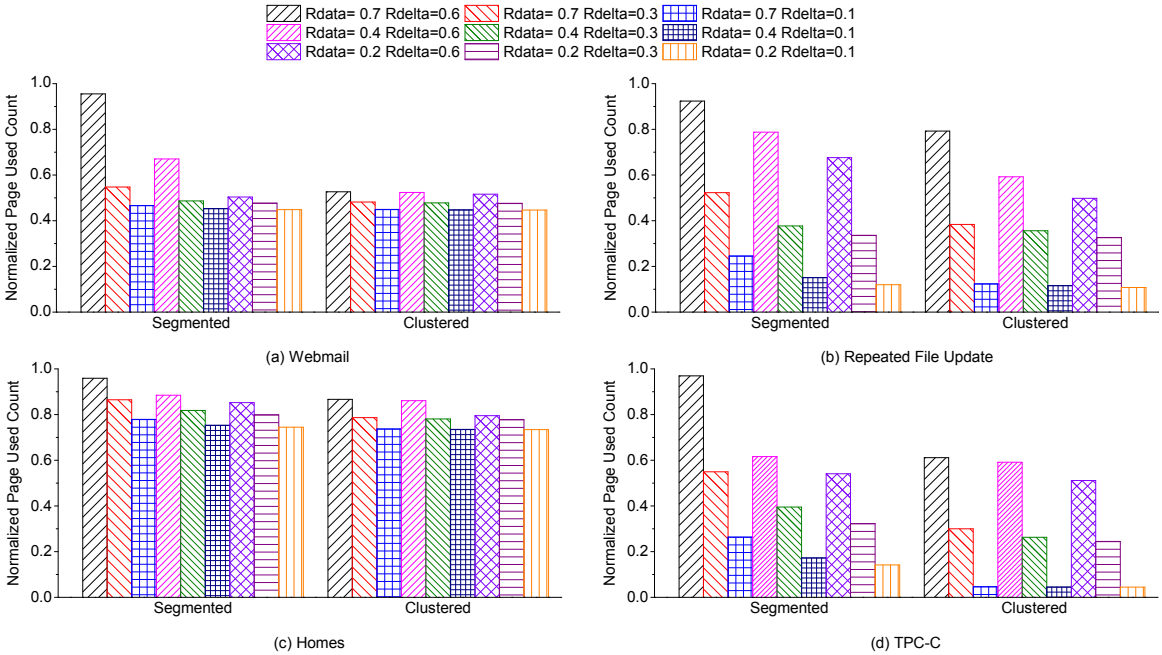


Figure 13: Reduction of the number of programmed flash memory pages under different workloads and over different data compressibility.

write stress reduction, we set that each SLC-mode flash memory page is 16kB and stores four compressed 4kB sectors and their deltas. Figure 12 shows the average number of flash memory pages that must be programmed in order to store 1000 consecutive versions of metadata pages. We considered the use of both segmented placement and clustered placement design strategies as presented in Section 3.1. Thanks to the very good per-sector compressibility and delta compressibility of metadata, the flash memory write stress can be reduced by over 20×. In addition, by allowing all the four sectors share the space for storing deltas, clustered placement can achieve higher write stress reduction than segmented placement, as shown in Figure 12.

4.2.2 Analytical Results for General Cases

Prior work [10, 13, 14] modeled delta compressibility to follow Gaussian-like distributions. To facilitate the evaluation over a broader range of data types, we follow this Gaussian distribution based model in these work as well. Let R_{data} denote the mean of the per-sector compression ratio of original data, and let R_{delta} denote the mean of delta compression ratio. Based upon the results shown in Section 4.1, we considered three different values of R_{data}, i.e., 0.2, 0.4, and 0.7. scenarios. According to prior work [10, 13, 14], we considered three different values of R_{delta}, i.e., 0.1, 0.3, and 0.6. Meanwhile, we set the value of deviation to 10% of the corresponding value of mean according to our measurements in Section 4.1.

In this section, we carried out simulations to estimate the flash memory write stress reduction over different workloads, and the results are shown in Figure 13. We chose the following four representative workloads:

- *Webmail Server*: We used Webmail Server block trace from [16], which was obtained from a department mail sever and the activities include mail editing, saving, backing up, etc.
- *Repeated File Update*: We enhanced the benchmark in [26] to generate a series of file updating in an Android Tablet, and accordingly captured the block IO traces.
- *Home*: We used the Homes Traces in [16], which include a research group activities of developing, testing, experiments, technical writing, plotting, etc.
- *Transaction*: We executed TPC-C benchmarks (10 warehouses) for transaction processing on MySQL 5.1 database system. We ran the benchmarks and use blktrace tool to obtain the corresponding traces.

As shown in Figure 13, the write stress can be noticeably reduced by using the proposed design solution (a smaller value in figure indicates a better stress reduction). In the "Repeated File Update" and TPC-C workloads, the number of programmed flash memory pages can be reduced by over 80%. The results clearly show that the flash memory write stress reduction is reversely proportional to R_{data} and R_{delta}, which can

be intuitively justified. When both the original data and delta information cannot be compressed efficiently (such as R_{data} is 0.7 and R_{delta} is 0.6), the write stress can be hardly reduced because the compressed delta cannot be placed in the same page with the original data. However, with the clustered data placement strategy, some deltas could be placed because of a larger shared spare space. Thus the clustered data placement strategy has a better performance than the segmented approach in most of the cases, especially when the compression efficiency is relatively poor.

The write stress reduction varies among different workloads and strongly depends on the data update operation frequency. For example, with a large percentage of data updates than "Homes", "Repeated File Update" can achieve noticeably better write stress reduction as shown in Figure 13. In essence, there exists a upper bound of write stress reduction, which is proportional to the percentage of update operations. This explains why the write stress reduction cannot be further noticeably reduced even with better data compressibility, as shown in Figure 13.

4.3 Implementation Overhead Analysis

This subsection discusses and analyzes the overhead caused by the proposed design solution in terms of read latency, update latency, and SSD controller silicon cost.

4.3.1 Read Latency Overhead

Figure 14 illustrates the read process to recover the latest data content. After the flash memory sensing and flash-to-controller data transfer, the SSD controller parses the data elements and accordingly carries out the ECC decoding and data/delta decompression, based upon which it combines the original data and all the subsequent deltas to obtain the latest data content. As explained in Section 3.2, different segments are protected by different ECC codes (LDPC codes or BCH codes) according to the length of information bits. Hence the controller must contain several different ECC decoders.

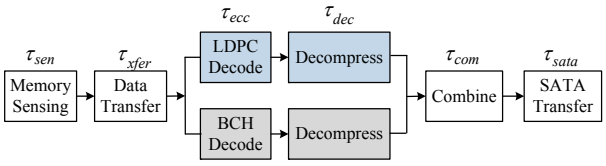


Figure 14: Illustration the process to obtain the latest data content.

Let τ_{sen} denote the flash memory sensing latency(the latency to read out the data content from flash cells using sensing circuits [29]), $\tau_{xfer}(\Omega)$ denote the latency of transferring Ω amount of data from flash memory chip to SSD controller, $\tau_{LDPC}^{(dec)}$ and $\tau_{BCH}^{(dec)}$ denote the LDPC and BCH decoding latency, $\tau_{sec}^{(dec)}$ and $\tau_{delta}^{(dec)}$ denote the latency of decompressing the original data and deltas, τ_{com} denote the latency to combine the original data and all the deltas to obtain the latest data content, and τ_{sata} denote the latency of transferring 4kB from SSD to host. In the conventional design practice without delta compression, to serve a single 4kB read request, the overall latency can be expressed as:

$$\tau_{read} = \tau_{sen} + \tau_{xfer}(4kB) + \tau_{LDPC}^{(dec)} + \tau_{sata}. \quad (1)$$

When using the proposed design solution to realize delta compression, the read latency can be expressed as:

$$\begin{aligned}\tau_{read} =& \tau_{sen} + \tau_{xfer}(n \cdot 4kB) + \max(\tau_{LDPC}^{(dec)}, \tau_{BCH}^{(dec)}) \\ &+ \max(\tau_{sec}^{(dec)}, \tau_{delta}^{(dec)}) + \tau_{com} + \tau_{sata},\end{aligned} \quad (2)$$

where n denotes the number of 4kB sectors being transferred from flash memory chip to SSD controller. We have that $n = 1$ in the case of segmented placement, and n is the number of 4kB in each flash memory physical page in the case of clustered placement. Since there could be multiple elements that are decoded by the LDPC decoder or the same BCH decoder, $\tau_{LDPC}^{(dec)}$ and $\tau_{BCH}^{(dec)}$ in Eq. 2 are the aggregated LDPC and BCH decoding latency. In addition, $\tau_{delta}^{(dec)}$ in Eq. 2 is the aggregated delta decompression latency because there could be multiple deltas to be decompressed by the same decompression engine.

We can estimate the read latency based on the following configurations. The SLC-mode sensing latency τ_{sen} is about 40μs in sub-20nm NAND flash memory. We set the flash memory physical page size as 16kB. Under the latest ONFI 4.0 flash memory I/O specification with the throughput of 800MB/s, the transfer latency $\tau_{xfer}(4kB)$ is 5μs. We set the throughput of both LDPC and BCH decoding as 1GBps. Data decompression throughput is set as 500MBps, and delta decompression throughput is set as 4GBps due to its very simple operations. When combining the original data and all the deltas, we simply use parallel XOR operations and hence set τ_{com} as 1μs. Under the SATA 3.0 I/O specification with the throughput of 6Gbps, the SSD-to-host data transfer latency τ_{sata} is set as 5.3μs.

Based upon the above configurations, we have that, to serve a 4kB read request, the overall read latency is 54μs under the conventional practice without delta compression. When using the proposed design solution, the overall latency depends on the number of deltas involved in the read operation. With the two different data placement

Table 2: Read/Update latency overhead comparison of different cases.

Operation	Technique	Average-case (μs)	Worst-case (μs)
Read	Conventional	54	
	Clustered	76	102
	Segmented	56	63
Update	Conventional	186	
	Clustered	246	272
	Segmented	226	233

strategies, we estimate the *worst-case* and *average-case* read latency as shown in Table 2:

- *Clustered placement*: In this case, the flash-to-controller data transfer latency is $\tau_{xfer}(16kB)$=20μs. In the worse case, the compressed 4kB sector being requested and all its deltas almost completely occupy the entire 16kB flash memory physical page, and are all protected by the same ECC (LDPC or BCH). And the total information bit length will be nearly 32kB at most due to ECC code word puncturing (as explained in Section 3.2). As a result, the decoding latency is 32μs at most and delta decompression latency is 4μs. Hence, the overall worst-case read latency is 102μs, representing a 88% increase compared with the conventional practice. In the average case, the latency of decoding/decompressing the original 4kB sector is longer than that of its deltas. Assuming the original 4kB sector is compressed to 3kB, we can estimate the decoding and decompression latency as 4μs and 6μs. Hence, the overall average-case read latency is 76μs, representing a 41% increase compared with the conventional practice.
- *Segmented placement*: In this case, the flash-to-controller data transfer latency is $\tau_{xfer}(4kB)$=5μs. The worst-case scenario occurs when the data compressibility is low and hence the compressed sector is close to to 4kB, leading to the decoding and decompression latency of 4μs (using $LDPC_{4kB}$) and 8μs, respectively. Hence, the worst-case overall read latency is 63μs, representing a 17% increase compared with the conventional practice. Under the average case, the compression ratio is modest and multiple deltas are stored, for which the latency could be about 2$\sim$4μs. Hence the average-case overall latency is about 56μs, representing a 4% increase compared with conventional practice.

4.3.2 Update Latency Overhead

In conventional practice without using delta compression, a data update operation simply invokes a flash memory write operation. However, in our case, a data update operation invokes data read, delta compression, and flash memory page partial programming. Let τ_{read} denote the latency to read and reconstruct one 4kB sector data (as discussed in the above), $\tau_{delta}^{(enc)}$ denote the delta compression latency, and $\tau_{program}$ denote the latency of flash memory page partial programming. Hence the update latency can be expressed as:

$$\tau_{write} = \tau_{read} + \tau_{delta}^{(enc)} + \tau_{ecc}^{(enc)} + \tau_{xfer} + \tau_{program} \quad (3)$$

Based upon our experiments with sub-20nm NAND flash memory, we set $\tau_{program}$ as of 150μs. We set the delta compression throughput $\tau_{delta}^{(enc)}$ as 4GBps and the ECC encoding throughput $\tau_{ecc}^{(enc)}$ as 1GBps. Therefore, the overall of writing one flash memory page is 186μs. When using the proposed design solution, as illustrated in Table 2, the value of τ_{read} could largely vary. In the case of clustered placement, the worst-case and average-case update latency is 272μs and 246μs, representing 32% and 46% increase compared with the conventional practice. In the case of segmented placement, the worst-case and average-case update latency is 233μs and 226μs, representing 25% and 22% increase compared with the conventional practice.

4.3.3 Silicon Cost

Finally, we evaluated the silicon cost overhead when using the proposed design solution. In particular, the SSD controller must integrate several new processing engines, including (1) multiple BCH code encoders/decoders, (2) per-sector lossless data compression and decompression engines, and (3) delta compression and decompression engines. As discussed in Section 3.2, we use three different BCH codes, BCH_{4B}, BCH_{128B}, and BCH_{512B}, which protect upto 4B, 128B, and 512B, respectively. Setting the worst-case SLC-mode flash memory bit error rate (BER) as 2×10^{-3} and the decoding failure rate as 10^{-15}, we constructed the code BCH_{4B} as the (102, 32) binary BCH code over GF(2^7), BCH_{128B} as the (1277, 1024) binary BCH code over GF(2^{11}), and BCH_{512B} as the (4642, 4096) binary BCH code over GF(2^{13}). To evaluate the entire BCH coding system silicon cost, we carried out HDL-based ASIC design using Synopsys synthesis tool set and results show that the entire BCH coding system occupies 0.24mm^2 of silicon area at the 22nm node, while achieving 1GBps throughput.

Regarding the per-sector lossless data compression and decompression, we chose the LZ77 compression

algorithm [30], and designed the LZ77 compression and decompression engines with HDL-based design entry and Synopsys synthesis tool set. The results show that the LZ77 compression and decompression engine occupies 0.15mm^2 of silicon area at the 22nm node (memory costs included), while achieving 500MBps throughput. Regarding delta compression and decompression, since they mainly involve simple XOR and counting operations, it is reasonable to expect that their silicon implementation cost is negligible compared with BCH coding and LZ77 compression. Therefore, we estimate that the overall silicon cost for implementing the proposed design solution is 0.39mm^2 at the 22nm node. According to our knowledge, the LDPC decoder module accounts for up to 10% of a typical SSD controller, meanwhile our silicon cost (including the logical resources such as gates, registers, memory, etc) is about 1/3 of an LDPC decoder. Therefore, we can estimate that the involved silicon area in proposed solution will occupy less than 5% of the silicon area of an SSD controller, which is a relatively small cost compared to the entire SSD controller.

5 Related Work

Aiming to detect the data content similarity and store the compressed difference, delta compression has been well studied in the open literature. Dropbox [31] and Github use delta compression to reduce the network bandwidth and storage workload using a pure application software level solution. Design solutions in [10, 11, 13] reduce the waste of space by detecting and eliminating the duplicate content in block device level while the proposed solution could further reduce the redundancy of similar but not identical writes. The FTL-level approach presented in [14] stores the compressed deltas to a temporary buffer and commits them together to the flash memory when the buffer is full, thus the number of writes could be reduced. Authors of [32] proposed a design solution to extend the NAND flash lifetime by detecting the identical writes. Authors of [33] developed an approach to utilize the content similarity to improve the IO performance while the proposed techniques pay more attention on the write stress reduction to extend the SSD lifetime. To improve the performance of data backup workloads in disks, authors of [9] proposed an approach to implement delta compression on top of deduplication to further eliminate redundancy among similar data. The key difference between proposed solution and existing solutions is that we can make sure the deltas and original data content locate in the same physical flash memory page, which will eliminate the read latency overhead fundamentally.

General-purpose lossless data compression also has been widely studied in flash-based storage system. The authors of [34, 35] presented a solution to realize transparent compression at the block layer to improve the space efficiency of SSD based cache. A mathematic framework to estimate how data compression can improve NAND flash memory lifetime is presented in [12]. The authors of [36] proposed to integrate database compression and flash-aware FTL to effectively support database compression on SSDs. The authors of [37] evaluated several existing compression solutions and compared their performance. Different from all the prior work, we for the first time present a design solution that cohesively exploits data compressibility and SLC-mode flash memory page partial-programmability to implement delta compression at minimal read latency and data management overhead.

6 Conclusion

In this paper, we present a simple design solution to most effectively reduce the write stress on SLC-mode region inside modern SSDs. The key is to leverage the fact that SLC-mode flash memory pages can naturally support partial programming, which makes it possible to use intra-page delta compression to reduce write stress without incurring significant read latency and data management complexity penalties. To further eliminate the impact on storage capacity, we combine intra-page delta compression with intra-sector lossless data compression, leading to the opportunistic in-place delta compression. Its effectiveness has been well demonstrated through experiments and simulations.

Acknowledgments

We would like to thank our shepherd Michael M. Swift and the anonymous reviewers for their insight and suggestions that help us to improve the quality and presentation of this paper. This work was supported by the National Science Foundation under Grants No. ECCS-1406154.

References

[1] X. Jimenez, D. Novo, and P. Ienne, "Libra: Software-controlled cell bit-density to balance wear in NAND Flash," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 2, pp. 28:1–28:22, Feb. 2015.

[2] Y. Oh, E. Lee, J. Choi, D. Lee, and S. Noh, "Hybrid solid state drives for improved performance and enhanced lifetime," in *IEEE Symposium on Mass Storage Systems and Technologies (MSST'13)*, May 2013, pp. 1–5.

[3] L.-P. Chang, "A hybrid approach to NAND-Flash-based solid-state disks," *IEEE Transactions on*

Computers, vol. 59, no. 10, pp. 1337–1349, Oct 2010.

[4] D. Sharma, "System design for mainstream TLC SSD," in *Proc. of Flash Memory Summit*, Aug. 2014.

[5] *Micron's M600 SSD accelerates writes with dynamic SLC cache*. http://techreport.com/news/27056/micron-m600-ssd-accelerates-writes-with-dynamic-slc-cache, Sept., 2014.

[6] *Samsung's 840 EVO solid-state drive reviewed TLC NAND with a shot of SLC cache*. http://techreport.com/review/25122/samsung-840-evo-solid-state-drive-reviewed, July, 2013.

[7] Y. Oh, E. Lee, J. Choi, D. Lee, and S. H. Noh, "Efficient use of low cost SSDs for cost effective solid state caches," *Science and Technology*, vol. 2010, p. 0025282.

[8] U. Manber, S. Wu *et al.*, "Glimpse: A tool to search through entire file systems." in *Usenix Winter*, 1994, pp. 23–32.

[9] P. Shilane, G. Wallace, M. Huang, and W. Hsu, "Delta compressed and deduplicated storage using stream-informed locality," in *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems (HotStorage'12)*, Berkeley, CA, 2012.

[10] C. B. Morrey III and D. Grunwald, "Peabody: The time travelling disk," in *Proceedings of 20th IEEE Conference on Mass Storage Systems and Technologies (MSST'03)*. IEEE, 2003, pp. 241–253.

[11] Q. Yang and J. Ren, "I-CASH: Intelligently coupled array of ssd and hdd," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, Feb 2011, pp. 278–289.

[12] J. Li, K. Zhao, X. Zhang, J. Ma, M. Zhao, and T. Zhang, "How much can data compressibility help to improve nand flash memory lifetime?" in *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Santa Clara, CA, Feb. 2015, pp. 227–240.

[13] Q. Yang, W. Xiao, and J. Ren, "Trap-array: A disk array architecture providing timely recovery to any point-in-time," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, 2006, pp. 289–301.

[14] G. Wu and X. He, "Delta-FTL: Improving SSD lifetime via exploiting content locality," in *Proceedings of the 7th European conference on Computer systems (Eurosys'12)*, New York, NY, USA, 2012, pp. 253–266.

[15] UMass Trace Repository. http://traces.cs.umass.edu/index.php/Storage/Storage/.

[16] FIU IODedup Trace-Home. http://iotta.snia.org/traces/391.

[17] D. Campello, H. Lopez, R. Koller, R. Rangaswami, and L. Useche, "Non-blocking writes to files," in *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Santa Clara, CA, Feb. 2015, pp. 151–165.

[18] S. Lin, T. Kasami, T. Fujiwara, and M. Fossorier, *Trellises and Trellis-Based Decoding Algorithms for Linear Block Codes*. Kluwer Academic Publishers, 1998.

[19] Z. Wang, Z. Cui, and J. Sha, "VLSI design for low-density parity-check code decoding," *IEEE Circuits and Systems Magazine*, vol. 11, no. 1, pp. 52–69, 2011.

[20] C. Fewer, M. Flanagan, and A. Fagan, "A versatile variable rate LDPC codec architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 10, pp. 2240–2251, Oct 2007.

[21] T. Richardson, "Error floors of LDPC codes," in *Proc. of 41st Allerton Conf. Communications, Control and Computing*, Oct. 2003.

[22] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST'13)*, San Jose, CA, 2013, pp. 257–270.

[23] Employees Sample Database. https://dev.mysql.com/doc/employee/en/index.html.

[24] BIRT Sample Database. http://www.eclipse.org/birt/documentation/sample-database.php.

[25] Linux Kernel 3.11.10. https://www.kernel.org/pub/linux/kernel/v3.x/.

[26] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *USENIX Annual Technical Conference (ATC'13)*, San Jose, CA, 2013, pp. 309–320.

[27] T. Tso, "Debugfs." [Online]. Available: http://linux.die.net/man/8/debugfs

[28] X. Zhang, J. Li, K. Zhao, H. Wang, and T. Zhang, "Leveraging progressive programmability of SLC flash pages to realize zero-overhead delta compression for metadata storage," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*, Santa Clara, CA, Jul. 2015.

[29] R. Micheloni, L. Crippa, and A. Marelli, *Inside NAND Flash Memories*. Dordrecht: Springer, 2010.

[30] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transaction on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[31] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012, pp. 481–494.

[32] 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. of USENIX Conference on File and Storage Technologies (FAST'11)*, vol. 11, 2011.

[33] R. Koller and R. Rangaswami, "I/O deduplication: Utilizing content similarity to improve I/O performance," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 13, 2010.

[34] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Using transparent compression to improve SSD-based I/O caches," in *Proceedings of the 5th European conference on Computer systems (Eurosys'10)*. ACM, 2010, pp. 1–14.

[35] T. Makatos, Y. Klonatos, M. Marazakis, M. Flouris, and A. Bilas, "ZBD: Using transparent compression at the block level to increase storage space efficiency," in *2010 International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'10)*, May 2010, pp. 61–70.

[36] D. Das, D. Arteaga, N. Talagala, T. Mathiasen, and J. Lindström, "NVM compression—hybrid flash-aware application level compression," in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, Oct. 2014.

[37] A. Zuck, S. Toledo, D. Sotnikov, and D. Harnik, "Compression and SSDs: Where and how?" in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, Oct. 2014.