Lightweight Data Compression for Mobile Flash Storage

CHENG JI, City University of Hong Kong
LI-PIN CHANG, National Chiao-Tung University
LIANG SHI and CONGMING GAO, Chongqing University
CHAO WU, City University of Hong Kong
YUANGANG WANG, Huawei Technologies Co. Ltd
CHUN JASON XUE, City University of Hong Kong

Data compression is beneficial to flash storage lifespan. However, because the design of mobile flash storage is highly cost-sensitive, hardware compression becomes a less attractive option. This study investigates the feasibility of data compression on mobile flash storage. It first characterizes data compressibility based on mobile apps, and the analysis shows that write traffic bound for mobile storage volumes is highly compressible. Based on this finding, a lightweight approach is introduced for firmware-based data compression in mobile flash storage. The controller and flash module work in a pipelined fashion to hide the data compression overhead. Together with this pipelined design, the proposed approach selectively compresses incoming data of high compressibility, while leaving data of low compressibility to a compression-aware garbage collector. Experimental results show that our approach greatly reduced the frequency of block erase by 50.5% compared to uncompressed flash storage. Compared to unconditional data compression, our approach improved the write latency by 10.4% at a marginal cost of 4% more block erase operations.

CCS Concepts: • **Computer systems organization** → Embedded systems; • **Software and its engineering** → *Operating systems*;

Additional Key Words and Phrases: Flash memory, mobile device, compressibility, data compression

ACM Reference format:
Cheng Ji, Li-Pin Chang, Liang Shi, Congming Gao, Chao Wu, Yuangang Wang, and Chun Jason Xue. 2017. Lightweight Data Compression for Mobile Flash Storage. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 183 (September 2017), 18 pages.
https://doi.org/10.1145/3126511

1 INTRODUCTION

Flash memory is the primary choice of data storage in mobile devices, such as smartphones, tablets, and wearable devices. Compared to solid state drives, flash storage for mobile devices do not have the luxury of sophisticated hardware and firmware features. Android is the most popular

This article was presented in the International Conference on Embedded Software 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' addresses: C. Ji, C. Wu, and C. Xue are with the Department of Computer Science, City University of Hong Kong, Hong Kong; emails: {chengji4-c, chaowu6-c, jasonxue}@cityu.edu.hk, L.-P. Chang is with the Department of Computer Science, National Chiao-Tung University, Taiwan; email: lpchang@cs.nctu.edu.tw; L. Shi (corresponding author) and C. Gao are with the College of Computer Science, Chongqing University, China; emails: shiliang@cqu.edu.cn, albertgaocm@gmail.com; Y. Wang is with Huawei Technologies Co. Ltd, China; email: wangyuangang@huawei.com.

https://doi.org/10.1145/3126511

OS for mobile devices. As reported in [8, 11], I/O workloads in Android devices are highly write-centric. This is because a middle-ware layer, i.e., SQLite, produces a high volume of synchronous write traffic through its database journaling mechanism. This intensive write pressure introduces a high variation in I/O latency, affecting the user experience for smartphones. In addition, frequent garbage collection reduces the flash memory lifespan. Prior work reported that data compression considerably improves the performance and lifetime of solid state disks [13, 18, 19]. However, there is little work considering data compression for mobile flash storage. There are two reasons: First, the common belief discourages data compression on smartphone storage volumes because multimedia files, which occupy the majority of the smartphone storage capacity, are not compressible. Second, the extra data compression hardware reduces the cost competitiveness of mobile flash storage, while naive firmware-based data compression can degrade the write latency. This paper aims to address the above challenges.

The first part of this study is to quantitatively analyze data compressibility in real smartphones. In addition to the snapshots of mobile storage volumes, the analysis also involves the write traffic bound for mobile storage. Our analysis results show that, even though a large portion of the storage capacity is occupied by incompressible multimedia contents, a large portion of the write traffic is contributed by the SQLite layer and file system and this write traffic is highly compressible. Therefore, although counter-intuitive, data compression could be beneficial to mobile flash storage.

The second part of this study introduces a lightweight data compression strategy for mobile flash storage. The proposed approach selectively compresses the incoming data stream whenever the reduction in write latency is larger than the overhead for data compression. Uncompressed data are written to flash and will later be processed by a compression-aware garbage collector in the background without impacting the performance of foreground I/O activities. Experimental results show that, the proposed approach can greatly reduce the total number of block erasures by up to 50.5% compared to uncompressed flash storage, and improve the write latency by up to 10.4% compared to unconditional data compression. The impact on read performance was marginal because data decompression is fast. To the best of our knowledge, this work is the first to investigate firmware-based data compression for mobile flash storage. In summary, this paper makes the following contributions:

(1) A study based on a rich set of mobile apps that reveals the high compressibility of the write traffic bound for mobile storage;
(2) A fast and accurate prediction method of data compressibility and a lightweight data compression policy for selective compression on the incoming data stream;
(3) A compression-aware garbage collection policy that compresses garbage-collected data in the background for long-term improvement of write performance;
(4) A series of experiments that evaluate the proposed lightweight compression, showing promising results in terms of flash lifespan and I/O performance.

The rest of this paper is organized as follows. Section 2 describes the background and related work. Section 3 presents the performance model and motivation. Section 4 presents the user study of compression on Android smartphones. Section 5 presents the design of the proposed lightweight data compression approach. Section 6 presents experiments and discussions, and this paper is concluded in Section 7.

2 BACKGROUND AND RELATED WORK

Flash memory characteristics: NAND flash-based storage devices exhibit superior I/O performance against traditional hard disks. Flash storage employs a firmware layer called Flash Translation Layer (FTL) for block device emulation. Due to the erase-before-write constraint, a flash

page cannot be overwritten unless the block that contains this page is erased. The FTL finds flash pages of free space and redirects updates to these pages. When the FTL runs low on free pages, it triggers garbage collection to reclaim free space. The reclaiming process involves a series of data migration and block erasure. After a block is erased, all its pages are available for writing new data again. While flash storage gains popularity for its superior performance, its performance is subject to the garbage collection overhead. There have been studies proposed to improve garbage collection performance, e.g., reducing page migration costs through separating cold data from hot data [7, 10]. Flash memory endure limited program/erase (P/E) cycles, and the lifetime issue becomes more challenging as cell density increases [4]. Embedded flash storage devices, such as eMMC and UFS, have become a mainstream solution for mobile storage devices. Due to the considerations of hardware costs and power consumption, they are equipped with high density but poor endurance multilevel flash memory and basic embedded controllers with limited embedded RAM [9, 16].

I/O characteristics of mobile devices: I/O performance has significant impacts on the overall performance of mobile smart devices [11]. Recently, the characterization of I/O behaviors in mobile devices drew a lot of attention. Lee et al. [12] provided a deep profile on I/O patterns of Android smartphones, and it was reported that the journal mechanisms of the SQLite and Ext4 file system are an I/O performance bottleneck. The double journaling problem, called Journaling of Journal [8], introduces excessive write traffic to flash storage and causes serious I/O performance degradation. These prior works focused on I/O behaviors, but there is little work regarding the characterization on the data contents written to mobile flash storage.

Data compression in flash storage: Because flash memory is of limited PE cycles, reducing the incoming write traffic is an effective means to increase the flash storage lifespan. Data compression is also beneficial to energy consumption of data storage, as reported in [17]. Data compression for flash storage not only reduces the incoming write traffic volume but also alleviates the pressure on garbage collection. Zuck et al. [21] investigated how compression granularity and allocation of compressed data could affect write traffic volume and flash memory lifespan. Park et al. [13] proposed a hardware-based compression design, called zFTL, which improves both write performance and power consumption. Our work differs from zFTL in some aspects: First, our approach is a firmware implementation, while zFTL requires hardware for compression and compressibility prediction. Second, our approach employs a dynamic threshold for data compression, while zFTL employs a static threshold. Third, garbage collection in our approach is aware of data compression, while that in zFTL is not. Zhang et al. [19] proposed to exploit the partial programmability of SLC-mode pages for reducing write pressure on flash memory through opportunistic in-place delta compression. These prior methods require extra hardware for data compression. However, cost reduction is a critical issue in mobile flash storage design, and these hardware-based solutions are less attractive to mobile flash storage. File systems can also enable transparent data compression [14], but many of the major file systems for mobile smart devices, such as Ext4, F2FS, and FAT, do not officially support data compression.

3 MODEL AND MOTIVATION

This section first presents a performance model of I/O requests with data compression. A motivating observation is then presented to address the need for selective compression.

3.1 Performance Model

The two key components involved in data compression inside of flash storage are the *controller* and *flash memory module*. The controller compresses host data, while the flash module writes compressed data in units of flash pages. They work together in a pipelined fashion to overlap

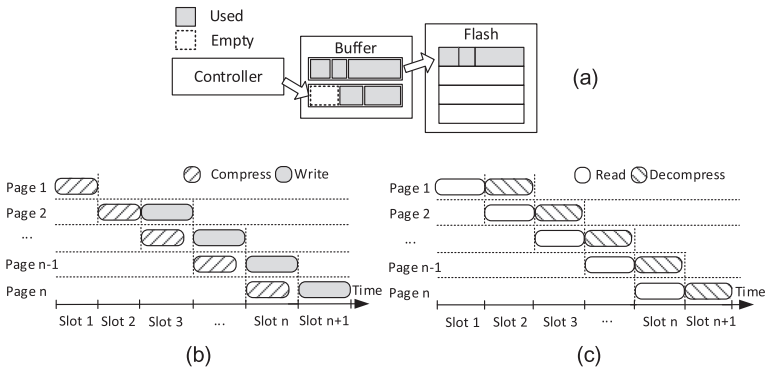


Fig. 1. (a) The controller and flash module share a write buffer. (b) Compression timeline with pipeline. (c) Decompression timeline with pipeline.

the time overheads of compression and programming. As shown in Figure 1(a), a write buffer consisting of (at least) two page slots is shared between the controller and the flash module. The controller compresses page data from the host and appends compressed pages to a buffer slot. When the buffer slot is full, the controller switches to another slot, and in the meantime, the flash module begins to program the buffer slot into a flash page. The write buffer is write-back, meaning that compressed pages are held in the current buffer slot until buffer slot switching. Data loss in the small write buffer can be prevented with a tiny capacitor, as previously discussed in [16].

Compression pipeline: Assume that the controller compresses one page of host data in t_c units of time, and that the flash module takes t_w units of time to write a flash page. Suppose that t_c is not longer than t_w. Figure 1(b) depicts an example timeline where the pipeline handles a write request of n pages. The first two time slots do not involve page programming because the controller is appending compressed pages to the first buffer slot, and the lengths of these two time slots are t_c. In time slot 3, the controller switches to another buffer slot and the flash module begins to program a flash page. The length of slot 3 is t_w, the longer one of page program and page compression times. If the controller fills up a buffer slot after compressing the last (n-th) page, then one more time slot (i.e., time slot $n + 1$) is necessary to write a flash page. The first and the last time slots involve no parallelism between the controller and flash module.

Here, the *compression ratio* Cr of a page of data is defined as the ratio of the compressed size to the uncompressed size of the data. The lower the C_r value, the better the compression performance. Let T_w be the expected (average) latency of a write request of n pages, and let C_r be the average data compression ratio of these n pages. Specifically, T_w of a write request is the duration between the arrival of the request and the time when the last page of the write request enters the write buffer. If the request has only one page, i.e., $n = 1$, the controller first spends t_c units of time to compress one page of data, and then the flash module spends $t_w \times C_r$ units of time to write flash. No parallelism can be exploited because these two steps are dependent. If the request consists of multiple pages, based on Figure 1(b), each of the first n time slots uses at least t_c units of time for data compression. Before the last time slot, there will be $(n - 1) \times C_r$ time slots that involve page writes, each of which requires additional $t_w - t_c$ units of time. The time length of the last time slot is still $t_w \times C_r$. Therefore, we have

$$T_w = \begin{cases} t_c + t_w \times Cr, & n = 1 \\ t_c \times n + (t_w - t_c)(n-1) \times Cr + t_w \times Cr, & n > 1. \end{cases} \quad (1)$$

This analysis does not involve any garbage collection activities. They will be discussed in the next section.

Decompression pipeline: Figure 1(c) shows how a read request is handled based on a similar pipelined design. The flash module reads a flash page into a buffer slot and then the controller decompresses the page data. The decompression time of a page (except the last page) can be overlapped with the read time of the subsequent page. Let the time overheads of reading and decompressing a page be t_r and t_d ($t_r \geq t_d$), respectively. Let the read request size be n pages. At least $n \times C_r$ flash pages are required to store the compressed data of the request. Therefore, on average the latency of the read request will be $T_r = t_r \times n \times C_r + t_d$.

3.2 Compression Overhead vs. Write Latency

Even for the simplest case of $n = 1$ in Equation (1), the write latency T_w involves t_c, t_w, and C_r. Since t_w is a constant that depends on flash memory physics, to achieve a better T_w, either t_c should be faster or C_r should be better (lower). As reported in prior work [18], the time to compress a 4KB page is 89.5μs on an ARM-based controller rated at 304MHz. Our industrial contacts confirm that the controllers of typical embedded MultiMedia Cards (eMMCs) are not faster than 200MHz. Because execution time is linearly proportional to clock rate, t_c will be 136μs on a 200MHz controller. It is possible to scale up the clock rate to speed up data compression. However, this option would exacerbate the power consumption of mobile flash storage [15] and is not reasonable for battery-based mobile devices.

The data compression ratio C_r depends on the compression algorithm and the data pattern. For the case $n = 1$, the write latency is t_w if no compression is involved. Because smartphone users are highly sensitive to latency, *our goal is to enable firmware-based data compression without degrading I/O latency*. For demonstration, let t_c be 60% of t_w. We can derive from Equation (1) that if C_r is better (lower) than 40%, then the write request can use data compression because the write latency through the pipelined compression is not worse than the write latency without compression. Otherwise, the write request should not be compressed because data compression could degrade its latency.

Latency-aware data compression: Compression on the incoming data should be selective because as previously shown in Figure 1(b), the pipelined compression design cannot hide the compression overhead of the first page of a write request. If a write carries data of poor compressibility, then data compression may degrade the latency of the request. Let us consider the latency of a single-page write request. Without compression, the latency of the request is t_w. Page writes may trigger garbage collection activities. Let t_{gc} be the time overhead of garbage collection amortized over every page write. Therefore, the expected latency of the write request would be $t_w + t_{gc}$. Similarly, according to Equation (1), the expected write latency with data compression of the request is $t_c + t_w \times C_r$, and this overhead is amplified by adding t_{gc} to it. Here, t_{gc} is a conservative overestimate because data compression helps reduce the frequency of garbage collection activities. The request can undergo compression without degrading the write latency if $t_c + t_w \times C_r + t_{gc} \leq t_w + t_{gc}$ holds. Based on this argument, we derive the following conditions for write requests of n pages:

$$\begin{cases} Cr \leq \frac{t_w - t_c}{t_w}, & n = 1 \\ Cr \leq \frac{t_w - t_c}{t_w - t_c + \frac{t_c}{n}}, & n > 1 \end{cases} \tag{2}$$

It can be observed that, the larger the request size n is, the higher the chance that a write request can use data compression without latency degradation.

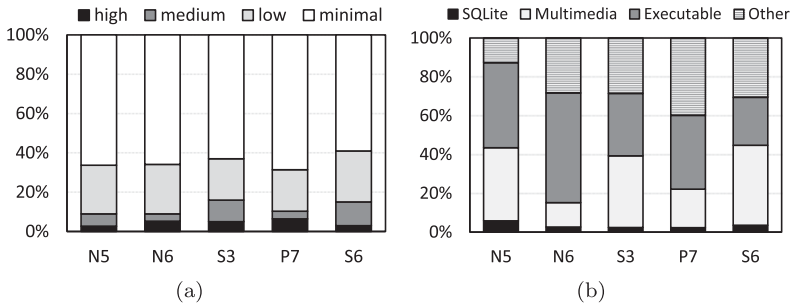


Fig. 2. (a) A breakdown of data with different compressibility. (b) A breakdown of data with different file types for data with *low* and *minimal* compressibility.

Because the criterium in Equation (2) involves multiple parameters, a static threshold on C_r for selective compression may lead to over-optimistic or over-pessimistic results. For example, zFTL [13] skips incompressible data (with hardware assistance), but this simple criteria can frequently create a degradation in write latency. The challenges here involve: 1) how to quickly predict the compression ratio of incoming data without actually compressing the data and 2) how to decide whether or not to compress the incoming data. In addition, if data is written to flash memory without compression, it should have a second chance to be compressed later at proper times.

4 EMPIRICAL STUDY OF DATA COMPRESSIBILITY

This section presents an empirical study on the data compressibility on smartphone. Compressibility is analyzed from two angles: disk volume snapshots and write traffic bound for flash storage. In this study, all data compression is based on the LZO lossless data compression algorithm [1]. The input block size of LZO is 4KB, which is as large as a flash page and a file system block. To better present the results, the compressibility of a file is classified as *high* if its compression ratio C_r is below 0.25, *medium* if C_r is between 0.25 and 0.65, *low* if C_r is between 0.65 and 0.95, and *minimal* if C_r is higher than 0.95.

4.1 Volume Snapshot Analysis

We collected five Android phones, including Nexus 5 (N5), 6 (N6), Galaxy S3 (S3), S6 (S6) and Ascend P7 (P7), from randomly selected engineering students. These phones had been used daily for at least six months. We took snapshot of the smartphone `DATA` partition and analyzed the compression ratio of files in the partition. Storage space contributions are classified according to the file types: `SQLite` denotes files with extensions .db, .db-journal, and .db-wal, `Multimedia` corresponds to files with extensions .jpg, .cnt, and .mp4, and `Executable` files have extensions .odex, .dex, .apk, and .so.

Figure 2(a) and (b) show the compressibility distribution of files and the file type distribution of data with *low* and *minimal* compressibility, respectively. As expected, files of *low* or *minimal* compressibility occupied the majority of the storage space in smartphones, between 84% and 91% of the disk volume capacity. This is because multimedia files contributed to large portions of the storage space in the five phones. Executable files also contributed to large portions of the data of low or minimal compressibility. The results in Figure 2 are consistent with the common belief that smartphone storage volumes are barely compressible.

4.2 Write Traffic Analysis

As reported in prior studies, the write traffic bound for flash storage is mostly contributed by file system metadata and SQLite files, and these writes exhibit highly overwriting behaviors [11, 12].

Table 1. Application scenarios

Application	Type	Scenarios
Facebook	Social networking	Browsing news feedsand posting new messages
Twitter	Social networking	Browsing news feedsand posting new messages
Messenger	Instant messaging	Chatting with contacts and browsing conversation history
Google Earth	Online map	Viewing online satellite maps
Chrome	Web browser	Web surfing
Camera	Multimedia	Taking 30 pictures

Therefore, the snapshot analysis does not necessarily reveal the true potential of data compression, and data compression could still be very beneficial if the write traffic bound for flash storage is highly compressible.

Evaluation Setup: To assess the compressibility of write traffic, we inserted a routine in the dispatch routine of the kernel I/O scheduler. The new routine performs LZO compression on the virtual memory pages (VM pages) associated with block write requests to collect compression ratio information without altering the data in VM pages. This analysis was performed on the N5. The Android and kernel versions are 4.4.4 and 3.4, respectively. The file system is Ext4.

We ran a collection of popular Android apps on the N5 to examine the write traffic compressibility. Table 1 summaries the application scenarios. All the applications were used for at least five minutes. The write traffic was classified into five different types according to the purposes of their destination disk blocks. `Journal` stands for the write traffic bound for the Ext4 journal space, and `Meta` corresponds to that bound for the Ext4 inodes, allocation bitmaps, and super blocks. The definitions of `SQLite` and `Multimedia` were the same as those in Figure 2, and the write traffic of `Others` goes to all the other files including executable files.

Figure 3 shows write traffic of different compressibility and its breakdown with respect to different data types. Interestingly, five out of the six applications produced high volumes of compressible write traffic. The results of Facebook and Twitter appeared highly similar: Large portions of their write traffics were of good compressibility. Specifically, about 46% of the Facebook write traffic was of high or moderate compressibility, and the write traffic was mainly contributed by SQLite files and the Ext4 journal. There are two important reasons for this phenomenon: Firstly, Facebook and Twitter heavily rely on the SQLite layer for transactional data management. The highly synchronous write behavior of the SQLite layer amplified the write traffic volume to SQLite files and the Ext4 journal through frequent commits of file system transactions [5]. Secondly, SQLite files and the Ext4 journal contain highly structured data [16], and these contents are highly compressible. By contrast, "other files" (i.e., `Others`) contributed to a large portion of the minimally compressible write traffic. This is because these two applications created temporary files to cache recently browsed image files, and the image files are not compressible. The compressibility distribution of Messenger appears similar to those of Facebook and Twitter, except that Messenger had much less minimally compressible data. This is because, even though Messenger also involved frequent SQLite operations, it is text-based and did not write many multimedia files.

Google Earth involved the highest percentage of SQLite-related write traffic among the six applications. This is because Google Earth saves all map tile images as Binary Large Object (BLOB). The images contributed a large volume of write traffic of low or minimal compressibility. In addition to the images, we found that Google Earth also stored map metadata in SQLite files, and

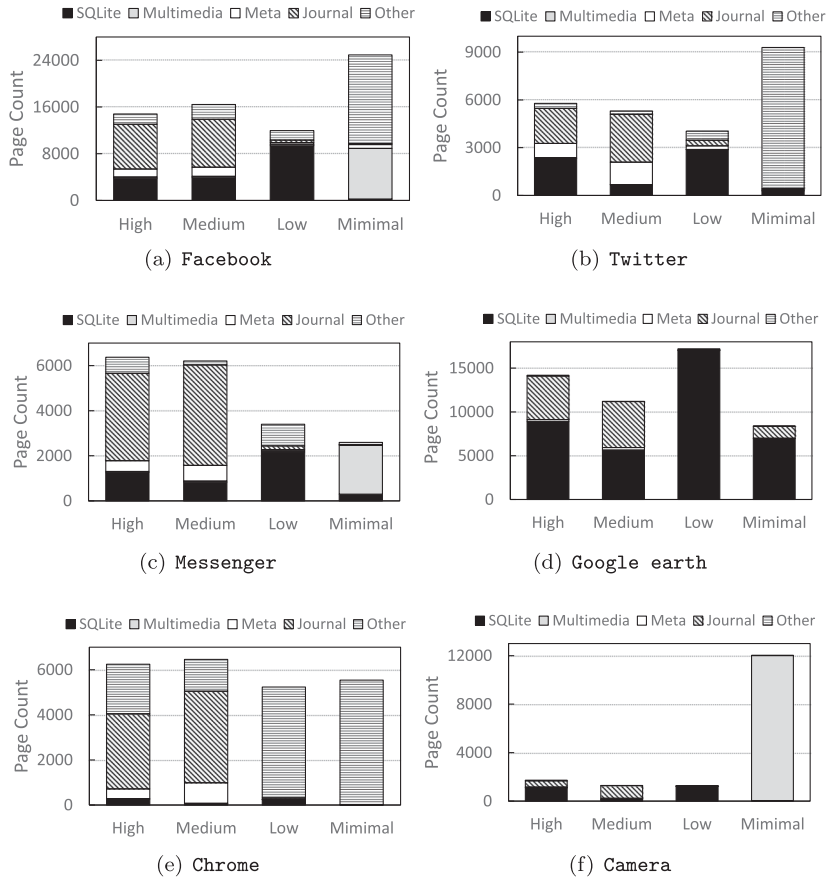


Fig. 3. Compressibility distribution with breakdowns of different types of data. The X-axis stands for different compressibility, while the Y-axis depicts the total number of virtual memory pages dispatched to the block device driver for writing.

the map metadata was highly or moderately compressible. Overall, about one half of the write traffic was of medium or high compressibility. Chrome showed a highly similar compressibility distribution, but the breakdowns appeared very different. Specifically, its minimally compressible write traffic were mainly contributed by the cache files of recently browsed images.

The write traffic of Camera was mostly contributed by JPEG pictures, and not surprisingly, most of its write traffic was not compressible. Camera maintained a database for picture organization, and updates to this database produced a fraction of highly or moderately compressible data.

In summary, Android smartphones produced a high volume of highly or moderately compressible write traffic through frequent SQLite operations and Ext4 journaling. This phenomenon is directly related to the mobile application behaviors and the file system design. Our observation suggests that, in contrast to widely held beliefs, *data compression can be beneficial to mobile flash storage* in terms of write traffic reduction and flash memory lifespan extension.

5 LIGHTWEIGHT DATA COMPRESSION

As previously discussed in Section 3, unconditional data compression could unexpectedly degrade write latency. Because smartphone users are highly sensitive to latency, data compression on

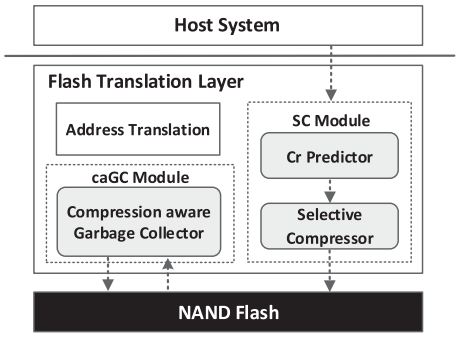


Fig. 4. An overall architecture of LDC.

incoming write requests should be selective. Our basic idea is to compress incoming data of high compressibility, while leaving data of low compressibility to a compression-aware garbage collector. Figure 4 shows the architecture of a mobile flash storage with the proposed approach. The proposed approach, Lightweight Data Compression (LDC), involves a few add-on components of the FTL. On the arrival of new data, *Cr Predictor* produces a prediction of the compression ratio of the data. Based on the predicted compression ratio, *Selective Compressor* decides whether or not to compress the data before writing it to flash. Uncompressed data in flash will later be examined by *compression-aware Garbage Collector*, which performs data compression during background garbage collection.

5.1 Fast Prediction of Compression Ratio

The first challenge is how to get the compression ratio of a piece of data before it is actually compressed. Instead of the actual compression ratio, our approach makes a prediction of compression ratio on the arrival of new data. The time overhead of predicting the compression ratio of a page must be negligibly small compared to that of compressing a page.

We propose to exploit the correlation between entropy and compression ratio for the prediction. Entropy is a metric on the randomness of data. The entropy of a variable X with a set of possible values $\{x_1, \ldots, x_n\}$ is calculated by

$$-\sum_{i=1}^{n} p(x_i) \times log_2(p(x_i)), \tag{3}$$

where $p(x_i)$ is the probability of value x_i. To calculate the entropy of a page, each x_i corresponds to an unique byte value, and $p(x_i)$ is the ratio of the appearance count of the byte value x_i to the page size in bytes. For example, if the byte value "123" appears 10 times in a 4KB page, then p(123) = 10/4096. As reported in [6], entropy reflects a lower bound of the compression ratio for data compression algorithms.

We shall next investigate the relation between entropy and compression ratio with mobile settings. We computed the compression ratio (using LZO) and entropy in terms of 4KB disk blocks of the volume snapshot of our N6. To observe the relation with an emphasis on good compression ratios (low C_r values), here the inverse of C_r is employed instead of C_r. Figure 5 reveals a highly predictable relation: The compression ratio is extremely good when the entropy is close to zero. As the entropy increases, the compression ratio dramatically degrades in the beginning, and then the degradation slows down as the entropy becomes large. The compression ratio approaches 1.0

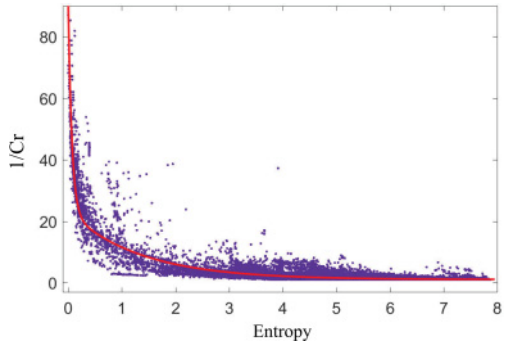


Fig. 5. Entropy and compression ratio of 4KB disk blocks of the volume snapshot of the N6. The X-axis and Y-axis plot the entropy values and the inverse of compression ratios, respectively. The larger the value $\frac{1}{C_r}$ is, the better the compression ratio is.

Table 2. A Fragment of the lookup table for mapping entropy values to compression ratios (C_r)

Entropy	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
C_r	0.11	0.12	0.13	0.14	0.16	0.17	0.18	0.20	0.22

when the entropy becomes the largest possible value 8. This relation can be well approximated using off-line regression analysis, and the approximation function is depicted by the red line.

To predict the compression ratio of a page, the controller first calculates the entropy of the page and then maps the entropy to a compression ratio. The calculation of entropy involves floating point operations, which are not often supported by embedded controllers. Because there are only 256 unique byte values, the logarithm function can be implemented using a small *Log Lookup Table*. All arithmetic operations can also be converted to integer operations through multiplying all variables by a large integer. We measured that the proposed integer operations introduced a small error less than 1.7% in entropy values. The mapping from entropy to compression ratio can be implemented using another lookup table, *Cr Lookup Table*, based on the approximation function in Figure 5. Table 2 shows a fragment of this table and the numbers in the table are converted to integers for implementation. We further propose randomly sampling 10% from the 4096 bytes of a page for fast entropy computation. We measured that the sub-sampling created a small error less than 3.3% in entropy values. Overall, the error with our fast entropy computation is less than 5%. The time overhead of predicting the compression ratio of a 4KB page is negligibly small, about 1.8% of that of compressing the page.

5.2 Selective Compression

We propose Selective Compression (SC) to handle data compression on incoming write requests. Since the request size is known upon the arrival of a write request, a reference compression ratio can be computed for the request using Equation (2). For each page of the write request, the compression ratio of the page is predicted using the proposed fast prediction method. The page is compressed if the prediction is lower than the reference compression ratio. Otherwise, the page will not be compressed. Pages of compressed data and uncompressed pages are written to separate flash blocks.

Figure 6 shows an example of how unconditional compression and our SC work. Let there be four single-page write requests, R_1 to R_4, which are numbered by order of arrival. The requests are

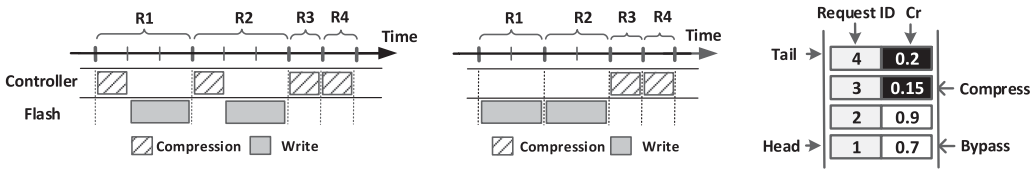


Fig. 6. The results of handling four write requests using (a) Unconditional Compression and (b) the proposed Selective Compression.

serviced in a first-come first-serve manner. Let the compression ratios of R_1, R_2, R_3, and R_4 be 0.7, 0.9, 0.15 and 0.2, respectively. Let the time overheads of page compression and page programming be one and two units of time, respectively. Figure 6(a) illustrates the unconditional compression approach that compresses all write requests regardless of their compression ratios. Since R_1 has a poor compression ratio, compressing R_1 does not save any flash page write. The flash module starts to write a flash page as soon as the compressed page is added to the write buffer. Because R_2 is also barely compressible, it is handled in the same way as R_1. Next, because R_3 and R_4 have good compression ratios, their compressed pages are added to the write buffer without triggering any flash page writes. The average latency of the four write requests is $(1\times4 + 2\times2)/4 = 2$ units of time.

The proposed SC approach predicts the compression ratio of data on the arrival of a write request. According to Equation (2), the predicted compression ratio must be smaller or equal to 0.5, and therefore SC does not compress R_1 and R_2 but writes two uncompressed pages to flash memory. R_3 and R_4 will be compressed and adding their compressed pages to the write buffer does not trigger a page write. Notice that the time overhead of compression ratio prediction is neglected because it is rather small compared to the overhead of page compression. The average latency of the 4 requests is $(1\times2 + 2\times2)/4 = 1.5$ units of time, which is much better than that of unconditional compression.

eMMC supports packed commands that could pack multiple small requests into a larger one. However, as reported in [20], this benefit is neutralized by the highly synchronous write behaviors of mobile applications, and the majority of write requests arriving at eMMC are small ($\leq$4 KB). Our write traffic study also confirmed this result: Under the Facebook workload, 77% of the write requests are not larger than 8 KB. Therefore, our current design assumes that a storage device processes a single request at a time.

Because compressed pages are of variable sizes, it is possible that a compressed page straddles two flash pages. Reading straddling compressed pages may amplify the read overhead. The proposed SC avoids writing compressed pages across page boundaries. Internal fragmentation in flash pages can later be reclaimed through garbage collection.

5.3 Compression-Aware Garbage Collection

The SC approach writes uncompressed pages if the pages carry less compressible data. These uncompressed pages are still compressible and can later be compressed at proper times. Because garbage collection involves a series of valid page migrations, uncompressed pages can be compressed during background garbage collection with no extra write costs. We propose compression-aware Garbage Collection (caGC) for background data compression. The primary goal of caGC is to improve garbage collection efficiency via compression.

The victim selection policy of caGC is a direct extension of the Greedy policy with the consideration of data compression. Let $S(P_i)$ be the amount of valid data stored in a flash page P_i and a flash

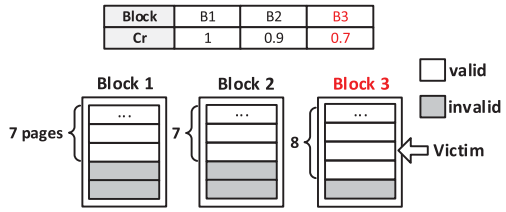


Fig. 7. Victim selection for the proposed caGC.

block has N flash pages. The value of $S(P_i)$ can be determined using the FTL mapping information. The flash page P_i may either contain an uncompressed page or stores a few compressed pages. In the latter case, some of the compressed pages may have been invalidated, and thus $S(P_i)$ returns a value between zero and the flash page size in bytes. For efficient implementation of caGC, we propose to employ a block-level compression ratio. We propose that SC maintains multiple (flash) log blocks. SC writes uncompressed pages of similar predictions of compression ratio to the same log block. Let $C_r(B)$ be the average compression ratio of block B. If block B stores uncompressed pages, then $C_r(B)$ is the average of the predicted compression ratios of all its pages. If block B stores compressed pages, $C_r(B)$ returns 1 (incompressible). caGC selects the block having the lowest score based on the following formula:

$$\sum_{i=1}^{N} S(P_i) \times Cr(B). \tag{4}$$

Figure 7 shows an example of victim selection for caGC. caGC groups data in terms of compressibility instead of data hotness. Although Block 3 has the largest amount of valid data, it will be selected for erasure because it has the lowest score ($8 \times 0.7 = 5.6$) and thus induces the least data migration cost. The compression ratio of Block 1 is 1.0 because it stores compressed data.

During garbage collection, if the migrated data is of low (but not minimal) compressibility, for space reduction, caGC will write compressed pages of the data across flash page boundaries. This is because, based on our snapshot analysis, most of the data of low compressibility are associated with executable files. Because executable files are accessed through large, sequential read requests [12], writing their compressed pages across page boundaries will not noticeably increase the page read frequency.

5.4 Implementation Remarks

5.4.1 Implementation Overhead. The proposed LDC approach (SC plus caGC) relies on a small write buffer to collect compressed pages and to form a pipeline of the controller and flash module. The RAM size required by this buffer is as large as two flash pages, e.g., 8KB. On the other hand, the compression ratio prediction requires two tables, namely, the *Log Lookup Table* and *Cr Lookup Table*. The more entries the tables have, the higher the precision of the results will be. On the other hand, the two tables should be small to save RAM space. In our current implementation, the *Log Lookup Table* has 1000 entries, and the *Cr Lookup Table* uses 81 entries. Each entry of the tables requires two bytes, and in total the two tables use 2162 bytes only.

5.4.2 Address Mapping. Because the storage size of a logical page is variable after compression, the address mapping scheme for compression-aware FTLs is different from a conventional one. To resolve the address mapping for multiple logical pages stored in a physical page, we must keep track of 1) the relation of multiple logical pages to one physical page and 2) the offsets of logical pages within the physical page. There have been a few excellent studies on this issue, and our

Table 3. Parameters of SSD simulator

Description	Configuration
Page Size	4KB
Pages per Block	64
Over Provision Space	1%
Block Erase Time	1.5ms
Page Read/Write Time	25μs/200μs
Page Data Transfer Time	100μs

Table 4. Workload Characteristics

Application	Req. Count	Write Ratio	IO Volume
Facebook (FB)	17428	90%	336.1MB
Twitter (TW)	7869	85%	119.7MB
Messenger (ME)	7573	66%	136.6MB
G. Earth (GE)	25831	95%	237.4MB
Chrome (CH)	6953	83%	140.1MB
Camera (CA)	2301	96%	65.3MB
Multiple (MA)	67955	88%	1034.2MB

approach adopts the address mapping table structures of zFTL [13]. zFTL extends the structure of page mapping table by adding two attributes: FLAG indicates whether the corresponding logical page is compressed, and IDX represents the index of each logical page within the physical page.

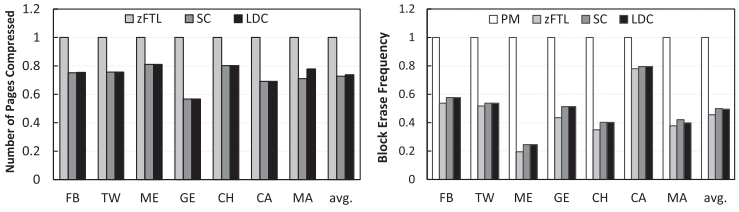
6 PERFORMANCE EVALUATION

6.1 Experimental Setup

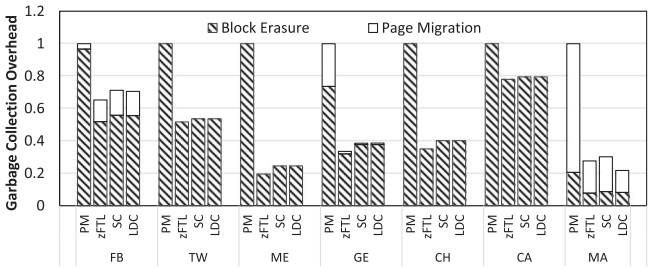
We implemented the proposed LDC approach in the DiskSim simulation environment [3] with the Microsoft Research SSD extension [2]. The capacity of the simulated flash storage was 16GB with parameters shown in Table 3. The flash storage consisted of a tiny byte-addressable write buffer of two page slots, as previously shown in Figure 1(a). Our simulation involved the pipelined execution of the controller and flash module. Since writing/reading a flash page involves data transferring and flash access, t_w and t_r used in the performance model were 300μs and 125μs, respectively. We assumed a controller clock speed of 200MHz. At this clock rate, the time overheads of compressing and decompressing a 4KB page were 136μs and 33μs, respectively, based on the performance data reported in [18].

The following methods were evaluated for performance comparison:

(1) Page Mapping FTL (**PM**): It is the original page mapping FTL without any data compression.
(2) Compression FTL (**zFTL [13]**): It is a hardware-based approach. For fair comparison, data compression in zFTL is performed by the firmware. As suggested in the original study, all incoming data are compressed unless the predicted compression ratio is extremely poor ($C_r > 0.95$). Basically, zFTL can still be seen as unconditional compression because the compression ratio threshold is very high.
(3) Selective Compression (**SC**): It extends zFTL with our proposed SC approach. SC selectively compresses incoming data according to the criterium shown in Equation (2). No background compression is involved.



(a) Number of Pages Compressed (b) Block Erase Frequency



(c) Garbage Collection Overhead

Fig. 8. Results of number of pages compressed, block erase frequency, and garbage collection overhead.

(4) Lightweight Data Compression (**LDC**): LDC enhances SC with compression-aware Garbage Collection (caGC). The data not compressed by SC will be treated by caGC for background compression.

The I/O workloads were collected from the application scenarios described in Table 1. For each I/O request, we recorded the block address, the request length, entropy and compression ratio of each page associated with the request. Each of the workloads was replayed three times on the flash storage simulator to create a sufficiently large write traffic volume. In addition to these workloads, a new workload, called Multiple Applications (MA), is created by combining all the single-application workloads. This workload aims to reflect long-term user behavior. Before each run of the simulation, the compression ratio of existing data in flash storage is initialized based on the compressibility distribution previously presented in the volume snapshot analysis.

6.2 Experimental Results

6.2.1 Number of Pages Compressed. This performance metric reflects how many pages are compressed, including the pages of new requests and the existing pages compressed during garbage collection. All results are normalized to those of zFTL. Not surprisingly, Figure 8(a) shows that zFTL compressed the largest number of pages, because it unconditionally compressed incoming pages (except those with poor compression ratios). In comparison, SC and LDC compressed noticeably fewer pages, because of their selective compression strategy. Notably, the difference is the largest under the GE workload. This is because, as shown in Figure 3, Google Earth produced the largest percentage of data of low compressibility. Such data were skipped by SC and LDC, but they were still compressed by zFTL. Under MA workload, LDC compressed about 7% more pages than SC did. This is because garbage collection under the MA workload was the most frequent, and LDC compressed more pages through the compression-aware garbage collection (caGC).

6.2.2 Block Erase Frequency. This metric indicates how many block erase operations are performed. The less the block erase frequency is, the larger gain in flash storage lifespan is. All results are normalized to those of PM. Figure 8(b) shows that LDC significantly reduced the block erase frequency by 50% on average. Roughly speaking, *LDC doubles the flash storage lifespan.* Not surprisingly, zFTL achieved the lowest block erase frequency because it compressed the largest number of incoming pages for maximum space reduction. However, the results of LDC and zFTL were very close and the difference was only about 4%.

6.2.3 Garbage Collection Overhead. This overhead consists of block erase time and page migration time. We shall focus on the result under the MA workload in Figure 8(c) because this workload imposed the highest pressure on garbage collection. LDC noticeably reduced the time overhead of page migration compared to SC. This is because LDC compressed valid data during page migration while SC did not. It is worth noting that LDC also outperformed zFTL, which compressed all pages before garbage collection. This is because LDC eliminates internal fragmentation in pages by writing compressed pages across page boundaries. zFTL left internal fragmentation in flash pages, and thus it wrote a larger number of flash pages than LDC did.

6.2.4 I/O latency. This part is focused on I/O latency. All results are normalized to PM. We expect that SC and LDC will not underperform PM in terms of I/O latency.

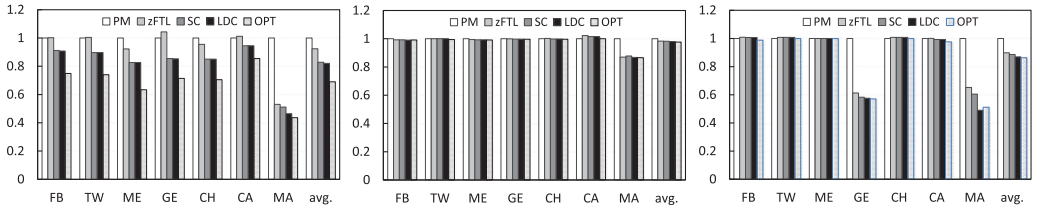
Write latency: As shown in Figure 9(a), compared to the non-compressing PM, zFTL suffered from longer write latencies under four out of all the seven workloads. For example, the write latency of zFTL was 4% longer under the GE workload. This is because zFTL compressed pages of low compression ratios, and in this case the compression overhead negatively impacted on the write latency.

On the other hand, the proposed SC and LDC achieved better write latency than PM: the average write latencies of SC and LDC were 17.2% and 18.0% lower than that of PM, respectively. The improvement can be discussed from two angles: Firstly, under the single-application workloads, the write latencies of SC and LDC were mainly affected by the overhead of compressing incoming write requests. In this case, benefiting from the selective compression strategy, SC and LDC outperformed the non-compressing PM. Secondly, the multi-application MA workload imposed a high pressure on garbage collection through its high write traffic volume, and LDC further outperformed SC because of the compression-aware garbage collection (caGC) strategy.

zFTL performed worse than SC and LDC by 9.6% and 10.4%, respectively. Even though zFTL should have the lowest garbage collection overhead because of its maximum space saving, the extra latency cost due to unconditional compression appeared larger than the benefit of garbage collection overhead reduction.

Read latency: Decompressing a page is significantly faster than compressing a page. Even though page decompressing and page reading are also pipelined, at least one page decompression operation cannot be overlapped with a flash page read. In addition, reading a compressed page that straddles two flash pages requires two flash page reads.

Interestingly, Figure 9(b) shows that the read latencies of the proposed SC and LDC approaches were comparable to or even better than that of the non-compressing PM. This is because firstly, many of the read requests involve uncompressed data. Secondly, the reduction in write latency improved the waiting time of all requests in the storage command queue, including read requests. This effect is particularly significant under the write-intensive MA workload. Thirdly, reading straddling compressed pages barely amplified the read latency. As discussed in the design of caGC, these straddling pages are of low compressibility, and our snapshot analysis shows that many such data are associated with sequential-reading executable files. Under CA workload, all the compressing methods experienced a very small read performance drop compared to PM. This is because



(a) Write Latency (b) Read Latency (c) Worst-case I/O Latency

Fig. 9. Results of write, read, and worst-case I/O latencies.

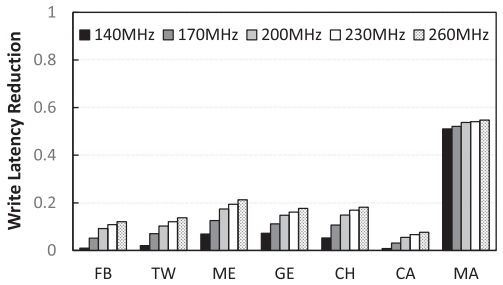


Fig. 10. The reduction in write latency compared to PM under different controller clock rates.

previously compressed pages were repeatedly read, and the decompression overhead sightly increased the read latency.

Worst-case I/O latency: The proposed SC and LDC approaches significantly improved the worst-case I/O latency under the GE and MA workloads. The GE workload produced long bursts of write requests, while the MA workload occasionally triggered on-demand garbage collection. The worst-case I/O latency was significantly amplified by the waiting time in the storage command queue when write bursts or on-demand garbage collection occurred. By reducing the write latency, SC and LDC decreased the waiting time of requests and improved the worst-case latency.

Figure 9 also includes OPT, which is unconditional data compression with a zero compression (and decompression) cost. Free data compression significantly benefited the write latency of OPT under the single-application workloads. However, OPT performed slightly worse than LDC under the MA workload in terms of the worst-case latency, because LDC achieved a lower garbage collection overhead by eliminating internal fragmentation in flash pages.

6.2.5 Compression Overhead. We varied the controller clock rate from 140MHz to 260MHz to evaluate how data compression overhead affects write latency. The time overhead of compressing a page under a particular clock rate is proportionally scaled with respect to 136μs at 200MHz. The controller clock rate has a direct impact on the selective compression method. Figure 10 shows the write latency reduction achieved by LDC with respect to PM. These results suggest that the reduction improves as the clock rate increases. This is because the smaller the value t_c in Equation (1) is, the more incoming pages can be compressed without any latency degradation. This reduction gradually saturated as the clock rate increased. On the other hand, when the clock rate was 140MHz, the time to compress a page is almost the same as that to program a flash page, and thus LDC barely compressed incoming pages through selective compression. Notably, increasing the clock rate under the MA workload did not improve the latency reduction as much as it did under

the single-application workloads. This is again because the MA workload imposed a high pressure on garbage collection, and caGC had a significant contribution to the latency reduction.

7 CONCLUSION

This study investigates data compressibility based on block write traffic of mobile applications. The analysis identifies that the write traffic bound for smartphone storage volumes is highly compressible. Based on our findings, a firmware approach for data compression is proposed. The proposed approach makes a fast prediction of the compression ratio on incoming data and decides whether or not to compress the data. Data of good compressibility are compressed upon their arrivals, while the other data are left to background compression through garbage collection. Experimental results show that the proposed approach significantly reduced the frequency of block erase by 50.5% compared to uncompressed flash storage and improved the write latency by 10.4% compared to unconditional compression. Most importantly, our approach does not require extra data compression hardware.

ACKNOWLEDGMENT

This work is partially supported by National 863 Program 2015AA015304 and Ministry of Science and Technology of Taiwan (MOST 104-2221-E-009-011-MY3), the Fundamental Research Funds for the Central Universities(106112016CDJZR185512), NSFC(61402059 and 61572411), and Huawei Innovation Research Program(HIRP).

REFERENCES

[1] 2017. LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/. (2017).
[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of ATC*. 57–70.
[3] Schindler J. Schlosser S. J. Bucy, and G. Ganger. 2008. DiskSim 4.0. http://www.pdl.cmu.edu/DiskSim. (2008).
[4] Hsin-Yu Chang, Chien-Chung Ho, Yuan-Hao Chang, Yu-Ming Chang, and Tei-Wei Kuo. 2016. How to enable software isolation and boost system performance with sub-block erase over 3D flash memory. In *Proceedings of CODES+ISSS*. ACM.
[5] Li-Pin Chang, Po-Han Sung, Po-Tsang Chen, and Po-Hung Chen. 2016. Eager Synching: A Selective Logging Strategy for Fast Fsync() on Flash-Based Android Devices. *ACM Trans. Embed. Comput. Syst* (2016).
[6] Georges Hansel, Dominique Perrin, and Imre Simon. 1992. Compression and entropy. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 513–528.
[7] Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. 2006. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage (TOS)* 2, 1 (2006), 22–40.
[8] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Proceedings of ATC*.
[9] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. 2016. An empirical study of file-system fragmentation in mobile storage systems. In *Proceedings of HotStorage*.
[10] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of Hotstorage*.
[11] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)* (2012).
[12] Kisung Lee and Youjip Won. 2012. Smart layers and dumb result: IO characterization of an Android-based smartphone. In *Proceedings of EMSOFT*. ACM, 23–32.
[13] Youngjo Park and Jin-Soo Kim. 2011. zFTL: power-efficient data compression support for nand flash-based consumer electronics devices. *IEEE Transactions on Consumer Electronics* 57, 3 (2011), 1148–1156.
[14] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* (2013).
[15] Assim Sagahyroon. 2006. Power consumption in handheld computers. In *Proceedings of APCCAS*. IEEE.
[16] Mungyu Son, Junwhan Ahn, and Sungjoo Yoo. 2015. A tiny-capacitor-backed non-volatile buffer to reduce storage writes in smartphones. In *Proceedings of CODES+ ISSS*. IEEE.

[17] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah. 2010. Analyzing the energy efficiency of a database server. In *Proceedings of SIGMOD*. ACM.
[18] Guanying Wu and Xubin He. 2012. Delta-FTL: improving SSD lifetime via exploiting content locality. In *Proceedings of EuroSys*. ACM.
[19] Xuebin Zhang, Jiangpeng Li, Hao Wang, Kai Zhao, and Tong Zhang. 2016. Reducing Solid-State Storage Device Write Stress through Opportunistic In-place Delta Compression. In *Proceedings of FAST*.
[20] Deng Zhou, Wen Pan, Wei Wang, and Tao Xie. 2015. I/O characteristics of smartphone applications and their implications for eMMC design. In *Proceedings of IISWC*. IEEE.
[21] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2014. Compression and SSDs: Where and How? In *Proceedings of INFLOW*.

Received April 2017; revised June 2017; accepted June 2017