Improving Performance and Lifetime of Solid-State Drives Using Hardware-Accelerated Compression

Sungjin Lee, Jihoon Park, Kermin Fleming, Arvind, *Fellow*, IEEE, and Jihong Kim, *Member*, IEEE

Abstract — *The performance and lifetime of high-performance solid-state drives (SSDs) can be improved by data compression, which can reduce the amount of data physically transferred from/to flash memory. In this paper, we present our experience of building a high-performance solid-state drive using a hardware accelerated compression module called BlueZIP. In order to fully exploit the BlueZIP module, we devise a compression-aware flash translation layer (FTL), called CaFTL, which supports compression-aware address mapping and garbage collection for BlueZIP. For poorly compressed pages, CaFTL supports selective compression so that unnecessary compression can be avoided. We have implemented a complete SSD prototype with BlueZIP on an FPGA-based custom SSD platform and evaluated its effectiveness using realistic workloads. Our evaluation results show that BlueZIP can increase the lifetime of the SSD prototype by 26% as well as improve read and write performance by 20% and 27%, respectively, on average*[1].

Index Terms — Solid-State Drive, NAND Flash Memory, Flash Translation Layer, Data Compression.

I. INTRODUCTION

NAND flash-based solid-state drives (SSDs) have recently emerged as an attractive solution for consumer devices and desktop systems thanks to the continued scale-down of a NAND memory cell size combined with the use of multi-level cell (MLC) technology. As the density of flash memory cells increases, however, the performance and reliability of flash memory may deteriorate significantly [1]. For example, single-level cell (SLC) flash memory fabricated with the 34 nm process allows a flash block to have 100,000 program/erase (P/E) cycles, whereas MLC flash memory at the same 34 nm process supports only 5,000 P/E cycles per block. The performance of MLC flash memory is also several times slower than that of SLC flash memory. Moreover, as the semiconductor process is further scaled down, it is expected that these problems will be getting worse.

One of the promising approaches that can mitigate these problems is to use hardware accelerated compression. Since the lifetime of flash-based SSDs strongly depends on the amount of data written to the SSDs, data compression, which reduces the actual amount of data written to the SSDs, can be an effective solution to improve the lifetime of the SSDs. Furthermore, if compression can be supported by a hardware acceleration unit, it can also improve the performance of SSDs because a smaller amount of data are physically transferred during I/O operations over uncompressed reads and writes.

The idea of using data compression for data storage is not new and has been widely studied. For example, many existing file systems support software-based data compression to expand the effective capacity of a storage device. Although software-based compression approaches can be useful in improving the lifetime of SSDs, they incur a considerable compression/decompression overhead, thus the overall SSD performance deteriorates significantly. Therefore, software-based compression is usually employed when the storage capacity is one of the most important design goals.

An obvious solution for the problem of software-based compression is to use a special hardware accelerator. Although some SSD companies are believed to employ hardware-accelerated compression in their products, there is no known literature that describes such SSDs in detail. For example, recent investigations [2], [3], which discuss data compression in flash memory, only focus on software-side design and implementation issues. For the hardware design issues, the existing techniques either assume a magic compression hardware accelerator with no performance penalty or do not present their design in detail.

In this paper, we describe our experience of building a hardware compression module, called BlueZIP, which was designed for flash-based SSDs. BlueZIP is implemented using an FPGA-based SSD prototype called BlueSSD [4], thus allowing us to evaluate the pros and cons of hardware accelerated compression in real settings. Efficient software support is another crucial issue in realizing the potential benefit of data compression. To this end, we propose a compression-aware flash translation layer, called CaFTL, which provides support for compression-aware address mapping and garbage collection. CaFTL also supports

[1] This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 20110020426, No. R33-10095, and No. 2011-0020514). The ICT at Seoul National University provided research facilities for this study.

Sungjin Lee, Jihoon Park, and Jihong Kim are with the School of Computer Science and Engineering, Seoul National University, Gwanak-ro, Gwanak-gu, Seoul 151-742, Korea (e-mail:{chamdoo, promar2, jihong}@davinci.snu.ac.kr).

Kermin Fleming and Arvind are with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 32 Vassar Street, Cambridge, MA 02139, USA (e-mail:{kfleming, arvind}@csail.mit.edu).

Contributed Paper
Manuscript received 09/09/11
Current version published 12/27/11
Electronic version published 12/27/11.

selective compression to avoid useless data compression for the data which exhibit a low compression ratio. Our evaluation results using various benchmark programs show that BlueZIP lowers the amount of data written to flash memory by 26%, improving the lifetime of SSDs by a similar amount. Read and write speed of SSDs are also improved as well by 20% and 27% on average, respectively.

The rest of this paper is organized as follows. In Section II, we briefly review previous works related to data compression in NAND flash memory. In Section III, we explain the architecture of the proposed BlueZIP in detail, including its hardware architecture and software architecture. Experimental results are given in Section IV. Section V concludes with a summary and directions for future work.

II. Related Work

Several research groups have investigated using data compression for NAND flash memory. Yim *et al*. [2] proposed a flash compression layer to increase the effective storage capacity. They focused on resolving the internal fragmentation problem that occurs when the size of compressed data is smaller than that of a single flash page (which is a unit of read and write operations in NAND flash memory). They mitigated this fragmentation problem by introducing an internal packing scheme. Park *et al*. [3] proposed another flash translation layer, called zFTL, which employs data compression to improve the endurance of NAND flash memory. zFTL takes account of compressed data in both address mapping and garbage collection. Unlike our proposed work, these two techniques are limited in their contributions because they assumed that hardware compression/decompression modules incur no performance penalty for compressing and decompressing data. They also did not explore important design issues such as compression/decompression speed and the amount of the required hardware resource. Furthermore, they have not considered interactions between the compression software and hardware layers for more optimized designs.

There also have been several studies for hardware-accelerated compression, especially for the main memory system. For example, X-Match is one of the representative compression algorithms designed for main memory compression [5]. X-Match achieves a reasonable compression ratio for memory data, effectively increasing the memory capacity. X-Match, however, is not suitable to be used for secondary storage like solid-state drives because of its design limitations. For example, X-Match is optimized to compress small-size data such as several bytes of data (e.g., 4 bytes), which is the unit of data transfer between CPU and main memory. When X-Match is used for compressing large-size data such as 8 KB pages in SSDs, it performs poorly. Unlike X-Match, BlueZIP is designed to provide better compressibility for a large amount of streaming data whose size is several KBs and, therefore, more suitable for a secondary storage device.

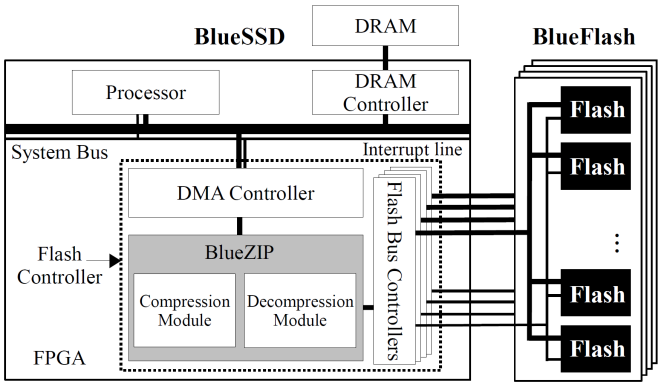

Fig. 1. An overall SSD architecture with BlueZIP.

III. BlueZIP

Fig. 1 shows an overall organization of our prototype SSD with BlueZIP. BlueZIP is implemented as one of the hardware modules within BlueSSD [4], which is an open SSD platform built on top of a general-purpose FPGA board combined with a custom flash board, called BlueFlash. BlueSSD uses the embedded processor included in the FPGA as a main SSD controller, so as to execute our proposed FTL and the Linux kernel. The flash controller is in charge of transferring the data from/to the BlueFlash board and is composed of two hardware modules: a DMA controller and a flash bus controller. The DMA controller receives commands from the processor and transfers data from/to DRAM through the system bus. The flash bus controller performs several flash operations, including read, write, and erase operations, and moves the data from/to flash chips in the BlueFlash board.

The BlueZIP module is implemented between the DMA controller and the flash bus controllers. The main role of BlueZIP is to perform compression or decompression for the data being transferred from the DMA controller or from the flash bus controller, respectively. BlueZIP uses the LZRW3 algorithm [6], a variant of the LZ77 algorithm, because it achieves a good compression ratio without high computational burden. The LZRW3 algorithm has been significantly modified in BlueZIP so that its hardware implementation becomes efficient.

A. Hardware Architecture of BlueZIP

In this subsection, we describe the hardware architecture of BlueZIP. We first explain the compression module of BlueZIP in detail and then briefly introduce data decompression steps. We also discuss the issues related to internal fragmentation that occurs when data compression is used in NAND flash memory [2] and then explain our approach to mitigate this fragmentation problem.

1) Compression Module

Fig. 2 shows an overall architecture of the compression module of BlueZIP, which is composed of four hardware submodules: a shift register, a dictionary, a compression logic, and a compression buffer. The shift register holds the data to be tested for compression and the dictionary table contains

repeated patterns previously seen. The compression logic converts the data in the shift register to symbols by referring to the dictionary. The compressed data, a sequence of symbols, are stored in the compression buffer and moved eventually to a flash chip.

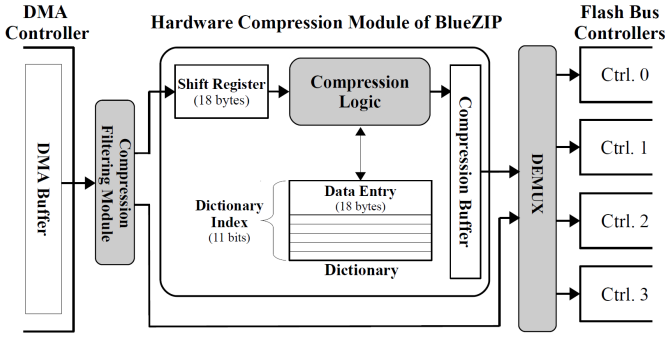

Fig. 2. An overall architecture of the compression module of BlueZIP.

BlueZIP fetches the data from the DMA buffer, which keeps the entire data sent from the host, until the shift register is fully filled. The compression logic creates a hash value using the first 3 bytes of the data in the shift register, which are used as a dictionary index for the dictionary table. The compression logic then checks the data entry where the dictionary index points. If the first 3 bytes of the corresponding data entry is equivalent to those of the shift register, we assume that we found a matching pattern from the dictionary. When the compression logic finds a matching data entry, it compares the remaining bytes in the shift register with those in the data entry and finds the common part of the data between the shift register and the data entry. This common part is called a data segment. The compression logic creates a symbol by combining the dictionary index and the length of the data segment, along with a compression flag whose value is '1'. The compression flag indicates if the symbol represents compressed data or uncompressed data. The symbol created is then written to the compression buffer. Finally, the whole data segment is discarded from the shift register, and the new data are transferred to the shift register from the DMA buffer.

When we do not find a matching pattern from the dictionary, we create a symbol only for the first byte of the data. A 9-bit symbol is created by adding one-bit compression flag (whose value is 0) to the first byte of the shift register. After the symbol created is written to the compression buffer, a new byte of the data from the DMA buffer is appended to the tail of the shift register, discarding the first byte of the shift register. Note that when a matching pattern is not available in the dictionary, the old pattern in the data entry to which the hash value points is replaced by the new pattern in the shift register for supporting newly found patterns.

In our compression scheme, a single byte of the data may be expanded into a 9-bit symbol (8 bits for the original data; 1 bit for a compression flag) if a single byte data cannot be compressed. If a compression ratio is low, the amount of the data actually written after compression can be bigger than the original data. In order to solve this problem, BlueZIP supports a selective compression function, which allows the FTL to determine whether the requested data should be stored in a compressed form or an uncompressed form. If the FTL decides to write data without compression, the compression filtering module of BlueZIP sends the requested data to the flash bus controller directly, bypassing the compression module.

The size of the shift register as well as the size of the dictionary has a significant effect on a compression ratio. As their sizes increase, a better compression ratio can be obtained at the cost of using more hardware resources. Currently, the shift register and the dictionary are set to 18 bytes and 36 KB (= 211×18 bytes), respectively. This setting requires a small amount of hardware resources, but provides a reasonable compression ratio. In Section IV, we will analyze the effect of the dictionary table size on a compression ratio in detail.

2) Decompression Module

Data decompression of BlueZIP is very similar to its data compression except that the processing steps are reversed. BlueZIP fetches the data from the flash bus controller and decides if the data are compressed or not by checking a compression flag. If the data are compressed, BlueZIP restores the original data by using the dictionary index which indicates the data entry in the dictionary, along with the length of the data segment. In BlueZIP, the dictionary is reconstructed on-the-fly at data decompression, so additional metadata for decompression is not required.

3) Granularity of Data Compression

In order to take full advantage of the benefit of data compression, BlueZIP should provide a meaningful compression ratio so that a large number of page writes can be eliminated. Unfortunately, NAND flash memory must be programmed and be read in a unit of one page. Therefore, it is difficult to avoid internal fragmentation that occurs when the size of compressed data does not fit into a unit of a page. This fragmentation problem wastes valuable storage capacity and, more importantly, reduces the overall compression ratio. The simple but effective way to mitigate such an internal fragmentation problem is to increase the number of pages compressed together, which is called a data chunk in this paper.

Data compression with a large data chunk, however, has two main drawbacks. First, a large data chunk hurts the read performance. In order to read a page from flash memory, the entire data chunk that holds the requested page should be decompressed first. Second, data compression with a large data chunk requires more hardware resources. As the size of a data chunk increases, a hardware accelerator needs a larger memory space because it keeps more data in the buffer memory while performing data compression/decompression. As will be discussed in Section IV in detail, the current implementation of BlueZIP uses a data chunk of 8 KB, which

corresponds to four 2 KB pages, because this setting achieves a good compression ratio with a small read penalty using a reasonable amount of hardware resources.

Since the basic unit of data transfer between the file system and a storage device is usually 2-4 KB, there is a size mismatch problem with the data chunk size. In order to resolve this problem, the FTL stores the data from the file system temporarily in its internal write buffer. When the write buffer becomes full, the stored pages are sent to the flash memory. In BlueZIP, the write buffer size is set to 8,208 bytes, which is slightly larger than 8 KB, because the chunk header information of 16 bytes is added to the data to be written. This header contains metadata for the pages stored on the same data chunk so that the FTL uses for garbage collection and decompression. In addition, the sizes of the DMA buffer and the compression buffer are set to 10 KB, respectively, which is large enough to hold the entire data chunk during compression.

B. Software Architecture of BlueZIP

In this subsection, we explain CaFTL, the proposed compression-aware flash translation layer. We first describe the address translation mechanism of CaFTL, which is devised to manage compressed data in NAND flash memory, and then discuss several issues related to garbage collection with CaFTL. Finally, we explain the selective compression scheme in detail.

1) Address Translation

CaFTL is based on a page-mapped FTL which maps a logical page into a physical page using a page mapping table. Unlike other page-mapped FTLs, CaFTL maintains a special data structure, called a data chunk table, which manages information about data chunks in flash memory. A data chunk table is composed of table entries, each of which is 8 bits: the first 3 bits are used for a valid page counter, the following 4 bits represent the number of physical pages allocated to a data chunk, and the remaining 1 bit is used as a compression indicator which indicates whether the data chunk is compressed or not. The number of table entries is the same as the number of physical pages and the table entries that belong to the same data chunk have the same values.

Fig. 3 shows an overall architecture of CaFTL with a page mapping table and a data chunk table. As mentioned before, CaFTL keeps the data of four pages in the write buffer and then flushes the stored pages to flash memory altogether, along with their header information. Header information includes the logical page addresses for four pages, which are used for decompression and garbage collection later. CaFTL updates the page mapping table so that each logical page entry indicates the first physical page address of the new data chunk. After the data have been written, CaFTL gets the number of physical pages actually used for writing the data chunk, updating the corresponding entries of the data chunk table. Note that a valid page counter is initially set to four because all the four pages are valid.

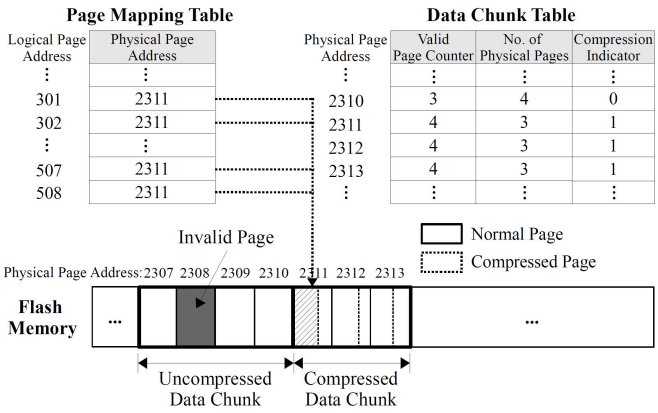

Fig. 3. An overall organization of CaFTL.

In order to read the data of a page from flash memory, CaFTL first finds the physical location of the data chunk containing the requested page by referring to the page mapping table. All the pages in the data chunk are then decompressed. CaFTL finds the requested logical page by looking at the header information and only the data of the requested page are transferred to the host. In addition, CaFTL maintains the read buffer to prevent repeated decompression for frequently accessed data chunks. CaFTL keeps four data chunks in the read buffer, which is managed by an LRU replacement policy.

When a certain logical page is updated by new data, CaFTL decrements the valid page counter of the corresponding entry in the data chunk table by 1 because the corresponding data chunk contains no longer valid data for that page. The new page is written to the newly allocated data chunk, along with other pages which are requested together.

Since CaFTL needs an additional data chunk table (whose size is increasing as the capacity of SSDs increases), it requires more memory space than the traditional page-mapped FTLs. However, we can mitigate the table size problem by adopting a demand-based mapping mechanism proposed in [7]. Furthermore, the data chunk table is smaller than the page mapping table because it requires only an 8-bit entry for each physical page.

2) Garbage Collection

CaFTL performs garbage collection to reclaim free space after all available free blocks are used up. Similar to a greedy policy used in existing FTLs, a block with the fewest valid pages is selected as a victim block. Once the victim block is chosen, CaFTL looks at the status of the data chunks in the victim block by referring to the data chunk table. If the data chunk has no valid pages (i.e., the valid page counter is 0), it is not necessary to move the pages in the data chunk because it contains only invalid pages. Therefore, CaFTL skips this data chunk and then sees the next one. If there is a data chunk with valid pages, CaFTL decompresses the data chunk and then stores only the valid pages on the temporary buffer. Similar to writing the data sent from the host, CaFTL evicts four valid pages to the new data chunk at once, updating the

page mapping table as well as the data chunk table. The victim block is erased and becomes a free block after moving all valid pages.

3) Selective Compression

The size of a data chunk compressed by BlueZIP can be larger than that of the original one because additional metadata (e.g., a chunk header and a compression flag) is included in a compressed data chunk. To prevent the size expansion problem, CaFTL exploits a selective compression function of BlueZIP. Since multimedia files, which were already highly compressed, are most likely candidates for the size expansion problem, CaFTL focuses on such data files when making selective compression decisions. For example, if CaFTL detects poorly compressed data streams in advance, it does not compress those data.

To detect a data stream whose write pattern is sequential and whose compression ratio is low, CaFTL monitors a compression ratio of a data chunk whenever it is written to a flash memory. If a compressed data chunk is larger than the original one and the logical addresses of the four pages in the data chunk are sequential (e.g., the logical addresses are 100, 101, 102, 103), this data chunk is regarded as a sequential data stream. CaFTL keeps the last logical address (e.g., 103) of the sequential data stream in a data structure, called a filtering table. The filtering table contains the information of each sequential data stream: a logical page address and a reference counter. If a new sequential data stream is observed and its first logical address is consecutive (e.g., 104) to the previous data stream in the filtering table, the previous data stream is replaced by the new one and its reference counter is increased by 1. Once a reference counter reaches to a certain threshold value (which is set to 4 in our implementation), CaFTL writes the following sequential data streams to the flash memory without compression. Note that a chunk header is not required for an uncompressed data chunk because a logical page address can be stored in a spare area of a page. In addition, a compression indicator of an uncompressed data chunk should be set to 0 so that decompression steps are bypassed when reading that chunk later.

The current version of CaFTL keeps only 20 sequential data streams in the filtering table. If the filtering table becomes full, the data stream with the smallest filtering counter is removed and the new one is inserted into the filtering table. The memory requirement for keeping the filtering table is as small as several-ten bytes.

V. Experimental Results

A. Experimental Environment

For the evaluation, we implemented the hardware and software modules of BlueZIP on the BlueSSD platform [4]. Fig. 4 shows a snapshot of the BlueSSD platform. As mentioned in Section III, BlueSSD is composed of two main components, the FPGA board and the custom flash board, called BlueFlash. The FPGA board is equipped with an FPGA fabric for implementing hardware logics and an embedded processor running at 400 MHz for executing software modules.

The BlueFlash board holds four identical flash buses and each bus supports 8 flash chips. Each flash chip can store up to 1 GB of data with 4096 blocks. Each block is composed of 128 2-KB pages. In the current implementation, four buses share a single compression/decompression module because our FPGA device is not large enough to implement a dedicated compression/decompression module for each flash bus. For a detailed analysis on the hardware resource utilization of BlueZIP, see Section IV-D.

The hardware modules of BlueZIP were modeled and synthesized with a rule-based hardware design language [8]. CaFTL was implemented as a device driver in Linux 2.6.25.3. The block size of a Linux file system [9], which is the unit of data transferred from/to BlueSSD, was set to 2 KB so that its size is the same as that of a page.

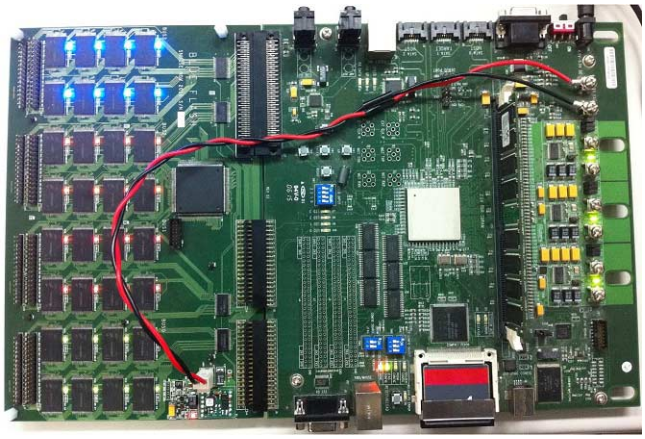

Fig. 4. A snapshot of an FPGA-based SSD prototype, BlueSSD.

B. Effect of the Design Parameters of BlueZIP on the Compression Ratio

We first investigated the effect of the dictionary table size and the data chunk size on the compression ratio while changing their sizes. For this evaluation, we have implemented a software simulator of BlueZIP's compression and decompression modules because it allows us to easily evaluate the effect of the design parameters of BlueZIP on the compression ratio.

Four types of data files with different compressibility were used for the evaluation: SENSOR, LINUX, DOCUMENT, and MP3. SENSOR is a set of sensor data files which were collected during a semiconductor fabrication process. These sensor data files contain a few text patterns which are repeated a large number of times, so it shows a very good compression ratio. LINUX is a subset of the Linux kernel 2.6.32 source files with a good compression ratio. DOCUMENT is a set of documents and image files with the file extensions (such as .ppt, .pdf, .doc, .bmp, and .jpeg.) DOCUMENT shows a medium compression ratio. MP3 is a set of MP3 files which were already highly compressed.

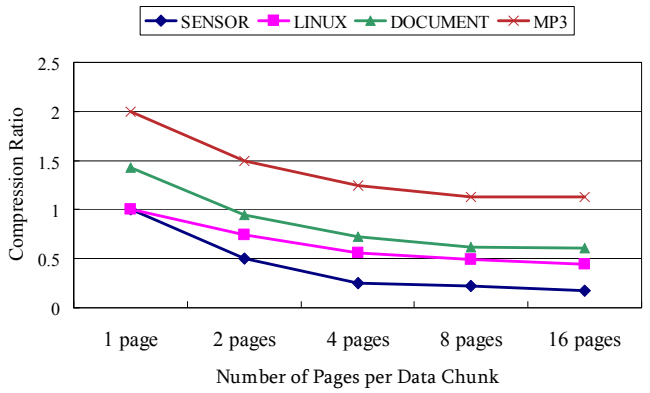

Fig. 5. The effect of different number of pages in the data chunk on compression ratios.

Fig. 5 shows the effect of varying sizes of a data chunk from 1 page to 16 pages on the compression ratio. When the data chunk consists of a single page, there was no benefit of data compression because of the internal fragmentation problem. As the number of pages compressed together increases, however, the compression ratio is accordingly improved because the wasted space by internal fragmentation is reduced. The improvement in the compression ratio becomes negligible when the number of pages in the data chunk gets larger than four pages. Another important observation from Fig. 5 is that the compression ratio is higher than 1.0 for DOCUMENT and MP3 files, thus making compressed files bigger than original uncompressed files. Note that this size expansion problem can be resolved using the selective compression technique of BlueZIP, which will be discussed in the following subsection.

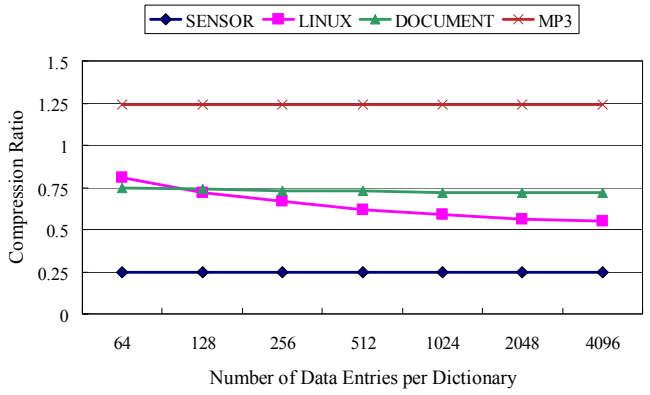

Fig. 6. The effect of different number of data entries in the directory on compression ratios.

Fig. 6 shows the effect of varying sizes of a dictionary from 64 to 4096 entries on the compression ratio. For the MP3 benchmark, which exhibits quite low compressibility, the dictionary size does not have a significant effect on the compression ratio. This is because same patterns are rarely repeated in MP3 because it was already highly compressed. On the other hand, in the case of LINUX, as the number of dictionary entries increases, the overall compression ratio is improved because more useful patterns can be kept in the dictionary. SENSOR and DOCUMENT have a high degree of compressibility, but their compression ratios are the same regardless of the number of data entries because a small dictionary table is sufficient enough to maintain useful bit patterns. As shown in Fig. 6, even though the optimal number of data entries is somewhat different depending on the types of the input files, the compression ratio is saturated for all the benchmarks when the number of entries reaches 2048.

Based on the results shown in Figs. 5 and 6, we have decided to use the dictionary table with 2048 data entries and the data chunk with four pages (i.e., 8 KB) because they exhibited a good compression ratio with a relatively small amount of hardware resource.

C. *Performance Evaluation*

In order to evaluate the performance and lifetime impact of BlueZIP, we have compared three configurations of our BlueSSD system: *Baseline*, $BlueZIP_{alwz}$, and $BlueZIP_{sel}$. *Baseline* is our baseline design, i.e., BlueSSD, without using BlueZIP. Both $BlueZIP_{alwz}$ and $BlueZIP_{sel}$ are BlueSSD combined with BlueZIP, but they differ in that $BlueZIP_{sel}$ supports selective compression, while $BlueZIP_{alwz}$ compresses all the data written to a flash chip.

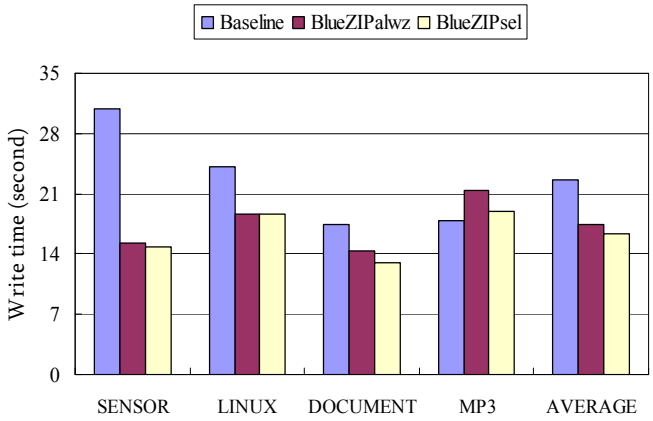

Fig. 7. Write performance for different file types.

Fig. 7 reports the comparison results for the write performance for the four test files when they are copied to BlueSSD. Even though selective compression was not used, $BlueZIP_{alwz}$ shows a fairly good performance for the test files with good compression ratios. $BlueZIP_{alwz}$ achieves about 50%, 23%, and 17% higher performance over *Baseline* for SENSOR, LINUX, and DOCUMENT. However, for compressed files, the performance of $BlueZIP_{alwz}$ somewhat deteriorates. For MP3, $BlueZIP_{alwz}$ is about 20% slower than *Baseline* because the number of the pages written to flash memory is increased if data were already highly compressed. By avoiding useless compression, however, $BlueZIP_{sel}$ shows a better write speed over $BlueZIP_{alwz}$, achieving almost the same performance as *Baseline*.

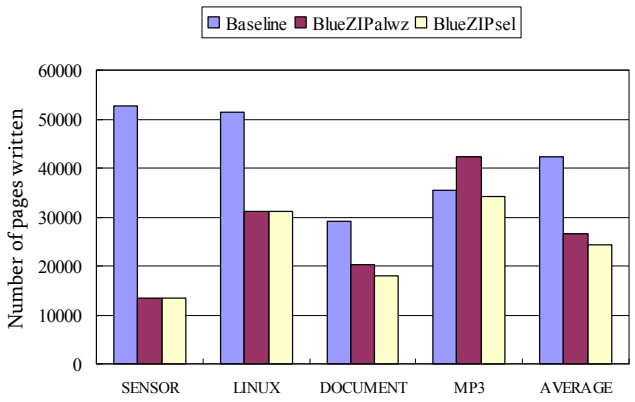

Fig. 8. The number of page written for different file types.

Fig. 8 shows the number of pages written to flash memory when copying the test data files. *BlueZIP*$_{sel}$ writes 26% less data to flash memory over *Baseline*, thus improving the overall lifetime of SSD by the same amount. One interesting observation is that *BlueZIP*$_{sel}$ writes less amount of data over *Baseline* for MP3. This is because copying MP3 files generates many metadata updates for the file system whose compressed size is a lot smaller than its original size. In addition, the selective compression function of BlueZIP also helps to prevent the size expansion by eliminating useless data compression.

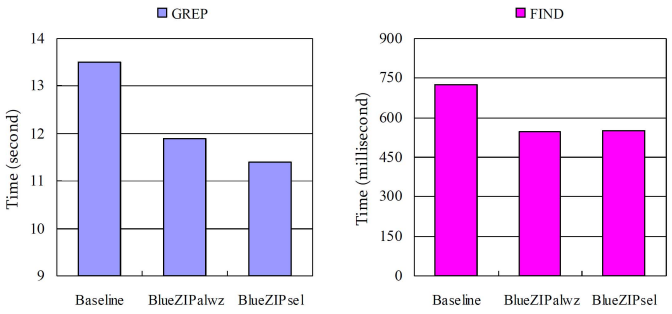

Fig. 9. Read performance with GREP (left) and FIND (right).

In Fig. 9, we have compared the read performance of *BlueZIP*$_{sel}$ while executing two read-intensive applications, GREP and FIND, on Linux kernel source files. GREP searches all the source files to find a matching string, whereas FIND searches for files in directories. Fig. 9 shows that *BlueZIP*$_{sel}$ improves the overall read performance by 20% on average. Although there is some decompression overhead during read operations, this result indicates that the reduction in the number of pages read sufficiently offsets the decompression overhead.

We have evaluated the overall performance of *BlueZIP*$_{sel}$ using a more complicated benchmark program. We have selected the Postmark benchmark because it is widely used to evaluate the performance of storage devices. We modified Postmark so that it generates three different types of data: TEXT$_{raw}$, TEXT$_{web}$, and IMAGE$_{web}$. TEXT$_{raw}$ is raw text data, TEXT$_{web}$ is Web text data, and IMAGE$_{web}$ is Web image data. Fig. 10 shows the results with Postmark. For TEXT$_{raw}$ and TEXT$_{web}$ (whose compression ratios are relatively high), the performance improvements by both *BlueZIP*$_{alwz}$ and *BlueZIP*$_{sel}$ are significant. Even for IMAGE$_{web}$ whose compression ratio is expected to be low, *BlueZIP*$_{alwz}$ and *BlueZIP*$_{sel}$ achieve a relatively good performance. Postmark is a metadata intensive benchmark; writes to metadata account for about 30% of all the write requests. By effectively reducing the amount of the metadata written, *BlueZIP*$_{alwz}$ and *BlueZIP*$_{sel}$ reduce about 14% of the data written to flash memory. Since Postmark generates many small-size transactions (whose sizes are less than 9 KB), the benefit of selective compression is negligible.

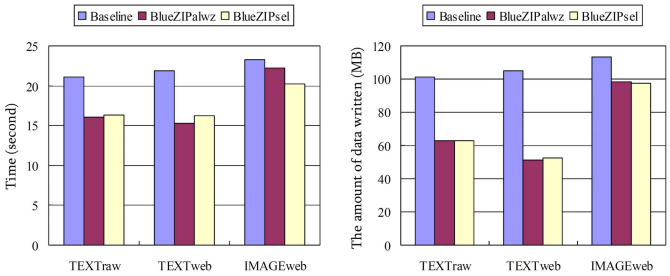

Fig. 10. Execution time (left) and amount of data written (right) with Postmark benchmark.

D. Hardware Utilization

Finally, we compared the hardware resource usage of BlueZIP with that of the standalone BlueSSD. Table I shows the utilizations of hardware resources according to their types; *Baseline* indicates the standalone design without BlueZIP, and *BlueZIP* denotes the BlueSSD design with hardware compression and decompression modules. As shown in TABLE I, compared to our *Baseline* design, *BlueZIP* requires 23% more Slices for the implementation of hardware compression/ decompression logics and consumes 33% more BRAMs, which are used for the compression/decompression buffer and the dictionary table. The utilizations of other resources including IOBs, GCLK, and DCMs are the same as the *Baseline* design.

TABLE I
HARDWARE RESOURCE UTILIZATION

Type of Resource (# of Available Resources)	# of Resources Used (%)	
	Baseline	BlueZIP
Slices (13696)	8180 (59%)	11234 (82%)
- Flip Flops (27392)	8062 (29%)	9686 (35%)
- 4 input LUTs (27392)	13740 (50%)	19175 (70%)
Bonded IOBs (556)	111 (19%)	111 (19%)
BRAMs (136)	76 (55%)	121 (88%)
GCLK (16)	4 (25%)	4 (25%)
DCMs (8)	1 (12%)	1 (12%)

V. CONCLUSION

In this paper, we have proposed a hardware accelerated compression module, called BlueZIP, and a compression-aware flash translation layer, called CaFTL. To show their feasibility and effectiveness in improving performance and

lifetime, we have implemented BlueZIP and CaFTL on an FPGA-based SSD prototype and have evaluated their performance with realistic benchmark programs. Our evaluation results show that BlueZIP supported by CaFTL can improve the lifetime of SSDs by 26% and improve read and write speed on average by 20% and 27%, respectively.

BlueZIP can be improved in several directions. To eliminate the extra overhead induced by hardware compression, we will investigate a pipelined architecture for BlueZIP so that the compression/decompression process is completely overlapped with I/O operations. This also allows us to investigate several design alternatives in terms of compression ratio, hardware cost, and operation speed. Integrating hardware compression with data de-duplication is also one of our future works. Data compression is beneficial in removing repeated patterns inside a data chunk, whereas data de-duplication helps us to eliminate duplicate chunks in a very large volume of data [10]-[12]. By exploiting complementary aspects of these two techniques, we can further improve the performance and reliability of flash-based SSDs.

REFERENCES

[1] S. Saeki and M. Oishi, "SSDs challenge HDDs, but quality a problem," *Nikkei Electronics Asia*, Jun. 2009.

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

[3] T. Park and J.-S. Kim, "Compression support for flash translation layer," in *Proceedings of the International Workshop on Software Support for Portable Storage*, pp. 19-24, Oct. 2010.

[4] S. Lee, K. Fleming, J. Park, K. Ha, A. Caulfield, S. Swanson, Arvind, and J. Kim, "BlueSSD: an open platform for cross-layer experiments for NAND flash-based SSDs," in *Proceedings of the International Workshop on Architectural Research Prototyping*, Jun. 2010.

[5] M. Kjelso, M. Gooch, and S. Jones, "Design and performance of a main memory hardware data compressor," in *Proceedings of the EUROMICRO Conference*, pp. 423-430, Sept. 1996.

[6] R. N. Williams, "An extremely fast ziv-lempel data compression algorithm," in *Proceedings of the Data Compression Conference*, pp. 362-371, Apr. 1991.

[7] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, pp. 229-240, Mar. 2009.

[8] R. Nikhil, "Bluespec system verilog: efficient, correct RTL from high level specifications," in *Proceedings of the International Conference on Formal Methods and Models for Co-Design*, pp. 69-70, Jun. 2004.

[9] R. Card, T. Ts'o, and S. Tweedie, "Design and implementation of the second extended filesystem," in *Proceedings of the Dutch International Symposium on Linux*, Dec. 1994.

[10] F. Chen, T. Luo, and X. Zhang, "CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proceedings of the USENIX Conference on File and Storage Technologies*, pp. 77-90, Feb. 2011.

[11] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pp. 91-103, Feb. 2011.

[12] Q. Yang and J. Ren, "I-CASH: intelligently coupled array of SSD and HDD," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 278-289, Feb. 2011.

BIOGRAPHIES

Sungjin Lee received the B.E. degree in electrical engineering from Korea University, Korea, in 2005, and the M.E. degree in computer science and engineering from Seoul National University, Korea, in 2007. He is currently working toward the Ph.D. degree at Seoul National University. From 1999 to 2002, he was a software engineer at Bridgetec Co., Seoul, Korea. His research interests include storage systems, operating system, and embedded software.

Jihoon Park received the B.E. degree and the M.E. degree in computer science and engineering from Seoul National University, Korea, in 2008 and 2011, respectively. His research interests include storage systems, hardware design, and reconfigurable logic.

Kermin Fleming is a graduate student at the Massachusetts Institute of Technology. He is a graduate of Carnegie Mellon University (BS 06, MS 06). His interests include hardware design, high-level synthesis, and reconfigurable logic.

Arvind (M'74-SM'85-F'94**)** is the Johnson Professor of Computer Science and Engineering at the Massachusetts Institute of Technology and a member of CSAIL (Computer Science and Artificial Intelligence Laboratory). From 1974 to 1978 prior to coming to MIT, he taught at the University of California, Irvine. Arvind received his M.S. and Ph.D. in computer science from the University of Minnesota in 1972 and 1973, respectively. He received his B. Tech. in electrical engineering from the Indian Institute of Technology, Kanpur, in 1969, and also taught there from 1977-78. Arvind's current research focus is on enabling rapid development of embedded systems. Arvind is a Fellow of both IEEE and ACM, and a member of the National Academy of Engineering.

Jihong Kim (M'00) received the B.S. degree in computer science and statistics from Seoul National University, Seoul, Korea, in 1986, and the M.S. and Ph.D. degrees in computer science and engineering from the University of Washington, Seattle, WA, in 1988 and 1995, respectively. Before joining SNU in 1997, he was a Member of Technical Staff in the DSPS R&D Center of Texas Instruments in Dallas, Texas. He is currently a Professor in the School of Computer Science and Engineering, Seoul National University. His research interests include embedded software, low-power systems, computer architecture, and storage systems.