Co-Optimizing Storage Space Utilization and Performance for Key-Value Solid State Drives

Yen-Ting Chen, *Student Member, IEEE*, Ming-Chang Yang, *Member, IEEE*, Yuan-Hao Chang, *Senior Member, IEEE*, Tseng-Yi Chen, *Member, IEEE*, Hsin-Wen Wei, *Member, IEEE*, and Wei-Kuan Shih, *Member, IEEE*

***Abstract*—Growing demand for key-value store applications is building a strong momentum for the commercialization of key-value hard disk drives. To achieve better performance, flash-based solid state drive is the next ideal candidate for commercialization in the foreseeable future. However, the existing fixed-sized management strategies of flash-based devices would potentially result in low storage space utilization when managing variable-sized key-value data. In addition, the low storage space utilization would further lead to the degradation of device performance, due to low invalid data space reclamation efficiency. The space utilization issue motivates this paper to propose a key-value flash translation layer design to improve storage space utilization as well as the performance of the key-value solid state drives. A series of experiments was conducted to evaluate the proposed design, and the experiment results of space utilization and device performance are very encouraging.**

***Index Terms*—Flash memory storage systems, flash storage device, key-value store, performance, storage space utilization.**

I. Introduction

RECENTLY, key-value store techniques have gained popularity in large-scale and data-driven storage applications. Under this trend, we observe not only the considerable emergence of different key-value store implementations but also the development of specific hard disk drives (HDDs) for key-value store applications. In the meantime, in order to achieve a higher degree of performance and satisfy the demands for high-performance applications, we can predict that flash-based solid state drives (SSDs) specifically designed for key-value store applications would also be commercialized in a foreseeable future. Nevertheless, when storing the variable-sized key-value pairs into flash memory, the existing fixed-sized mapping techniques would potentially result in very low storage space utilization, especially when the average size of key-value pairs is usually no longer than a few hundred bytes [1]. To tackle this low space utilization issue, we propose a new key-value store aware flash management design for future key-value flash storage devices. We place special emphasis on devising a new space management strategy to store the variable-sized key-value pairs into the fixed-sized flash pages, with the goal to ultimately co-optimize the storage space utilization and device performance.

In a flash drive, a block is the basic unit to perform the erase operation over flash memory, and each block is usually made up of hundreds of pages, each of which is the basic unit of read/write operations. Owing to the *write-once property*, a programmed/written page can not be overwritten before its residing block is erased. Due to the property of flash pages, the updated data are usually stored into other free pages. Pages that have not been programmed (since their residing block was erased) are commonly referred to as *free pages*, while pages containing the up-to-date (resp. out-of-date) version of data are usually called live/valid pages (resp. dead/invalid pages). Therefore, in order to manage data scattered in different flash pages/blocks, a flash device usually incorporates a management software, namely, *flash translation layer* (FTL), to perform the *address translation* from the logical addresses referenced by the host to the physical addresses in flash memory. Moreover, FTL also incorporates a *garbage collector* to recycle the space occupied by the invalid data. In other words, FTL software plays an important role in flash management, so that the flash device can behave like a block-based device.

Many FTL designs have been proposed to resolve various design issues. In particular, address translation strategies have drawn the most attentions to achieve a good balance between the device performance and the required RAM space [2], [3], [10], [13], [15], [23], and most of the existing address mapping designs manage the storage space of a flash device based on the fixed-sized flash blocks/pages. On the other hand, the key-value store technique has recently been widely applied to various large-scale, data-driven applications, such as Google's BigTable [5] and Amazon's Dynamo [8]. However, since the key-value store adopts a new data access model (i.e., API), the size of the to-be-stored and to-be-retrieved data can be variable. Notably, the key-value interface is quite different from the block-oriented interface, in which data can be only accessed based on the fixed-sized blocks/sectors. Accordingly, storage devices supporting key-value interface have attracted lots of attention from the storage industry [22]. The first commercialized product, namely Kinetic Drives [20], is produced by Seagate and is based on the

Manuscript received July 10, 2017; revised October 7, 2017 and January 10, 2018; accepted January 10, 2018. Date of publication February 2, 2018; date of current version December 19, 2018. This work was supported in part by Ministry of Science and Technology under Grant 105-2221-E-001-013-MY3, Grant 105-2221-E-001-004-MY2, Grant 106-3114-E-002-008, Grant 105-2221-E-007-066-MY2, and Grant 105-3111-Y-001-041. This paper was recommended by Associate Editor Z. Shao. *(Corresponding author: Yuan-Hao Chang.)*

Y.-T. Chen and W.-K. Shih are with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan.

M.-C. Yang is with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong.

Y.-H. Chang, and T.-Y. Chen are with the Institute of Information Science, Academia Sinica, Taipei 115, Taiwan (e-mail: johnson@iis.sinica.edu.tw; tsengyi@iis.sinica.edu.tw).

H.-W. Wei is with the Department of Electrical and Computer Engineering, Tamkang University, New Taipei 25137, Taiwan.

Digital Object Identifier 10.1109/TCAD.2018.2801244

traditional hard disk technology. Expectedly, the observed success of the hard disk based key-value devices would also spur the commercialization of key-value flash drives [i.e., flash-based solid-state drive (SSD)] supporting key-value interface] in the near future. However, to the best of our knowledge, only few researches [16], [18], [19], [27], which consider the flash memory as the main storage media, have been proposed to improve the performance of key-value store applications based on the special characteristics of flash memory. It is worth noting that these researches were merely proposed to redesign the storage management to fully utilize the characteristics of raw flash memory chips, but they do not consider the variable-sized key-value pairs as the directly access interface for SSDs to the host system [12], [17], [24], [26]. That is, little work was proposed to introduce *a new flash management software embedded in a key-value-specific flash-based solid state drive*.

This paper proposes a key-value FTL (KVFTL) design to co-optimize the storage space utilization and device performance for the flash-based key-value SSDs (KVSSDs). Notably, the low space utilization problem is mainly caused by storing variable-sized key-value pairs into fixed-sized flash pages with the existing FTL designs. Moreover, the low space utilization would also make the whole SSD be occupied with wasted storage space and trigger the SSD to reclaim the space of invalid data much more frequently. As a result, the device performance would be inevitably degraded because of the low efficiency on recycling the wasted storage space by the garbage collector. Thus, to optimize the storage space utilization, KVFTL introduces a brand new storage concept, called *partition*, which creates different sized storage units (from the fixed-sized flash pages) to store variable-sized data. Through slicing the value into multiple different-sized partitions and storing/putting them into the flash device according the partition size, the proposed KVFTL could achieve the optimal storage space utilization, namely 100%. However, reading/getting the value from these multiple partitions in distinct flash pages could potentially lead to the read amplification issue and result in lower read performance. Therefore, we further proposed three revised mechanisms based on KVFTL to trade storage space for lower read amplification and higher device performance. In this paper, wear-leveling is outside the scope of discussion because the primary focus of the proposed design aims to address the low space utilization and performance issue of flash devices when supporting key-value store applications. Furthermore, workloads of key-value store applications are usually read-dominant and as such have lower urgency when it comes to considering the lifetime of the flash devices.

To evaluate the capability of KVFTL, a series of experiments were conducted. The results show that the proposed design not only achieves extremely high storage space utilization, but also has great flexibility to trade little storage space for device performance. Specifically, the proposed design can achieve 93% of the storage space utilization while effectively reduce the total execution time by 31% when compared to the traditional page-based FTL designs.

The rest of this paper is organized as follows. Section II presents the motivation. Section III presents the new KVFTL, to optimize the storage space utilization for key-value flash storage devices. Section IV further investigates the potential read amplification issue of KVFTL, and puts forward three new strategies to alleviate the read amplification issue of KVFTL by trading little space storage utilization for the improved device performance. Section V presents the experimental results, and Section VI gives more details to compare the differences between existing approaches and our design. Finally, we concludes this paper in Section VII.

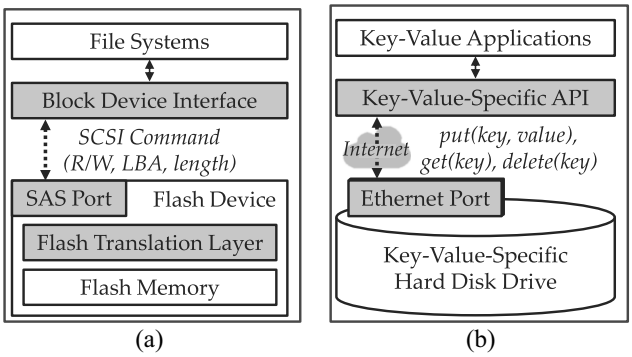

Fig. 1. Architectures of (a) block-based and (b) key-value-based devices.

II. Background and Motivation

The block device interface has led the storage industry for several decades from the tape drives, HDDs, and CD-ROMs, to nowadays flash-based SSDs. As shown in Fig. 1(a), the block device interface enables the *initiator* (i.e., the host system or the computer) to send read/write commands (e.g., small computer system interface (SCSI)[1] commands [28]) to the *target* storage device based on the logical block addresses (LBAs), where the size of an LBA is fixed and is usually 512 bytes [4]. In addition, the storage devices are usually physically connected/attached through some standard connection ports such as serial attached SCSI (SAS).[2] On the other hand, as depicted in Fig. 1(a), in order to make a flash-based device behave like a regular block device, a software-based management layer, namely, FTL, should be incorporated to overcome the write-once property of flash pages. That is, a flash page cannot be directly overwritten unless its residing block is erased first. Notably, a flash page is the basic operational unit of read/write operations; while a flash block is the basic operational unit of erase operations.

The key-value store technique has recently gained popularity in large-scale and data-driven storage applications. Under this trend, we observe not only the considerable emergence of different key-value store implementations [5], [8], [14] but also the commercialization of HDDs made specifically for key-value store applications (e.g., Kinetic Open Drives [20]). In a key-value store, data are stored as key-value pairs. Each key-value pair is stored as an arbitrary sequence of bytes and can be of arbitrary length; each key-value pair is associated with a *key* and its content is referred to as *value*, so that the content of a key-value pair can be accessed/retrieved by using the associated key. This brand new data access model brings not only the advantages on scalability and the flexibility to handle key-value store applications, but also some fundamental changes to the system architecture and application behavior. First, as

[1]SCSI is a set of standards for physically connecting and transferring data between computers and peripheral devices.

[2]SAS is a point-to-point serial protocol that moves data to and from computer storage devices.

shown in Fig. 1(b), a key-value storage device can be directly attached through the standard Ethernet port, rather than the SAS port for conventional block devices. This allows the key-value storage devices to be easily deployed over the network or Internet, and to easily build up a large-scale key-value store. Moreover, realizing the key-value store management at the device level can also avoid and simplify the software design of the host side, since under this kind of architecture, the key-value store applications can directly communicate with key-value storage devices. Second, the key-value storage devices can be directly accessed by the applications through the key-value API (e.g., put, get, and delete operations) based on the (key, value) pair, rather than the LBA-based read/write commands in block interface. Third and the most importantly, the accessed value from/to a key-value storage device can be variable length (varied between 0 bytes and 1 MiB [20]), while the size of the read/written data of a block device could be only multiple times of the size of a logical block (e.g., 512 B).

In order to further achieve a higher degree of performance and satisfy the demands of high-speed key-value applications, the commercialization of key-value SSDs seems inevitable. However, according to our investigation, the variable-length data access model of key-value stores/pairs might result in low storage space utilization in the flash-based storage devices. The rationale behind this is that most of the existing space management strategies of flash devices are based on flash pages, so that data could be only accessed/stored in the unit of a flash page. Nonetheless, key-value store applications would store variable-sized values (i.e., key-value pairs) into the storage devices. According to the workload analysis presented in [1], the average value size of key-value pairs in a large-scale key-value store is just few hundred bytes. This size is smaller than the size of a sector (e.g., 512 B). That is, the average value size in realistic key-value store applications could be much smaller than the flash page size (e.g., 4 KB or 16 KB). Furthermore, the flash page size for the next-generation flash memory [9] is on a continuously expanding trajectory (e.g., 32 KB and even 64 KB) to accommodate larger storage capacity. Consequently, *storing the variable and relatively small size of values into the fixed and relatively large size of flash pages would result in an extremely low storage space utilization*. This phenomenon would also affect the efficiency of reclaiming storage space from invalid data and trigger the garbage collection much more frequently. As a result, the device performance would degrade severely because of the long latency to copy live pages and erase blocks.

The low storage space utilization and the performance problem of the key-value flash storage devices inspire this paper to put forward a new space management strategy for the variable-sized values of key-value pairs in key-value store applications. Our goal is to present a new *key-value store aware* FTL design to store variable-sized values into fixed-sized flash pages of key-value flash storage devices, so as to ultimately improve storage space utilization and the overall device performance. The key technical challenge here is how to tackle the internal fragmentation of flash pages caused by the invalidation of key-value pairs, so that the efficiency of space management and garbage collection can be improved. To the best of our knowledge, this paper is one of the pioneers to present the new type of FTL design for key-value flash devices in order to co-optimize the space utilization and device performance.

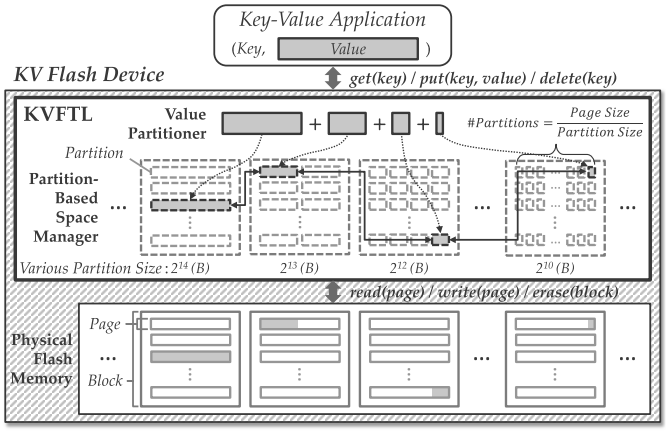

Fig. 2. Overview of KVFTL.

III. Key-Value Flash Translation Layer

A. Overview

In this section, a new *key-value flash translation layer design* (referred to as KVFTL) is proposed to improve the storage space utilization of storing variable-sized key-value pairs into fixed-sized flash pages for key-value flash devices (such as KVSSDs). In this section (i.e., Section III), we will put more focus on optimizing the storage space utilization for KVSSDs. Based on this design, we will further investigate how to trade little storage space utilization for the improved device performance in the next section (i.e., Section IV). Notably, the low storage space utilization problem of the existing FTL designs mainly results from the wasted storage space when the variable-sized key-value pair is directly stored into the fixed-sized flash pages, where the sizes of key-value pairs are usually relatively small or non-integer multiple of the page size. In particular, according to the development trend of the next generation flash memory chips, the storage space utilization issue could get even worse because the size of flash pages still keeps increasing [9]. The proposed KVFTL design is specifically tailored for the coming key-value flash devices. Toward that end, our design supports the novel key-value store APIs (such as get, put, and delete operations) rather than the conventional LBA-based read/write requests.

As shown in Fig. 2, our key idea is to dynamically create different sizes of new storage units, namely *partitions*, out of the fixed-sized flash pages. As a result, the variable-sized key-value pairs can be accordingly partitioned, and the partitioned value of any given key can be stored into multiple partitions of different sizes (in the form of a "linked chain") without sacrificing the storage space utilization. When a value is going to be put into the flash device, the value partitioner would be in charge of the process for partitioning. Partitioning of the value would slice this value into different-sized partitions and separate these partitions into distinct flash pages across different blocks according to their partition size. Thus, in order to efficiently manage the storage space comprised of different sizes of partitions, Section III-C shall introduce a *partition-based space manager* to present how to store the partitioned values in different sizes of partitions, and how to dynamically allocate/reclaim the storage space to proceed different types of key-value operations.

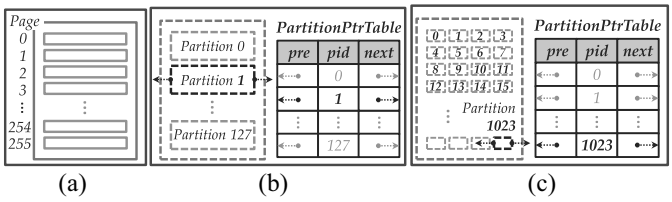

Fig. 3. Examples of *partitioned block* and *PartitionPtrTable*. (a) Flash block page size: 2^{14} *B*. Partitioned block partition size (b) 2^{15} *B* and (c) 2^{12} *B*.

B. New Storage Units: Various Sizes of Partitions

As shown in Fig. 3(a), a flash block is usually composed of hundreds of flash pages, and each flash page is of a fixed storage space, such as 16 KB. To avoid wasting storage space when storing variable-sized values, we propose to create a range of different sizes of storage units, namely *partitions*, out of the fixed-sized flash pages. The partition size is the powers of 2 (in bytes), because flash page size is usually a power of 2 [e.g., 2^{14} bytes in Fig. 3(a)]. In practice, the range of possible partition sizes could vary from 1 B to 1 MB (i.e., 2^0 to 2^{20} bytes), since the maximal value size of a key-value pair is at most 1 MB according to the specification of the key-value hard disk [20]. Furthermore, all the pages in the same flash block are partitioned and allocated into the same-sized partitions to ease the complexity of space management and garbage collection (please see Section III-C1 for details), since the garbage collection will recycle the flash space in the unit of a flash block (please see Section III-C2 for details). Moreover, as shown in Fig. 3(b) and (c), the partitions of the same flash block will be arranged in an ordered pattern (i.e., from left to right, and from top to bottom). Furthermore, if the partition size is larger than the page size, an (external) partition could be composed of multiple flash pages [as shown in Fig. 3(b)]; otherwise, a flash page will be equally partitioned into multiple (internal) partitions [as shown in Fig. 3(c)], and the number of internal partitions in a page is equal to the quotient of the page size divided by the partition size. That is, a *partition* can be the continuous storage space within or across flash pages.

To keep track of the key-value pairs stored in partitions, a *PartitionPtrTable* should be maintained for each *partitioned block*, where a partitioned block is a flash block whose pages have been partitioned into equal-sized partitions based on a given partition size. As shown in Fig. 3(b) and (c), the *PartitionPtrTable* has three fields: The "pid" field is used to indicate the ordinal number of a partition in the partitioned block; and the "pre" and "next" fields are two pointers to link the previous and the next partitions of the value of a key-value pair, respectively. Notably, the objective of the two pointers is to provide an efficient mechanism to retrieve the entire value scattered across multiple partitions of different partitioned blocks. As we mentioned in Section III-A, the variable-sized value will be partitioned and stored into multiple partitions. Hereafter, in the following examples, we will directly use a *PartitionPtrTable* to represent a partitioned block. Meanwhile, the *PartitionPtrTables* themselves, which contain information for partitioned blocks, are stored in an isolated area on the flash device, and can be loaded/stored on demand to limit the required run-time RAM space.

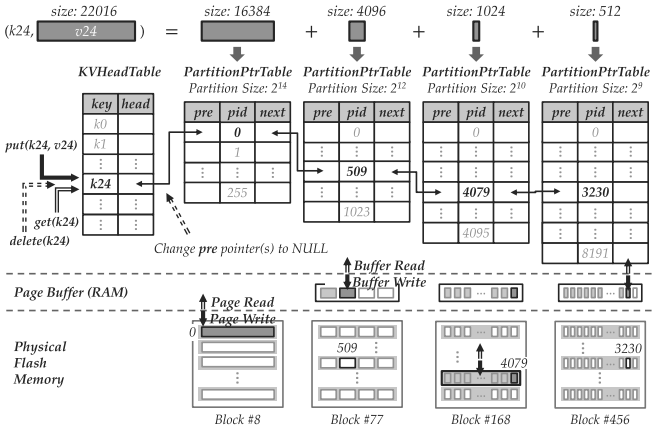

Fig. 4. Examples of *put()*, *get()*, and *delete()* operations.

C. Partition-Based Space Manager

Different from the traditional page-based space management, a new *partition-based space manager* is proposed to maintain the storage space consisting of different sizes of *partitions*. Section III-C1 will first present how to dynamically allocate different sizes of partitions to accommodate the variable-sized key-value pairs, and elaborate how to handle the three most important key-value operations (i.e., put, get, and delete operations). Then, Section III-C2 will introduce how to perform garbage collection over the "partitioned pages" to accommodate for more value updates.

1) Partition-Based Space Allocation [put(), get(), and delete() Operations]: This section will present how to allocate storage space, based on partitions, by elaborating on the detailed procedures of the three most important key-value operations (i.e., put, get, and delete).

a) put(key, value): The *put(key, value)* operation is the origin of all other operations, where a *key* is used to associate the to-be-stored *value*. As shown in Fig. 4, when a *put* operation asks to store the value *v*24 of a size equal to 22 016 B (and its associated key is *k*24) into a key-value flash device, a *value partitioner* will iteratively cut the received value into partition(s) of different partition sizes. Notably, in each iteration, the value partitioner should always slice the value into two parts. The larger part can be completely stored into a partition as large as possible without introducing any space waste, and the remaining part will be iteratively taken into the same partitioning process until the size of the smaller part is zero. For example, in the first iteration of value partitioning, the value *v*24 with a size of 22 016 B will be sliced into a larger part of 16 384 B in size and a smaller part of 5632 B in size. Then, following the same partitioning process, the value *v*24 with a size of 22 016 B will be eventually partitioned into four partitions of sizes equal to 16 384 B, 4096 B, 1024 B, and 512 B, without causing any space waste. In other words, *the value partitioner could achieve* 100% *storage space utilization, when the possible value sizes during the partitioning process are not smaller than the smallest partition size*.

Next, when a value is partitioned, the well-cut partitions should be stored into the partitioned blocks of the corresponding partition size. For example, the partition of size equal to 16 384 B should be stored into a partitioned block whose partition size is 2^{14} bytes. Furthermore, the free partitions of a partitioned block should be used from the first partition to the

last one; meanwhile, if all of the partitions are used, another free block should be allocated (and partitioned) to store the following partitions. However, due to the hardware constraint, data written to the same flash page should be programmed at the same time. To overcome this hardware constraint, a page buffer is needed for each partitioned block whose partition size is smaller than the page size. The rationale behind this is that data of partition size greater than and equal to the page size can be directly written into the flash pages without the help of page buffers. In fact, the required RAM space for page buffers is small, since there is at most one "running" partitioned block for each partition size, and the number of partition sizes, which are smaller than the page size, is also limited. For example, if the smallest partition size is set as 2^0 B and the page size is 2^{14} B, our design requires only 14 page buffers to collect data of the 14 partition sizes. Notably, for high integrity applications (i.e., database), the page buffers should be implemented by using DRAM with a super capacitor, so that the data stored in the page buffers can be safely flushed to non-volatile flash pages when unexpected error or power failure occurs. The objective of the page buffer is to postpone the actual physical write operation to a flash page until all of the partitions that will be written into the same flash page are received/collected in the page buffer. As a result, all the partitions buffered in the same page buffer can be thereby programmed into the same flash page at once. As the examples in Fig. 4 show, the partitioned data for partition 0 of partitioned block 8 will be directly written into the corresponding flash page(s) since the partition size is equal to (or larger than) the page size. Comparatively, the two partitions (whose partition sizes are smaller than the page size) for partition 509 of partitioned block 77 and partition 3230 of partitioned block 456 will be temporarily stored in their corresponding page buffers; meanwhile, the content of a page buffer will not be written into the corresponding flash page until the last partition (e.g., the data that is being stored in partition 4079 of partitioned block 168) for the same flash page is received.

Finally, to keep the integrity of the value and to ease the value retrieval, all the corresponding pre and next pointers of different *PartitionPtrTables* should be linked by the order of partition sizes. Notably, the next pointer of the last partition should point to NULL to indicate its corresponding partition as the last one. Furthermore, in Fig. 4, a *KVHeadTable* is maintained for all keys to keep track of the first partition of a value through the "head" field. In this paper, we assume the keys in KVFTL are of 16-bits and could be stored in a linear table. Therefore, we could search through the table from top to bottom to find a specific key in this table. Considering the RAM usage, this linear table is stored in the flash memory and will be loaded into the on-device RAM space on demand. In addition, to support keys with larger sizes (e.g., 64-bits), this linear table could be maintained as multi-level tables.

Note that a *put* operation to an existing key will be considered as an update. In particular, an update will first invoke the *delete* operation to invalidate the existing but out-of-date value, before storing the updated value into other free partitions. Then, the head pointer, corresponding to the updated key, in the *KVHeadTable* should be also updated to link to the first partition of the newly updated value. Moreover, the pre pointers for the out-of-date partitions in the *PartitionPtrTables* should be pointed to NULL to mark all out-of-date partitions as *invalid partitions*, to accelerate the garbage collection process.

Algorithm 1: PUT(KEY, VALUE)

```
   Input: (key, value): The to-be-stored key-value pair.
1  if the key already exists then
2  |  Invoke DELETE(key);
3  end
4  PartitionSize ← 2^MaxPartitionPower(bytes);
5  RemainingSize ← the total size of value ;
   // Iteratively partition the value
6  while RemainingSize is larger than zero do
7  |  if RemainingSize is larger than PartitionSize then
   |  |  // Partition the value
8  |  |  partition ← value[0 : PartitionSize − 1];
9  |  |  value ← value[PartitionSize : RemainingSize];
10 |  |  Decrease the RemainingSize by PartitionSize;
11 |  |  if this is the first partition then
12 |  |  |  Make double links between the "head" field in
   |  |  |  KVHeadTable and the first partition's "pre" field in
   |  |  |  PartitionPtrTable;
13 |  |  end
14 |  |  else
15 |  |  |  Make double links between the current partition's "pre"
   |  |  |  field in PartitionPtrTable and the previous partition's
   |  |  |  "next" field in PartitionPtrTable;
16 |  |  end
17 |  |  if PartitionSize is a multiple of the page size then
18 |  |  |  Perform page write(s);
19 |  |  end
20 |  |  else
21 |  |  |  Write this partition into the page buffer of the
   |  |  |  corresponding partitioned block;
22 |  |  |  if the page buffer of the corresponding partitioned block
   |  |  |  is full then
23 |  |  |  |  Perform page write;
24 |  |  |  end
25 |  |  end
26 |  |  if the corresponding partitioned block is full then
27 |  |  |  Allocate a new block for this partition size ;
28 |  |  end
29 |  end
30 |  Divide the PartitionSize by 2 ;
31 end
32 return Successful;
```

The detailed procedure of the *put* operation is summarized in Algorithm 1, where *(key, value)* denotes the to-be-stored key-value pair of the *put* operation. If this *key* has already existed on the device (line 1), a *delete* operation will be performed to invalidate the existing but out-of-date value (line 2). Otherwise, the *put* operation will be accomplished by partitioning/cutting the received value into partitions of different sizes iteratively (lines 6–30). Notably, the variables *PartitionSize* and *RemainingSize* are used to indicate the "partition size" that is currently tried to slice the *value* and the value's "remaining size" that is still waiting to be dissected, respectively. Meanwhile, the *PartitionSize* will be initiated as $2^{MaxPartitionPower}$ bytes (i.e., the maximal value size), and the *RemainingSize* shall be initially set as the total size of the received *value* (lines 4 and 5). In each iteration, if the value's remaining size (i.e., *RemainingSize*) is larger than the trying-to-cut partition size (i.e., *PartitionSize*) (line 7), the proposed *value partitioner* will cut the value's first *PartitionSize* bytes (i.e., $value[0 : PartitionSize - 1]$) as a *partition*; Subsequently, the remaining part (i.e., $value[PartitionSize : RemainingSize]$) will be treated as the *value* and the *RemainingSize* should be also decreased by *PartitionSize* accordingly. Moreover, the first partition of the received *value*, should have double links

Algorithm 2: GET(KEY)

```
Input: key: The key to retrieve the required value.
1  NextPointer ←"head" pointer of KVHeadTable[key] ;
2  if NextPointer IS NULL then
3  |   return Non-existent ;
4  end
5  else
6  |   while NextPointer is not NULL do
7  |   |   Perform page read to read out the pointed partition;
8  |   |   NextPointer ← "next" pointer of this partition in
   |   |   PartitionPtrTable ;
9  |   end
10 end
11 return Successful ;
```

Algorithm 3: DELETE(KEY)

```
Input: key: The key to delete.
1  NextPointer ←"head" pointer of KVHeadTable[key] ;
2  if NextPointer IS NULL then
3  |   return Fail ;
4  end
5  else
6  |   while NextPointer is not NULL do
7  |   |   Invalid this partition's "pre" pointer in PartitionPtrTable ;
8  |   |   NextPointer ← "next" pointer of this partition in
   |   |   PartitionPtrTable ;
9  |   end
10 end
11 return Successful ;
```

maintained between the head field in the *KVHeadTable* and the pre field of the corresponding *PartitionPtrTable* of this partition. (lines 11 and 12). For the following partitions, double links should be maintained in the pre field of this partition and the next field of previous partition in the corresponding *PartitionPtrTable* (lines 14 and 15). For the partitions whose sizes are the multiple of a flash page size, the partition will be directly written into flash page(s) (line 18); otherwise, partitions of a size smaller than a page should be temporarily cached/written into the page buffer (line 21), and only when the page buffer of the corresponding partitioned block is full should the flash page write be performed to physically write data into the flash device (lines 22 and 23). Then, a new block for this partition size should be allocated when the corresponding partitioned block is full (lines 26 and 27). At the end of each iteration, the *PartitionSize* will be divided by 2 so that the value can be iteratively partitioned into partitions of smaller sizes that are also power of two (line 30).

b) get(key): By traversing the maintained pointers and tables (i.e., the *KVHeadTable* and *PartitionPtrTables*), we can easily get the stored value associated with the given key. For example, in Fig. 4, the *KVHeadTable* is first looked up to find the head partition of the value associated with the given key *k*24. Then, we just need to read out all of the partitions linked by all next pointers of the related *PartitionPtrTables*, until we encounter a next pointer pointing to NULL. That is, the contents of partition 0 of block 8, partition 509 of block 77, partition 4079 of block 168, and partition 3230 of block 456 should be read out to have the complete value. Additionally, if the requested partition is still in the page buffer, the content should be directly read from the page buffer with a "buffer read" rather than from the corresponding flash page.

The procedure of the *get* operation is shown in Algorithm 2. Let *key* denotes the received key for retrieving the associated stored value. To iteratively read out all partitions of the associated value, the variable *NextPointer* is used to indicate the address of the next partition, and *NextPointer* should be initiated by looking up the *KVHeadTable* to get the head pointer for the given *key*, and thus retrieve the address of the first partition (line 1). Please note that if the head pointer points to NULL, it means that the value of the given key may be either deleted or has not been *put* yet (lines 2 and 3). Otherwise, we need to iteratively read out the partition pointed by the *NextPointer* from the corresponding flash page (lines 5–8). Notably, for each iteration, we need to visit the *PartitionPtrTable* of each partition, and set *NextPointer* as the next pointer of each partition until the last partition of the value is reached (line 8).

c) delete(key): The purpose of a *delete(key)* operation is to invalidate the stored value associated with the given key. Since it is more efficient to reclaim the invalid storage space of flash memory in the unit of a flash block, when performing a *delete* operation, we will only mark a partition as an *invalid partition* by pointing the corresponding pre pointer to NULL (similar to the update of a value). This does not immediate free up the space taken by the partitions, nevertheless, the actual storage space occupied by the invalid partitions will be reclaimed during the garbage collection process (please see Section III-C2).

The procedure of the *delete* operation is shown in Algorithm 3. Let *key* denotes the to-be-deleted key. The procedure of *delete* basically follows the steps of the *get* operation. The difference is that we do not need to access the physical flash memory of these partitions to invalidate them. Please note that we can perform the *delete* operation on the given key only if this key has been *put*. This means that the head pointer of the *KVHeadTable[key]* is not NULL (lines 1–3). Then, we only need to make each partition's pre pointer in its corresponding *PartitionPtrTable* point to NULL to invalidate it (lines 6–8). As a result, the garbage collection process can efficiently distinguish whether a partition is valid or not by checking the corresponding pre pointer of a partition.

2) Partition-Based Space Reclamation (Garbage Collection): Because of the write-once property of flash pages, updated data is usually stored in other free flash pages. Similarly, when more and more put and delete operations are processed, invalid partitions will be gradually accumulated in flash blocks/pages, and thus free partitions will be used up to store the newly or updated key-value pairs. Under this situation, garbage collection must be performed to reclaim the storage space occupied by the invalid partitions. However, the storage space of flash memory could only be recycled in the unit of a flash page, and be erased in the unit of a flash block. Thus, this section will present a *partition-based garbage collection* to bridge the gap between the partition sizes and flash block/page sizes, and to efficiently reclaim invalid partitions over flash blocks.

In order to avoid performing garbage collection over-aggressively, the partitioned-based garbage collection could be invoked only when the number of the remaining free blocks in the device is less than a predefined threshold. When the garbage collection is invoked, the first step is to find a worthwhile flash block (also referred to as *victim block*) to reclaim

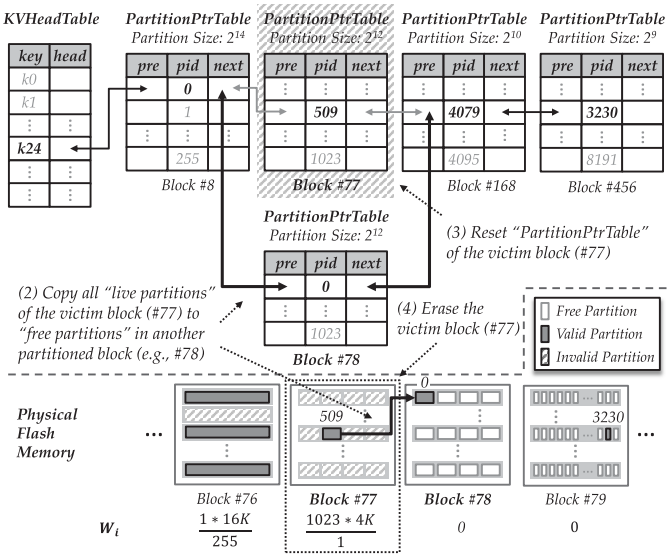

Fig. 5. Illustration of block-based garbage collection.

Algorithm 4: Partition-Based Garbage Collection

```
1  while The number of remaining free blocks is smaller than the
   predefined threshold;
2  do
3      Find a victim block to recycle ;
4      foreach partition(s) ∈ this victim block do
5          if the "pre" pointer of this partition in PartitionPtrTable is
           not NULL then
6              Re-put this partition into the currently-available
               partitioned block of the same partition size ;
7              Copy this partition's "pre" and "next" pointers to the new
               corresponding PartitionPtrTable;
8              Update the previous partition's "next" pointer;
9              if This partition's "next" pointer is not NULL then
10                 Update the next partition's "pre" pointer;
11             end
12         end
13     end
14     Erase the victim block ;
15 end
16 return Success ;
```

for achieving a better garbage collection efficiency. Therefore, in order to have a good way to determine the best victim block, we introduce a value W_i, as follows, to reflect the efficiency of recycling a $block_i$

$$W_i = \frac{\#\text{invalid partitions} \times \text{partition size of block}_i}{\#\text{required live page copyings}}. \quad (1)$$

The efficiency of recycling a block can be considered as how many live-page copyings we have to performed to reclaim the invalid space of that block. It is worth noting that a page should be regarded as a live-page as long as there is at least one valid partition in this page, while the number of required live-page-copyings can be considered as the time overheads to migrate all of the valid partitions of the recycled block into other free partitions. Therefore, a higher W_i reflects the better recycling efficiency because we consume less time on migrating valid partitions to reclaim more invalid space in $block_i$. As shown in the example of Fig. 5, among the four considered blocks 76–79, block 77, which reflects the largest W (e.g., $1023 \times 4\text{K} \div 1$), should be selected as the victim block for space reclamation.

Next, when a block is selected as a victim block, all the valid partitions in the victim block should be copied to other same-sized free partitions. By looking up the *PartitionPtrTable* of the victim block, we can easily distinguish valid partitions from invalid partitions, since the pre pointers of invalid partitions would point to NULL, as presented in Section III-C1. On the other hand, as Fig. 5 shows, the "live-partition-copyings" can be easily achieved by "re-putting" the valid partitions into the free partitions of the same-sized partitioned block (e.g., block 78), and then "re-pointing" the pre and next pointers to the newly partitioned block 78. For example, the valid partition (i.e., partition 509) in the victim block, block 77, will be copied into the first free partition (i.e., partition 0) of the same-sized partitioned block 78, and the corresponding pre and next pointers will be changed to ensure the partitioned values can be still linked in a chain. Finally, the *PartitionPtrTable* of the victim block should be reset and the victim block should be also erased to completely reclaim the victim block as a free one. Additionally, to consider hot/cold separation, KVFTL stores those "re-put" partitions separately from "newly-put" partitions.

The pseudo code of the *garbage collection* operation is shown in Algorithm 4. The garbage collection will be invoked only when the number of the remaining free blocks is less than a predefined value (line 1). After the garbage collection is invoked, we first find a flash block with the largest W_i, which is determined by our proposed strategy, as a *victim block* for space recycling (line 3). For each valid partition of this *victim block* (line 5), we will re-put it into the corresponding partitioned block with the same size that is currently available. To maintain the information of this partition, the pre and next pointers of this partition and it is previous partition's next pointer should be updated accordingly (lines 6–8). Also, if the next partition of this partition exists, it is pre pointer should be redirected to this partition (lines 9–10). After the procedure to preserve these valid partitions of this *victim block* is finished, an erase operation will be performed on this block to recycle the physical flash space (line 14).

IV. Co-Optimization of Storage Space Utilization and Device Performance

A. Overview

Although storing/putting the variable-sized value into multiple different-sized partitions optimizes the storage space utilization, reading/getting the value from these multiple partitions could potentially result in severe *read amplification*, which describes the phenomenon that the total size of the actual flash page reads is larger than the demanded value size. Basically, the read amplification of the proposed KVFTL design mainly comes from the significant number of partitions we sliced for a value. Since the smallest read unit of flash memory is a flash page, an entire page would inevitably be read even though we just want to read out a relatively smaller size of partition. Thus, this hardware characteristic of flash memory also worsens the read amplification. As shown in Fig. 6(a), the proposed KVFTL could achieve 100% storage space utilization when storing the value $v24$ of size equal to 22 016 B; however, when reading the value $v24$, a total four

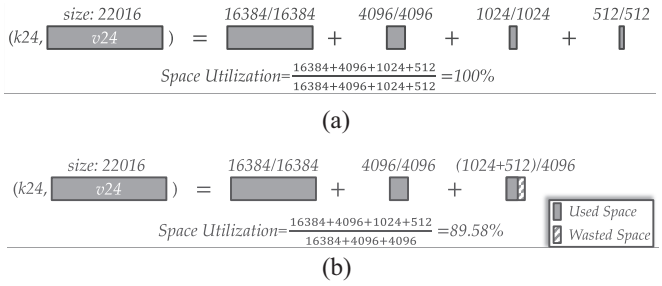

Fig. 6. KVFTL under different sizes of the smallest partition. (a) Smallest Partition Size: 2^0 *B*. (b) Smallest Partition Size: 2^{12} *B*.

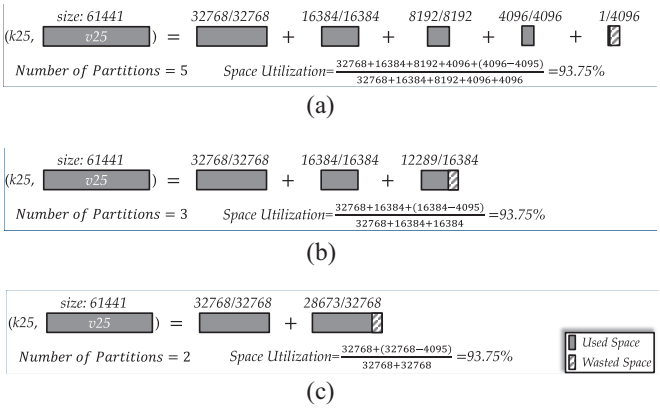

Fig. 7. KVFTL with three different read-intensive methods. (a) Smallest partition size: 2^{12} *B*. (b) Maximum number of partitions: 3. (c) Maximum wasted storage space: 2^{12} *B*.

flash page would be read, even though the size of the requested value is less than two flash pages.

The read amplification issue of the proposed KVFTL drives us to investigate the *tradeoff between the storage space utilization and the device performance, with the goal to ultimately co-optimize the storage space utilization and the device performance*. After all, even though the space utilization is one of the most important design issues for future heavily loaded storage applications, read performance is also an important design consideration, especially for those read-centric key-value store applications. In order to essentially alleviate the read amplification problem, Sections IV-B–IV-D further put forward three revised mechanisms, based on the KVFTL presented in Section III. Please note that, as we will see in the following sections, the three revised mechanisms will advocate different strategies to effectively *limit the number of partitions used for storing (i.e., slicing) a value by trading off a little storage space*. Although, the read performance improves with the conceding of some storage space, the performance of garbage collection and write operation will degrade accordingly because of the lower storage space utilization. However, Section V-B3 will further show that, by choosing proper configurations, these revised KVFTL design can be tailored for applications of different access patterns with encouraging performance results.

B. Restriction on the Smallest Partition Size

To avoid partitioning the value into too many smaller partitions, an intuitive strategy is to designate a configurable threshold size for restricting the smallest partition size. That is, *we will stop continuing to partition/slice a value when the size of the "non-partitioned part" of the value is less than the predefined smallest partition size*. Nevertheless, this strategy would waste the storage space that stores the "tail" of the value in the restricted smallest partition, since the size of the "value tail" could be less than the smallest partition size. Notably, the value tail is referred to as the last portion (of a value) remained in the value partitioning process, since there is no smaller partition to contain this portion of the value without wasting storage space. As shown in Fig. 6(b), if the smallest partition size is limited to 2^{12} bytes, the last $(1024 + 512)$ bytes of the value *v*24 will be directly stored in the smallest partition, whose size is 4096 bytes; therefore, around 3021 bytes of the storage space would be wasted. In other words, under this example, the total storage space utilization will be slightly decreased by 10%, but the read performance can be improved by 25% due to the reduction of one flash page read.

C. Restriction on the Number of Partitions

Another approach to mitigate the read amplification problem is to *directly limit the number of partitions used to partition a value*. Notably, this method would also produce the value tail introduced in the previous section, and waste some storage space. The major difference between restricting the smallest partition size and restricting the number of partitions is that the latter gives more reliable guarantee on reducing the read amplification. The rationale behind this is that even though the smallest partition size is restricted, there is still a probability to create considerable number of partitions in some cases. For example, as shown in Fig. 7(a), suppose the smallest partition size is 2^{12} bytes and there is a value *v25* of 61 441 bytes. Under this case, the value *v25* will be partitioned into five partitions of sizes equal to 32 768, 16 384, 8192, 4096, and 4096 bytes with 4095 bytes wasted. It is clear to see that even though we have restricted the smallest partition size, we still produce five partitions for a value. By contrast, if we directly restrict the number of partitions to be 3 as shown in Fig. 7(b), we will only partition this value into 3 partitions of sizes equal to 32968, 16384, and 16384 bytes with 4095 bytes wasted. That is, this new revised approach can store the same value with the same wasted storage space, but with a smaller number of partitions. However, it is still worthy to note that, although this method can effectively decrease the read amplification, it could potentially sacrifice more storage space in some cases, as compared to the method presented in the previous section.

D. Restriction on the Wasted Storage Space

Although two effective methods have been introduced in Sections IV-B and IV-C to alleviate the read amplification issue, both of them might sacrifice too much storage space utilization and are not able to co-optimize the space utilization and device performance. In other words, by increasing of the smallest partition size (or the decreasing of the number of partitions), the read amplification will keep reducing as the storage space utilization degraded. The essential rationale behind this phenomenon is that both strategies try to limit the number of partitions after partitioning a value without considering how many storage space they waste.

Thus, to further prevent the storage space utilization from indefinitely degrading while reducing the number of partitions

of a value, we put forward another revised method to simultaneously restrict the wasted storage space and reduce read amplification, so as to ultimately co-optimize the storage space utilization and the device performance. That is, *the process for partitioning a value will immediately halt, when the remaining value could be stored into the former partition size, with the wasted storage space less than the predefined "acceptable wasted storage space."* For example, as shown in Fig. 7(a) and (b), by adopting the methods described in the previous two sections, the value *v25* would be partitioned into 5 and 3 partitions, respectively. Nevertheless, as shown in Fig. 7(c), by restricting the maximum wasted storage space as 2^{12} bytes, the value *v25* would be only partitioned into two partitions with 4095 bytes of space wasted. This is exactly the same as the other two methods.

In summary, in the original KVFTL, a value would be recursively sliced into small(er) partitions, until it has been completely partitioned or the partition size reaches the smallest partition size (i.e., 2^0 byte). On the other hand, either of the two revised methods presented in Sections IV-B or IV-C will continue partitioning a value until it achieves the predefined smallest partition size or the maximum number of partitions. By contrast, the revised approach presented in this section will halt the partitioning process whenever the size of the wasted storage space is acceptable (i.e., less than the predefined acceptable wasted storage space). This is also the main reason why the third revised approach can explicitly co-optimize the storage space utilization and the device performance.

V. Performance Evaluation

A. Experimental Setup

In this section, we evaluate the storage space utilization and device performance (i.e., the total execution time, the number of page reads, page writes, and block erases) of KVFTL, when it is applied to a key-value flash device. The experiment was conducted in a customized simulator that was modified from the simulator used in BET [6] since it could offer the most basic framework for our key-value flash device. Based on the simulator, a key-value flash device of three different flash pages (i.e., 16 KB, 32 KB, and 64 KB) and blocks of 256 pages was under investigation. The detailed configuration of the evaluated flash-memory chip is shown in Table I. Serial access time is the time to shift data from the data register out over the data bus or write data from the data bus into the data register sequentially. Notably, serial access time of each read/wrtie request is calculated according to the size of the data register which is always aligned to the page size in our evaluation. To read data from a flash page, data must be read to the data register first and shifted to the data bus byte by byte. Therefore, the time required to perform a page read is the total time of serial access time plus the read latency. Similarly, the time to perform a page write is the total time of serial access time plus the write latency. Besides, in this experiment, the possible range of partition sizes is set from 2^0 bytes to 2^{20} bytes and the maximum partition size is referred to the specification of Seagate Kinetic Drive [20]. Moreover, the size of each page buffer is equal to the page size (i.e., 16 KB, 32 KB, or 64 KB); meanwhile, in this evaluation, only 14 page buffers and 224 KB (i.e., 14×16 KB) RAM space are allocated for the partitioned blocks whose partition sizes are from $2^0, 2^1, \ldots,$ to 2^{13} bytes.

TABLE I
Evaluated Flash Chip [25]

Page size	16KB	Read latency	$115\mu s$
Block size	4MB (256 pages)	Write latency	$1600\mu s$
Serial access	$10ns$/ byte	Erase latency	$3ms$

Furthermore, we also show the evaluated results of KVFTL-SP, KVFTL-NP, and KVFTL-WS, which are the revised versions of the original KVFTL presented in Sections IV-B–IV-D, respectively. In our evaluations, KVFTL-SP, KVFTL-NP, and KVFTL-WS, respectively, denote the original KVFTL of the restrictions on the smallest partition size (set as 2^{12} bytes), the maximum number of partitions (set as 3), and the maximum wasted storage space (set as 2^{11} bytes). Notably, these settings are selected as representative configurations of the three revised versions of the original KVFTL without losing of generality, since it can achieve a great balance between the storage space utilization and the device performance, under the evaluated key-value flash device.

In addition, to compare the proposed design with the most common existing solution, DFTL was also implemented in this evaluation. Based on DFTL [10], DFTL is altered to support key-value operations. Since key-value pairs cannot be directly mapped (or translated) to the logical page addresses (LPAs) in DFTL, an additional mapping table called key-value global translation directory (KGTD) is adopted to map keys to the LPAs. Notably, the number of the LPAs depends on the value size of a key. When the revised DFTL receives a key-value pair, the value would be first partitioned into numbers of LPAs. For value that is insufficient to fill up a whole LPA, it would be directly stored with an LPA and waste the remaining logical page space. After the value are translated into the LPAs, they would be mapped by the CMT of the revised DFTL. In the revised DFTL, each entry of CMT is modified to include the fields of value size and key identifier for retrieving the value of a specific key. Furthermore, to record the locations of these separated LPAs of a key, each entry of KGTD maintains a bitmap to indicate which translation pages (recorded in GTD of DFTL) that the LPAs are stored into. To retrieved the value of a specific key, KGTD is first traced to figure out the index of the translation pages. After that, CMTs in these translation pages are scanned to obtain the complete key-value pair. Note that, DFTL is the baseline in our experiments because it is a page-based management design. On the other hand, we used the well-known Yahoo Cloud Serving Benchmark [7] to generate the key-value workloads based on the HBase database [21]. Notably, the workloads were configured to contain five hundred thousands of *put* operations and five hundred thousands of *get* operations. In addition, three representative distributions of value sizes were adopted: in the "small" and "large" distributions, the evaluated value sizes are mainly smaller or larger than the flash page size (i.e., 16 KB), respectively; in the "uniform" distribution, the evaluated value sizes are uniformly varied from the smallest value size (i.e., 1 B) to the largest value size (i.e., 1 MB) [20].

Notably, in order to demonstrate the optimal device performance that can be achieved by the two evaluated approaches, the experimental results presented in Sections V-B1 to V-B3 are collected by giving a large enough RAM space to store all the mapping information (i.e., GTD and CMT of DFTL, and *KVHeadTable* and *PartitionPtrTable*

of the proposed KVFTL) without the need to load/store them from/to the flash space. In the meantime, in order to further investigate how the device performance would be affected by the amount of RAM space, Section V-B4 further analyzes the static and the dynamic RAM requirement for both evaluated approaches and compares the performance results of both approaches by giving a different amount of on-device RAM space.

B. Experimental Results

1) Storage Space Utilization: Fig. 8(a) shows the storage space utilization achieved by DFTL, KVFTL, KVFTL-SP, KVFTL-NP, and KVFTL-WS under different distribution with 16 KB flash pages, where the *x*-axis denotes the distribution of value sizes and the *y*-axis denotes the storage space utilization (in percentage). Under all evaluated distributions, KVFTL could always achieve the optimal storage space utilization (i.e., 100%), while DFTL would result in the lowest space utilization (i.e., 50% under the small distribution). This is because DFTL wastes too much storage space on storing the variable-sized values. On the other hand, KVFTL-SP, KVFTL-NP, and KVFTL-WS could lead to almost 90% storage space utilization under most of the distributions. Based on our best knowledge, nowadays flash storage devices usually have more than 10% over-provisioned storage space. For example, if the storage space is 100 GB and the over-provisioned portion is 10 GB, it means the physical storage space is 110 GB. Thus, with the storage space larger than the claimed capacity, 90% of space utilization should be acceptable. It is also interesting to see among these three revised versions of KVFTL, KVFTL-WS achieves the best storage space utilization, and even achieves almost 100% storage space utilization under large distribution.

Fig. 8(b) further shows the storage space utilization achieved by DFTL, KVFTL, KVFTL-SP, KVFTL-NP, and KVFTL-WS under different sizes of a flash page, where the *x*-axis denotes the size of flash page and the *y*-axis denotes the storage space utilization (in percentage). Under all the evaluated flash page sizes, KVFTL could always achieve the optimal storage space utilization while storage space utilization of all other designs are getting worse as the page size becomes larger and larger. Especially, the decline of storage space utilization of DFTL is the most dramatic since it stores those variable-sized values into a relatively bigger flash pages and waste more storage space with the increasing of the page size. On the other hand, KVFTL-SP, KVFTL-NP, and KVFTL-WS could still remain at least 80% storage space utilization under larger flash page size. In particular, KVFTL-WS could still achieve almost 90% storage space utilization due to the strategy on partitioning is focusing on the wasted storage space, which makes the degradation of this design is smaller than KVFTL-SP and KVFTL-NP.

2) Performance: Fig. 9(a) shows the total execution time introduced by the five evaluated FTL designs under all evaluated distributions, where the *x*-axis denotes the distribution of value sizes and the *y*-axis denotes the total execution time to complete the evaluated workload. Owing to the read amplification issue, KVFTL increases extra execution time by 18% as compared with DFTL. However, according to the results, KVFTL-SP, KVFTL-NP, and KVFTL-WS, which have close

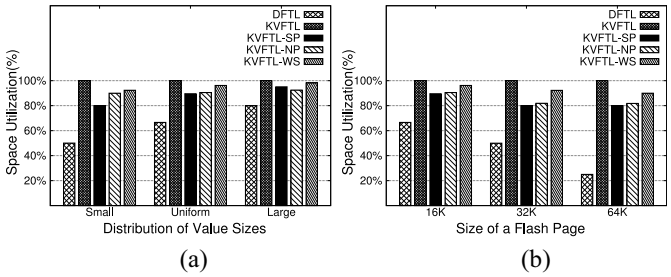

Fig. 8. Storage space utilization. (a) Under different distribution. (b) Under different page size.

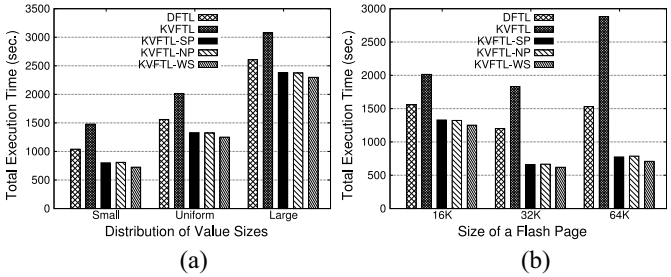

Fig. 9. Total execution time. (a) Under different distribution. (b) Under different page size.

results, could lead to the shortest total execution time compared to KVFTL and DFTL. Specifically, under all evaluated distributions, these three FTL designs could at most reduce 31% and 51% total execution time over that of DFTL and KVFTL, respectively.

Fig. 9(b) shows the total execution time introduced by the five evaluated FTL designs under all three different flash page sizes and uniform distribution, where the *x*-axis denotes the flash page size and the *y*-axis denotes the total execution time to complete the evaluated workload. Under different flash page sizes, the trend of execution time introduced by these five designs are similar. KVFTL still performs the worst due to the read amplification issue while KVFTL-SP, KVFTL-NP, and KVFTL-WS have close results and perform better than DFTL. Execution time of all designs under page of 32 KB are shorter than the results under page of 16 KB because each page could store more data and introduce less write operations and garbage collection, even though the read/write latency is larger due to the flash page size. On the other hand, since rapid growth of the read/write latency as the increasing of the page size, execution time of all designs under page of 64 KB are longer than the results under the other two page sizes even though less write operations and nearly zero garbage collection are needed.

On the other hand, Fig. 10 demonstrates the total amounts of page accesses (i.e., reads and writes) and block erases generated by the five evaluated designs with 16 KB flash page size, where the *x*-axis denotes the operation type and the *y*-axis denotes the total number of operation counts. Note that DFTL introduced the least amount of read operations, but suffered from the largest amount of page writes and block erases. This is because the lowest storage space utilization of DFTL would increase the frequency of garbage collection.

On another hand, KVFTL has the smallest amount of write and erase operations among these five FTL designs,

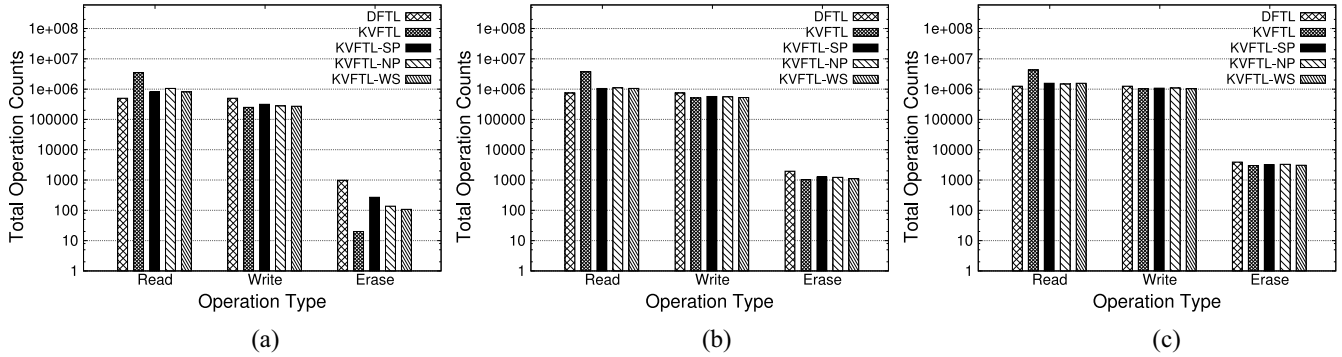

Fig. 10. Total operation counts of read/write/erase operations. (a) Small. (b) Uniform. (c) Large.

especially under small distribution, which is reasonable due to the 100% storage space utilization. Here some attentive readers might be wondering why the number of write and erase operations of KVFTL would be lower than all other FTL designs, despite having the smallest possible partition size. It is because for those partitions whose size are smaller than a page, we adopt page buffers as we described to collect them according to the partition size. Partitions in a page buffer would be written into the physical flash page once the buffer is full and need only one page write. This mechanism makes KVFTL have the smallest number of write operations. Furthermore, due to less write operations and higher storage space utilization, KVFTL would have a better efficiency for garbage collection and result in much smaller number of erase operations. However, even though KVFTL effectively reduces the write amplification to improve the performance, KVFTL still performs the worst in terms of the total execution time since the severe read amplification largely degrades the overall performance. Comparatively, KVFTL-SP, KVFTL-NP, and KVFTL-WS could achieve a great balance between amount of read and write operations. That is, as compared with DFTL, these three FTL designs could still effectively reduce the counts of write and erase operations, but the read amplification was quite limited (i.e., at most 1.6 times of the total page reads than that of DFTL). Moreover, considering the write and erase latencies are usually much longer than the read latency, these three FTL designs could be thereby expected to conduct the shortest execution time as we just shown.

3) Tradeoff Between Space Utilization and Performance: Fig. 11 shows the variations of the read/write performance and the space utilization under different configurations of the smallest partition size, where the *x*-axis denotes the different smallest partition sizes applied to KVFTL. On another hand, Fig. 12 shows the variations of the read/write performance and the space utilization under different configurations of the maximum number of partitions, where the *x*-axis denotes the different maximum number of partitions applied to KVFTL. Similarly, Fig. 13 shows the variations of the read/write performance and the space utilization under different configurations of the maximum bytes of wasted storage space, where the *x*-axis denotes the different maximum bytes of wasted storage space applied to KVFTL. The left-hand-side and the right-hand-size *y*-axis of these three Figures all denote the storage space utilization (in percentage) and the total page read/write counts, respectively.

When the smallest partition size of KVFTL-SP kept increasing, the read amplification would be rapidly decreased, while

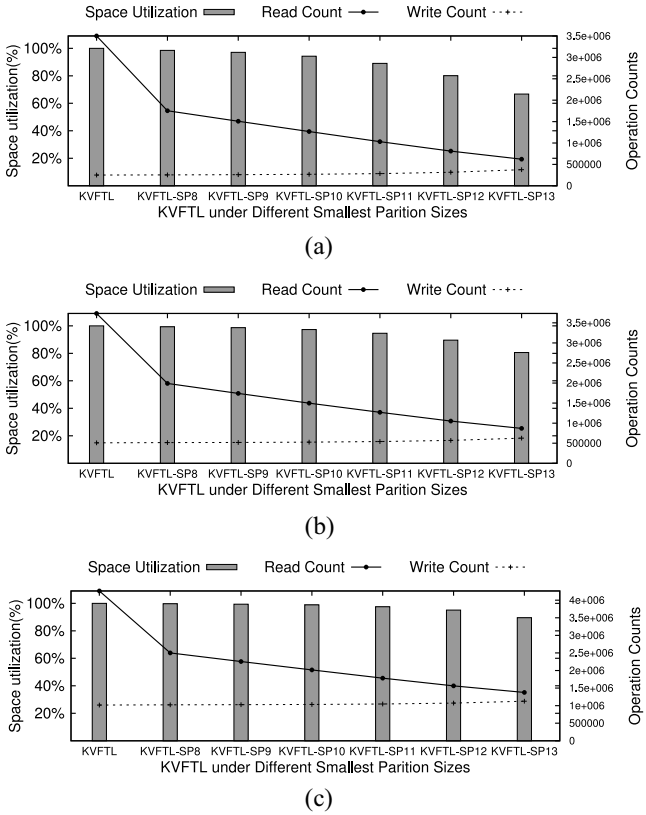

Fig. 11. Trade-off between the storage space utilization and read/write performance with restriction on the smallest partition size. (a) Small distribution. (b) Uniform distribution. (c) Large distribution.

the storage space utilization and the write performance would become worse and worse. Notably, storage space utilization of KVFTL-SP has dropped below 80% under small distribution. Although we can further raise the smallest partition size to gain better performance on read operations with a little more amount of write operations, the storage space utilization would become unacceptable. The same phenomenon occurs to KVFTL-NP while the maximum number of partitions is set to 2, namely, KVFTL-NP2. Differently, KVFTL-WS still has an outstanding space utilization even though the maximum bytes of wasted storage space is raised to 2^{12}, namely KVFTL-WS12. KVFTL-WS12 can achieve over 90% storage space utilization under all distributions with almost the same effect on reducing the read amplification of KVFTL-SP and

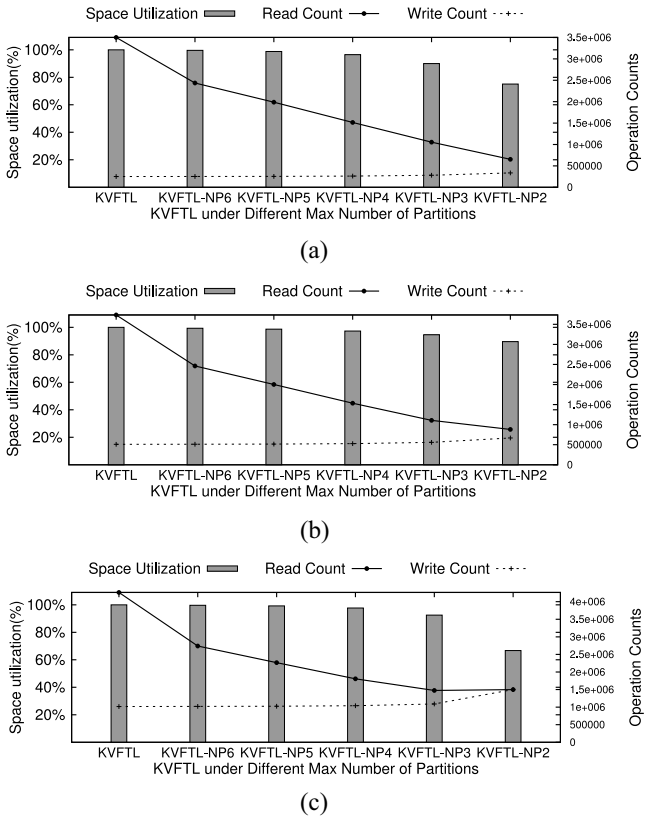

Fig. 12. Trade-off between the storage space utilization and read/write performance with restriction on the maximum number of partitions. (a) Small distribution. (b) Uniform distribution. (c) Large distribution.

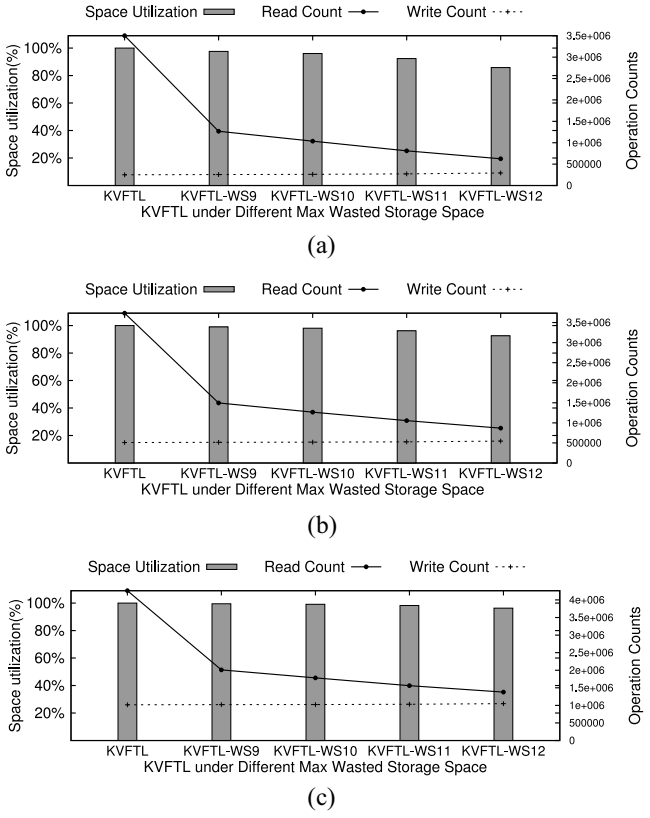

Fig. 13. Trade-off between the storage space utilization and read/write performance with restriction on the maximum wasted storage space. (a) Small distribution. (b) Uniform distribution. (c) Large distribution.

TABLE II
EVALUATED STATIC RAM USAGE

	KGTD+GTD	*KVHeadTable*
RAM Usage	(25298+832) B	5838 B

KVFTL-NP, which shows KVFTL-WS would be the best choice among these three KVFTL designs in most cases.

The tradeoff between the space utilization and the performance demonstrates the great flexibility of KVFTL design. If the space utilization is the most critical design issue, KVFTL, which achieves the optimal space utilization, would be the best choice. On the other hand, if little waste on the space utilization is acceptable, the KVFTL-WS design should be considered to strike a good balance to meet the different design requirements.

4) RAM Space Investigation:

a) RAM requirement analysis: First of all, the overall RAM usage can be classified into two parts in practice. One part is the static part, which is the global mapping information and needs to reside in RAM, such as the *KGTD+GTD* in DFTL and the *KVHeadTable* in the proposed KVFTL. The other part is the dynamic part, which is the complete mapping information and will be stored in flash memory and be only loaded into the limited on-device RAM space when needed, such as the CMT in DFTL and the *PartitionPtrTable* in KVFTL.

Table II shows the static part of RAM requirement under the uniform distribution for the revised DFTL and the proposed KVFTL. The static part of RAM requirement of KGTD and *KVHeadTable* is calculated by summarizing the RAM usage of all valid entries. In addition, in KGTD, each entry consumes 26 bytes in total to record the key identifier and the bitmap. Notably, since additional fields of value size and key identifier are required for CMT in DFTL to perform key operations, the total usage of these mapping tables is bigger than traditional DFTL and also enlarges the total size of GTD, which is about 1 KB in our evaluation. On the other hand, in *KVHeadTable*, each entry consumes 6 bytes in total to record the key identifier and the pointer to the first partition of a value. Since there are finally 973 valid entries remained in the RAM, the RAM usage of DFTL and KVFTL are 26 130 B and 5838 B as shown in Table II, respectively. This result shows that KVFTL could outperform the revised DFTL design with regard to the RAM requirement of the global mapping part, through simplifying the management on the mappings for keys and their value.

b) Performance evaluation under different RAM space: On the other hand, Table III shows the total execution time under the uniform distribution considering different total RAM sizes for the revised DFTL and the proposed KVFTL-WS. Also, the overheads to perform load/store mapping information for both designs are represented by calculating the percentage of the execution time introduced by load/store mapping information to the total execution time in Table III. Please note that, the evaluated result under unlimited RAM size is the baseline in Table III because it is an optimal result taken from Section V-B2.

As shown in Table III, under RAM size of 16 KB, KVFTL-WS spends 19.450%/0.079% of the total execution time on performing the load/store operation for mapping information.

TABLE III
EVALUATED TOTAL EXECUTION TIME UNDER DIFFERENT TOTAL RAM SIZES

RAM Size	Design	*Load*	*Store*	*Execution Time*
16K	DFTL	12.327%	0.059%	1780.250 sec.
	KVFTL-WS	19.450%	0.079%	1559.278 sec.
32K	DFTL	12.325%	0.059%	1780.191 sec.
	KVFTL-WS	19.450%	0.079%	1559.220 sec.
128K	DFTL	12.220%	0.058%	1778.052 sec.
	KVFTL-WS	19.350%	0.077%	1557.313 sec.
512K	DFTL	1.039%	0.002%	1576.163 sec.
	KVFTL-WS	2.718%	0.008%	1289.938 sec.
1M	DFTL	0.072%	0.000%	1560.866 sec.
	KVFTL-WS	1.060%	0.000%	1268.226 sec.
32M	DFTL	0.000%	0.000%	1559.742 sec.
	KVFTL-WS	0.005%	0.000%	1254.843 sec.
Unlimited	DFTL	0.000%	0.000%	1559.742 sec.
	KVFTL-WS	0.000%	0.000%	1254.775 sec.

Thus, it introduces more time overhead than the revised DFTL on loading/storing mapping information. However, since more efforts are paid for KVFTL-WS on performing the load and store operation, KVFTL-WS could still outperform the revised DFTL by at least 12% total execution time. With the increasing of the RAM size, both designs would spend less and less execution time on performing the load and store operations and thus reduce the total execution time. As shown in Table III, RAM size of 32 MB is large enough in our evaluation to contain almost all the valid entries of both designs and make our proposed KVFTL-WS design outperform the revised DFTL by about 19% total execution time.

VI. RELATED WORK

Many FTL designs have been proposed to resolve various design issues of flash storage devices. Well-known examples are BAST [13] and FAST [15] that were proposed to leverage the advantage of coarse-grained and fine-grained address translation; DFTL [10] was proposed to store page-level address mappings in flash memory and cache in RAM on demand to improve the performance of random writes for flash devices.

Under the changes of data access model from fixed-sized to variable-sized data in the host system, researches started to study how to improve the performance of key-value store or object-based file systems on using flash device as the storage media. OFTL [18] proposed to offload the storage space management from object file systems to OFTL, which accesses raw flash chips in the page-unit and exports the byte-unit access interface to the file system. p-OFTL [27] proposed to merge the storage management in the object-based file systems with the flash management design. Then, SILT [16] further presents a key-value store system that significantly reduces memory consumption of each key and achieves high-performance key-value lookups. LOCS [26] is an LSM-tree-based key-value store that exploits the in-device parallelism of an open-channel SSD (SDF) whose internal channels can be directly accessed at the host side. By directly managing the flash channels, techniques of scheduling and dispatching policies are redesigned to optimize the traffic control of write requests so that I/O performance and throughput of the SDF can be improved. Wisckey [17] is a key-value store, where keys are kept sorted in the LSM-tree and values are separately stored in a log. DIDACache [24] proposed a highly optimized key-value cache system above the operating system; it removes unnecessary intermediate layers for flash storage devices.

More recently, KAML [11] proposed a key-value multi-log architecture for variable-sized records to support atomic transactions so as to leverage the host DRAM to improve performance. However, it focuses on handling the transactions and does not tackle the issues on: 1) how to effectively improve the storage space utilization based on the variable value size of keys and 2) how to efficiently boost garbage collection on storing key-value records in fix-sized pages. These two issues are the main focus of the proposed KVFTL. KVFTL considers key-value-specific SSDs; it receives key operations through the key-value interface of SSDs, rearrange the variable-sized key-value pairs, and resolve the low space utilization and garbage collection overhead issues.

VII. CONCLUSION

This paper proposes a new KVFTL for key-value flash storage devices. In particular, a partition-based space manager is proposed to optimize the space utilization of storing variable-sized values into fixed-sized flash pages. The results show that the proposed design can not only achieve high storage space utilization, but also have a great flexibility to trade little space utilization for better performance. Specifically, the proposed design can achieve 93% of the storage space utilization while effectively reducing the total execution time by 31%, as compared to the traditional page-based FTL design. In the future, we shall investigate how to further optimize the performance for the advanced key-value operations, such as the *getnext*, *getprev*, and *range* operations. Furthermore, we shall also investigate how to employ the parallelism of flash devices based on the proposed strategies to improve the throughput so as to speed up the performance. More research on RAM-based caches for the KVFTL design should be also included since the granularity of access to a flash device should be variable-sized key-value pairs instead of fix-sized pages or sectors. Last but not least, KVFTL could consider the adoption of nonvolatile memories for the page buffers, where nonvolatile memories are more reliable but might have the lifetime issue.

REFERENCES

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. SIGMETRICS*, 2012, pp. 53–64.
[2] A. Ban, "Flash file system," U.S. Patent 5 404 485, Apr. 1995.
[3] A. Ban, *Flash-Memory Translation Layer for NAND Flash (NFTL)*, M-Systems, Kfar Saba, Israel, 1998.
[4] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A design for high-performance flash disks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 88–93, Apr. 2005.
[5] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, Jun. 2008.
[6] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design," in *Proc. 44th Annu. Design Autom. Conf. (DAC)*, San Diego, CA, USA, 2007, pp. 212–217.
[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. SoCC*, 2010, pp. 143–154.
[8] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proc. SOSP*, 2007, pp. 205–220.
[9] J. Elliott and B. Brennan, "Industry innovation with Samsung's next generation V-NAND," in *Proc. Flash Memory Summit*, 2014. [Online]. Available: https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2014/20140805_Keynote2_Samsung.pdf
[10] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. ASPLOS*, 2009, pp. 229–240.

[11] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A flexible, high-performance key-value SSD," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 373–384.
[12] Y. Kang, J. Yang, and E. L. Miller, "Object-based SCM: An efficient interface for storage class memories," in *Proc. MSST*, May 2011, pp. 1–12.
[13] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Trans. Consum. Electron.*, vol. 48, no. 2, pp. 366–375, May 2002.
[14] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[15] S.-W. Lee *et al.*, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, Jul. 2007, Art. no. 3.
[16] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proc. SOSP*, 2011, pp. 1–13.
[17] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *Proc. FAST*, Santa Clara, CA, USA, Feb. 2016, pp. 133–148.
[18] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. FAST*, 2013, pp. 257–270.
[19] L. Mármol *et al.*, "NVMKV: A scalable and lightweight flash aware key-value store," in *Proc. HotStorage*, 2014, p. 8.
[20] Seagate. *Kinetic HDD Repository*. Accessed: Nov. 16, 2015. [Online]. Available: http://www.seagate.com/products/enterprise-servers-storage/nearline-storage/kinetic-hdd/
[21] The Apache Software Foundation. *Apache HBase Repository*. Accessed: Nov. 23, 2015. [Online]. Available: https://hbase.apache.org/
[22] Toshiba. *Key Value Drive Repository*. Accessed: Nov. 19, 2015. [Online]. Available: https://toshiba.semicon-storage.com/content/dam/toshiba-ss/asia-pacific/docs/product/storage/kv/technology-whitepater.pdf
[23] Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan, "MNFTL: An efficient flash translation layer for MLC NAND flash memory storage systems," in *Proc. DAC*, Jun. 2011, pp. 17–22.
[24] Z. Shen, F. Chen, Y. Jia, and Z. Shao, "DIDAcache: A deep integration of device and application for flash based key-value caching," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, Santa Clara, CA, USA, 2017, pp. 391–405.
[25] Hitachi Data Systems, "Hitachi accelerated flash-an innovative approach to solid-state storage," Tokyo, Japan, Hitachi, White Paper, Nov. 2015. https://community.hds.com/docs/DOC-1005695
[26] P. Wang *et al.*, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proc. EuroSys*, 2014, pp. 1–14.
[27] W. Wang, Y. Lu, and J. Shu, "p-OFTL: An object-based semantic-aware parallel flash translation layer," in *Proc. DATE*, Mar. 2014, pp. 1–6.
[28] K. Y. Yun and D. L. Dill, "A high-performance asynchronous SCSI controller," in *Proc. ICCD*, Oct. 1995, pp. 44–49.

Yen-Ting Chen (S'18) received the B.S. degree from the Department of Electrical Engineering, National Chung Cheng University, Minhsiung, Taiwan, in 2014 and the M.S. degree from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2016, where he is currently pursuing the Ph.D. degree with the Department of Computer Science.

He is a Research Assistant with the Institute of Information Science, Academia Sinica, Taipei, Taiwan. His current research interests include flash memory storage systems and embedded systems.

Ming-Chang Yang (S'12–M'17) received the B.S. degree from the Department of Computer Science, National Chiao-Tung University, Hsinchu, Taiwan, in 2010, and the master's and Ph.D. degrees from the Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan, in 2012 and 2016, respectively.

He is currently an Assistant Professor with the Department of Computer Science and Engineering, the Chinese University of Hong Kong, Hong Kong. His current research interests including emerging nonvolatile memory and storage technologies, memory and storage systems, and the next-generation memory/storage architecture designs.

Yuan-Hao Chang (SM'14) received the Ph.D. degree from National Taiwan University, Taipei, Taiwan, in 2009.

He joined the Institute of Information Science, Academia Sinica, Taipei, as an Assistant Research Fellow from 2011 and 2015, and has been promoted as an Associate Research Fellow since 2015. His current research interests include memory/storage systems, operating systems, embedded systems, and real-time systems.

Dr. Chang is a Lifetime Member of ACM.

Tseng-Yi Chen (M'13) received the Ph.D. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 2015 and the M.S. degree from Institute of Information Systems and Application, National Tsing Hua University in 2010.

He is currently a Post-Doctoral Fellow with the Institute of Information Science, Academia Sinica, Taipei, Taiwan. His current research interests include storage systems, operating systems, embedded systems, cloud computing systems, and fault-tolerant storage systems.

Hsin-Wen Wei (M'09) received the B.S. degree in information and computer engineering from Chung Yuan Christian University, Taoyuan City, Taiwan and the Ph.D. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 2008.

She is currently an Assistant Professor with the Department of Electrical Engineering, Tamkang University, New Taipei, Taiwan. Her current research interests include real-time and embedded systems, multimedia systems, communication and networking, and graph theory.

Wei-Kuan Shih (M'13) received the B.S. and M.S. degrees in computer science from the National Taiwan University, Taipei, Taiwan, and the Ph.D. degree in computer science from the University of Illinois, Urbana-Champaign, Champaign, IL, USA.

He is a Professor with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan. He has published over 130 articles in professional journals and conferences. His current research interests include real-time system, wireless sensor networks, distributed file systems, and embedded file systems, and energy issues pertaining to cloud computing.