Leveraging Progressive Programmability of SLC Flash Pages to Realize Zero-overhead Delta Compression for Metadata Storage

Xuebin Zhang, Jiangpeng Li, Kai Zhao, Hao Wang and Tong Zhang
ECSE Department, Rensselaer Polytechnic Institute, USA

Abstract

This paper presents a method to implement delta compression for metadata storage in flash memory. With the abundant temporal redundancy in metadata, it is very intuitive to expect flash-based metadata storage can significantly benefit from delta compression. However, straightforward realization of delta compression demands the storage of the original data and the deltas among different versions in different flash memory physical pages, which leads to significant overhead in terms of read/write latency and data management complexity. Through experiments with 20nm NAND flash memory chips, we observed that, when operating in SLC mode, flash memory page can be programmed in a progressive manner, i.e., different portion of the same SLC flash memory page can be programmed at different time. This motivates us to propose a simple design approach that can realize delta compression for metadata storage without latency and data management complexity overheads. The key idea is to allocate SLC-mode flash memory pages for metadata, and store the original data and all the subsequent deltas in the same physical page through progressive programming. Experimental results show that this approach can significantly reduce the metadata write traffic without any latency overhead.

1 Introduction

Although metadata occupy relatively small percentage of storage space, they tend to account for a large percentage of storage I/O traffic. It is well known that consecutive metadata update operations have abundant temporal redundancy. Intuitively, this can be leveraged to realize delta compression and hence largely reduce the write traffic on the I/O stack and/or on the physical storage media. Many design solutions have been proposed in the literature to exploit data storage temporal redundancy at different levels, such as file systems [1, 2], block device [3–6] and FTL (Flash Translation Layer) [7]. These existing solutions detect the data similarity and then store the compressed difference between consecutive versions. The host must fetch more than one pages to retrieve the latest data. In addition, the host needs to keep the page mapping information between old data and the compressed deltas. Therefore, the file system and/or firmware have to support sophisticated data structure. As a result, although these existing solutions can be directly applied to metadata storage in flash memory, they inevitably result in metadata data access latency overhead and hence system performance penalty.

This paper presents a design solution that can effectively implement delta compression for metadata storage in flash memory. First, through experiments with leading-edge 20nm NAND flash memory chips, we observed that, when being used in SLC mode, flash memory pages can support progressive programming, i.e., different portions of the same SLC flash memory page can be programmed at different time. This clearly makes it possible to store the original data and the subsequent deltas in the same physical page. Since the runtime delta compression/decompression can be carried out by SSD controllers much faster than flash memory page read, this can essentially eliminate the data access latency overhead in the realization of delta compression.

We carried out experiments and simulations to evaluate the effectiveness of proposed design solution. First, we verified the feasibility of SLC-mode flash memory page progressive programming using a PCIe FPGA-based flash memory characterization hardware prototype with 20nm MLC NAND flash memory chips. We further implemented a metadata analyzer under Linux to monitor and collect a large set of consecutive versions of metadata. The results show that the delta compression efficiency can be up to 1:0.069. In addition, the proposed solution can significantly reduce the metadata write data volume to flash memory. Finally, we studied the latency incurred by decompression and the results further verified that they do not add noticeable latency overhead.

2 Background and Motivation

2.1 Conventional Methods for Realizing Delta Compression

Delta compression aims to detect the data content similarity and store the compressed difference. The file system level approach presented in [2] stores and indexes the deltas of similar regions. Design solutions in [3, 4, 6] reduce the waste of space by detecting and eliminating the duplicate content in block device level. The FTL-level approach presented in [7] stores the compressed deltas to a temporary buffer and commits them together to the flash memory when the buffer is full.

In spite of their good delta compression efficiency, these existing solutions have some significant limitations. First, the new data is stored in a different location from the old data. So the host must visit at least two different locations to retrieve one page of data, which brings significant read overhead. Second, extra mapping information must be stored as well to maintain the correlation among different versions. This leads to sophisticated data structures and data management, especially for read-intensive applications.

2.2 Characteristics of Metadata

A fundamental feature in flash storage systems is that the data access must be page-aligned, i.e., the basic access unit of flash devices is one physical page. Even though the host only needs to update a single byte, it must read and write one entire flash memory page. It is obvious that there could be certain redundancy because of the write-back of the same data, and this problem is the most noticeable for metadata storage due to its following distinct characteristics.

- Small update size: Although the page size is as large as 16KB even 32KB, the possible access size may be only several bytes even bits. A study [8] shows that more than 80% of write operations in some Android applications are partial page overwrites less than 4KB.

- Frequent updates within a short time: Metadata updates are triggered frequently. Every I/O action, even a simple file opening, will trigger a metadata update. For instance, as shown in [9], to update merely 6 bytes within a text file on disk triggers the write of 11 physical pages for updating various content including bitmapping, inode, data, etc.

- Large percentage of disk traffic: According to a recent study [10], the metadata region only occupies 5% of the storage space but consumes more than 20% of the write traffic.

2.3 Progressive Programming

In this work, we observed that SLC NAND flash memory can support progressive programming, i.e., different portions in a SLC flash memory page can be programmed at different time. As illustratd in Figure 1, we first write the logical page D_0, consisting of data $d_0 d_1 \dots d_n$, into the physical page P_0 and leave the rest portion unwritten. Later on, we can write a new data L_1, consisting of data $l_0 l_1 \dots l_m$, into the physical page P_0, i.e., append L_1 with the existing data D_0 in the same physical page. This can be realized through a read-append-overwrite procedure: We read out the original data content $d_0 d_1 \dots d_n$, append the new data $l_0 l_1 \dots l_m$ and pad with '1' to form the data for an entire physical page, and then write the data back to the physical page P_0.

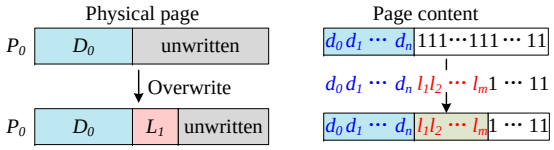

Figure 1: Illustration of flash memory progressive programming.

Using 20nm MLC NAND flash memory chips, we carried out experiments and the results show that the chips can support the progressive programming when being operated in the SLC mode. In our experiments, we define "one cycle" as progressively programming the flash page for 8 times before it is filled up and then being erased. In contrast, the conventional "one cycle" is to fully erase before each programming. Figure 2 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 programming can work for more than 7000 cycles, which indicates that the progressive programming mechanism does not bring noticeable extra physical damage to flash memory cells.

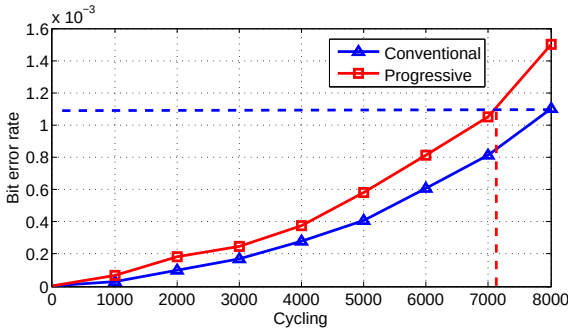

Figure 2: Comparison of the bit error rate of conventional programming and progressive programming.

3 Proposed Solution

3.1 Metadata Update on NAND Flash

Leveraging the progressive programming feature of SLC-mode flash memory, we can effectively realize delta compression for metadata. Here we assume that the host is able to provide certain hint information to help the storage device to distinguish metadata from normal data. The proposed design solution is illustrated in Figure 3. A physical flash memory page is divided into two portions (denoted as p_0 and p_1). We only use the first portion p_0 to store the original metadata content, and leave the second portion p_1 to store the deltas. Suppose D_0 is stored in the portion p_0 and p_1 is left unwritten at the beginning, Figure 4 shows the flow diagram for realizing metadata delta compression. Once the host writes a new version data D_1 , we read the existing data D_0 and compress the difference between D_0 and D_1 as L_1. Then we append L_1 with the original data D_0 in the same physical page. The storage system only needs to record the location of P_0 to recover the data block D_1 at any time.

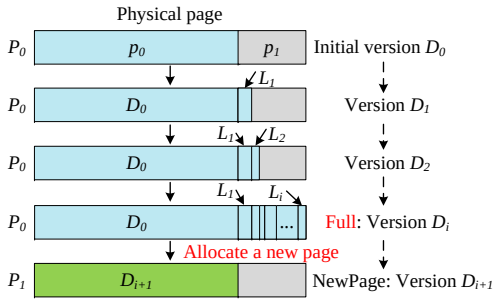

Figure 3: Illustration of proposed update procedures.

As shown in Figure 3, when the host again writes a newer version data D_2, we first read D_0 and L_1 from the physical page to reconstruct the latest data block D_1. Then we compress the difference between D_1 and D_2, and append the compressed difference L_2 into the same physical page. Clearly, the storage system still only needs to record the location of the physical page P_0 to recover D_2. The same process repeats as the host continues to write new data blocks until the physical page P_0 is full.

Once the physical page is full after the i-th version D_i is written, we allocate a new physical page, write the latest version data D_{i+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 page to retrieve the latest version data.

From the above description, we can see that this proposed design solution can eliminate all the extra index/mapping information in conventional realization of delta compression. In addition, we do not need to record

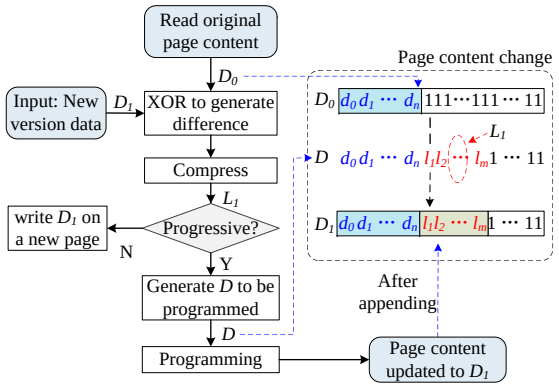

Figure 4: Flow diagram for realizing metadata delta compression.

the offset location of L_i because the decompressed L_i has a same length with D_0. L_i is decompressed bit by bit serially and the host can determine the ending when the decompressed data length reaches the length of D_0.

Compared with the conventional techniques described in Section 2.1, the proposed design solution has three major advantages:

- Since the original data and all the subsequent compressed differences are stored in the same flash physical page, it does not require special data management and hence does not complicate the design of file system and device firmware.
- When we read the data at one logical address, we only need to read one physical page based upon which we carry out successive decompression. This fundamentally eliminates the read latency penalty.
- The updated page can be written back to the flash memory any time without waiting in the buffer. The proposed technique can be used together with data buffering to fully minimize the write traffic to flash memory chips.

3.2 Implementation of Delta Compression

It is clear that the delta compression contains two steps: the first step is to generate the difference by bit-wise XOR operation and the second step is to compress it using entropy coding (e.g., run-length encoding). In this work, we use a simple method to improve the compression efficiency of the difference, which is illustrated in Figure 5. We partition each data page into a number of equal-sized segments, e.g., each segment can be few bytes, and compare the two data pages with the unit of segment. The segments with different content are called mismatched segments. In stead of using bit-wise XOR and entropy coding, we simply record the index of each mismatched

segment and the segment content. This is referred to diff-index compression. Clearly, given the old version of data page and the diff-index compression result, we can directly reconstruct the new version of data.

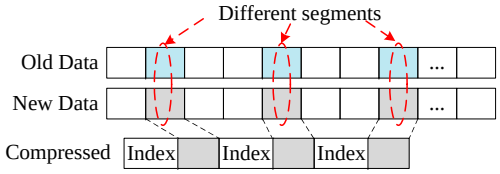

Figure 5: Illustration of the proposed diff-index delta compression.

4 Evaluations

In this section, we evaluated the compression efficiency of temporal redundancy based delta compression and the performance gains of the proposed design solution. File system metadata includes superblock, block group descriptor, bitmaps and file inodes, but we only focus on the file inodes here because superblock and bitmap occupy a rather small space on disk.

We modified Mobibench [10] to make it work as the I/O workload benchmark under Linux Ubuntu 14.04 Desktop. We use a large set of SQLite workload (`create, insert, update, delete`) and general file system tasks (file `read, update, append`). The database and file metadata are updated consecutively triggered by I/O workloads. To monitor the characteristics of metadata, based upon the existing tool `debugfs` [11], we implemented a metadata analyzer tool[1] to track, extract, and analyze the file system metadata.

We use ext4 file system 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 collect 1000 consecutive versions of metadata.

4.1 Compression Efficiency

Based on the collected consecutive versions of metadata, we investigated the metadata compression efficiency of each kind of I/O operations. The results are shown in Figure 6. We use both the conventional compression scheme (i.e., bit-wise XOR followed by run-length encoding) and the proposed diff-index compression scheme described in Section 3.2. For the case of using the run-length coding, the average compression ratio is 1:0.087 and the standard deviation of is 0.0096. In comparison, when using the proposed diff-index compression scheme, the average compression ratio is 1:0.069 with the standard deviation of 0.0087. We note that a smaller standard deviation means a more stable delta compression efficiency. The results clearly demonstrate the significant data volume reduction potential by applying delta compression for metadata. The compression efficiency of the proposed diff-index compression scheme is better than the conventional run-length coding based scheme because modified content tend to concentrate around certain fields in the metadata structure.

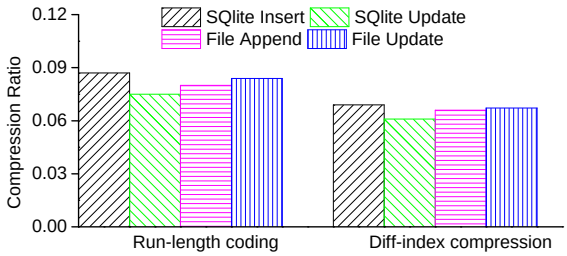

Figure 6: Average delta compression ratio.

4.2 Write Traffic Reduction of Metadata

As described earlier, in order to ensure that the original data and its subsequent deltas are stored in the same physical page, the delta compression must be reset once the physical page has been filled up. As a result, the write traffic reduction is less than the delta compression gains presented above. We carried out further study to evaluate the write traffic reduction. We set each physical page size as 16KB and leave 512B for storing the compressed deltas. Figure 7 shows the number of flash memory physical pages bing written when writing 1000 versions of metadata under different workloads. The results show that, when using the proposed design solution, about 40 SLC pages would be enough to store 1000 consecutive versions of metadata under SQlite/File workload. For the baseline without using delta compression, we have to allocate 1000 MLC or TLC pages.

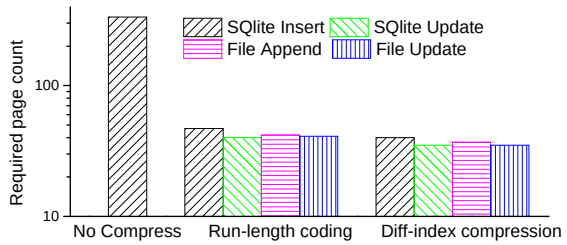

Figure 7: Number of flash memory pages required to write 1000 versions of metadata.

[1] metaAnalyzer: https://github.com/SmartStorage/metaAnalyzer

4.3 Practical Considerations

As discussed in Section 2.1, the conventional practice of implementing delta compression must store extra information relating the original data and the compressed deltas, and must fetch multiple flash pages in order to reconstruct the latest version. In contrast, the proposed solution can guarantee that the original data and deltas are located in the same physical page. Thus, the read latency overhead can be fundamentally eliminated.

We further investigate the latency induced by delta decompression/compression when being implemented by ARM processors. To emulate their execution on ARM platforms, we cross-compile the compression algorithm source code and run them on the Android tablet's bottom-level Linux system. The tablet's CPU frequency is set as 1026MHz with only one core enabled, and the average encoding and decoding latency for a 16KB physical page is shown in Table. 1. Recall that the read/write latency for a SLC mode flash page is $41\mu s/150\mu s$ [12], we can conclude that the decompression and compression do not noticeably affect the I/O performance. In fact, the latency in Table. 1 is just a very conservative worst-case estimation. In practice, the compression/decompression should be implemented by an ASIC accelerator in storage device controller, which can outperform ARM-based implementation by at least few orders of magnitudes.

Table 1: ARM-based delta compression and decompression latency.

Algorithm	Run-length coding	Diff-index comp.
Compression(μs)	20.7	5.2
Decompression(μs)	8.1	4.1

Practical implementation of the proposed design solution demands the change on ECC management. The ECC redundancy bits will be generated and appended together with the deltas. Before the deltas are appended to the flash page, the length of compressed delta should be written in the page, which will be used for the ECC decoding in the future.

An obvious overhead of the proposed solution is that the flash memory is used at its SLC mode, which leads to half of its storage space waste. However, this would not be a problem if more than one version of delta can be compressed into this physical page, which is very likely because of the abundant temporal redundancy in metadata as demonstrated in Section 4.1. In addition, since metadata occupy a small percentage of storage space, the use of SLC-mode flash storage will not incur significant storage space overhead. Finally, it should be pointed out that this proposed technique is only eared to flash-based storage devices and is not applicable to storage devices using byte-addressable non-volatile memory (e.g., phase-change memory).

5 Conclusion

In this paper, we present a simple design solution to support delta compression for metadata storage in flash memory. The key is to leverage the fact that SLC-mode flash memory pages can naturally support progressive programming. This makes it possible to store the original data and its subsequent deltas in the same physical page. Its effectiveness has been well demonstrated through experiments and simulations. Future research is directed to expanding this design solution to other update-intensive applications such as database operations, cloud storage sync-up, etc.

References

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

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

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

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] G. Wu and X. He, "Reducing SSD read latency via nand flash program and erase suspension." in *Proceedings of 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012, p. 10.