zFTL: Power-Efficient Data Compression Support for NAND Flash-based Consumer Electronics Devices

Youngjo Park and Jin-Soo Kim, *Member*, IEEE

Abstract — *Flash translation layers play an important role in determining the storage performance and lifetime of NAND flash-based consumer electronics devices. In this paper, we present a flash translation layer called zFTL, which reduces the amount of data written to NAND flash memory by supporting on-line, transparent data compression based on the X-Match algorithm. To minimize compression overhead and power consumption, we also propose a novel prediction scheme that identifies incompressible data in advance without going through full compression.*

Our evaluations with five real-world workloads show that zFTL successfully enhances storage performance and lifetime by improving the write amplification factor (WAF) by a factor of 2.6 (geometric mean) compared to the case without compression support. In addition, we find that the proposed prediction scheme is effective in reducing power consumption by skipping compression for incompressible data[1].

Index Terms — NAND flash memory, flash translation layer (FTL), data compression, incompressible data prediction.

I. INTRODUCTION

Recently, NAND flash memory has become a necessity as a storage medium for mobile consumer electronics devices, thanks to its non-volatility, superior performance, shock resistance, and low-power consumption. With technology advancing, the capacity of NAND flash memory is getting larger and its price is getting lower.

However, NAND flash memory has several limitations. First, previous data should be erased before a new data can be written in the same place. This is usually called *erase-before-write* characteristic. Second, normal read and write operations are performed on a per-*page* basis, whereas erase operations on a per-*block* basis. The erase block size is larger than the page size by 64-128 times. In MLC (Multi-Level Cell) NAND flash memory, the typical page size is 4KB and each block consists of 128 pages. Finally, flash memory has limited lifetime; MLC NAND flash memory wears out after 1K to 5K write/erase cycles.

[1] This work was supported by Future-based Technology Development Program (No. 2010-0020730) and by Mid-career Researcher Program (No. 2010-0026511) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology.

Youngjo Park is with Memory Division, Samsung Electronics Co., Hwasung 445-701, South Korea (e-mail: yj113.park@samsung.com).

Jin-Soo Kim (corresponding author) is with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon 440-746, South Korea (e-mail: jinsookim@skku.edu).

Contributed Paper
Manuscript received 07/15/11
Current version published 09/19/11
Electronic version published 09/19/11.

The aforementioned limitations are effectively hidden through the use of an intermediate software layer called Flash Translation Layer (FTL) [1]. The basic role of the FTL is to emulate the traditional block interface on top of NAND flash memory so that the existing disk-based file systems can be used without any modification. For this reason, many NAND flash-based consumer electronics devices, such as MP3 players, in-car navigators, smartphones, tablets, and digital TVs, implement the FTL in the operating system.

Most FTLs employ an *address remapping* technique, which writes incoming data into one or more pre-erased pages and maintains the mapping information between the host's logical sector number and the on-flash physical page number. As the new data are written, the previous version is invalidated, and those obsolete pages are collected and then eventually converted to free pages via the procedure known as *garbage collection*. To cope with the limited write/erase cycles, FTLs also perform *wear-leveling* which distributes erase operations evenly across the entire flash memory blocks [2], [3].

Although garbage collection and wear-leveling improve the overall performance and lifetime, they cause additional writes. One way to quantify the added cost of an FTL is to measure the *write amplification factor (WAF)* [4]. The WAF is defined as the ratio of actual data written into NAND flash memory as compared to the actual data written by the host system. A lower WAF is a measure of efficient storage and housekeeping algorithms inside FTL, improving the overall life expectancy of NAND flash memory by lowering the total write/erase cycles required to manage the data stored in flash memory. Although the WAF of hard disks is 1.0, the WAF can be as high as 10 on low-end flash memory cards.

In this paper, we present the design and implementation of a flash translation layer called zFTL, which internally compresses or decompresses data. Data compression is an effective way to lower the WAF further down to below 1.0, thus improving FTL performance and lengthening flash lifetime. Specifically, this paper discusses and evaluates several design issues arise when we support on-line, transparent compression/decompression inside FTL. zFTL is based on page-level address remapping [5] and the compression unit size is set to 4KB. We focus on the management of the compressed data, assuming the actual compression/decompression is done by dedicated hardware. For this reason, zFTL uses the X-Match [6] algorithm which allows for fast hardware implementation. In addition, this paper proposes a novel prediction scheme called Incompressible Data Predictor (IDP) for the X-Match

algorithm, which identifies incompressible data before they are fully compressed. The purpose of the IDP is to avoid data compression for incompressible data, thereby saving time and power consumption.

zFTL is evaluated with five real-world workloads. Our results show that the use of data compression improves the WAF by a factor of 1.8 to 4.7. We also find that the proposed IDP is effective in reducing power consumption especially when many of the input data are incompressible. From these results, we believe zFTL is a power-efficient way of enhancing the storage performance and lifetime of NAND flash-based consumer electronics devices.

The rest of the paper is organized as follows. The next section discusses the related work. Section III introduces the overall architecture and design issues of zFTL. Section IV describes the proposed IDP scheme in detail. Section V presents the experimental results and section VI concludes the paper.

II. Related Work

Data compression techniques have been studied in various layers in computer systems. JFFS2 [7] is a representative flash-aware file system inspired by the log-structured file system [8]. JFFS2 provides an option to use Zlib-based data compression [9]. CramFS [10] and SquashFS [11] are compressed read-only file systems, mainly targeting the root file system in small embedded systems. Hyun et al. [12] proposed LeCramFS which modifies CramFS for NAND flash memory. These flash-aware file systems do not require FTL, as they work directly on NAND flash memory.

Yim et al. [13] studied a flash compression layer for SmartMedia card system, proposing an internal packing scheme (IPS) to manage internal fragmentation. The IPS best-fit scheme can reduce the internal fragmentation effectively, but it may incur some read overhead as unrelated logical sectors are packed together to minimize internal fragmentation. Chen et al. [14] proposed another internal packing scheme called IPS real-time. In the IPS real-time scheme, the compressed data can be stored into consecutive flash pages, but it has no consideration for random reads; it needs to access two flash pages to read a sector which spans two pages. Both approaches focused only on reducing internal fragmentation, without considering other issues such as mapping information management and garbage collection under the presence of compressed data. In addition, they are devised for old 512-byte flash page size, which has been outdated by new generations of NAND flash memory chips.

Special hardware compressor/decompressor engines have been proposed in several literatures. The Memory Expansion Technology (MXT) [15] performs compression and decompression between the shared cache and the main memory, to expand the effective main memory size using hardware implementation of the LZ77 algorithm [16], [17]. Benini et al. [18] investigated a hardware-assisted data compression for memory energy minimization. They describe the implementation of hardware compression algorithms including LZ-like one in detail and show no penalty in performance. Kjelso et al. [6] proposed the X-Match compression algorithm for main memory, which is easy to implement in hardware. X-Match is another variant of LZ77, differing in that phrases matching works in four bytes unit [19]. For brevity, we assume data compression and decompression is assisted by special hardware that is fast enough to hide its overhead.

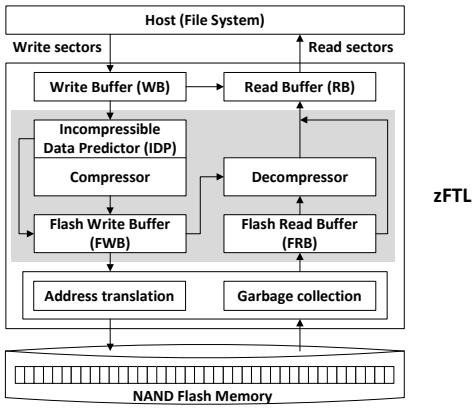

Fig. 1. The overall system architecture of zFTL. The shaded area indicates the added components to support data compression in zFTL.

III. zFTL

A. System Architecture

Fig. 1 shows the overall architecture of zFTL. File systems issue read/write requests to zFTL. The size of each request is a multiple of the disk sector size (512B). For write requests, zFTL aggregates the requested data in the Write Buffer (WB), whose size is equal to the compression unit size. If the WB is full, the data in the buffer are compressed and the compressed data (called a "*chunk*") are appended into the Flash Write Buffer (FWB). The size of the FWB is a multiple of the flash page size (4KB in MLC NAND). Although the compression unit size is fixed, the resulting chunk is highly variable in size. Hence, the FWB may hold a number of chunks depending on the compression ratio.

When the FWB has not enough space for the incoming chunk, the FWB is flushed into flash memory. Before flushing data, the corresponding logical sectors are remapped to new physical pages by zFTL. In case the previous data are available in any of buffers, they are removed from the buffer, ensuring data consistency and preventing the invalidated data from being flushed into flash memory. If the number of free blocks is below a certain threshold, zFTL initiates garbage collection to reclaim erase blocks. We will discuss the garbage collection process of zFTL in section III.D.

For reads, zFTL first searches for the requested data in the WB as it may have the most recent version of the data. When the search fails, zFTL looks up the data in the FWB. If the data are found in the FWB, the corresponding chunk is decompressed and then loaded into the Read Buffer (RB). When the data are still not found in the FWB, zFTL examines

the RB and the Flash Read Buffer (FRB). Note that the two write buffers (WB and FWB) should be looked up before the two read buffers (RB and FRB), as they may keep the up-to-date data. While the requested data are stored in any of these buffers, the read request can be satisfied without issuing any flash read operations. Otherwise, zFTL needs to decompress the requested chunk after reading the corresponding page from flash memory.

Some data are inherently incompressible. This happens if the data belong to multimedia files (such as *.jpg, *.mp3, *.avi, *.mpg, etc.) or compressed archive files (such as *.zip, *.rar, *.gz, etc.). zFTL identifies those incompressible data based on the resulting size after compression. When the compression ratio is not good enough to justify the overhead of storing the data in compressed form (currently, 6 bytes for each chunk), zFTL stores the original (uncompressed) data into flash memory in order to save space. While reading these data, zFTL directly copies the contents of the FRB to the RB, bypassing the decompressor. This also saves the time and energy that might be spent on decompressing such data.

One problem with this approach is that it is uncertain whether the current data are sufficiently compressible or not until the entire data are processed by the compressor engine. If we could determine whether the incoming data are incompressible or not in advance before the actual compression, the compression overhead can be avoided for incompressible data. In Fig. 1, the Incompressible Data Predictor (IDP) is introduced for this reason. The IDP examines a small subset of data in the WB and predicts whether the current data are incompressible or not. If the data are predicted incompressible, the compressor engine is bypassed and the original data are forwarded to the FWB. The prediction scheme used in the IDP will be described in detail in section IV.

B. Compression Algorithms

The choice of compression algorithms is one of the important design issues, because it determines the speed of compression/decompression, the compression ratio, and the complexity of hardware implementation. Many hardware implementations of LZ77 [16] or variants have been proposed in previous studies. Among them, we choose a variant of LZ77 called the X-Match [6] algorithm for zFTL. X-Match not only shows fairly reasonable compression ratio across the workloads, but also allows for efficient hardware implementation. Moreover, we show that it is possible to develop an effective Incompressible Data Predictor (IDP) for the X-Match algorithm in section IV.

The unit of data compression is another important factor affecting the compression ratio and speed. In particular, dictionary-based algorithms such as LZ77 and X-Match have the characteristic that the bigger compression unit tends to yield the better compression ratio. This is because these algorithms replace a repeated pattern of strings within the compression unit by a much shorter but uniquely identifiable string.

We have considered two options related to the unit of compression. One is to compress the variable-sized data as a whole as it is delivered by a single write request from the file system. The number of sectors written by a write request is usually a multiple of the file system block size and can be as large as 256 sectors (i.e., 128KB) for sequential writes. Thus, this scheme can improve the overall compression ratio and reduce the number of mapping entries. However, the use of the variable-sized compression unit presents a number of issues that need careful handling. For example, when a portion of the compressed data is read by a read request, the entire compressed data should be fetched from flash memory for decompression. An even worse scenario occurs when the compressed data are partly updated by a later write operation. In this case, the original data should be merged with the new data after decompression. Then, it can be either recompressed and stored into flash memory as a single compression unit, or split into two or three pieces each of which is separately compressed and stored.

Another option is to compress a fixed size of data at a time. In fact, any power of two multiple of the sector size, such as 512B, 1KB, 2KB, 4KB, 8KB, etc., can be used as the compression unit size. As discussed before, the use of larger compression unit size is favored for better compression ratio. However, if the compression unit size becomes too large, the system suffers from unnecessary overhead when the compressed data are partly read or updated. Moreover, enlarging the compression unit size has a diminishing return in the compression ratio. Burrows et al. [20] and Yim et al. [13] have shown that there is no significant difference in the compression ratio for 2KB to 8KB compression unit sizes.

For the above reasons, zFTL uses a fixed compression unit size of 4KB. Since most file systems use at least 4KB as the file system block size, they rarely issue I/O operations smaller than this size and the read/write request sizes are usually a multiple of 4KB. In addition, the compression unit size of 4KB is large enough to achieve good compression ratio.

C. Address Mapping

zFTL employs a page-level mapping technique [5] where a per-page mapping entry from the logical page number to the physical flash page number is maintained in the Page Mapping Table (PMT). Similar to other FTLs with page-level mapping, PMT is accessed by the logical page number. To support data compression, zFTL extends the structure of PMT slightly. Each 32-bit mapping entry includes the incompressible data flag (FLAG) and the page index (IDX), as well as the physical page number (PPN) where the page is stored. FLAG indicates whether the corresponding logical page is compressed or not. Since a single flash page may accommodate compressed chunks from several logical pages in zFTL, IDX is used to represent the relative position of each logical page within the physical page. Fig. 2 illustrates an example of PMT in zFTL. Note that PMT entries for the logical page number 100, 101, and 102 have the same value for the PPN field, representing that the data for those logical pages are compressed and stored

in the same physical page number 320 in the order indicated by the IDX value. For incompressible data, the corresponding FLAG is set to 1 (cf. the PMT entry of the logical page number 103 in Fig. 2).

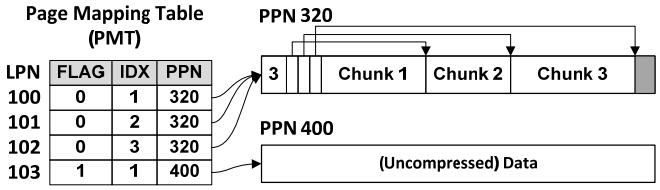

Fig. 2. The structure of page mapping table (PMT) and on-flash layout in zFTL. The contents of logical page numbers (LPNs) 100-102 are compressed and stored in physical page number (PPN) 320. The data in LPN 103 are incompressible (FLAG==1), hence they are stored uncompressed in PPN 400.

Depending on FLAG, the physical flash page has two different structures. For incompressible data (FLAG = 1), the entire page is devoted to the (uncompressed) original data. When the page size is larger than the compression unit size, each data block is identified by IDX. On the other hand, when the value of FLAG is 0, the related physical page includes such information as the total number of chunks in the page, a set of offsets for each chunk, and a set of chunks, as depicted in Fig. 2. The offset indicates the last byte position of the corresponding chunk in the page.

D. Garbage Collection

As in other FTLs, zFTL reserves a set of erase blocks (5% of the total erase blocks, by default) to absorb the incoming write requests. When zFTL runs out of available erase blocks, garbage collection is invoked to reclaim the space allocated to obsolete pages. zFTL uses the greedy policy to choose a victim erase block, i.e., the erase block which has the smallest number of valid pages is selected as a victim. During garbage collection, the remaining valid pages in the victim erase block are copied into another erase block and the victim erase block is cleared to be used later.

Since each physical page normally contains the data from more than one logical page in zFTL, it can be partially invalidated by subsequent write operations. Therefore, zFTL should be able to identify the current status of each chunk stored in the same physical page, in order to copy only the valid chunk during garbage collection. For this reason, zFTL maintains the Page Status Table (PST) in memory. Unlike PMT, PST is indexed by the physical page number, and each PST entry keeps track of the number of valid chunks and the bitmap for each chunk stored in the given physical page number. The bitmap indicates whether the corresponding chunk is valid or not.

Fig. 3 shows an example 8-bit PST entry designed for 4KB physical pages. Fig. 3 represents that two chunks (the second and the third one) are currently valid in the physical page number 330. Under this PST structure, up to five logical pages can be packed into a 4KB physical page. Our experiments show that about three chunks are stored in a single 4KB flash page on average for the most of well-compressed workloads. Thus, we believe the 8-bit entry is sufficient for 4KB flash pages. If the page size is increased, we can add a few more bits to each PST entry.

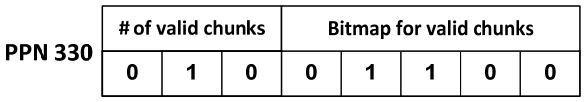

Fig. 3. An example PST (Page Status Table) entry. This example shows that there are two valid chunks (the second and the third one) in physical page number (PPN) 330.

E. Internal Fragmentation

The flash page size is fixed whereas the resulting chunk size varies after compression. Unless we allow a chunk to be stored in more than one page, internal fragmentation is unavoidable. The relative amount of internal fragmentation will be getting smaller as the page size becomes larger than the compression unit size. Considering the recent trend in NAND flash memory architecture where the page size grows progressively larger, the impact of internal fragmentation can be of minor significance, compared to the benefit of compression support.

Currently, zFTL does not implement any special scheme to reduce internal fragmentation. zFTL simply packs the incoming data in the order they are issued from the upper layer. We leave a more comprehensive analysis and possible optimization on internal fragmentation for future work.

F. Memory Requirement

The memory requirement of zFTL is comparable to other FTLs with page-level mapping. The use of block-level mapping can decrease the memory requirement by a factor of 64 to 128, but the increasing number of flash-based storage is adopting page-level mapping due to its superior performance and higher flexibility. Since other page-mapping FTLs also keep page-level address mapping information in memory (i.e., PMT in zFTL), only the memory used by PST is the added cost in zFTL, which requires 512KB for 2GB flash memory with 4KB page size.

If PMT and PST are too large to be accommodated in memory, zFTL may use the selective caching method used in DFTL [5], where the whole mapping table is stored in flash memory and only the needed part of the mapping table is loaded into memory.

IV. PREDICTING INCOMPRESSIBLE DATA

A. Overview of the X-Match Algorithm

The goal of the Incompressible Data Predictor (IDP) shown in Fig. 1 is to identify the incompressible data in advance without going through full compression. To design an effective predictor, it is necessary to investigate the characteristics of the underlying X-Match compression algorithm.

The X-Match algorithm is a dictionary-based lossless data compression algorithm. X-Match maintains a dictionary of

data previously seen and attempts to match the current data with an entry in the dictionary [6]. Its dictionary is composed of up to 128 entries and each entry has 4 bytes. X-Match reads 4 bytes from the input data (referred to as a "*tuple*") at a time for matching. Fig. 4 illustrates the basic idea of the X-Match algorithm with example cases.

If the incoming tuple fully matches with an entry in the dictionary as shown in Fig. 4(a), a single bit of '0' is emitted first as an output to indicate a match, followed by the information on the match location (<ML>) and the match type (<MT>). The match location is encoded with the phased binary code and represents the location of the matched directory entry. The match type denotes the Huffman code for the full match. Note that the matched directory entry is moved to the top of the dictionary.

A partial hit occurs when at least any two characters of the incoming tuple match with a dictionary entry, as depicted in Fig. 4(b). In this case, the match type encodes (using the Huffman code) which characters from the incoming tuple matched a dictionary entry. Any unmatched characters from the incoming tuple are then sent literally ('Z' in Fig. 4(b)). Otherwise, a miss occurs and a single bit of '1' is transmitted followed by the tuple itself as illustrated in Fig. 4(c). For a partial hit or a miss, the incoming tuple is inserted at the top of the dictionary.

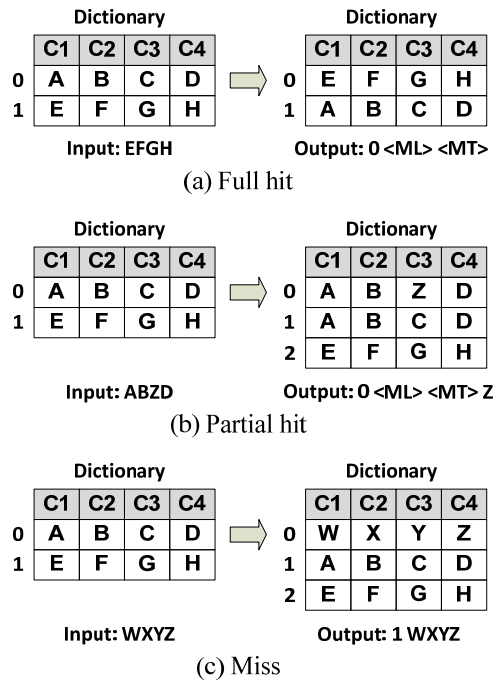

Fig. 4. Example cases of the X-Match algorithm. <ML> represents the match location (i.e., the location of the matching entry in the dictionary) encoded with the phased binary code. <MT> indicates the Huffman code for the match type.

B. Incompressible Data Predictor (IDP)

As described in the previous subsection, X-Match repeatedly matches a 4-byte tuple against the dictionary entries. For a tuple to be compressed, at least two characters should be matched with any of the dictionary entries. Our IDP is based on the following observations. First, since the dictionary is gradually filled with the incoming tuples, X-Match works better if there are many overlapped characters between tuples. Second, only the overlap of characters in the same byte position in a tuple matters. For example, although two tuples "AACC" and "CCAA" have many characters in common, it is not helpful for X-Match as they have different characters in each byte position.

The basic idea behind the proposed IDP is to count the number of distinct characters in each byte position for the incoming tuples, and then use this count to predict whether the data will be compressed or not. Fig. 5 presents the case when the number of tuples is eight. Each tuple is arranged vertically, and we count the number of unique characters in each column, C1, C2, C3, and C4. If this count is small, it means that many characters are overlapped in the particular column. The large count indicates that there are many unique characters in that column, lowering the possibility of full hits or partial hits.

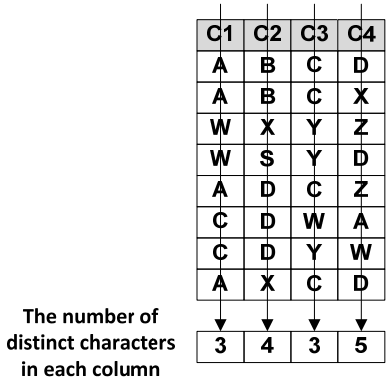

Fig. 5. The basic idea of predicting incompressible data. The number of unique characters in each column can be used to predict whether the data are incompressible or not.

To confirm this idea, we have conducted an experiment with real data. Fig. 6 illustrates the cumulative distribution of the number of unique characters in the third column for the data produced while office productivity software is installed. As we use the compression unit size of 4KB, there are 1,024 tuples per compression unit to be processed by the X-Match algorithm. Since each byte can have the value between 0 and 255, the count of each column will have the value between 1 and 256. Out of the total 243,897 compression units generated during this experiment, 137,474 units (56.4%) were incompressible, i.e., the size of the compressed data plus 6 bytes (the metadata size for each chunk) exceeded the size of the original data (4KB). Fig. 6 shows that 99.7% of these incompressible units have the number of unique characters greater than 239 characters in the third column. On the other hand, the number of unique characters is distributed over a much wider range for compressible units.

Fig. 6 suggests that the number of unique characters can be an effective means to predict whether a certain compression unit is compressible or not. For example, we may use a

prediction policy such that a compression unit is incompressible if it has more than 239 unique characters in the third column. If we use this policy, only 0.4% of incompressible units are mispredicted as compressible and 4.5% of compressible units are mispredicted as incompressible, according to Fig. 6. Although we count the number of unique characters only for the third column, the results for the other columns are similar.

This idea can be extended further to minimize overhead. Instead of looking at all the tuples to count the number of distinct characters in each column, we found that the prediction using only a subset of tuples works quite well. Specifically, the proposed Incompressible Data Predictor (IDP) only counts the number of distinct characters in the third column for the first 32 tuples and predicts that a compression unit is incompressible if the count is greater than 25 characters. When we use this prediction policy, 99.4% of incompressible units and 86.2% of compressible units are predicted correctly for the same workload shown in Fig. 6. Although the misprediction rate is slightly higher, our prediction policy has the benefit that it can make a decision whether the compression for the current data should be continued or stopped after looking at just 32 tuples out of the total 1,024 tuples.

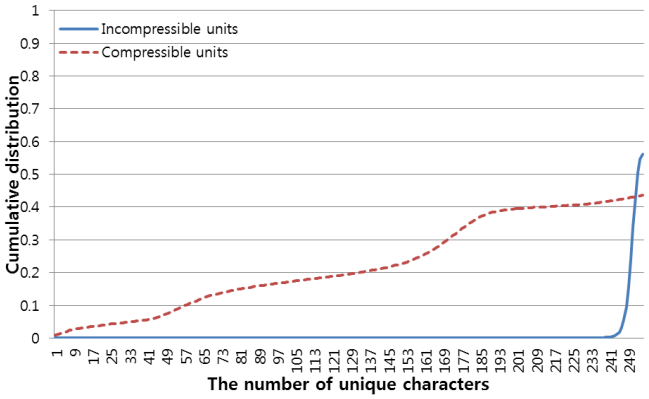

Fig. 6. The cumulative distribution of the number of unique characters in the third column for incompressible/compressible units. Each unit is 4KB in size and the data are collected while installing office productivity software.

C. Hardware Implementation of IDP

Because the prediction for incompressible data should be performed as fast as the hardware compressor, the IDP also needs to be implemented in hardware. The IDP hardware is placed within the compressor and synchronizes its clock cycle with the compressor to minimize the prediction delay. The IDP hardware has 256 1-bit registers as shown in Fig. 7. The prediction hardware takes the third byte of the tuple read by the X-Match compressor for each cycle, and sets the corresponding register to the value of '1'. The number of registers which have the value of '1' represents the number of unique characters. To eliminate the delay for counting the registers whose values are '1', the original value of each register is inverted and then added to the counter before the value is updated in the selected register. After the first 32 cycles, the prediction hardware compares the counter value with the threshold configured beforehand (25 by default). If the counter value is greater than the threshold, the prediction hardware sends the stop signal to the controller of the X-Match compressor.

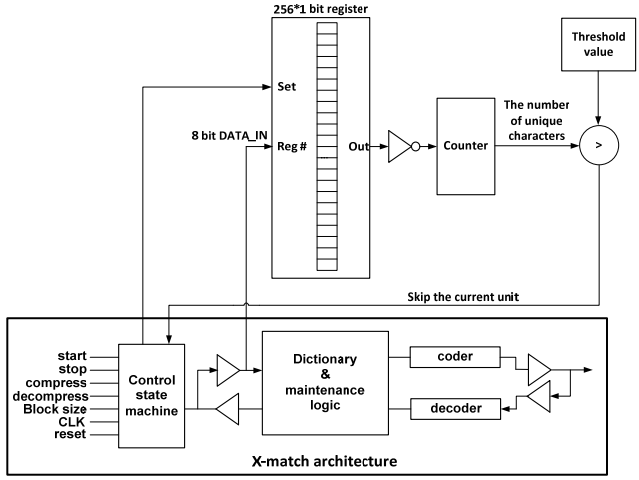

Fig. 7. The hardware implementation of the Incompressible Data Predictor (IDP). The third byte of each tuple is used to set the value of the corresponding register to '1'. When the value is transitioned from '0' to '1', the counter is incremented. If the counter value is greater than the threshold value after 32 cycles, the current unit is predicted incompressible and the compressor is stopped.

We have designed and implemented the prediction hardware using an FPGA. The cost of the IDP hardware is about 482 LUTs which is approximately 2K in ASIC gates. It is very small compared with the X-Match compressor/decompressor which is known to cost 110K gates [6]. The prediction takes 165 ns for a single 4KB unit according to the simulation result. Therefore, the performance degradation due to the prediction is almost negligible.

V. Evaluation

A. Experimental Setup

zFTL is implemented as one of block device drivers in an open-source operating system. The compression support can be turned off anytime using a special kernel interface. Instead of using bare NAND flash chips, we use the generic kernel subsystem which emulates the behavior and timing of various memory devices including NAND flash chips. We configured the parameters of the subsystem to model a 2GB MLC NAND flash memory chip where the page size is 4KB and each erase block has 128 pages. The latency of read, write, and erase operation is assumed to be 60 μs, 800 μs, and 1.5 ms, respectively, according to the data sheet of a representative MLC NAND flash memory chip.

Table 1 shows the basic information of five workloads used in this paper. UNTAR and COMPILE are the real workloads executed on the evaluation platform, which untar and compile the source code of a version of open-source operating system,

respectively. TEMP denotes the set of files downloaded from the Internet while a web browser visits social network sites, e-commerce sites, video sharing sites, and Internet portal sites. We periodically collect the files in the browser's temporary directory and then copied them onto zFTL.

SYSTEM and INSTALL workloads are mainly used to investigate the compression ratios of the files used in a commercial operating system. The SYSTEM workload is obtained by copying system files to zFTL which are frequently used by the commercial operating system. The INSTALL workload represents the storage access requests generated while office productivity software is installed in the commercial operating system. The installation process has been mimicked by extracting file system access traces with a profiling tool and replaying them on the evaluation platform.

TABLE I
WORKLOADS USED IN THIS PAPER

Workload	Write Requests	Read Requests	Sectors Written	Sectors Read
UNTAR	35,720	16,184	967,232	129,472
COMPILE	63,976	33,216	937,232	265,728
TEMP	63,064	8	1,864,112	64
SYSTEM	77,384	9	2,319,160	72
INSTALL	65,704	8,281	1,948,008	66,248

To model an aged file system, we initialize zFTL by running Postmark 1.51 [21] before each experiment. Postmark is configured with 25K files, 50K transactions, and file sizes ranging from 30KB to 80KB. The total amount of data written by Postmark is about 3GB. During this preconditioning phase, we turn off the compression support in zFTL.

The performance of X-Match with the proposed Incompressible Data Predictor is compared to those of Zlib [9], LZ77 [16], and the original X-Match [6]. Zlib and LZ77 are very well-known compression algorithms for their performance and reliability. Zlib is a representative software library used for data compression. Although it is expensive to implement the Zlib algorithm in hardware, we incorporate it into our evaluation as it shows the best compression ratios for the workloads shown in Table I. Overall, LZ77 exhibits slightly worse compression ratio than Zlib, but efficient hardware implementations of LZ77 or variants have been proposed in several previous studies. In fact, the X-Match algorithm is also a variant of LZ77.

B. Average Compression Ratio

Fig. 8 shows the average compression ratios for each workload with Zlib, LZ77, X-Match, and X-Match with IDP. The compression ratio is defined as the ratio of the compressed chunk size to the original (uncompressed) data size (4KB). Hence, the lower the compression ratio, the better. The average compression ratio varies from workload to workload, but Zlib always results in the best compression ratio. In particular, workloads which manipulate text-based files such as UNTAR and COMPILE exhibit fairly good compression ratios as low as 27% with Zlib. Because X-Match compresses the data in four bytes unit, the compression ratios of X-Match is not as good as those of Zlib or LZ77 in these text-based workloads.

On the other hand, TEMP shows much worse compression ratio since most files are image files and movie clips which have been already compressed. We find that system files touched in the SYSTEM workload also reveal good compression ratios. The compression ratio of INSTALL is higher than that of SYSTEM by 21% (Zlib) or by 24% (LZ77). This is because INSTALL handles many files in a special file format that stores a library of compressed files. X-Match with IDP presents almost the same compression ratio as the original X-Match. The difference between X-Match and X-Match with IDP comes from a small number of mispredictions in X-Match with IDP, but the difference is hardly noticeable in many cases.

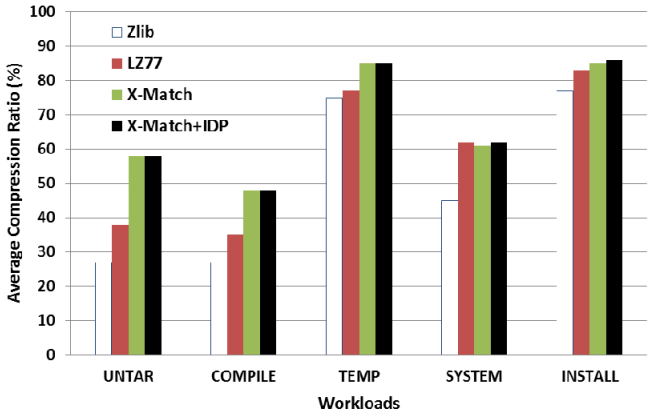

Fig. 8. Average compression ratio. The compression ratio depends on the contents of the data and the compression algorithms. X-Match with IDP shows slightly worse compression ratio than the original X-Match due to mispredictions, but the difference is hardly noticeable.

C. Write Amplification Factor (WAF)

Fig. 9 compares the Write Amplification Factor (WAF) before and after the compression support is enabled. The WAF breaks down according to the source of writes; it is either for the actual data writes or for the writes issued during garbage collection. The upper bar indicates the amount of additional writes caused by garbage collection, which is as high as 4.19 (in COMPILE) when the compression is not enabled.

UNTAR and COMPILE show very low WAFs under zFTL due to their low compression ratios. Since the amount of data written into NAND flash memory is reduced effectively, garbage collection hardly occurs. As a result, their WAFs are improved by a factor of 11.3 (UNTAR) and 15.0 (COMPILE) with the Zlib algorithm. The WAFs for TEMP, SYSTEM, and INSTALL are also improved by a factor of 3.5, 5.5, and 2.6, respectively, with Zlib. LZ77 and X-Match perform slightly worse than Zlib, resulting in improvements in WAFs by a factor of 2.1 (INSTALL) to 10.8 (COMPILE) with LZ77, and by

a factor of 1.9 (INSTALL) to 5.0 (COMPILE) with X-Match. X-Match with IDP reduces the WAFs for UNTAR, COMPILE, TEMP, SYSTEM, and INSTALL workloads by a factor of 2.1, 4.7, 2.0, 3.1, and 1.8, respectively, with the geometric mean of 2.6. Compared to X-Match, X-Match with IDP increases WAFs by 5.0% on average due to mispredicted data.

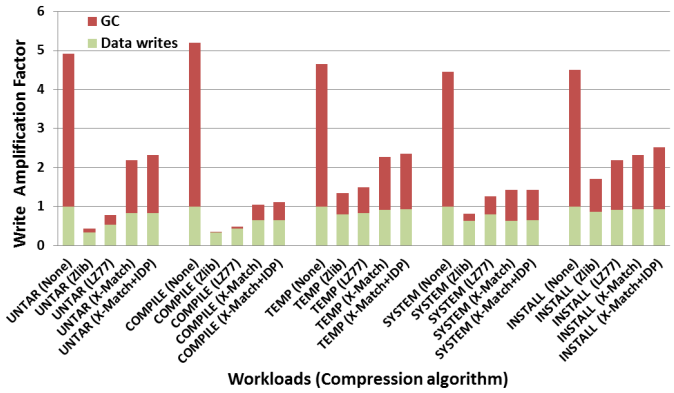

Fig. 9. Write amplification factor (WAF). The leftmost bar in each workload shows the WAF without compression support. X-Match with IDP improves WAFs by a factor of 2.6 (geometric mean).

D. Garbage Collection Overhead

Fig. 10 illustrates the total time spent on garbage collection. It is estimated by multiplying the number of flash read, write, and erase operations during garbage collection by the respective operational latencies of MLC NAND flash memory. The final results are normalized to the values obtained when the compression support is disabled.

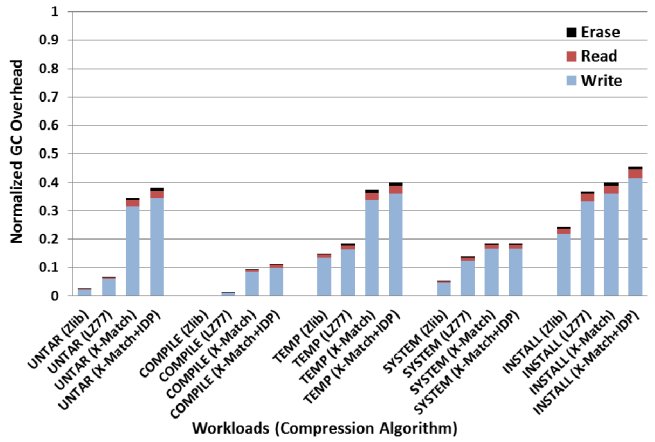

Fig. 10. Normalized garbage collection overhead. The final results are normalized to the values obtained when the compression support is disabled. We can see that the use of compression effectively reduces the time spent on garbage collection.

In UNTAR and COMPILE workloads, the garbage collection overhead is almost negligible for Zlib and LZ77 because of good compression ratios. X-Match with IDP shows the largest overhead, but it is still much better than the case without any compression. We observe that the overall trend of Fig. 10 is highly correlated to that of Fig. 8.

E. Power Consumption

Fig. 11 shows how much power is saved by using the proposed Incompressible Data Predictor (IDP) with the X-Match algorithm with respect to the original X-Match algorithm. In the original X-Match algorithm, all tuples in each 4KB compression unit should go through the compressor engine for 1,024 cycles. Thus, the power consumption, P_{org}, required to process the total N compression units by the original X-Match algorithm can be given by

$$P_{org} = N \cdot P_{comp} \cdot 1024 \quad (1)$$

where P_{comp} represents the unit power consumed by the compressor hardware per cycle. Under the X-Match algorithm with the proposed IDP, the data predicted incompressible stop using the compressor engine after 32 cycles. Therefore, the power consumption of the proposed approach can be approximated as follows:

$$P_{IDP} = P_{comp} \cdot (N_{pi} \cdot 32 + N_{pc} \cdot 1024) + N \cdot P_{pred} \cdot 32 . \quad (2)$$

In (2), N_{pi} and N_{pc} denote the number of compression units that are predicted incompressible and compressible, respectively, where $N = N_{pi} + N_{pc}$. P_{pred} indicates the unit power spent by the prediction hardware per cycle. We estimated that P_{pred} is one-fiftieth of P_{comp}, assuming that the power consumption is roughly proportional to the number of logic gates required to implement the hardware (cf. section IV.C).

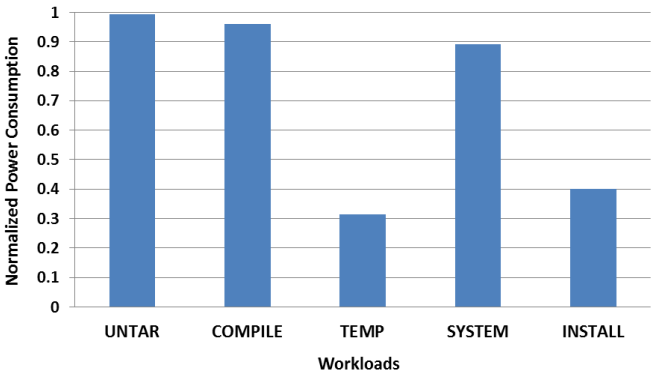

Fig. 11. Estimated power consumption of X-Match with the proposed IDP. The values are normalized to the estimated power consumption in the original X-Match algorithm. In UNTAR, COMPILE, and SYSTEM, there are no significant benefit as most of the data are compressible in these workloads. However, INSTALL and TEMP show power savings by 60% and by 69%, respectively.

As can be seen in (2), the power saving due to the proposed IDP greatly depends on N_{pi}, the number of compression units that are predicted incompressible. One extreme workload is UNTAR, where only 0.02% of the input data are predicted incompressible. In this case, virtually no power saving has been achieved as shown in Fig. 11. In COMPILE and SYSTEM, 3.2% and 5.6% of the data are predicted incompressible, respectively, resulting in 3.9% (COMPILE) and 10.8% (SYSTEM) of power savings. When there are modest number of incompressible data as in TEMP and INSTALL, the use of

IDP achieves 68.6% and 60.0% of power savings, respectively. In TEMP and INSTALL, 63.0% and 56.1% of the data are predicted incompressible. With our evaluation with another extreme workload which is composed of 99% of incompressible data, we observe that the proposed approach saves power consumption by 97%.

V. Conclusion

Due to inherent characteristics of NAND flash memory which does not allow in-place update and wears out after repeated write/erase cycles, flash translation layers have been using a variety of techniques to enhance the overall performance and lifetime of NAND flash-based consumer electronics devices. Many previous researches on flash translation layers have focused on efficient address mapping and garbage collection schemes. However, another orthogonal issue that can reduce the amount of data written into NAND flash memory is to support data compression inside the flash translation layer.

In this paper, we present zFTL, a flash translation layer which supports on-line, transparent data compression based on the X-Match algorithm. We have examined several design issues to support data compression in the flash translation layer, including some required extensions in address mapping and garbage collection. To reduce the compression overhead and power consumption associated with incompressible data, we have also proposed a novel scheme called Incompressible Data Predictor (IDP) that can predict whether the input data are incompressible or not by examining only a subset of data.

Through the use of five real-world workloads, we confirm that zFTL improves the WAF by a factor of 2.6 (geometric mean) compared to the case without compression support. The proposed IDP is effective in reducing power consumption especially when there are many incompressible units among input data. When 63.0% of the data are predicted incompressible, zFTL reduces power consumption by 68.4% compared to the original X-Match algorithm without any prediction scheme.

References

[1] Intel Corporation, "Understanding the flash translation layer (FTL) specification," *Application Note AP-684*, Dec. 1998.

[2] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," *In Proc. of the USENIX Winter Technical Conference*, pp. 155–164, 1995.

[3] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp.366-375, 2002.

[4] X.-Y. Hu, E. Eleftheriou, R. Haas, I Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," *In Proc. Of the Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[5] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," *In Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 229–240, 2009.

[6] M. Kjelso, M. Gooch, and S. Jones, "Design and performance of a main memory hardware data compressor," *In Proc. of the 22nd EUROMICRO Conference*, pp.2-5, Sep. 1996.

[7] D. Woodhouse, "JFFS: the journaling flash file system," *In Proc. of the Ottawa Linux Symposium (OLS)*, 2001.

[8] M. Rosenblum, and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[9] J.-L. Gailly and M. Adler, *Zlib general purpose compression library (version 1.2.5)*, Apr. 2010.

[10] N. Goyal and R. Mahapatra, "Energy characterization of CRAMFS for embedded systems," *In Proc. of the International Workshop on Software Support for Portable Storage (IWSSPS)*, 2005.

[11] A. I. Pavlov and M. Cecchetti, *SquashFS HOWTO (revision 1.9)*, Jul. 2008.

[12] S. Hyun, H. Bahn, and K. Koh, "LeCramFS: an efficient compressed file system for flash-based portable consumer devices," *IEEE Transactions on Consumer Electronics*, vol. 53, no. 2, pp. 481–488, May 2007.

[13] K. S. Yim, H. Bahn, and K. Koh, "A flash compression layer for SmartMedia card systems," *IEEE Transactions on Consumer Electronics*, vol. 50, no.1, pp. 192–197, Feb. 2004.

[14] C. H. Chen, C. T. Chen, and W. T. Huang, "The real-time compression layer for flash memory in mobile multimedia devices," *In Proc. of the International Conference on Multimedia and Ubiquitous Engineering,* pp. 171–176, Apr. 2007.

[15] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM memory expansion technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp.271–285, Mar. 2001.

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

[17] P. A. Franaszek, J. Robinson, and J. Thomas, "Parallel compression with cooperative dictionary construction*," In Proc. of the Data Compression Conference,* pp. 200–209, 1996.

[18] L. Benini, D. Bruni, A. Macii, and E. Macii, "Hardware-assisted data compression for energy minimization in systems with embedded processors," *In Proc. of Design, Automation and Test in Europe (DATE)*, pp. 449–453, 2002.

[19] C. D. Benveniste, P. A. Franaszek, and J. T. Robinson, "Cache-memory interfaces in compressed memory systems," *IEEE Transactions on computers,* vol. 50, no. 11, pp. 1106–1116, Nov. 2001

[20] M. Burrows, C. Jerian, B. Lampson, and T. Mann, "On-line data compression in a log-structured file system," *In Proc. of the Architectural Support for Programming Languages and Operating System (ASPLOS)*, pp. 2–9, 1992.

[21] J. Katcher, "PostMark: a new filesystem benchmark," *Technical Report TR3022, Network Appliance,* 1997.

BIOGRAPHIES

Youngjo Park received his BS degree in computer engineering from Kookmin University, Korea, in 2009, and the MS degree in embedded software from Sungkyunkwan University (SKKU), Korea, in 2011. He is currently an assistant engineer in Samsung Electronics Co. His research interests include NAND flash memory, storage systems, and embedded systems.

Jin-Soo Kim (M'89) received the BS, MS, and PhD degrees in computer engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He is currently an associate professor in Sungkyunkwan University (SKKU). Before joining SKKU, he was an associate professor at Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of research staff, and with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems. He is a member of the IEEE and the IEEE Computer Society.