**ABSTRACT**  One of the most important features of solid-state drives (SSDs) is the ability to update data sectors out of place, performing garbage collection operations within its physical blocks, to mitigate limited flash memory lifespan. In SSDs, a firmware called the Flash Translation Layer (FTL) is used to hide such features from the operating system. One way to extend the lifespan of flash memories is to use data compression at the FTL level. In compression-based FTLs, the address mapping type has a significant effect on the efficiency and read/write speed. The type of address mapping in previous similar schemes is based on page-level address translation and requires high memory space for the mapping table. In this paper, we propose a compression-based hybrid FTL called In Block Update Address Mapping, in which the pages of each logical block are compressed, and fewer physical pages are written in physical blocks. Therefore, a small number of free pages remain. In the proposed scheme, the required memory of the mapping table has been reduced by 78% when compared to similar schemes. Moreover, extensive simulation results reported in this paper show that the proposed FTL scheme outperforms other FTL schemes in the read and write operations under realistic benchmark workloads.

**INDEX TERMS**  Flash memory, SSD, flash translation layer, garbage collection, data compression in SSD.

**I. INTRODUCTION**  The use of NAND Flash memory technology in external storage has led to the emergence of solid-state drives (SSDs). The most important features of this type of storage include its small size, low power consumption, and shock resistance. In SSDs, several NAND Flash Memory chips are connected to the flash controller module via a common channel. The flash controller module is responsible for generating read, write and erase signals for NAND Flash Memory chips. Each SSD has multiple channels for transmitting data, therefore it can serve read and write requests in parallel [1]–[3]. Since the input rate of requests is usually higher than the servicing rate, a DRAM is used as a buffer for I/O requests in SSDs [4], [29].

In flash memory chips, there are usually 2 to 8 dies, and each die has its own control signals. Each die contains several planes and there are thousands of physical blocks per plane. Each block usually comprise 64, 128, or 256 physical pages. By activating the erase signal for a physical block, all pages of that block are erased together. Each physical page is made up of main data section and spare data section. The main data block size can be 4 KB, 8 KB, 16 KB or 32 KB and spare data blocks usually contain control data and error correcting codes, etc [2], [3], [5], [30].

In NAND Flash memories, read, write and erase operations are supported. Reading and writing operations can be performed over each physical page, but erasing operations are performed over each physical block. Also, the number of erase operations in physical blocks is limited, and the physical blocks age and loose their functionality after a limited number of cycles. The NAND Flash chips, depending on the internal structure and the number of bits stored in each cell, are classified into three categories: SLC, MLC, and TLC. SLC-type flash memory has the most extended lifespan and can store of a single bit of data. Wearing out by being erased, the MLC and TLC flash memories gradually lose the ability to save data reliably, making them more suitable for personal use than being a memory of choice in commercial applications [3], [6].

Flash memories have several limitations such as limited lifespan, the inability to overwrite data on pre-written pages, and the asymmetry of read and write operations.
To address these limitations, a firmware called the Flash Translation Layer (FTL) is used between the file system and the flash memory. The Flash translation layer consists of an address translation unit, a garbage collection unit, and a wear-leveling unit [6], [7]. An out of place writing method is used to overwrite a physical page in flash memory. In this method, the new data is written on a free page and then the previous page becomes invalid. Physical blocks containing invalid pages are called dirty blocks. When the free spaces of a flash memory reaches a threshold, then the garbage collection module cleans up the dirty blocks. Garbage collection operations transfer the valid pages to the free blocks and then erase the dirty blocks. The extra capacity of flash memory space that is always kept free is called over provisioned space (OPS) [7], [31].

The physical blocks of flash memory can be written and erased for a limited number of times [8]. One way to extend the lifespan of flash memory is to distribute write requests evenly across physical blocks. These operations are called wear-leveling and impose a high overhead on the SSD [29]. Another way to extend the lifespan of flash memory is to use data compression techniques at the SSD level [9], [10].

The design of common file systems such as FAT32 and NTFS are based on a typical hard disk structure. In a common file system, any logical data address is equivalent to a hard disk sector address. But flash memories do not have the ability to overwrite pre-written pages and data must be written in free pages. For this reason, flash memories use a module called “Address Translation” that maps the logical addresses to the physical pages of the flash memory [6], [7].

In flash memories, the architecture of the address translation unit has a very significant impact on the reading and writing requests delays. Also, one of the suggested methods for managing the memory space of a flash device and using it optimally is to use data compression. Generally, FTLs that use data compression need very large address translation tables, and that is a weakness. In this paper, we propose a new FTL that uses data compression compatible with personal computers SSDs. This proposed FTL scheme comprise a new type of address translation unit whose mapping table is much smaller, thus the speed of reading and writing requests are significantly better than with previously reported schemes. Our scheme also uses a new technique for detecting data compression capabilities, which is based on recognizing the type of write requests.

Section 2 introduces the address translation related works and latest FTLs that utilize data compression techniques and in Section 3, the new proposed scheme called in-block update mapping is introduced. Section 4 provide simulation results confirming the advantages of the proposed FTL and Section 5 concludes the paper.

II. RELATED WORKS

The address translation unit plays a very important role in a flash memory, and FTL refers to the address translation in the flash memory. The mapping operations in the address translation unit can be classified into three categories, page level, block level, and hybrid. In page mapping, the entry of the mapping table is a LPN (Logical Page Number), and the content of each entry is a PPN (Physical Page Number). The advantage of page mapping is that data can be freely written on any physical page. However, the disadvantages of this scheme are that the mapping table is very large. For example, 1 TB of SSD requires 1 GB of RAM for a mapping table [11], [12], [30].

Write requests that are sent to the flash memory over a time period include a collection of sequential and random write requests. Hybrid address translation designs usually use the sequential or random feature of the requests and translate sequential write requests into block mapping and random writing requests as page mapping. FAST and LAST schemes are two examples of combining translation addresses for flash memory [4], [13], [14].

Since the main problem with page address translation is the large size of the mapping table in schemes such as DFTL, CDFTL propose that the address translation operation be based on demand and that the mapping table be stored on the physical pages of the flash memory. Then, to translate a logical address into a physical page address, the related mapping information is read from the physical pages (called translation pages) and loaded into a small mapping table. Such schemes require a very small mapping table, but their main problem comes from the reading and writing requests, which require reading and rewriting translation pages, and this creates overhead in the form of additional processing time [15], [16], [30].

In some previous related works, a type of page-level address translation has been suggested. It supports the idea of compressing data in flash memory. Among the schemes that support data compression at the level of flash memory, the ZFTL, LDC and EDC schemes have been proposed. These schemes are based on page-level address translation and therefore require a lot of memory for mapping tables. In the following, these schemes are briefly described.

A. FTLS THAT SUPPORTED DATA COMPRESSION IN FLASH MEMORY

The ZFTL scheme proposed by Park et al. [9], [10] is an innovative flash translation layer scheme that supports compression of write requests at the level of the Flash translation. In this scheme, several compressible logical pages are written into a write buffer. When the write buffer does not have enough space for the new compressed data, then it is flushed into a physical page of the flash memory. The data compression unit in this scheme is fixed, and it is equal to the size of a logical page (4KB). The zFTL scheme detects incompressible data using a hardware module and avoids the compression of such data. This module is called incompressible data predictor. Since in the ZFTL scheme, multiple logical pages are written within a physical page, therefore, a page state table (PST) is required to perform the garbage collection operation. The most significant disadvantage of the ZFTL
scheme is the large size of the Page Map Table. This table is relatively large, about 25% of the Page Map Table (PMT), which leads to a surging memory overhead when new data is written in SSDs frequently. Its second disadvantage is its incompressible data predictor module which increases the complexity of the system hardware.

The LDC scheme is an improved version of the ZFTL scheme proposed by Ji et al. [17]. This scheme eliminates the idea of using the incompressible data predictor hardware module in the ZFTL scheme and uses a software module to predict the data compression capability [17], [18]. The second improvement in this scheme is about garbage collection, for which a new garbage collection algorithm called compression aware garbage collection was proposed. The main idea behind this algorithm is that most data that are detected as incompressible have little compression potential. During garbage collection operation in a physical block, if this type of data is valid, then this valid data is compressed if possible and written to the boundary between two consecutive pages using the internal fragment space of the physical pages. By compressing this type of data, fewer physical pages are written during garbage collection and thus more free pages remain in a physical block.

Another recently proposed scheme is called EDC [19]. In this scheme, three different types of compression algorithms have been used according to the input data rate of the flash memory. If the input data rate is too high, it uses a compression algorithm that has a high compression speed but that offers less compression. By contrast, if the input data rate is low, it uses a compression algorithm that has low speed but that offers better compression.

B. MOTIVATION

Among the writing requests in flash memory, some have random target location while others are sequential. Random write requests cause many pages to be processed by the garbage collection mechanisms, which causes a lot of overhead in flash memory [4], [14]. Page-level FTLs are recommended for applications where accesses are composed in majority of random writes and are characterized by very high update rates (such as SSDs in data centers). The reason for using page mapping in previous compression-based FTLs is that the number of logical pages compressed per physical page varies [9], [17], and therefore it challenges other address mapping schemes (block or hybrid mapping) in a way that has not been discussed so far. Therefore, the size of address mapping tables in such FTLs is very large, and this is a major shortcoming [9].

However, in many personal computers (PCs), the number of random write requests to flash memory is not high and most of the requests are sequential. Also, most of the requests for writing to multimedia or compressed archive files are sequential, and they comprise few random writes [14]. Since PCs perform many write requests that are sequential and incompressible, implementing a hybrid address translation can be very effective. For this reason, the best type of address mapping in FTLs that use data compression capabilities in PCs is a hybrid address translation that requires a relatively small address translation table.

The proposed scheme is a hybrid log-based FTL that supports data compression at the level of the Flash translation layer for PCs. In this scheme, the pages within each logical block are compressed using a data compression technique, and in total fewer physical pages are written in each physical block. Therefore, in the physical blocks of flash memories, there are usually a number of free pages left, which are used for data updating requests. Also, this scheme uses a new technique for detecting data compression capabilities, which is based on recognizing the type of requests. Since the proposed method allows the data to be updated and overwritten within each physical block, it is called in block update address mapping or IBU Mapping.

III. PROPOSED METHOD (IN BLOCK UPDATE ADDRESS MAPPING)

The proposed IBU Mapping scheme exploits the concept of data compression in the FTL layer. Since many files written in a flash memory are incompressible, and these files are usually large, their write requests are mostly generated in the form of sequential requests. One of the main features of the proposed scheme is the block-level mapping of sequential incompressible write requests. The general structure of the IBU Mapping scheme is shown in Figure 1.

To write the received requests, compressibility of requests is predicted using the Incompressible Request Predictor module (1 see Figure 1) and the writing requests are categorized into two groups: incompressible requests (2) and compressible requests (3). Next, each request is categorized into a number of sequential sub-requests and small sub-requests using the Request Splitter module (4). Each of these sub-requests is equivalent to the data of a logical block. By definition, a sequential sub-request contains all pages of a logical block, while a sub-request contains only a small portion of a logical block. As a result of these categories, writing requests are categorized into three groups: small incompressible sub-requests (5), sequential incompressible sub-requests (6) and compressible sub-requests (7).

Each compressible sub-request is equivalent to a logical block, and by compressing the pages of each logical block (8), several logical pages are written into one physical block page (9). This scheme uses a two-level address mapping table called IBU Map Table (10) to translate the address, in which each logical block is mapped to a physical block and a shared log block (belonging to the over-provision area). After servicing the compressible request, there are usually a few free pages left in each physical block. These free pages are reserved for writing update requests and thus allows the rewrite of sub-request within each physical block. If the sub-request is tagged as sequential incompressible, then it is completely written to a new physical block and its addresses is updated in the IBU Map Table. The advantage of consider-
ing incompressible sequential requests is that it dramatically reduces the Garbage Collection overhead.

If several small incompressible sub-requests are placed together in a way to complete a logical block, then each of them is called a small incompressible sequential sub-request. These types of sub-requests are detected using the Locality Detector module (11) and then aggregated into a number of log blocks called Sequential Log blocks using a block mapping table (12). After each sequential log block is fully written, the associated mapping addresses are then transferred to the IBU Map Table. Collecting this type of sub-requests in sequential log blocks reduces the overhead of garbage collection operations. However, if a small incompressible sub-request is not recognized as a sequential sub-request by the Locality Detector module, then it is called a random incompressible sub-request and written by the IBU Map Table into a data block or shared log block. Thus, by separating the write requests, in terms of the type of request (sequential or random) and their compressibility, it is possible to use a hybrid address translation architecture in the proposed scheme whose mapping memory is about 25% of that used by the previous schemes.

A. COMPRESSING REQUESTS AND PREDICTING INCOMPRESSIBLE REQUESTS

A very critical issue in FTLs that use a data compression approach is the data compression capability of write requests. In related schemes such as ZFTL and LDC, statistical analysis techniques are used to predict data compression capability and this process is done for all logical pages of a request [9], [17].

As mentioned in the motivation section, incompressible data at the flash memory level usually belongs to multimedia files, compressed document files, and compressed archive files. The studies in [20]–[23] show that the histograms of data value frequencies found in files with the same format are similar. These studies also show that these results also apply to fragments of a file. These approaches form a dataset of sample histograms as representatives of each file type. Then, the type of input files can be predicted by comparing its histogram with the samples in the dataset. The previous studies indicate this prediction method can be applied to every part of a file.

The experimental results show that the histograms of incompressible files are quite similar. The reason behind this is that each write request of an incompressible file is a part of that file. Thus, the histograms of incompressible files resemble each other. By utilizing this idea in the proposed scheme, the compression capability of the incoming write requests could be predicted with a high success rate.

The proposed method shapes a small dataset of representative histograms. After that, the histogram of each request is compared to these samples, and the histogram most similar to the input histogram is specified. In the case, if the
difference between the two histograms is less than a predefined threshold, the write request is marked as “Incompressible.” If a request is assigned to an incompressible file type, then all pages belonging to that request are considered incompressible. Thus, in the proposed scheme, checking for the data compression capability is done at the request level.

Figure 2 shows the general compressibility detection procedure. First, 10% of each of the data comprised in the write request is selected and a histogram is calculated for this data (1). The calculated histogram is then compared with each of the histograms in the FileType_Dataset (2) and the distance between the two histograms is calculated (3). Here, the “Root Mean Square Error” is used to measure the distance between two histograms. For each incompressible file type, a representative histogram is included in the FileType_Dataset. Each of the representative histograms is obtained by averaging the histograms of many files that have the same file format. This is done through a preprocessing step. At each iteration, if the distance between the histogram of the write-request and one of the histograms in the FileType_Dataset is less than or equals to a predefined threshold, the incoming write request is considered as an incompressible write request. Otherwise, the same procedure is repeated for the rest of the histograms in the FileType_Dataset. If there is no histogram in the FileType_Dataset that satisfies the Distance <= Threshold condition, the incoming write request is considered to be a compressible write request.

One of the most essential factors in selecting a compression algorithm is its speed of compression/decompression, compression rate, and its hardware implementation complexity. The proposed scheme uses the LZ77 algorithm, which is a very popular compression algorithm for which many optimized hardware implementations have been reported [9], [24]. The data compression unit is a vital factor in its compression rate and speed. Some compression algorithms, such as the LZ77, are such that the compression rate improves for larger compression units. These compression algorithms replace a repeating pattern in the compression unit with a shorter but unique pattern [9]. Although it is more appropriate to use a larger compression unit to achieve a better compression rate. The proposed scheme uses a fixed size compression unit (16KB), which is the size of a logical page in MLC NAND Flash. Several studies indicate that this block size offers a suitable compression rate [REFERENCE].

**B. LOCALITY DETECTOR MODULE AND WRITE INCOMPRESSIBLE DATA**

In the proposed scheme, the set of incompressible sub-requests includes two groups: the incompressible sequential sub-requests and the small incompressible sub-requests. The incompressible sequential sub-requests are written completely into a physical block and then the physical block address is transferred to the IBU Mapping Table with a switch merge operation. Additionally, incompressible sub-requests contain small incompressible sub-requests. This section meticulously discusses how to serve these write requests. Among the set of small incompressible sub-requests, there are usually several small sub-requests that arrive sequentially. Each of these sub-requests is called a small sequential sub-request and is detected with the Locality Detector module.

This module writes a small incompressible sub-request into several log blocks called a Sequential Log block using a spatial locality detector algorithm and a block address mapping table. After filling each of these sequential log blocks, the IBU scheme transfers the physical block address to the IBU mapping table with a switch merge operation.

The Locality Detector module of the IBU draws on the locality detection solution of LAST [14]. Moreover, before detecting the spatial locality, it checks the compressibility of the requests. Thus, only short incompressible files can be fed into this module. As a result, the IBU method can take advantage of compressing compressible files along with aggregating small incompressible requests in the same physical block.
FIGURE 3. Illustration of the locality detector module workflow.

The structure of the Block Map Table is shown in Figure 3. This table consists of two fields, LBN and PBN. This table uses the LBN field as a search key, and the PBN field refers to a Sequential Log Block containing sequential small sub-requests associated with a logical block (LBN). The spatial locality detection algorithm of each of the small sub-request is very simple and is as follows:

1. If the start address of a small sub-request is at the beginning of a logical block, then this sub-request is considered a small sequential sub-request and its logical block address is added to the Block Map Table and then the sub-request is written into a Sequential Log block.

2. If the logical block address of the small sub-request is already available as a search key in the Block Map Table, it can be written along with the data in the corresponding log block, and this sub-request will be a small sequential sub-request.

3. Otherwise, the small sub-requests are considered as random sub-requests and are written directly to the data blocks or shared log blocks according to the information in the IBU Map Table.

C. LOCALITY DETECTOR MODULE AND WRITE INCOMPRESSIBLE DATA

In a typical flow of incoming write requests, there are many compressible sub-requests. Their writing and management methods are described thoroughly in this section. In the proposed scheme for compressible sub-requests, there is a data compressor module that compresses the data of logical sub-requests and flushes them into a temporary buffer called the Container Buffer. As shown in Figure 4, the capacity of this buffer is equal to the size of a physical page and comprises two parts: Compressed Data Buffer and Metadata Buffer. The Compressed Data Buffer part holds logically compressed pages, and the Metadata Buffer part holds the metadata needed for address translation operations and garbage collection operations. When the Container Buffer capacity is sufficiently filled to prevent the next logically compressed page to flush in, then this buffer data is written to the last free page of a physical block. The Container Buffer is then reset and made ready for the next logical compressed page.

In compression-based FTLs, it is necessary to separate logical pages and validate compressed logical pages within each physical page during garbage collection and data reading operations. To do this, some additional information, called metadata, needs to be stored. Most of the time, there is some internal fragmented space left when compressing data into a physical page, which can be used to hold the metadata needed [5], [25]. The adopted solution in the IBU Mapping scheme is to use this fragmented space to store such metadata. This metadata includes a Number field (one byte) that represents the number of logical compressed pages, the offset of the compressed pages (one byte per page) and the size of the compressed pages (two bytes per page). With this metadata, one can easily separate valid compressed logical pages. The Metadata Buffer part is used in the container buffer to store this metadata.

The suggested address translation table is shown in Figure 4. This table has two separate sections for address translation. The first part of this table translates logical addresses into block-level mappings and contains two Data PBN and Log PBN fields per logical block. The Data PBN field refers to a physical block that belongs to the flash memory data space, and in the proposed scheme, all its pages are assigned to only one logical block.

The scheme uses a second address field in the mapping table called Log PBN to serve write requests without the need for multiple garbage collection operations. These log blocks are in fact the physical blocks of the over provision area that belongs to the flash memory inaccessible space and they are shared by the logic blocks in this proposed scheme. If the physical data block mapped to an LBN is fully written when
a new rewrite sub-request arrives for that LBN, then a shared log block is used to write a sub-request.

The second part of the mapping table writes the pages of a logical block as a page mapping into a physical block. This table contains fields for storing the physical offset of data written inside a physical block. The number of offset fields is the same as the number of pages in a physical block. For example, if there are 64 pages within a physical block, then 64 offset fields will be considered for the second part of the address mapping table. Next to each offset field is a single-bit label field called Log, and its activation indicates that the physical offset belongs to a physical page within the shared log block. Translating the address of a logical page to a physical page in the IBU Map Table is performed as follows:

1. Logical Block Number (LBN) a sub-request is obtained by dividing the logical page number by the number of pages in each block and then the physical block (Data block or Log block) mapped to the LBN is extracted from the block address translation section (equation 1, 2).

2. The offset of the page within the logical block is then obtained by dividing the remaining of the logical page number by the number of pages in each block. The offset of the physical page mapped to the logical page is then obtained by the page translation section of the mapping table. The physical page number mapped to the logical page is then calculated using the physical block number and the physical offset (equation 3, 4, 5).

\begin{align*}
1. \quad \text{LBN} &= \left(\frac{\text{LPN}}{\text{Page per Block}}\right) \\
2. \quad \text{PBN} &= (\text{Data Block or Log Block mapped to LBN}) \\
3. \quad \text{LPN Offset} &= \left(\frac{\text{LPN}}{\text{Page per Block}}\right) \\
4. \quad \text{PPN Offset} &= (\text{PPN Offset mapped to LPN Offset}) \\
5. \quad \text{PPN} &= (\text{PBN} \times \text{Page per Block}) + \text{PPN Offset}
\end{align*}

D. AN ALGORITHM FOR MANAGING WRITE REQUESTS IN THE IBU SCHEME

This section explains in detail how the write requests are managed in the IBU. Figure 5 shows the pseudo-code of the write requests algorithm for the flash storage device. In this scheme, the compressibility of each request is predicted first by the Incompressible Request Predictor module. Then each request is divided by the Request splitter module into a number of sub-requests. Each of these sub-requests is related to a logical block.

If the request is compressible, then the sequentially of the request is checked. If the sub-request is compressible sequentially, but the data block mapped to the sub-request does not have enough space to allow writing the compressed pages, then a new data block is allocated to write the sub-request compressed pages. Then with a switch merge operation, the new data block addresses are replaced by the old data block in the mapping table.
To help understand the operation of the IBU scheme, several detailed write request examples are provided in Figure 6. In these examples, the sub-request 0 is related to LBN 0 and is a compressible sequentially sub-request. According to the IBU Map Table, the data block mapped to this LBN (Data Block 10) does not have enough space to write this sub-request. So, a new data block is assigned to this request (Data Block 16) and then the logical pages of this sub-request are compressed and written to physical pages with offset 0 and 1 in the new physical block. Finally, with a switch merge operation, the IBU Map Table is updated.

However, if the sub-request is sequentially compressible and the data block mapped to the sub-request has enough space to write the compressed pages, then the logical pages of sub-request are compressed and written into the last pages of the data block, respectively. For example, in Figure 6, sub-request 7 is related to LBN 5 and is a sequentially compressible sub-request. The data block mapped to this log block (Data Block 15) has enough space to write the sub-request. Therefore, the logical pages of this sub-request are compressed and written to physical pages with offset 1 and 2 in Data Block 15, respectively.

If the sub-request is a small compressible sub-request, then the logical pages of the sub-request are compressed and written into a physical block mapped to the sub-request (data block or log block). For example, in Figure 6, sub-request 1 is a small compressible sub-request and related to LBN1. The data block mapped to this sub-request (Data Block 11) has enough space for its compressed pages and so, the logical compressed pages of the sub-request are written to the physical page with offset 1 within Data Block 11.

If the sub-request is incompressible, then the sequentially of the sub-request is checked. If the sub-request is an incompressible sequential type, then a new data block is selected for it and all logical pages of the sub-request are written in the new data block, respectively. Then, with a switch merge operation, the new data block address is replaced with the old data block in the IBU Map Table. For example, in Figure 6, sub-request 2 is an incompressible sequential sub-request. This is a sub-request related to LBN2. According to this example, a new block (Data Block 17) is selected to sub-request 2 and then all logical pages of this sub-request are written into it. Finally, IBU Map Table fields are updated with a switch merge operation for the LBN 2.

If the sub-request is incompressible small type, then the spatial locality of the small sub-request is checked according to the algorithm in Section B. If the sub-request has spatial locality, then the incompressible small sub-request is considered sequentially and its logical pages are written in a sequential log block, respectively. When a sequential log block is filled, then the sequential log block is replaced with a data block mapped to the LBN of a sub-request by the switch merge operation. For example, in Figure 6, sub-requests 3, 4 and 5 are all of an incompressible small sub-requests type. However, these sub-requests have the property of spatial locality and overall sub-requests are equal to the logical block (LBN3). These sub-requests after being written to the sequential log block (PBN 18) with the switch merge operation, the IBU Map Table is updated.
merge operation in IBU Map Table is replaced instead of data block 13.

However, an incompressible small sub-request may not have the feature of spatial locality, which we call an incompressible random sub-request. This type of sub-request is written within a data block or shared block mapped to the sub-request. For example, in Figure 6, sub-request 6 is an incompressible random type and relates to LBN 4. Since the data block mapped to this LBN (Data Block 14) is completely filled, the logical pages of this sub-request are then written to the shared log block mapped to LBN 4 (Log Block 0).

E. GARBAGE COLLECTION AND MERGE OPERATION FOR IBU SCHEME

Flash memories cannot overwrite physical pages, so garbage collection and erasing dirty blocks are inevitable to make free space for overwriting data. The proposed IBU Mapping scheme does not require a PST table for garbage collection of any physical block, and the status of valid logical pages within each physical page can be extracted directly through the IBU Map Table and the metadata stored in the fragment area of that page. In this scheme for every LPN stored in the metadata area of a physical page, if there is a mapping to that physical page in the IBU Map Table, then the logical page is valid and must be migrated in the garbage collection operations. For example, in Figure 7, there is only one address mapping in the IBU Map Table for LPN 6 data stored in PPN 44 and there is no mapping for the other LPNs. As a result, only LPN 6 data are valid in the PPN 44 physical page. Since the proposed scheme is a hybrid address translation scheme, it uses merge operations for garbage collection.

In the proposed scheme, there are three possible situations for switch-merge operations. This operation does not impose any time overhead to the system when transferring the data.
and only updates the addresses in the IBU Map Table. Generally, in the proposed scheme, switch-merge operations occur when:

1. If the sequential write sub-request is not compressible, then the sub-request is written to a new physical block with a switch-merge operation and the previous data block is cleared.
2. If small incompressible sub-requests have high spatial locality, such that they could fill a whole sequential log-block, then a switch-merge operation occurs between the previous data blocks and the sequential log block.

In this scheme, if small incompressible sub-requests have spatial locality but could not fill a whole sequential log-block, then a partial-merge operation is required to complete the sub-request writing process. In this case, the valid logical pages in the previous data block are moved to the sequential log-block, and then the address of the sequential log-block is recorded as a data block in the IBU Map Table. Partial-merge operations in this scheme have a low time overhead and are less likely to occur.

However, the most expensive type of merge operation that is likely to occur in the proposed scheme is the full-merge operation. The IBU Mapping scheme assigns a physical block and a shared log-block to each logical block in the system. When a shared log-block does not have enough space to store a new sub-request, then a full-merge operation is required for that shared log-block. Since each shared log-block is involved with several logical blocks, therefore several merge operations need to be performed among data blocks and shared log-block to erase a shared log-block.

### IV. SIMULATION RESULTS

This section presents simulation results comparing the proposed scheme (IBU Mapping), the ZFTL, and the LDC scheme. Among the FTLs that support data compression, the ZFTL scheme is usually chosen as the main and best-known benchmark. The LDC scheme is one of the latest designs and an upgraded version of the ZFTL scheme. The I/O workloads used in the simulation are collected according to the scenarios described in Table 1. Also, this table shows the distribution of the number of compressed logical pages that can be stored in a physical page for each workload data.

The Windows workload includes a combination of the file-types and requests that a PC user creates. This workload includes read and write requests caused by copying and creating different types of Windows files. The main feature of Windows workloads is that many of the generated write-requests are pre-compressed and thus cannot be compressed at the level of the Flash translation layer.

The second workload includes program installation workloads. This workload involves the process of installing different applications in a Windows environment. Installed programs include executable (.exe) and DLL files and various types of binary files. An important feature of this workload is that the majority of the generated write-requests are very compressible. The third workload belongs to the IOZone program [26]. The IOZone software simulates the workloads created by OLTP programs and includes various types of I/O operations (write, read, overwrite, sequential overwrite, etc.).

#### A. SIMULATION SETUP

An extended version of the SSDSim simulator [1], [25], [27] was used to evaluate the proposed scheme. SSDSim Simulator is a standard tool for simulating the Flash translation layer on SSD storage devices. The most important parameters set in the SSDSim simulator are shown in Table 2. These parameters are based on an MLC NAND Flash Memory chip [6] and a 200MHz integrated processor in a commercial SSD [17]. The size of the logical and physical pages is equal to 16 KB. This size is very common for physical pages of MLC NAND Flash memory. According to a previous scheme report and calculations performed, the data compression time of 16KB on a 200MHz controller is 540 µs [17].

As depicted in Table 2, small-sized flash memories were used. This is because to overwrite more data to flash so the performance of the proposed method can be evaluated better when more garbage collections are invoked.

Additionally, one of the main applications of the proposed method is in external SSDs connecting to PCs via USB connections. To this end, a flash-based SSD using USB2.0...
connection was simulated, and the data rate was 48 MB/s under simulation.

### B. SIMULATION EVALUATION OF GARBAGE COLLECTION OVERHEAD

Figure 8 shows the time overhead caused by garbage collection operations. This overhead includes the time needed for erasing physical blocks and the time for migrating valid pages. In this bar chart, the average garbage collection time of ZFTL was normalized to 100. Then, the corresponding figures of other methods were normalized based on the ZFTL’s normalized figure for average garbage collection time.

![Image: Figure 8. Garbage collection overhead.](image)

The garbage collection overhead for the IBU scheme for Windows workloads is about 4.3% less than with the ZFTL scheme and about 1.7% less than with the LDC scheme. Results also show that the higher the data compression rate in a workload, the better the IBU’s performance in garbage collection operations. The results of Figure 8 show that the garbage collection overhead in the IBU scheme for the Program Installation workload decreased by about 7.4% compared to the LDC scheme and by about 4.75% when compared to the ZFTL scheme. Also, for the IOZone workload in the IBU scheme, this reduction is about 4.89% when compared to the LDC scheme and about 7.14 when compared to the ZFTL scheme. The most important reason to improve the performance of garbage collection operations in the IBU scheme under program installation workload and IOZone is that the IBU serves compressed update data requests as the in-block requests, which causes increased switch merge operations in such write-intensive workloads. These factors reduce the number of garbage collection operations and reduce the overhead caused by valid data migration.

### C. EVALUATION OF WRITE LATENCY AND READ LATENCY

Figure 9 shows a comparison of write latency between IBU Mapping, ZFTL, and LDC schemes. For the Windows workload, the write-latency in the IBU Mapping scheme is about 3.2% lower than with the ZFTL scheme, and 1.3% lower than with the LDC scheme. However, the write-latency in the IBU compared to the LDC scheme was reduced by about 4% for the Program Installation workload and by about 3.3% for the IOZone workload. Generally, the proposed scheme has less write-latency in the Program Installation and IOZone workloads than in the ZFTL and LDC schemes, but there is no significant difference in the Windows workload. Figure 9 shows that for workloads with higher data compression rates, the IBU Mapping scheme has less write-latency than other schemes. For Program Installation and IOZone workloads, the write-latency in the IBU Mapping scheme is lower than in the LDC scheme, due to less overhead of the garbage collection operations when executing write requests. The ZFTL scheme also has the highest write latency, mainly due to the overhead for compressing the incompressible data and doing more garbage collection operations when writing new requests.

Figure 10 shows the read latency for IBU Mapping, ZFTL, and LDC schemes. Generally, for the Windows workload, Program Installation and IOZone, the LDC scheme has the highest read-latency compared to other schemes. Because this scheme writes compressed data at the boundary of physical pages during garbage collection, and to read some logical
pages, it needs access to two physical pages. But the IBU scheme has the best performance for Program Installation and IOZone workloads. For example, the IBU scheme reduces the read-latency by about 2.3% and 3.2% compared to the LDC scheme for Program Installation and IOZone workloads, respectively. Analysis of read-latency and write-latency diagrams, as well as garbage collection time diagrams, show that for heavy workloads, as the time overhead of garbage collection operations is lower, the read and write latency is reduced accordingly.

D. EVALUATION OF WRITE AMPLIFICATION FACTOR (WAF)
The Write Amplification Factor (WAF) is an important metric used to quantify the efficiency of various storage drives management schemes. The lower the WAF value, the less write overhead is imposed on the flash memory and therefore, the proposed FTL is more efficient. Although WAF is 1 for a hard disk (due to its in-place write capability), this amount can be increased up to 10 in flash memory. Data compression at the level of flash memory can significantly reduce the WAF value but will never be equal to 1. Figure 11 shows the WAF parameter value for ZFTL, LDC, and IBU schemes. This figure shows that for program installation and IOZone workloads, the IBU scheme has a lower WAF than the ZFTL and LDC schemes due to reduced numbers of garbage collection operations and the ability to perform large amounts of switch-merge operations. Evaluations of these Workloads show that the WAF value in the proposed IBU scheme is about 1% less than with the LDC scheme.

E. THE REQUIRED MEMORY FOR ADDRESS MAPPING
The most important feature of the proposed scheme is a significant reduction in the amount of memory required for the address mapping table compared to similar designs. As shown in Figure 12, the mapping memory in the ZFTL and LDC schemes contains the memory required for the Page Map Table and Page State Table tables. This value for ZFTL and LDC schemes is 10 MB, while the mapping memory for the IBU Mapping scheme is 2.125 MB. As a result, the amount of mapping memory in the proposed scheme is reduced by about 78%. This significant reduction obtained with the proposed design is due to changes in the structure of the mapping table.

Indeed, the mapping table of IBU takes up considerably less flash memory space than other similar methods. For example, in ZFTL and LDC, when a 1 TB flash-based SSD is used, the mapping table takes 1280 MB of memory space, while only 268 MB of memory space is occupied using the proposed FTL.

\[
Block Number = \left( \frac{\text{SSD Capacity}}{(\text{Page Per Block}) \times (\text{Page Capacity})} \right)
\]

\[
Block Number \text{ for } 1\ TB = \left( \frac{2^{40}}{2^7 \times 2^{24}} \right) = 2^{19}
\]

\[
Page Number = \frac{(\text{SSD Capacity})}{\text{Page Capacity}}
\]

\[
Page Number \text{ for } 1\ TB = \left( \frac{2^{40}}{2^{24}} \right) = 2^{26}
\]

\[
PMT \text{ Size in IBU} = \text{Block Number} \times \left( \frac{(\text{Data Block Address})}{\text{Page Per Block}} \right) + \left( \frac{(\text{Log Block Address})}{\text{Page Per Block}} \right) + \left( \frac{(\text{Page Offset Addresses})}{\text{Page Capacity}} \right)
\]

\[
PMT \text{ Capacity for IBU (1TB)} = 2^{19} \times (3 + 3 + (128 \times 1) \text{ bytes}) = 67\text{MB}
\]

\[
PMT \text{ capacity in ZFTL} = \text{Page number} \times (\text{Page Addresses})
\]

\[
PMT \text{ capacity in ZFTL} = \frac{1}{4} \times (\text{Page Capacity})
\]

\[
PMT \text{ capacity for ZFTL (1TB)} = 2^{26} \times 4\text{bytes} = 256\text{MB}
\]

\[
PST \text{ Capacity for ZFTL (1TB)} = \frac{1}{4} \times (256\text{MB}) = 64\text{MB}
\]

\[
Total \text{ RAM Capacity for ZFTL (1TB)} = 256 + 64 = 320\text{MB}
\]

F. THE IMPACT OF THE WORKLOAD CHARACTERISTICS ON THE COMPRESSION PERFORMANCE OF
Table 1 shows that the compression rate is low under some workflows, such as Windows, while access patterns tend to be
sequential. This lends the compression rate prediction module a helping hand to detect which requests are incompressible competently. As a result, a time-consuming compression operation will not be applied to a considerable number of requests.

Workloads containing binary data, such as IOZONE and Program Installation, can be compressed to a substantially small size. In the proposed method, the more sequential requests these workloads generate, the more logical pages can be placed in a single physical block, leaving more free storage space. This free space can be filled with data from incoming update requests, which can postpone garbage collections. Moreover, sequential data will be invalidated around the same time. Therefore, the small number of valid pages in such physical blocks will reduce the garbage collection overhead.

It should be noted that the proposed method, along with other similar methods, uses logical page-sized compression units. Therefore, access patterns and the size of the requests do not impact the compression rate.

V. CONCLUSION AND FUTURE WORK

Due to the inherent features of NAND flash memories, they have a limited lifespan and their physical pages are not capable of in-place updates. To resolve these limitations, a firmware called the Flash Translation Layer (FTL) is used. Among the FTL components, the address translation module and the garbage collection module have a significant impact on the performance of flash memories, and much of the previous research has focused on improving these two modules. The large size of address mapping tables and the need for sophisticated garbage collection operations are major disadvantages of earlier designs.

In this paper, a new Flash Translation layer (called IBU Mapping) is proposed for NAND Flash memories that support data compression at the FTL level. In this scheme, the new hybrid address translation unit that was introduced reduces significantly the amount of memory required for the address mapping table (78% reduction over previous schemes). Evaluation of the proposed method by standard workloads showed that the write delay on the IBU scheme has improved by an average of 1.86% compared to the best previously reported comparable scheme. Also, the time overhead for garbage collection operations in the IBU scheme has improved by an average of about 1.66% compared to the best previously reported related scheme. The most important advantage of the IBU scheme over previous drive management schemes is in the amount of memory required for the address mapping table, making it suitable for use on External SSDs and SSDs for PCs.

As the metadata of the mapping tables is stored in RAM, it is extremely important to have a solution in case of sudden power failures in flash memories. Providing efficient methods for backing up metadata as well as a fast system recovery after a sudden power failure are needed. These issues were left for future works.

REFERENCES

[1] A. Tavakkol, M. Arjomand, and H. Sarbazi-Azad, “Design for scalability in enterprise SSDs,” in Proc. 23rd Int. Conf. Parallel Architectures Computation, Aug. 2014, pp. 417–429.
[2] A. Tavakkol, M. Arjomand, and H. Sarbazi-Azad, “Network-on-SSD: A scalable and high-performance communication design paradigm for SSDs,” IEEE Comput. Archit. Lett., vol. 12, no. 1, pp. 5–8, Jan. 2013.
[3] C. W. Tsao, Y. H. Chang, M. C. Yang, and P. C. Huang, “Efficient victim block selection for flash storage devices,” IEEE Trans. Comput., vol. 64, no. 12, pp. 3444–3460, Dec. 2015.
[4] A. S. Ramasamy and P. Karanaraj, “RFFE: A buffer cache management algorithm for flash-memory-based SSD to improve write performance,” Can. J. Electr. Comput. Eng., vol. 38, no. 3, pp. 219–231, 2015.
[5] D. Kim, Y. Won, J. Cha, S. Yoon, J. Choi, and S. Kang, “Exploiting compression-induced internal fragmentation for power-off recovery in SSD,” IEEE Trans. Comput., vol. 65, no. 6, pp. 1720–1733, Jun. 2016.
[6] M.-C. Yang, Y.-H. Chang, T.-W. Kuo, and P.-C. Huang, “Capacity-independent address mapping for flash storage devices with explosively growing capacity,” IEEE Trans. Comput., vol. 65, no. 2, pp. 448–465, Feb. 2016.
[7] 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 Proc. 9th USENIX Conf. File Storage Technol., San Jose, CA, USA, Feb. 2011, p. 6.
[8] Z. Xu, R. Li, and C.-Z. Xu, “CAST: A page-level FTL with compact address mapping and parallel data blocks,” in Proc. IEEE 31st Int. Perform. Comput. Commun. Conf. (IPCCC), Dec. 2012, pp. 142–151.
[9] Y. Park and J.-S. Kim, “zFTL: Power-efficient data compression support for NAND flash-based consumer electronics devices,” IEEE Trans. Consumer Electron., vol. 57, no. 3, pp. 1148–1156, Aug. 2011.
[10] T. Park and J.-S. Kim, “Compression support for flash translation layer,” in Proc. Int. Workshop Softw. Support Portable Storage, Oct. 2010, pp. 19–24.
[11] D. Ma, J. Feng, and G. Li, “A survey of address translation technologies for flash memories,” ACM Comput. Surv., vol. 46, no. 3, pp. 1–39, Jan. 2014.
[12] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, “A survey of flash translation layer,” J. Syst. Archit., vol. 55, nos. 5–6, pp. 332–343, Mar. 2009.
[13] S.-W. Lee, W.-K. Choi, and D.-J. Park, “FAST: An efficient flash translation layer for flash memory,” in Proc. Int. Conf. Embedded Ubiquitous Comput., 2006, pp. 879–887.
[14] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, “LAST: Locality-aware sector translation for NAND flash memory-based storage systems,” ACM SIGOPS Operating Syst. Rev., vol. 42, no. 6, pp. 36–42, Oct. 2008.
[15] A. Gupta, Y. Kim, and B. Urgaonkar, “DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings,” ACM SIGARCH Comput. Archit. News, vol. 37, no. 1, pp. 229–240, Mar. 2009.
[16] Z. Qin, Y. Wang, D. Liu, and Z. Shao, “A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems,” in Proc. 17th IEEE Real-Time Embedded Technol. Appl. Symp., Chicago, IL, USA, Apr. 2011, pp. 157–166.
[17] C. Ji, L.-P. Chung, L. Shi, C. Gao, C. Wu, Y. Wang, and C. J. Xue, “Lightweight data compression for mobile flash storage,” ACM Trans. Embedded Comput. Syst., vol. 16, no. 5s, pp. 1–18, Oct. 2017.
[18] G. Hansel, D. Perrin, and I. Simon, “Compression and entropy,” in Proc. 9th Annu. Symp. Theor. Aspects Comput. Sci. (STACS), in Lecture Notes in Computer Science, vol. 577, A. Finkel and M. Jantzen, Eds. Cachan, France: Springer, Feb. 1992, pp. 515–528.
[19] B. Mao, S. Wu, H. Jiang, Y. Yang, and Z. Xi, “EDC: Improving the performance and space efficiency of flash-based storage systems with elastic data compression,” IEEE Trans. Parallel Distrib. Syst., vol. 29, no. 6, pp. 1261–1274, Jun. 2018.
[20] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog, “Fileprints: Identifying file types by n-gram analysis,” in Proc. 6th Annu. IEEE Syst. Man Cybern. (SMC) Inf. Assurance Workshop, Jun. 2005, pp. 64–71.
[21] M. Karresand and N. Shahrmehr, “File type identification of data fragments by their binary structure,” in Proc. IEEE Inf. Assurance Workshop, Jun. 2006, pp. 140–147.
[22] W. C. Calhoun and D. Coles, “Predicting the types of file fragments,” Digit. Invest., vol. 5, pp. S14–S20, Sep. 2008.
[23] R. G. Taghizadeh, R. G. Taghizadeh, F. Khakpash, M. Binesh Marvasti, and S. A. Asghari, “CA-Dedupe: Content-aware deduplication in SSDs,” J. Supercomput., vol. 76, no. 11, pp. 8901–8921, Nov. 2020.
[24] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” IEEE Trans. Inf. Theory, vol. IT-23, no. 3, pp. 337–343, May 1977.

[25] M. Nazari, R. T. Zadeh, S. A. Asghari, M. B. Marvasti, and A. M. Rahmani, “FRCD: Fast recovery of compressible data in flash memories,” Comput. Electr. Eng., vol. 78, pp. 520–535, Sep. 2019.

[26] Accessed: Jan. 12, 2021. [Online]. Available: http://www.iozone.org/docs/IOzone_msword_98.pdf

[27] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, “Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity,” in Proc. ICS, vol. 11, May/Jun. 2011, pp. 96–107.

[28] X. Kong, Y. Yao, N. Gu, W. Feng, and X. Xu, “Wear-aware out-of-order dynamic scheduling for NAND flash-based consumer electronics,” IEEE Trans. Consumer Electron., vol. 67, no. 2, pp. 149–157, May 2021.

[29] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate lossless coding,” IEEE Trans. Inform. Theory, vol. 24, pp. 530–541, Sep. 1978.

[30] Y. Pan, Y. Li, H. Zhang, H. Chen, and M. Lin, “GFTL: Group-level mapping in flash translation layer to provide efficient address translation for NAND flash-based SSDs,” IEEE Trans. Consumer Electron., vol. 66, no. 3, pp. 242–250, Aug. 2020.

[31] H. Yong, J. Lee, and J.-S. Kim, “Design and implementation of virtual stream management for NAND flash-based storage,” IEEE Trans. Consumer Electron., vol. 67, no. 2, pp. 149–157, May 2021.

REZA GHOZAMI TAGHIZADEH received the M.Sc. degree in computer engineering (computer architecture major) from the Department of Electrical and Computer Engineering, Kharazmi University, Tehran, Iran, in 2018. His current research interests include data storage systems, software architecture, embedded system design, and operating systems.

MOHAMMADREZA BINESH MARVASTI received the M.Sc. degree from the Department of ECE, University of Tehran, Iran, in 2007, and the Ph.D. degree in ECE from McMaster University, Canada, in 2013. He worked as a Faculty Member with the Department of Electrical and Computer Engineering, Kharazmi University. His research interests include computer architecture, low-power digital design, FPGAs, approximate computing, and on-chip interconnection networks.

SEYYED AMIR ASGHARI received the B.Sc. degree in hardware engineering and the M.Sc. and Ph.D. degrees in computer architecture from the Amirkabir University of Technology, in 2007, 2009, and 2013, respectively. He worked as a Faculty Member with the Department of Electrical and Computer Engineering, Kharazmi University. His current research interests include fault-tolerant design and real-time embedded system design.

RAMIN GHOLAMI TAGHIZADEH received the M.Sc. degree in computer engineering (artificial intelligence major) from the Department of Electrical and Computer Engineering, Kharazmi University, Tehran, Iran, in 2019. His current research interests include data storage systems, software architecture, operating systems, and data mining.

MORTEZA NABAVI received the B.S. degree in computer engineering from the Amirkabir University of Technology, Tehran, Iran, in 2008, the M.A.Sc. degree in computer engineering from Boston University, Boston, MA, USA, the M.A.Sc. degree in electronics from Carleton University, Ottawa, ON, Canada, in 2012, and the Ph.D. degree from the University of Waterloo, Waterloo, ON, Canada. He joined the Research and Development Group, Sidense Company, Ottawa, where he is currently involved in designing subthreshold one-time programmable (OTP) memories. His current research interests include subthreshold circuits and memories.

YVON SAVARIA (Fellow, IEEE) received the B.Eng. and M.Sc.A. degrees in electrical engineering from the École Polytechnique Montreal, in 1980 and 1982, respectively, and the Ph.D. degree in electrical engineering from McGill University, in 1985. Since 1985, he has been with Polytechnique Montréal, where he is currently a Professor with the Department of Electrical Engineering. He is also affiliated with the Hangzhou Innovation Institute, Beihang University. He has carried out work in several areas related to microelectronic circuits and microsystems, such as testing, verification, validation, clocking methods, defect and fault tolerance, effects of radiation on electronics, high-speed interconnects and circuit design techniques, CAD methods, reconfigurable computing and applications of microelectronics to telecommunications, aerospace, image processing, video processing, radar signal processing, and the acceleration of digital signal processing. He is also involved in several projects related to embedded systems in aircraft, radiation effects on electronics, asynchronous circuit design and testing, green IT, wireless sensor networks, virtual networks, software-defined networks, machine learning, computational efficiency, and application-specific architecture design. He holds 16 patents, has published 185 journal articles and 470 conference papers, and was the thesis advisor of 170 graduate students who completed their studies. He has been working as a Consultant or was sponsored for carrying out research by Bombardier, Ciena, CNRC, Design Workshop, Dolphin DREW, Ericsson, Genesis, Gerunum, Huawei, Hyperchip, Intel, ISR, Kaloom, LTRIM, Miranda, MiroTech, Nortel, Noviflow, Octasic, PMC-Sierra, Technocap, Thales, Tundra, and Wavelite. He is a fellow of the Canadian Academy of Engineering. He is a member of the Regroupement Stratégique en Microélectronique du Québec (RESSMQ) and of the Ordre des Ingénieurs du Québec (OIQ). He is a member of the CMC Microsystems Board. In 2001, he was awarded a Tier 1 Canada Research Chair (www.chairs.gc.ca) on the designs and architectures of advanced microelectronic systems that he held until June 2015. He also received a Synergy Award of the Natural Sciences and Engineering Research Council of Canada in 2006. He was a Program Co-Chairperson of NEWCAS’2018.

***