

# Compression Support for Flash Translation Layer

Youngjo Park and Jin-Soo Kim

Department of Embedded Software  
Sungkyunkwan University (SKKU), Suwon 440-746, South Korea  
{ thepark, jinsookim } @ skku.edu

## ABSTRACT

NAND flash memory has many advantageous features as a storage medium, such as superior performance, shock resistance, and low-power consumption. However, the erase-before-write nature and the limited number of write/erase cycles are obstacles to the promising future of NAND flash memory. An intermediate software layer called Flash Translation Layer (FTL) is used to overcome these obstacles.

In this paper, we present a flash translation layer called zFTL which has data compression support to reduce the amount of data written to flash memory. zFTL is based on page-level mapping, but it is extended to support on-line, transparent data compression and decompression. We implement zFTL on the MTD layer of the Linux kernel. Our experimental results with five workloads from Linux and Windows show that zFTL improves the write amplification factor by 27% ~ 92%.

## Categories and Subject Descriptors

### General Terms

Design, Reliability, Performance, Experimentation

### Keywords

Flash Memory, Flash Translation Layer (FTL), Storage Systems, Data Compression, Embedded System

## 1. INTRODUCTION

Recently, NAND flash memory has become a necessity not only in mobile devices but also in high-end laptops and desktop systems, thanks to its superior performance, shock resistance, and low-power consumption. With technology advancing, NAND flash memory's capacity is getting larger and its price is getting lower.

However, NAND flash memory has several limitations: 1) The previous data should be erased before a new data can be written in the same place. This is usually called *erase-before-write* characteristic. 2) Normal read/write operations are performed on a *per-page* basis, while 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. 3) Flash memory has limited lifetime; MLC NAND flash memory wears out after 5K to 10K write/erase cycles.

The aforementioned limitations are effectively hidden through the use of an intermediate software layer called Flash Translation Layer (FTL). Most FTLs employ *address remapping*, which writes an incoming data into one or more pre-erased pages and maintains the mapping information between the logical sector number and the physical page number. As the new data is 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 [1, 2].

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)*. 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 [3]. Basically, the WAF of hard disks is 1.0. The WAF can be as high as 10 on low-end SSDs, while Intel claims that its X25-M SSD keeps the WAF down to 1.1 [4].

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. We consider two compression algorithms, namely Zlib [6] and LZ77 [7].

We implement zFTL on the MTD layer of the Linux kernel 2.6.32.4 and evaluate it on NANDSim [8], which emulates the behavior and timing of NAND flash memory with RAM. zFTL is evaluated with five realistic workloads from Linux and Windows

---

This work was supported by Mid-career Researcher Program through NRF grant funded by the MEST (No. 2010-0000114).

XP. Our results show that the use of data compression improves the WAF up to 92%.

The rest of the paper is organized as follows. The next section discusses the related work. Section 3 introduces the overall architecture and design issues of zFTL. Section 4 presents the experimental results and Section 5 concludes the paper.

## 2. RELATED WORK

Data compression techniques have been studied in various layers in computer systems. JFFS2 [9] is a representative flash-aware file system inspired by the log-structured file system [10]. JFFS2 provides an option to use zlib-based data compression. CramFS [11] and SquashFS [12] are compressed read-only file systems, mainly targeting the root file system in small embedded systems. Hyun et al. [13] 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. [14] 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 [14]. Chen et al. [15] proposed another internal packing scheme called IPS real-time. The IPS real-time scheme splits the compressed data and stores them into two 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 the 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 flash memory chips. An SSD controller from SandForce Inc. is known to use real-time data compression and data deduplication to lower the WAF as low as 0.5 [4]. However, its internal architecture has not been published in detail.

Special hardware compressor/decompressor engines have been proposed in several literatures. IBM's Memory Expansion Technology (MXT) [16] 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 [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. [19] 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 [20]. For brevity, we assume data compression/decompression is assisted by special hardware that is fast enough to hide its overhead.

## 3. zFTL

### 3.1 System Architecture

Figure 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 (512 bytes). For write requests, zFTL aggregates the requested data in the write temporary buffer, whose size is equal to the compression unit size. If the write temporary buffer is full, the data in the buffer is compressed and then appended into the flash write buffer. The size of the flash write buffer is a multiple of the flash page size (4KB in MLC NAND). A single flash page size may hold a number of compression units depending on the compression ratio.

When the flash write buffer has not enough space for the incoming compressed data, the write buffer 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 is available in any of buffers, it is 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 detail in Section 3.5.

For reads, zFTL first searches the requested data in the write temporary buffer as it has, if any, the most recent version of the data. When the search fails, zFTL looks up the data in the flash write buffer. If the data is found in the flash write buffer, the compressed data is decompressed and then loaded into the read temporary buffer. When the data is still not found in the flash write buffer, zFTL examines the read temporary buffer and the flash read buffer. Note that the two write buffers should be looked up before the two read buffers, as they may keep the up-to-date data. While the requested data is 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 data after reading the corresponding page from flash memory. The detailed read/write flows in zFTL are depicted in Figure 2.

### 3.2 Compression Unit

The unit of data compression is an important factor affecting the compression ratio and speed. In particular, dictionary-based algorithms such as LZ77 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. In Linux, 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 is 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.

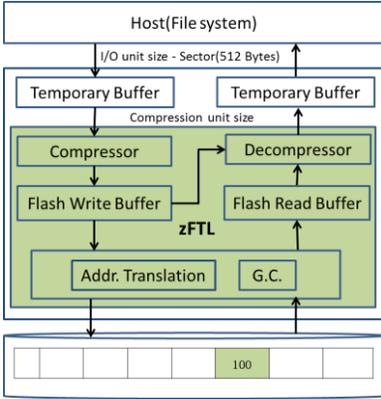


Figure 1. System architecture

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 is partly read or updated. Moreover, enlarging the compression unit size has a diminishing return in the compression ratio. Burrows et al. [21] and Yim et al. [14] 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 (simply called a *block*, for short). Since most file systems in Linux use 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.

### 3.3 Compression Algorithms

The choice of compression algorithms is also one of the important design issues, because it determines the speed of compression/decompression, the compression ratio, and the complexity of hardware implementation. However, the efficiency of various compression algorithms is beyond the scope of this paper. We currently implement zFTL with Zlib [6] and LZ77 [7] algorithms. Both algorithms are very well known for their performance and reliability. Especially, efficient hardware implementations of LZ77 or variants have been proposed in several previous studies.

### 3.4 Address Mapping

zFTL is based on page-level mapping 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 block 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

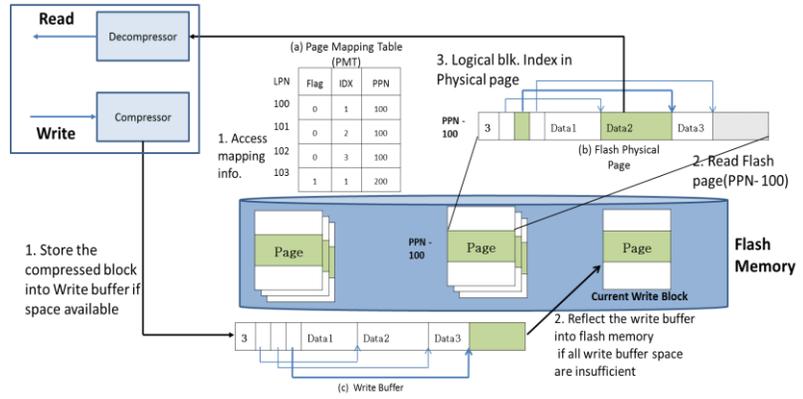


Figure 2. Read/Write flows in zFTL

single flash page may accommodate compressed blocks from several logical pages in zFTL, IDX is used to represent the relative position of each logical page within the physical page. Figure 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 100 in the order indicated by the IDX value.

Some data are inherently incompressible as they come from multimedia files or compressed files. It is pointless to compress these data again inside zFTL as it will not save any space. zFTL identifies these incompressible blocks based on the resulting size after compression. If the size of a block after compression is not small enough, the block is stored in flash memory as is, setting the corresponding FLAG to 1 (cf. the PMT entry of the logical page number 103 in Figure 2).

Depending on FLAG, the physical flash page has two different structures. For incompressible blocks (FLAG = 1), the entire page is devoted to the (uncompressed) original data block. 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 compressed blocks in the page, a set of offsets for each compressed block, and a set of compressed blocks, as depicted in Figure 2. The offset indicates the last byte position of the corresponding compressed block in the page.

### 3.5 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 compressed data stored in the

same physical page, in order to copy only the valid data 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 logical pages and the bitmap for each logical page stored in the given physical page number. The bitmap indicates whether the corresponding logical page is valid or not.

| PPN 100 | # of valid pages |   |   | Bitmap for valid pages |   |   |   |   |
|---------|------------------|---|---|------------------------|---|---|---|---|
|         | 0                | 1 | 0 | 0                      | 1 | 1 | 0 | 0 |
|         | 0                | 1 | 0 | 0                      | 1 | 1 | 0 | 0 |

Figure 3. An example PST (Page Status Table) entry

Figure 3 shows an example 8-bit PST entry designed for 4KB physical pages. Figure 3 represents that two logical pages (the second and the third one) are currently valid in the physical page number 100. Under this PST structure, up to five logical pages can be packed into a 4KB physical page. Our experiments show that about three compressed logical pages are stored in a single 4KB flash page on average for the most 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.

### 3.6 Internal Fragmentation

The flash page size is fixed whereas the resulting data block size varies after compression. Unless we allow the compressed block 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.

### 3.7 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 64~128, but the increasing number of SSDs are 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.

## 4. EVALUATION

### 4.1 Experimental Setup

We evaluate the performance of zFTL on an x86-based Linux system equipped with Intel Core2Duo E8400 3.0GHz CPU, 4GB DDR2 DRAM, and 64-bit Ubuntu 9.04 distribution. zFTL is implemented as one of block devices in the MTD (Memory Technology Devices) layer of the Linux kernel 2.6.32.4, on top of which the ext4 file system is mounted. The compression support can be turned off anytime using the /proc interface. Instead of bare NAND flash chips, we use the MTD NANDSim module which emulates the behavior and timing of NAND flash memory with the host RAM. We configure the parameters of NANDSim to model a 2GB MLC NAND flash memory where the page size is 4KB and each erase block has 128 pages. The latency of read, write, and erase operation is set to 60usec, 800usec, and 1.5ms, respectively [22].

Table 1. Workloads used in this paper

| Workload | Write Requests | Read Requests | Sectors Written | Sectors Read |
|----------|----------------|---------------|-----------------|--------------|
| UNTAR    | 116,922        | 15,722        | 935,376         | 125,776      |
| COMPILE  | 112,775        | 32,739        | 902,200         | 261,912      |
| TEMP     | 145,388        | 7             | 1,163,104       | 56           |
| WINDOWS  | 45,788         | 4             | 366,304         | 32           |
| OFFICE   | 243,503        | 8,284         | 1,948,024       | 66,272       |

Table 1 shows the basic information of five workloads used in this paper. UNTAR and COMPILE are the real workloads executed on ext4/zFTL, which untar and compile the source code of the Linux kernel 2.6.32.4, respectively. TEMP denotes the set of files downloaded from the Internet while the Firefox web browser visits such sites as Facebook, E-bay, Amazon, Yahoo, Youtube, etc. We periodically collect the files in the browser's temporary directory and then copied them onto zFTL.

WINDOWS and OFFICE workloads are mainly used to investigate the compression ratios of the files used in Windows XP. Since zFTL is implemented only on the Linux platform, we first extract file system access traces using the ProcessMonitor tool [23] while Excel, Word, and Powerpoint from Microsoft Office 2007 are installed on Windows XP. Then, the installation process has been mimicked on Linux using the following steps: 1) The trace is postprocessed to identify the existing Windows files which are touched during the installation. Those files are copied to zFTL (WINDOWS). 2) File access traces obtained from Windows are replayed (OFFICE).

To model the aged file system, we initialize zFTL by running Postmark 1.51 [24] 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.

### 4.2 Average Compression Ratio

Figure 4 shows the average compression ratios for each workload with Zlib and LZ77 algorithms. The compression ratio is defined as the ratio of the compressed block size to the

uncompressed data block size (4KB)<sup>1</sup>. The compression ratio varies from workload to workload, but Zlib shows slightly better compression ratios than LZ77 in general. Workloads which manipulate text-based files such as UNTAR and COMPILE exhibit fairly good compression ratios as low as 27% with Zlib. On the other hand, TEMP shows the worst compression ratio since most files are JPEG files and movie clips which have been already compressed. We found that Windows files touched during the installation of Microsoft Office also reveal good compression ratios. The compression ratio of OFFICE is higher than that of WINDOWS by 29% (Zlib) or 24% (LZ77). This is because OFFICE handles many CAB files which are in the Microsoft compressed archive format.



Figure 4. Average compression ratio

### 4.3 Write Amplification Factor (WAF)

Figure 5 compares the 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 3.29 (in COMPILE) when the compression is not enabled.

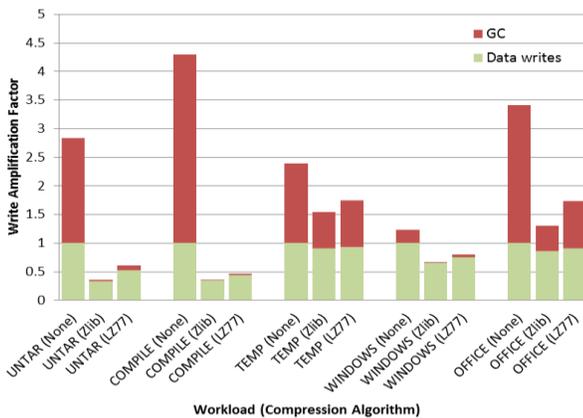


Figure 5. Write amplification factor (4KB page size)

<sup>1</sup> The compressed block size includes the housekeeping information such as the number of compressed blocks and the offset of each compressed block, but does not include the wasted space in a page due to the internal fragmentation.

UNTAR and COMPILE show very low WAFs under zFTL due to their low compression ratios. Since the amount of data written into NAND flash is reduced effectively, garbage collection hardly occurs. As a result, their WAFs are improved by 88% (UNTAR) and 92% (COMPILE) with the Zlib algorithm. The WAFs for TEMP, WINDOWS and OFFICE are also improved by 36%, 46% and 62%, respectively, with Zlib. As expected, TEMP shows the least performance improvement. The LZ77 algorithm performs slightly worse than Zlib, resulting in improvements in WAFs by 27% (TEMP) ~ 89% (COMPILE).

### 4.4 Garbage Collection Overhead

Figure 6 presents the total time spent for garbage collection. It is estimated by multiplying the number of flash read, write, and erase operations during garbage collection by the respective operational latency of MLC NAND flash memory. The final results are normalized to the values obtained when the compression support is disabled.

In UNTAR and COMPILE workloads, the garbage collection overhead is almost negligible because of good compression ratios. TEMP has the largest overhead, but it is still better than the case without any compression. We observe that the overall trend of Figure 6 is highly correlated to that of Figure 4.

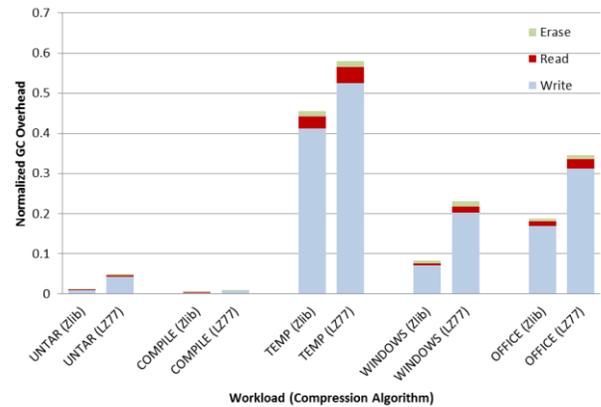


Figure 6. Normalized G.C. overhead with 4KB page size

### 4.5 Internal Fragmentation

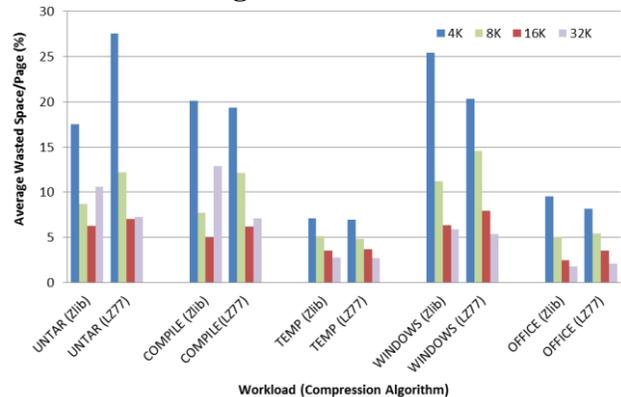


Figure 7. Average percentage of wasted space in a page

Figure 7 illustrates the average percentage of wasted space in a flash page. The amount of wasted space is measured when the contents of the flash write buffer is flushed into NAND flash memory. The overall tendency is that the percentage of wasted space gets smaller as the page size is increased from 4KB to 32KB. The sudden increases in UNTAR and COMPILE for the 32KB page size are due to the limit in the bitmap size of the PST entry. When the page size is 32KB, we expand the bitmap size to 32 bits so that each page can hold up to 32 compressed blocks. However, this was not sufficient for UNTAR and COMPILE.

One way to improve space utilization is to use multiple flash write buffers. With more than one flash write buffer, the incoming compressed block is more likely to find a flash write buffer that can accommodate its data. However, according to our experiments, this scheme reduces the wasted space only by up to 7% with the 4KB page size. The benefit gets even smaller as the page size becomes larger. Moreover, the use of multiple flash write buffers may harm the sequential read bandwidth since the sequentially-written data can reside in different flash pages.

## 5. CONCLUSION AND FUTURE WORK

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 endurance. 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. We have examined several design issues to support data compression in flash translation layer, including some required extensions in address mapping and garbage collection. We have implemented zFTL in the MTD layer of the Linux kernel. Through the use of real and emulated workloads, we confirm that zFTL improves the write amplification factor by up to 92%.

Our future work includes the analysis of hardware compressor/decompressor engine in terms of cost, performance and energy consumption. We also plan to evaluate zFTL with more diverse workloads.

## 6. REFERENCES

- [1] 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.
- [2] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification," *Application Note AP-684*, Dec, 1998.
- [3] Intel Corporation., <http://www.intel.com/cd/channel/reseller/asm-na/eng/products/nand/feature/index.htm>
- [4] AnandTech, <http://www.anandtech.com/show/2899/3>
- [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*, pp. 229–240, 2009.
- [6] J-L Gailly and M. Adler, "Zlib Compression Library," [www.zlib.net](http://www.zlib.net). Accessed Jan, 2010.
- [7] 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.
- [8] NAND simulator, <http://linux-mtd.infradead.org/faq/nand.html>.
- [9] D. Woodhouse, "JFFS:The Journaling Flash File System," *In Proc. of the Ottawa Linux Symposium(OLS)*, Red Hat, Inc., 2001.
- [10] M. Rosenblum, and J.K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10(1), pp. 26–52, Feb, 1992.
- [11] CramFS Documents, <http://lxr.linux.no/source/fs/cramfs/README>
- [12] SquashFS Homepage, <http://squashfs.sourceforge.net>
- [13] 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.
- [14] 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.
- [15] C.H. Chen, C.T. Chen, and W.T. Huang, "The real-time compression layer for flash memory in mobile multimedia devices," *International Conference on Multimedia and Ubiquitous Engineering*, pp. 171–176, Apr, 2007.
- [16] 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*, 45(2):271–285, Mar, 2001.
- [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, Mar/Apr, 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 Conference and Exhibition*, pp. 449–453, 2002.
- [19] 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.
- [20] 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
- [21] M. Burrows, C. Jerian, B. Lampson, and T. Mann, "On-line data compression in a log-structured file system," *In Proc. of Architectural Support for Programming Languages and Operating System*, pp. 2–9, 1992.
- [22] Samsung Elec., 2Gx8 Bit NAND Flash Memory, (K9GAG08U0M-P), 2006.
- [23] Windows Sysinternals, <http://technet.microsoft.com/en-us/sysinternals/default.aspx>
- [24] J. Katcher, "PostMark: a New Filesystem Benchmark," *Technical Report TR3022, Network Appliance*, 1997.