Workload-Aware Elastic Striping With Hot Data Identification for SSD RAID Arrays

Yongkun Li, Biaobiao Shen, Yubiao Pan, Yinlong Xu, Zhipeng Li, and John C. S. Lui, *Fellow, IEEE*

***Abstract*—Redundant array of independent disk (RAID) offers a good option to provide device-level fault tolerance for solid-state drives (SSDs). However, parity update with either read–modify–write or read–reconstruct–write may introduce a lot of extra I/Os and thus significantly degrades SSD RAID performance. To reduce the parity update cost, elastic striping chooses to reconstruct new stripes with only the newly updated data chunks instead of directly updating parity chunks. However, it necessitates an RAID-level garbage collection (GC) process, which may incur a very high cost due to the mixture of hot and cold data chunks. To address this problem, we follow the idea of elastic striping and propose a workload-aware scheme (WAS) to reduce the RAID-level GC cost so as to improve the performance and endurance of SSD RAID. In particular, we first develop a novel lightweight hot data identification scheme which requires only a very small computation time and memory cost, then propose a hotness-aware elastic striping approach to separately write data chunks with different hotness to different regions in SSD RAID. To evaluate the effectiveness and efficiency of our WAS, we implement a prototype system on RAID-5 and RAID-6 arrays composed of commercial SSDs. Experimental results show that compared to original elastic striping, our scheme reduces 30.0%–70.6% (and 23.9%–63.2%) of chunk writes under the RAID-5 (and RAID-6) settings, and also reduces the average response time by 60.9%–79.3% (and 56.8%–80.9%) for RAID-5 (and RAID-6), respectively. Besides, our scheme also improves the endurance and reliability of SSD RAID compared to original elastic striping.**

***Index Terms*—Elastic striping, redundant array of independent disks (RAID)-level garbage collection, solid-state drive (SSD) RAID, workload awareness.**

I. Introduction

WITHOUT mechanical parts, solid-state drives (SSDs) can provide higher input/output operations per second, lower power consumption, and higher shock resistance than traditional hard-disk drives (HDDs). Therefore, SSDs have been replacing traditional HDDs and are widely used in both personal computers and enterprise servers of various IT companies, such as Google, Amazon, Dropbox, Facebook, and Baidu [21], [24]. However, SSDs still have limitations due to their physical characteristics. In particular, bit errors are very common for SSDs [7], [8], [22], so device-level fault tolerance becomes necessary, especially for applications with high reliability requirements.

Parity-based redundant array of independent disks (RAID) [26] offers a good option to provide device-level fault tolerance. In particular, data redundancies, which are usually called parity chunks, are generated for each group of data chunks, and then both the data chunks and parity chunks are striped across multiple SSDs with one on each to form a stripe. As a result, lost data chunks can be recovered from other data chunks and parity chunks within the same stripe, and so device-level fault tolerance is provided. Note that in parity-based RAID arrays, to update a data chunk, all parity chunks within the same stripe need to be updated so as to preserve data consistency. We call the operation of updating parity chunks *parity update*, which requires to first preread old data and/or parity chunks, and then write back the newly updated parity chunks to SSDs.

However, parity update not only severely degrades the write performance of SSD RAID, but also significantly reduces the SSD RAID endurance. On the one hand, parity update introduces a lot of extra I/Os, and these I/O operations must delay user requests, and so significantly prolong the I/O response time and reduce the system throughput. On the other hand, each block in an SSD can only sustain a limited number of erasures [3], [8], so extra writes caused by parity update also consume a lot of erasures and thus reduce system endurance. Therefore, how to reduce the number of writes caused by parity update still remains as a critical problem to SSD RAID.

To address the above problem, one good choice is elastic striping [12], which chooses to reconstruct new stripes with the newly updated data chunks instead of directly updating the parity chunks in old stripes, and then mark the obsolete data chunks in old stripes as invalid at the RAID level. Thus, it can reduce the I/Os caused by parity update by writing the reconstructed stripes with full-stripe writes. Besides, elastic striping uses only SSDs without extra devices.

However, elastic striping necessitates garbage collection (GC) to reclaim the space of invalid data chunks in old stripes. Specifically, when GC is triggered, it selects a GC unit, e.g., multiple stripes, according to the GC algorithm,

Manuscript received March 13, 2016; revised June 14, 2016; accepted July 31, 2016. Date of publication August 30, 2016; date of current version April 19, 2017. The work was supported in part by the National Nature Science Foundation of China under Grant 61303048 and Grant 61379038, in part by the Anhui Provincial Natural Science Foundation under Grant 1508085SQF214, and in part by the Huawei Innovation Research Program under Grant HIRPO20140301. An earlier conference version of the paper appeared in IEEE/IFIP DSN 2015 [25]. This paper was recommended by Associate Editor S. Bhunia. *(Corresponding author: Yinlong Xu.)*

Y. Li, B. Shen, Y. Pan, Y. Xu, and Z. Li are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China (e-mail: ykli@ustc.edu.cn; ustcshen@mail.ustc.edu.cn; pyb@mail.ustc.edu.cn; ylxu@ustc.edu.cn; lizhip@mail.ustc.edu.cn).

J. C. S. Lui is with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong (e-mail: cslui@cse.cuhk.edu.hk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2016.2604292

then writes back all the valid data chunks in the selected GC unit by reconstructing new stripes, and finally releases the space of the GC unit for future allocation. We call this process RAID-level GC, and it may introduce extra reads and writes. Thus, the overall system performance heavily depends on the RAID-level GC cost. In particular, because of the mixture of hot and cold data chunks in a workload, invalid data chunks may be scattered over the stripes in an RAID array. In this case, it may incur a very large RAID-level GC cost, and thus degrades both the performance and endurance of SSD RAID. *This motivates us to develop a WAS to separate data chunks of different hotness values, so as to reduce the RAID-level GC cost when deploying elastic striping for SSD RAID.*

To achieve the above goal, we first develop a *lightweight* hot data identification scheme, and then develop a hotness-aware elastic striping for parity update in SSD RAID by leveraging data hotness. Not like the direct counting-based approach for hot data identification, which may result in a large overhead for large systems as it records the information of all data chunks in the whole system, our hot data identification scheme requires a very small computation time and memory cost. We emphasize that the lightweight feature is really important and necessary to implement the WAS in SSD RAID. We make the following contributions in this paper.

1) We first develop a novel lightweight hot data identification scheme to classify data chunks into multiple types according to their hotness. Specifically, our scheme leverages a hash-based grouping technique with multiple least recently used (LRU) lists, and it requires only a very small memory cost and computation time. Thus, our scheme is practical and also efficient to be deployed with elastic striping in RAID controller due to its lightweight feature.
2) We then propose a hotness-aware elastic striping scheme to separately store data chunks with different hotness values in different regions in SSD RAID. To achieve this, we first cache data chunks in different groups in buffer according to their hotness, and then write data chunks to SSDs by constructing full stripes with only the data chunks in the same group in buffer via elastic striping, which are supposed to have similar hotness. By separating hot/cold data chunks, our scheme significantly reduces the RAID-level GC cost, and thus improves both the performance and endurance of SSD RAID.
3) We also develop a prototype to implement our WAS on RAID-5 and RAID-6 arrays with commercial SSDs. With the prototype, we validate the effectiveness of our scheme by using both synthetic and real-world workloads. Experimental results show that compared to elastic striping without workload awareness, our WAS reduces 30.0%–70.6% (and 23.9%–63.2%) of chunk writes for RAID-5 and RAID-6 arrays, and also reduces the average response time by 60.9%–79.3% (and 56.8%–80.9%) for RAID-5 and RAID-6, respectively. In addition, our scheme also improves the endurance and reliability of SSD RAID.

The rest of this paper is organized as follows. In Section II, we first provide necessary background on SSD RAID, then introduce elastic striping in detail, and finally motivate our design. In Section III, we describe the detailed design of our WAS. In Section IV, we describe our system prototype and discuss several implementation issues. In Section V, we show experimental results, and in Section VI, we conclude this paper.

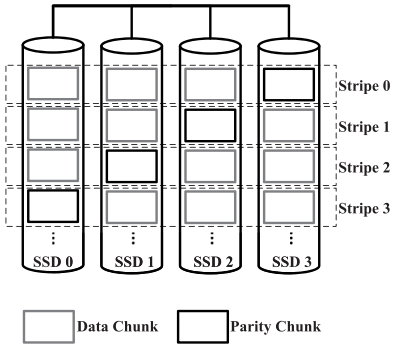


Fig. 1. Layout of an SSD RAID-5.

II. Background and Motivation

In this section, we first provide the necessary background on SSD RAID, and then overview the elastic striping scheme for improving parity update. Finally, we discuss the performance degradation problem caused by RAID-level GC in SSD RAID with elastic striping and motivate the design of our scheme.

A. SSDs and SSD RAID

We first give a brief introduction to the structure of SSDs. A flash-based SSD is usually organized into blocks, each further contains 64 or 128 pages with 4 KB or 8 KB each. Read, write, and erase are three fundamental operations in SSDs. In particular, read and write are performed in unit of page, while erase is performed in unit of block. To perform write, SSDs adopt the out-of-place overwrite scheme. That is, to update a page in an SSD, it writes the new data to another free page first, and then marks the original page as invalid. In order to reclaim the space occupied by those invalid pages, GC operation inside SSDs is needed. When GC is triggered, it first chooses a candidate block, reads out all the valid pages in this candidate block, then writes them back to another free block, and finally calls an erase operation to set all pages in the chosen block to free.

As bit error is still common in SSDs, RAID is usually considered to provide system-level fault tolerance. We take an SSD RAID-5 as an example, as shown in Fig. 1. The whole RAID is divided into stripes, each of which contains multiple chunks. Chunks in a stripe are distributed across all SSDs in the RAID to protect device failures. Each stripe has one parity chunk which is computed from data chunks in the same stripe. Parity chunks in the RAID are usually stored in different SSDs with a round-robin manner for load balance. When data chunks are updated, their corresponding parity chunks will be updated as well via either read–modify–write (RMW) or read–reconstruction–write (RRW). We let the chunk size be equal to the page size in this paper.

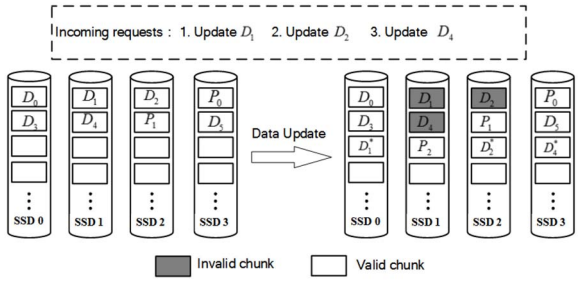


Fig. 2. Elastic striping: $P_2 = D_1^* \oplus D_2^* \oplus D_4^*$.

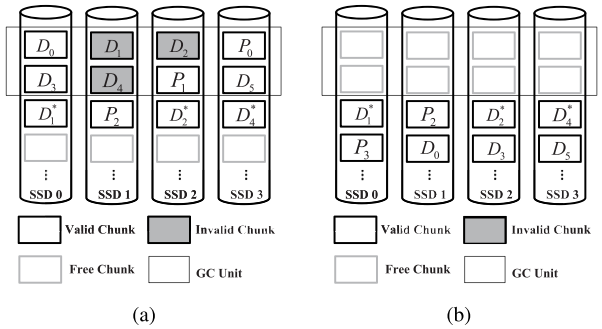


Fig. 3. RAID-level GC. (a) Before GC operation. (b) After GC operation.

However, parity update in an RAID will introduce extra I/Os and degrade both the performance and endurance of SSD RAID. To address this issue, three types of RAID schemes are developed to reduce the number of I/Os caused by parity update, e.g., parity logging, parity caching, and elastic striping. Specifically, parity logging [29] usually uses a dedicated device to log writes so as to delay parity update and reduce the amount of I/Os, and it has also been used for deploying RAID for SSDs. For example, Mao *et al.* [20] proposed a design for RAID-4, which uses one HDD as the parity device to absorb parity writes and uses another HDD as a mirror to absorb small write requests, and similar idea was also designed for RAID-6 in [36]. Li *et al.* [17] proposed EPLog which extends parity logging with an elastic feature and uses HDDs as log devices to absorb writes so as to reduce the writes to SSDs. Differently, parity caching [4], [10], [14], [16] uses a buffer, e.g., nonvolatile memory, to delay parity updates so as to reduce the writes to SSDs. At last, elastic striping [12] chooses to construct new stripes with updated new data chunks instead of immediately updating the parity chunks in the old stripe. Considering the good feature of requiring no additional devices, we focus on the scheme of elastic striping in this paper. In the following of this section, we first review how elastic striping works, and then discuss its problem and motivate our design.

B. Elastic Striping

Elastic striping was first proposed for chip-level RAID in single SSDs. Its main idea is to reconstruct new stripes with the newly updated data instead of immediately updating the parity chunks in the old stripe, so it requires no additional devices such like nonvolatile memories. Fig. 2 shows an example which illustrates the scheme.

Suppose that there are six data chunks D_0–D_5 and two parity chunks P_0 and P_1 in an SSD RAID at the beginning, and the incoming requests are: 1) updating D_1 to D_1^*; 2) updating D_2 to D_2^*; and 3) updating D_4 to D_4^*. We assume that the three update requests arrive sequentially.

Instead of immediately updating D_1, D_2, and D_4 and their corresponding parity chunks, elastic striping manages write requests in a log-structured manner. Precisely, it appends the updated data D_1^*, D_2^*, and D_4^* into the RAID array by constructing a new stripe without performing update to the old stripes. Note that D_1, D_2, and D_4 are out-of-date, but still need to be kept in SSDs for data protection. For space consideration, elastic striping marks these chunks as invalid at RAID level and calls GC to reclaim the space occupied by invalid chunks in future.

The GC works as follows. When it is triggered, it selects a GC unit, which represents the smallest unit for GC and can be a multiple of stripes, according to a GC algorithm, then writes back all the valid data chunks in the selected GC unit by reconstructing new stripes, and finally releases the space of the GC unit for future allocation. We call this GC process RAID-level GC so as to differentiate the GC process inside single flash chips. We further define the average number of valid data chunks that need to be written back during each GC operation as RAID-level GC cost.

To further illustrate the RAID-level GC process, we consider an example shown in Fig. 3, where a GC unit consists of two stripes. As shown in the example, there are three valid data chunks, D_0, D_3, and D_5, in the selected GC unit in Fig. 3(a). An RAID-level GC operation first reads D_0, D_3, and D_5, then writes them into free places, and reclaims those invalid chunks for future allocation as shown in Fig. 3(b).

C. Motivation

We note that skewness and temporal locality exist in real-world I/O workloads [6], [15], [28]. Skewness indicates that some data are accessed and updated frequently while others are updated rarely. Temporal locality means if a data chunk is accessed at present, it will be accessed with a high probability in the near future. In particular, many workloads of real-world applications exhibit the characteristic that 80% of accesses are directed to only 20% of data, which is the so called "80/20 Rule" [6], [23]. Our analysis of real-world workloads also validates this property (see Table I). As a result, the mixture of hot and cold data in SSD RAID will potentially increase the number of chunk rewrites per RAID-level GC operation, and finally aggravates the RAID-level GC cost. In order to explain this, we present a simple numerical analysis by using the example in Fig. 4.

Now we compare the RAID-level GC cost in two hypothetic cases so as to study the impact of workload-awareness on elastic striping: 1) hot and cold data are evenly mixed together [see Fig. 4(a)] and 2) hot and cold data are perfectly separated [see Fig. 4(b)]. In this example, we consider an 80/20 Rule scenario, in which 20% of data in an SSD RAID is hot and receives 80% of write requests, and the remaining 80% of data is cold and receives 20% of writes, and use the number of write accesses to measure hotness so as to ease the

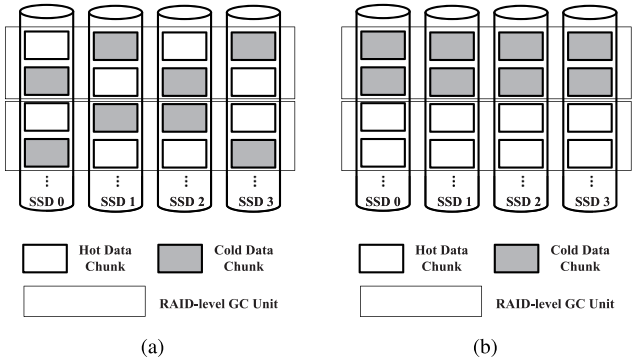


Fig. 4. Impact of hot/cold data separation on RAID-level GC cost. (a) Mixture of hot/cold data. (b) Separation of hot/cold data.

analysis here. Based on the 80/20 Rule, suppose that each cold data chunk receives one write on average, then each hot data chunk receives 16 writes. So hot data chunks are more likely to be updated and marked invalid than cold data chunks. We use p_h (p_c) to denote the probability of a hot (cold) data chunk being turned into invalid until the GC unit containing the chunk is selected for GC, and we have $p_c = (p_h/16)$ approximately. If we define the average probability of a chunk being turned into invalid (i.e., being updated) until the time when the unit containing the chunk is selected for GC as p, then for the example in Fig. 4(a), we have $p = (p_h + p_c/2) = (17/32)p_h$, and the expected RAID-level GC cost, which we denote as $\mathcal{C}_1$, can be expressed as follows.

RAID-Level GC Cost in Case 1 (Data Mixture):

$$\mathcal{C}_1 = N(1-p) = N(1-17/32p_h)$$

where N denotes the number of chunks in a GC unit.

In contrast, if hot data chunks and cold data chunks are perfectly separated as in Fig. 4(b), and we suppose that the RAID always selects a GC unit containing the most invalid chunks for GC, then the GC units with full of hot data chunks will potentially contain the most invalid chunks, and they are most likely to be selected for RAID-level GC as in Fig. 4(b), so the average probability p is $p = p_h$, and we can also derive the RAID-level GC cost, which we denote as $\mathcal{C}_2$, as follows.

RAID-Level GC Cost in Case 2 (Data Separation):

$$\mathcal{C}_2 = N(1-p) = N(1-p_h) < \mathcal{C}_1.$$

Based on the simple analysis on the above hypothetic example, we see that if hot and cold data chunks are mixed together in SSD RAID, then RAID-level GC requires more chunk rewrites, and thus degrades both the performance and endurance of SSD RAID. On the other hand, separating the hot and cold data may significantly reduce the RAID-level GC cost due to the skewness and locality within workload.

We point out that even though the above example just shows an extreme case, it serves as a good motivating example to help understand the problem. The experiments with real-world workloads in later section further validate the above argument for general cases. In short, we see that even though elastic striping can effectively reduce writes caused by RMW and RRW during parity update, it may suffer from high RAID-level GC cost, especially when invalid chunks are scattered over the stripes in the whole RAID array. Besides, according to the analysis on GC cost in single SSDs (e.g., [9], [13], [18], [19], [32], [35], and [31]) it is a general consensus that the GC performance in SSDs can be improved via separately storing hot/cold data. Inspired by this insight, we believe that the RAID-level GC cost in SSD RAID can also be reduced by separating hot and cold data. *This motivates us to develop a WAS with elastic striping to reduce the RAID-level GC cost so as to finally improve the SSD RAID performance and endurance.*

III. Workload-Aware Elastic Striping

In this section, we introduce our WAS in detail. To achieve workload-awareness in SSD RAID with elastic striping, we first classify data chunks into multiple types according to their hotnesses by developing a novel lightweight hot data identification scheme, and then exploit hotness awareness in parity update with elastic striping. In particular, we address the following key issues.

1) How to identify different types of data chunks efficiently (see Section III-A for lightweight hot data identification).
2) How to achieve hotness awareness when performing writes to SSDs with elastic striping (see Section III-B for hotness-aware elastic striping).

A. Lightweight Hot Data Identification

To achieve workload awareness, we first propose a lightweight hot data identification scheme by leveraging a hash-based grouping technique with multiple lists, which are managed to maintain the LRU property. For ease of presentation, we call these lists LRU *lists*, and call our identification scheme which relies on multiple grouping-based LRU lists GLRU with "G" standing for grouping.

The main idea of GLRU is to maintain a group of LRU lists to keep tracking of data chunks, and each LRU list contains a fixed number of data items, each of which records the logical page number (LPN) and a write counter of a particular data chunk. Specifically, the information of a data chunk is recorded by a data item which contains a 32-bit LPN and a 4-bit write counter. The association between data chunks and LRU lists are determined with a hash function, which is the division method [$f(x) = x \bmod K$, where x is the LPN of the requested data chunk and K is the number of LRU lists]. That is, for a data chunk, we compute the hash value of its LPN and take the hashing result as its group ID which is the identity of the LRU list in which the LPN is stored. We note that a data chunk belongs to at most one LRU list. In the whole system, we keep K LRU lists with N items in each. For ease of presentation, we collectively call them a *hot data table*. We emphasize that not all data chunks are recorded in the hot data table, and the rationale is that only potential hot data chunks are needed to be recorded so as to save the computation time and memory cost. Hot data chunks can be identified by looking up the hot data table. To further illustrate, we also show an example in Fig. 5 to depict the structure of GLRU which contains four LRU lists.

We now introduce the process of measuring the hotness of a single data chunk. Specifically, to determine whether a data

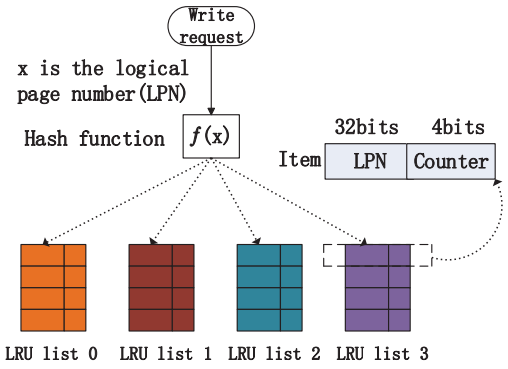


Fig. 5. Structure of GLRU, where $K = 4$ and $N = 4$.

Algorithm 1 Hot Data Identification: GLRU

Input: LPN of a data chunk;
Output: the hotness of the data;

```
Determine the group G in which the data belongs by hashing LPN;
if LPN exists in G then
    Increase the Counter by one;
    Adjust the position of the items in G to maintain the LRU property;
    if Counter ≥ Threshold then
        return HOT;
    else
        return COLD;
    end if
else
    if the probability test passed then
        Evict the last item in G;
        Insert an item with the LPN in the head;
        Set the Counter to one;
    end if
    return COLD;
end if
```

chunk is hot or not, we first hash its LPN to determine its group ID, and then look up the LPN in the corresponding LRU list. If the LPN is found and its counter is greater than or equal to a predefined threshold, then we take this data as hot; otherwise, we take it as cold. Before returning the identification result, we need to update the LRU list. In particular, if the LPN already exists in the LRU list, then we increase its counter by one and move this item to the head of the list so as to maintain the LRU property. We note that the write count will not be incremented if it reaches the maximum value of 15 as we use 4 bits to represent the counter. Otherwise, i.e., the LPN does not exist in the list before, then we evict the item in the tail with probability 0.5 and insert the new LPN into the list if the tail item is evicted. The rationale of this probabilistic evicting policy is that we can not be sure that the newly accessed data chunk is always hotter than the data chunk in the list tail, so we do the eviction with a probability.

Algorithm 1 provides the pseudo-code of the above described hot data identification scheme GLRU. Its time complexity is $O(N)$ and its space complexity is $O(N \times K)$, where N denotes the number of data items in each LRU list and K denotes the number of LRU lists. Considering that N directly affects the computation time of our scheme, in implementation, we usually set N as a small constant, say less than ten, so as to reduce the time cost for maintaining the LRU property in each LRU list.

We emphasize that our hot data identification scheme possesses at least three benefits. First, it is really fast in identifying whether a data chunk is hot or not, this is because we use a simple hash function to identify the group ID and each group (i.e., each LRU list) contains only a small number of items. Thus, our algorithm incurs a very small time overhead. Second, we only record the information of data chunks in the hot data table instead of all data chunks in the whole system, so the memory cost is also small. More importantly, the memory cost is independent from the system size, i.e., the total number of data chunks in the system. That is, as long as the number of LRU lists K and the number of items in each list N are fixed, the memory cost of our scheme is also fixed. Last, our algorithm is also highly efficient. As shown in the experiments with real-world workloads, even compared to the direct counting based approach which records the information of all data chunks, our algorithm helps further reduce the RAID-level GC cost by up to 16.7%.

Note that with the hot data identification scheme, data chunks are classified into different types according to their hotness, and the number of data types is clearly equal to the number of hotness levels. For example, if we set two hotness levels (hot and cold), then data chunks are also classified into two types, one type is hot, and the other type is cold. We point out that by also keeping write counters in the hot data table, we can easily extend GLRU to realize a multitier classification scheme by setting more than one threshold, so as to classify data chunks into multiple types. In our experiment, we implement the scheme by considering 2–4 tiers, so data chunks are classified into 2–4 types, respectively.

B. Hotness-Aware Elastic Striping

1) Data Flow: To achieve hotness awareness in elastic striping, after identifying hot/cold data, we first cache data chunks in buffer with a grouping-based approach. Specifically, we divide the whole buffer into multiple groups with the number of groups being equal to the number of data types, and use each group to cache only one type of data. After determining the type of a data chunk, we append it to the corresponding group in the system buffer, and we use in-place overwrite to update data in buffer. We note that all chunks in the same group should have similar update frequency. That is, these chunks are expected to be updated with similar likelihood in future. We call this caching approach hotness-aware caching, and Algorithm 2 describes the data flow.

After a chunk is appended into a group in buffer and the buffer for caching this group is full, then we generate full stripes by using data chunks in this group, and flush them to the underlying SSDs. In particular, we use a log-structured approach by appending the constructed data stripes to the end of the SSD RAID array. Precisely, we use data chunks from the same group in buffer to construct new full stripes, no matter whether the accesses to these data chunks are new writes or updates, and then append the new stripes to the SSD

Algorithm 2 Caching

```
for Each chunk C with its type being identified via the hot identification scheme do
    if The old version of C exists in a group in buffer then
        Delete it from that group;
    else if The old version of C has already flushed to SSDs then
        Mark it as invalid in metadata;
    end if
    Assign C into a group according to its type;
    if The buffer for caching the group to which C belongs is full then
        Perform encoding to generate full stripes;
    end if
end for
```

Algorithm 3 Hotness-Aware Elastic Striping

```
for Each group G to be flushed do
    if Space utilization exceeds the threshold then
        Perform RAID-level GC;
    end if
    Flush all chunks in group G with a log-structured manner by using elastic striping;
end for
```

RAID array. Clearly, before flushing chunks into the underlying SSDs, we need to check whether the space utilization of the whole RAID array exceeds a predefined threshold. If it is, then RAID-level GC will be triggered to reclaim space for future writes, and we will introduce RAID-level GC in detail in the next section. Algorithm 3 formally presents the process of writing data chunks to SSDs, and we call it hotness-aware elastic striping.

2) Benefits and Limitations: First of all, by separating data chunks into different groups in buffer, our hotness-aware caching scheme has the following advantages.

1) We can generate only full-stripe writes without triggering RMW and RRW when flushing data chunks, thus the performance degradation caused by these two operations can be eliminated.
2) Each full-stripe write only generates one parity write, so the number of parity writes gets minimized.
3) We can perform large write operations on the underlying SSDs by buffering small random writes and grouping them into sequential writes, and this improves the performance of single SSDs.

However, the loss of data chunks stored in cache at a sudden power off presents as a constraint for this design. In order to avoid this disaster, one may choose nonvolatile memory for caching, while we post this as a future work.

Second, by using a log-structured approach to write data chunks to SSDs, load balance between SSDs can be achieved, because log-structured approach always appends writes to SSD RAID in a round-robin mode no matter whether the request is a new write or an update. This also helps to achieve system-level wear-leveling among SSDs. Note that to flush $\text{BlockSize} \times (\text{RAIDSize} - 1)$ data chunks in the RAID-5 setup, extra BlockSize parity chunks are generated. Thus, the system totally flushes $\text{BlockSize} \times \text{RAIDSize}$ chunks with the log-structured write policy. That is, multiple full-stripe writes at the system level are issued, and the size of the write to each SSD is exactly equal to the full block size. Besides, note that each update operation inside SSDs triggers an out-of-place overwrite, the log-structured manner at RAID level, which takes updates as new writes, is similar to the out-of-place overwrite policy, so it does not have negative impact on SSD performance.

3) Discussions: We note that two key issues in the caching design need to be addressed so as to achieve high system performance. The first issue is how many chunks a group should contain. We denote the number of chunks in each group in cache as Size_G, and it is set as $\text{BlockSize} \times (\text{RAIDSize} - 1)$ for RAID-5 and $\text{BlockSize} \times (\text{RAIDSize} - 2)$ for RAID-6, where BlockSize is the number of pages in a block of an SSD and RAIDSize is the number of SSDs in the RAID array. Thus, one group can contain Size_G chunks in our setting. With this configuration, one can generate only full-stripe writes at system level.

The second key issue is how many groups we should have. We note that the more groups a system keeps, the higher accuracy the hot data identification scheme can achieve. However, the marginal gain, which denotes the additional improvement of identification accuracy by increasing one more group, should decrease. Besides, the more groups a system has, the more memory it consumes. As shown in the experiment section, using less than five groups is usually enough to achieve a good performance, and in particular, setting two groups already works well in most cases.

In terms of the memory cost of the caching scheme, note that if an RAID-5 is composed of 16 SSDs where each block contains 128 pages, then one group can buffer $128 \times (16 - 1)$ chunks at most. In this case, the size of each group is 7.5 MB if the page size is 4 KB. Therefore, the total cache size required by our algorithm is only tens of megabytes even if we classify data chunks into multiple types, say less than five. That is, our algorithm incurs only a small memory cost for caching.

IV. System Prototype and Implementation Details

To evaluate the performance of our WAS, we implement a system prototype by deploying our scheme on SSD RAID with commercial SSDs. In the following of this section, we first describe the overall system architecture of our prototype, then describe the RAID-level garbage collection component, and finally discuss several implementation issues.

A. Overview of System Architecture

Our system prototype consists of five key modules: 1) hot/cold identification module; 2) caching module; 3) coding module; 4) write module; and 5) GC module. The overall system architecture is illustrated in Fig. 6. Note that our design is implemented at the device level, so it does not need to access the SSD internals.

The main function of the hot/cold identification module is to track the access information of data and then categorize different data chunks into different groups. In particular, when a write request arrives, the hot/cold identification module first divides the data contained in the request into chunks whose size is set as the page size, then for each data chunk, the

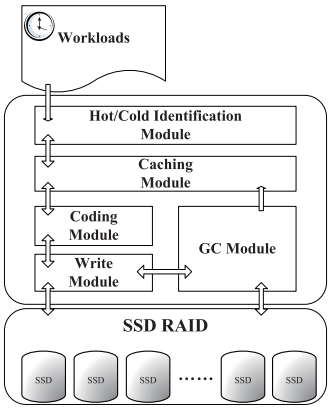


Fig. 6. Overview of system architecture.

hot/cold identification module checks the accessing addresses of each data chunk to determine whether it is a random request or a sequential request. If it is a sequential request, i.e., the addresses being accessed are successive to the ones accessed by the last request, the hot/cold identification module will simply return the identification result of the last request. The rationale is to leverage the temporal locality to reduce identification overhead. On the other hand, for a random request, the hot/cold identification module will search for its write frequency in our grouping-based LRU lists to decide which group this chunk belongs to. That is, for random requests, this module implements the hot identification scheme GLRU described in Section III-A to identify hot/cold data.

The caching module simply appends data chunks from the hot/cold identification module to different groups in the system buffer. When a chunk is appended into a group, if the cache for buffering this group of data chunks is not full, the caching module will simply cache this chunk in the buffer. Otherwise, it will flush all data chunks in this group to the underlying SSDs by calling the coding module.

The coding module adopts different codes (e.g., we implement RAID-5 code and RDP [5] code for RAID-6 in different experimental setups) to generate parity chunks. Note that our RAID scheme can also be applied for different RAID levels, and we just use RAID-5 and RDP code of RAID-6 as examples to validate the effectiveness of our design in this paper. In fact, the idea can be directly applied when constructing other RAID systems like EVENODD [2] code, X-Code [34], and Reed–Solomon [27] code, and we only need to make small changes in the implementation.

The write module is responsible for writing data to SSDs by implementing the hotness-aware elastic striping scheme described in Section III-B. It uses a log-structured approach to append chunks from the same group into SSD RAID. Precisely, data chunks from the same group in the caching module are always grouped to construct full new stripes, no matter whether these data chunks correspond to new writes or updates. The write module balances the write operations performed on all SSDs, and has the potential to benefit the system-level wear-leveling among SSDs. Before writing chunks into SSDs, the write module will check the space utilization of the whole array. If the space utilization exceeds a predefined threshold (e.g., 90% in our experimental setup), the write module first triggers RAID-level GC. Otherwise, it simply appends data and parity chunks into SSDs directly.

The GC module works for reclaiming the space occupied by invalid data, i.e., the data that has been updated before. When RAID-level GC is triggered, the GC module selects a candidate GC unit, which may be a multiple of stripes, and then writes back all the valid data chunks in the candidate GC unit by reconstructing new stripes. We further describe the RAID-level GC operation in the next section.

B. RAID-Level Garbage Collection

We note that the chunks that are marked invalid at the RAID level still need to be retained for protecting other data in the stripes. Therefore, we need RAID-level GC operation to reclaim the space occupied by invalid chunks. We further elaborate on two key issues when implementing the RAID-level GC in our prototype.

The first issue is how to set the size of GC unit. As we described before, a GC unit represents the smallest unit selected for RAID-level GC operation, and it is defined as a multiple of stripes in this paper. In particular, in our implementation, we let the size of a GC unit be equal to BlockSize $\times$ RAIDSize. Based on this setting, our RAID-level GC operates on a group of BlockSize stripes. With the help of the WAS, the RAID-level GC cost can be reduced.

The second issue is choosing which GC algorithm. We note that GC cost comes from the rewrites of valid chunks, so we use the greedy algorithm for performance consideration. That is, we always select the GC unit containing the least valid chunks for reclamation. When RAID-level GC is triggered, we first read out all valid chunks in the selected GC unit, then treat them as the coldest chunks, and finally assign them into the over-provisioned space of the RAID. Besides, this greedy algorithm is quite simple and easy to be implemented.

C. Implementation Issues

In this section, we discuss three implementation issues.

Metadata: We use (LBA, PDN, PBN, PPN) to record a chunk, where LBA denotes the logical block address of a chunk, PDN denotes physical device number or SSD number, PBN denotes physical block number, and PPN denotes physical page number. Note that PDN, PBN, and PPN are used to record the exact location of a chunk. We use 4 bytes to record LBA, and 4 bytes to store PBN. Additionally, we make several optimizations.

1) We use only several bits to record PDN instead of an int variable. For example, if an SSD RAID consists of 16 SSD, just 4 bits are enough to indicate all integers ranging from 0 to 15.
2) We also use only a small number of bits for PPN. If one block in an SSD contains 64 pages, 6 bits are able to represent all integers from 0 to 63 by using a binary string.

We note that only about 10 bytes are enough to store the metadata of one chunk. Therefore, our design requires only around 2.5 MB memory space for storing metadata for every gigabyte of data on SSDs.

We point out that we encode some parts of the metadata into one or two bytes so as to save the memory consumption

TABLE I
STATISTICS OF I/O WORKLOADS

Workloads	Total # of requests	Average request size	Write ratio	# of chunk writes	# of unique chunk writes	Frac. of acc. to top 20% data
Financial1	5334987	3465B	0.768391	7384396	135147	50.96%
Online	5700499	4096B	0.738849	4211806	69142	86.39%
Webmail	7795815	4096B	0.818642	6381985	218969	91.26%
Online+Webmail	13496314	4096B	0.784940	10593791	249026	80.88%
Synthetic	1000000	8192B	1	2875039	160282	80%

of metadata, while the decoding operation requires additional processing overhead. Thus, there is a tradeoff between memory consumption and processing overhead. However, the processing overhead is usually small as decoding only requires bitwise shift operations. Besides, both the memory usage and the processing overhead can be reduced if the chunk size is set to be larger. However, if the chunk size is too large, then there will be more partial writes, which may result in a degradation of the write performance. Thus, there is also a tradeoff when setting the chunk size.

To manage the metadata, note that the metadata will be frequently queried and modified. Compared to the inefficiency of a liner list in searching, a red–black tree, as a kind of self-balancing binary search tree, can achieve a higher efficiency in searching and insertion. Therefore, we use a red–black tree to manage metadata.

1) Partial Writes: Chunk is the smallest write unit in our implementation. In order to make the prototype be compatible with different chunk sizes, we also implement partial writes in our prototype. In particular, when the offset of a write request does not align with one chunk, our prototype first reads data from the remaining part of this chunk, combines them with the new data from the incoming request, then writes the combined chunk into the corresponding device. This process helps maintain data consistency when performing write operations.

2) Parallel Programming: Because our prototype is implemented on an RAID composed of multiple commercial SSDs, parallel I/Os are essential for system performance. We make an effort to provide parallelism for our prototype in which read, write, and RAID-level GC operations can be simultaneously executed among SSDs with a multithreading implementation.

V. PERFORMANCE EVALUATION

We conduct extensive experiments to show the effectiveness of our prototype by deploying it on an RAID consisting of commercial SSDs. In the experiments, we use four real-world workloads and one synthetic workload for evaluations.

Note that our design follows the idea of elastic striping [12] to do parity update and aims to reduce the RAID-level GC cost, so we take eSAP developed in [12] as the baseline scheme. We compare our scheme with eSAP mainly in the following aspects: RAID-level GC cost, average I/O response time, endurance, and reliability.

A. Workloads

We consider the following I/O workloads.

1) Financial1 [30]: It is an I/O trace collected from OLTP applications at a large financial institution.
2) Online [33]: It is a workload collected from a course management system of a university department.
3) Webmail [33]: It is a workload collected from Web interface of a department mail server.
4) Online+Webmail: It is the workload that combines the Online and Webmail workloads.
5) Synthetic: It is a synthetic workload in which addresses are accessed in normal distribution with 20% of addresses being accessed by 80% of requests.

Because the evaluation of RAID-level GC cost is the most important part in our experiments, all workloads considered in the evaluations are write-dominant. In particular, we only generate write requests for the Synthetic workload. Based on our analysis on the four real-world workloads, we find that the minimal write ratio is about 0.74 (the Online workload), and each workload writes several gigabytes of data. For example, there are about 40 GB of data chunks being written in the Online+Webmail workload, and all other workloads write more than 10 GB of data. Also, skewness exists in all workloads. For example, around 80%–90% of requests access only 20% of data except for the Financial1 workload which is random write dominant. The statistics further validate the widely assumed 80-20 Rule used in storage workloads, and also show the high skewness exhibited in real-world workloads. Table I shows the detailed workload statistics.

TABLE II
SPECIFICATION OF SSDS

Specification	SSD
Manufacturer	Intel
Model	530 Series(2.5-inch)
Capacity	120GB
Interface	SATA
Flash Memory	MLC

B. System Configuration

In our implementation, all experiments except for the endurance tests are performed on a Dell PowerEdge T620 server equipped with four 2.40 GHz Intel Xeon CPUs and 8 GB of physical memory. There are nine commercial SSDs being attached to the server. One SSD is installed with Debian 8 OS and works as the host. Other eight SSDs are taken as raw block devices and are configured as an RAID array. Table II shows the detailed specifications of the SSDs.

Because we cannot access the exact number of erasure operations for commercial SSDs, we run simulations to evaluate

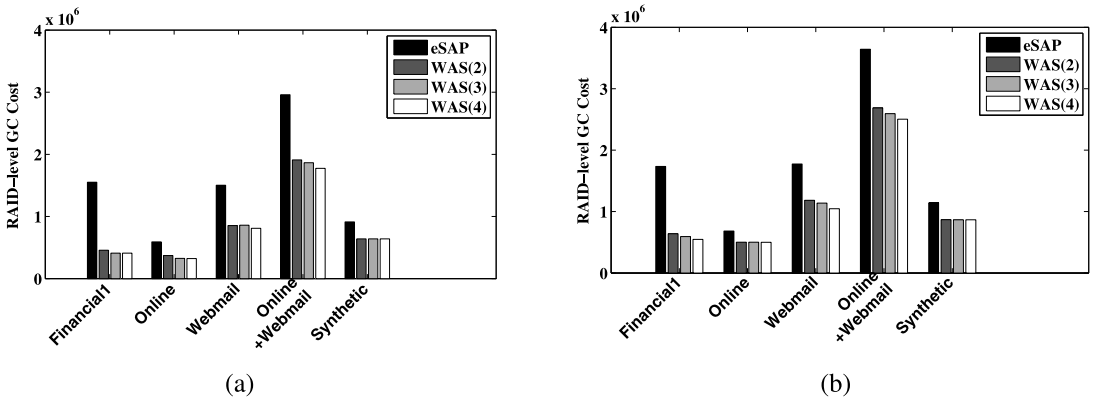


Fig. 7. RAID-level GC cost. (a) RAID-5. (b) RAID-6.

the endurance of SSD RAID by using the widely accepted SSD simulator, DiskSim [11] with SSD extensions [1].

We implement our prototype using C programming language with the gcc version 4.9.2.0. For comparison, we also implement the eSAP scheme. We set the chunk size as 4 KB, and so the stripe size is 32 KB as we have eight SSDs in the RAID. We set the parameter P_{wait} used in eSAP as 200 ms, which means that data chunks will first be buffered in a cache and wait for at most 200 ms before flushing to SSDs. We fix the working set size in each SSD as 256 MB in our experiments, thus, it is a total of 2 GB in the whole RAID composed of eight SSDs. According to the statistics of the I/O workloads in Table I, we find that each workload writes more than 10 GB and less than 40 GB data, and the total size of unique chunk writes for each workload is no more than 1 GB. Therefore, 2 GB is large enough to store all data chunks, and so the system needs to trigger a large amount of RAID-level GC operations under this setting. Furthermore, our prototype and eSAP will call RAID-level GC operations when more than 90% of space in the array has been utilized. In order to validate the effectiveness of our scheme in different RAID levels, we implement the RAID-5 code and RDP code of RAID-6 in the prototype.

In the implementation of our hot data identification scheme, we set the number of LRU lists and the number of data items in each list as 128 and 8, respectively. Based on this setting, the memory consumption of GLRU is around 5 KB. In terms of the threshold setting in our hot data identification scheme, we note that many exiting hot data identification schemes require to set a similar threshold. In particular, one simple counting-based method that is previously used to identify a hot data page is to check whether the page that is being accessed received writes or not before, i.e., whether the page receives more than one access or not. Besides, due to the temporal locality in real-world workloads, the frequently updated data is usually accessed more than once in a short time. Thus, we set the threshold as two in our experiments. We point out that different thresholds correspond to different definitions of hot data. If the threshold is larger, then a smaller amount of data is defined as hot. By using the same threshold as in a counting-based method, which implies to use the same definition of hot data, our scheme almost identifies all the hot data, but it can save a lot of memory consumption than counting-based method, because it does not need to record the information of all logical addresses, but only needs to maintain the hot data table.

C. RAID-Level GC Cost

In this evaluation, in order to generate full-stripe writes at the RAID level and have the potential to issue full-block writes to SSDs in the write module, we initialize the cache size, Size_G, as BlockSize $\times$ (RAIDSize $-$ 1) $=$ 64 $\times$ (8 $-$ 1) $=$ 448 for each group in an RAID-5 array, and set it as BlockSize $\times$ (RAIDSize $-$ 2) $=$ 64 $\times$ (8 $-$ 2) $=$ 384 for RAID-6. Since we set the chunk size as 4 KB, each group needs the memory space of 448×4 KB $=$ 1792 KB and 384×4 KB $=$ 1536 KB for RAID-5 and RAID-6, respectively. We set the number of groups as 2–4. In this section, we compare the RAID-level GC cost of our scheme under different settings with eSAP. In particular, for each workload, we evaluate the RAID-level GC cost by calculating the total number of chunk rewrites incurred by RAID-level GC. For ease of presentation, we denote our scheme as WAS.

Fig. 7 shows the results of RAID-level GC cost which denotes the total number of chunk rewrites caused by RAID-level GC operations. The number in the brackets after WAS indicates the number of groups used in the experiment. For example, WAS(2) means that we configure our prototype to have two groups in the caching module. That is, data chunks are classified into only two types, either hot or cold. Results show that our prototype with WAS can reduce the RAID-level GC cost compared with eSAP under all settings. In particular, even if we only classify data chunks into two types, our scheme WAS(2) shown in Fig. 7(a) can still reduce up to 70.6% of chunk writes compared with eSAP under Financial1 workload, and it reduces 30.0%–43.2% of chunk writes for other four workloads. For RAID-6 as shown in Fig. 7(b), WAS(2) can also reduce 23.9%–63.2% of chunk writes under all workloads.

Besides, we can also note that as the number of groups increases by classifying data chunks into more categories, the RAID-level GC cost further reduces. The main reason is that with more groups, the hotness classification of data chunks is more accurate, and data chunks in the same group should be updated with more even probability. However, we see that the improvement becomes negligible when we have more than

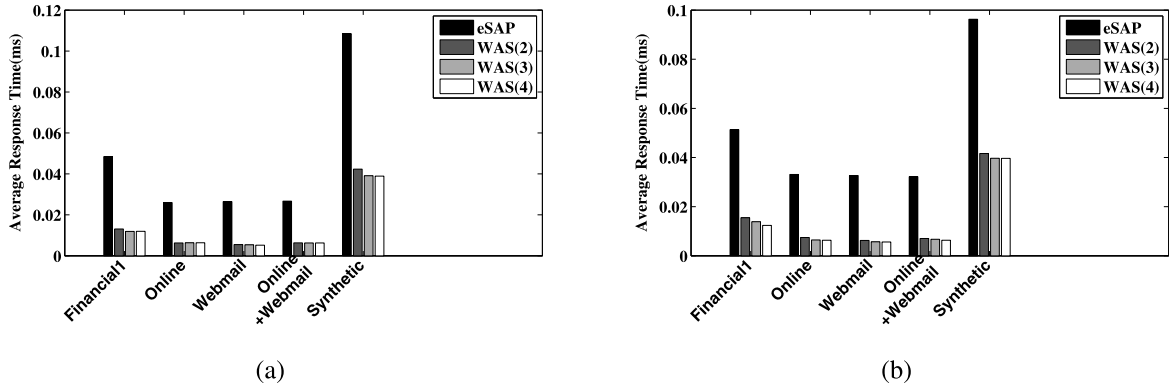


Fig. 8. Average response time. (a) RAID-5. (b) RAID-6.

two groups. The main reason is that after dividing data chunks into two groups, most of data chunks in the same group already have similar hotness, that is, we already efficiently separate the hot and cold data and well utilize the locality information. So there is no big benefit to further divide each group into multiple groups.

Considering that using more groups may also incur larger overhead, as the more groups a system has, the more memory consumption and higher computational cost it requires. In particular, according to the settings in our experiment, WAS(3) and WAS(4) consume one and two more groups of memory than WAS(2), and the group size is 1792 KB for RAID-5 and 1536 KB for RAID-6, respectively. Thus, we suggest to set the number of groups as two in general cases as WAS(2) already achieves a good performance. However, if there is no resource limitations in deploying a system, then more than two groups can be used for pursuing the best performance.

D. Average Response Time

In this section, we evaluate the average response time under different workloads. We vary the number of groups from 2 to 4. We record the total number of requests and the response time of each request, then compute the average response time. We still compare our scheme with eSAP so as to show the effectiveness and efficiency of workload awareness.

Fig. 8 shows the experimental results under different RAID settings and workloads. First of all, we see that compared with eSAP which is more busy in doing RAID-level GC operations, the average response time of the SSD RAID deployed with our scheme can be significantly reduced in both RAID-5 and RAID-6 setups. In particular, WAS(2) in the RAID-5 setup reduces the average response time by 60.9%–79.3% compared with eSAP under different workloads, while it reduces the response time by 56.8%–80.9% for RAID-6.

Second, we see that for our scheme, as the number of groups increases, e.g., from 2 to 4, the average response time further decreases, mainly because the hotness of data chunks within the same group is more similar when the classification scheme is configured in a more fine-grained manner by setting more groups. However, the further reduction of response time when setting more groups becomes small, e.g., the average response time under the settings of 2–4 groups is very close. The main reason is that the hotness of most data chunks in a group already becomes similar even under the setting of two groups, so further separating data chunks into more groups does not bring big benefit. This trend conforms to our previous analysis on RAID-level GC cost.

At last, we can observe that the average response time under the Synthetic workload is much longer than that under other workloads for both eSAP and our scheme. The main reason is that the Synthetic workload does not possess high temporal locality, so the ratio of the number of chunk writes caused by RAID-level GC to the total number of chunk writes becomes bigger. In other words, the SSD RAID is more busy in doing RAID-level GC operations under the Synthetic workload than other traces.

In a summary, based on the comparison results of RAID-level GC cost and average response time, we conclude that our scheme, e.g., WAS(2), significantly improves the performance of SSD RAID with eSAP.

E. System Endurance and Reliability

We first evaluate the system endurance by measuring the total number of erasures performed on the whole SSD RAID array. We also compare our scheme with eSAP under different workloads and RAID settings. Note that since we cannot access the exact information about the number of erasures being performed for commercial SSDs, we conduct this evaluation by using the DiskSim simulator with SSD extensions.

Fig. 9 shows the results of total number of erasures performed on the whole array by using our scheme and eSAP. Again, we also consider 2–4 groups in hot data identification for our scheme. From Fig. 9, we see that compared with eSAP, WAS(2) can reduce 22.9%–63.2% of erase operations under all workloads under RAID-5 setting, and it reduces 23.0%–61.8% of erase operations for RAID-6. Besides, we can also observe that by separating data chunks into more groups, we can reduce more erasures to the whole array, but the marginal reduction, i.e., the additional reduction of erasures by increasing one more group, is very small when setting more than two groups. This observation and its rationale both conform to the analysis on RAID-level GC cost and average response time in the above two sections.

Note that we adopt a log-structured manner to append writes into SSD RAID, so our scheme also balances writes

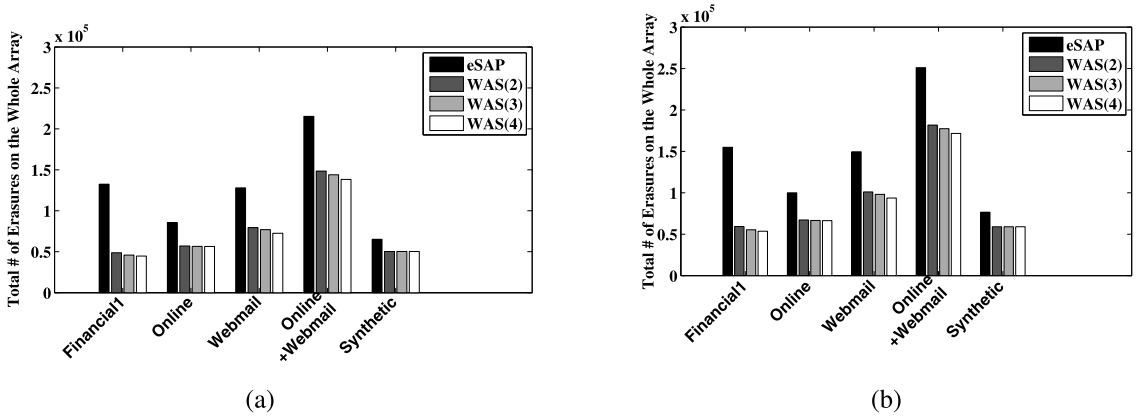


Fig. 9. System endurance. (a) RAID-5. (b) RAID-6.

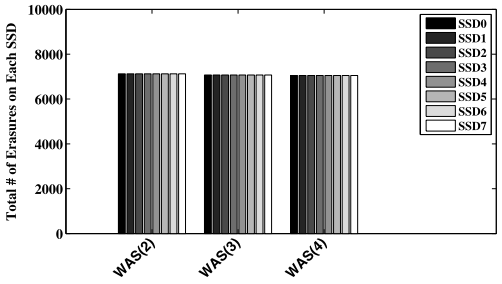


Fig. 10. System-level wear-leveling.

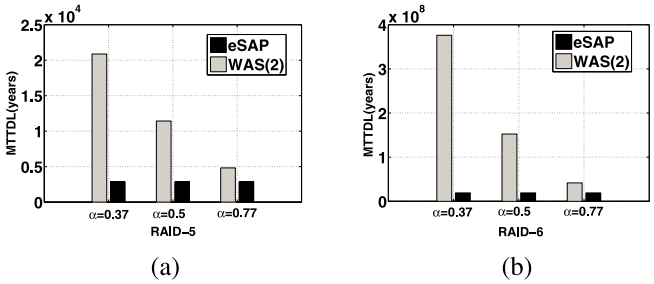


Fig. 11. RAID reliability. (a) RAID-5. (b) RAID-6.

among SSDs. This benefits the system-level wear-leveling among SSDs in the whole RAID array. To validate this, we further show the number of erasures performed on each SSD under different workloads. Fig. 10 shows the results of our scheme in the RAID-5 setup under the Online workload with different configurations. Results under the other four workloads and different setups share the same trend, so we ignore them in the interest of space. We see that our scheme achieves a very good system-level wear-leveling under all settings. Therefore, each SSD will get a similar aging rate with our RAID scheme.

Now we consider the reliability of SSD RAID with eSAP and our scheme WAS(2). We first characterize the error rate of each SSD after using eSAP and our scheme for parity update. Note that it is a common consensus that the error rate of an SSD increases as it undergoes more erasures, so we assume that the error arrive rate of each SSD after handling the same workload by using eSAP and our scheme is proportional to the number of erasures being performed. Mathematically, we denote the error arrive rate of each SSD by using eSAP to handle a workload as λ, then the error arrival rate of each SSD by using our scheme to handle the same workload can be computed as $\alpha\lambda$, where α is ratio of erasures with our scheme to that of eSAP. So α can be set as [0.37, 0.77] for RAID-5, and [0.38, 0.77] for RAID-6, as WAS(2) can reduce 22.9%–63.2% of erase operations for RAID-5 and 23.0%–61.8% of erase operations for RAID-6 than eSAP.

To compare the reliability of eSAP and WAS(2), we define a metric based on mean-time-to-data-loss (MTTDL). In particular, after handling the same workload, the error arrival rates of each SSD by using eSAP and WAS(2) are denoted as λ and $\alpha\lambda$, and we define RAID reliability by evaluating how long the RAID system can sustain until data loss happens if SSD failure arrives according to the rates λ and $\alpha\lambda$, respectively. We assume that the recovery rate for eSAP and WAS(2) is the same, which we denote as μ, and we denote the total number of SSDs in the RAID array as n. Now we can derive the MTTDL of SSD RAID with eSAP, which we denote as $\text{MTTDL}_{\text{eSAP}}$, as follows:

$$\text{MTTDL}_{\text{eSAP}} = \begin{cases} \dfrac{\mu + (2n-1)\lambda}{n(n-1)\lambda^2}, & \text{RAID-5} \\ \dfrac{\mu^2 + 2(n-1)\lambda\mu + \left(3n^2 - 6n + 2\right)\lambda^2}{n(n-1)(n-2)\lambda^3}, & \text{RAID-6.} \end{cases}$$

Similarly, by replacing λ with $\alpha\lambda$ in the above equation, we can derive the reliability of SSD RAID with WAS(2).

Now we perform numerical analysis to compare the reliability of eSAP and WAS(2) by using the above equations. In particular, we set n as 8, and set λ as 0.25 by assuming a four year average lifetime for SSDs. For the recovery rate, we set it as $\mu = 10^4$ by configuring the SSD capacity as 400 GB and the I/O throughput for recovery as 100 Mb/s. For the parameter α, based on the above analysis, we set it as 0.37, 0.5, and 0.77. Fig. 11 shows the reliability results under different settings. We see that more erasures a scheme can reduce, then the higher reliability it can achieve. In particular, compared to eSAP, WAS(2) improves RAID reliability from 2× to 20× under different RAID settings in terms of MTTDL.

F. Design Tradeoffs of Parity Update Schemes

As we introduced in Section II-A, parity update schemes for SSD RAID can be classified into three

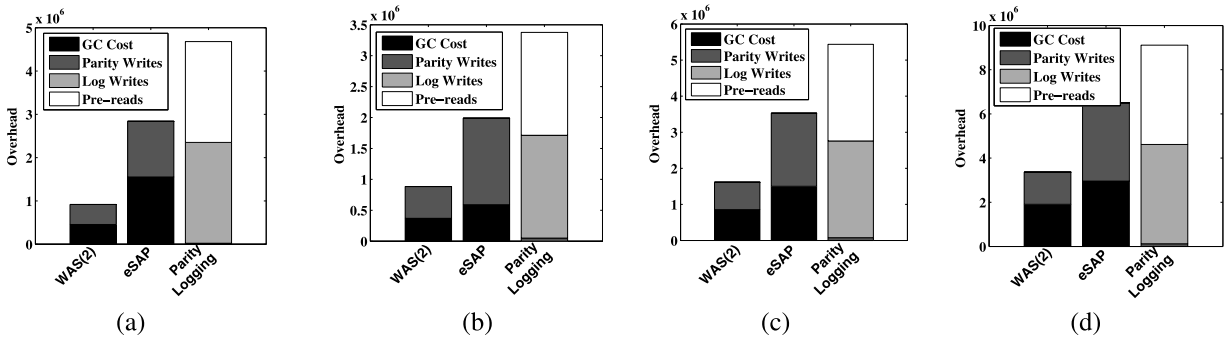


Fig. 12. Comparison of eSAP, WAS(2), and parity logging. (a) Financial1. (b) Online. (c) Webmail. (d) Online+Webmail.

categories: 1) parity logging; 2) parity caching; and 3) elastic striping, and they pose different design tradeoffs. In this section, we discuss their tradeoffs in details by evaluating their I/O overhead, which is defined as the amount of additional reads and writes required to handle all user requests of a workload.

We first focus on parity logging, which uses a dedicated device to store log chunks that are computed as the delta between the old version and the new version of data chunks belonging to the same stripe. Note that generating log chunks requires to preread the old data/parity chunks, and write back the log chunks to log devices. For performance consideration, we directly write data and parities to SSD RAID for full-stripe writes, rather than generating log chunks, so as to reduce the amount of preread and log chunks. Besides, we assume that the capacity of log device is large enough so that all log chunks can be kept in log devices without writing back to SSD RAID. Therefore, the overhead of parity logging includes three parts: 1) log chunk writes; 2) prereads, which are required when generating log chunks; and 3) parity writes, which are caused by full-stripe writes.

We now consider eSAP and our WAS, which are in the category of elastic striping. For our scheme, we focus on WAS(2) in this experiment, which classifies data into two types only (either hot or cold). As we described in Section II-A, the overhead of eSAP and WAS(2) mainly consist of two parts: 1) GC cost, which corresponds to the data movement caused by RAID-level GC and 2) parity writes, which are caused by writing full stripes to SSD RAID.

Fig. 12 shows the overhead of parity logging, eSAP, and WAS(2) under different workloads. First of all, we see that compared to eSAP, our scheme WAS(2) can significantly reduce the I/O overhead by exploiting workload awareness, and the benefit mainly comes from the reduction of RAID-level GC cost as analyzed before. Besides, different from elastic striping, parity logging generates a lot of log chunks and also requires a lot of prereads. Note that log chunks are written to log devices only, but not distributed across SSDs, so even though parity logging can reduce the amount of parity writes, it requires a careful design to efficiently address the writes of log chunks. Otherwise, the log device is very likely to be posed as a bottleneck and severely degrades device-level parallelism and system performance.

At last, for parity caching, which first caches updated data in buffer and then flushes to SSD RAID when buffer becomes full, its overhead heavily depends on the buffer size. In particular, if the buffer is extremely small, then this scheme corresponds to the conventional RAID scheme, which updates parities by using either RMW or RRW. As shown in [12], eSAP can outperform conventional RAID-5 by up to an order of magnitude in terms of the parity overhead for random write dominant workload, and we already show that WAS(2) can further reduce the RAID-level GC cost than eSAP. On the other hand, if the buffer size is large enough, and suppose that all updated data can be kept in memory, then no overhead is introduced. However, if more data is kept in memory for parity caching, then the lower reliability the system can achieve if DRAM is used for caching. Therefore, there is a tradeoff between performance and reliability for parity caching, so it is unfair to directly compare it with parity logging or elastic striping schemes.

In summary, we see that all the three classes of parity update schemes pose a design tradeoff. On the one hand, they reduce parity writes by deferring the updates to parity chunks in SSD RAID in different ways. On the other hand, each of them has its own drawbacks. Parity caching requires to keep data in memory, and so it sacrifices system reliability. Parity logging requires additional log devices, and it also necessitates a careful design to guarantee system performance when applying to SSD RAID. Elastic striping does not need additional devices, but it requires RAID-level GC, which may incur a large cost. We emphasize that the main focus of this paper is to reduce RAID-level GC cost of elastic striping by exploiting workload awareness, so as to make the best use of its advantages and bypass its disadvantages simultaneously.

G. Discussions on Hot Data Identification Schemes

To further show the effectiveness and efficiency of our hot data identification scheme GLRU, we compare it with a direct counting-based approach which records the information of all data chunks. In particular, we implement the counting based approach by extending the hotness computation algorithm used in previous work [25]. Hereafter, we call this scheme hotness computation algorithm (HCA).

HCA works as follows. It first defines the hotness of a data chunk C as $H_C = (W_F/N_F + W_C/\Delta\text{Age}_C)$, where W_C is the total number of updates to chunk C, ΔAge_C denotes the time interval between the last two updates to chunk C, W_F is the total number of updates to the file to which chunk C belongs, and N_F is the total number of chunks in this file. Second, for a group of data chunks G, HCA defines the group similarity as $S_G = (1/N_C)\sum_{C\in G} H_C$, where N_C is the number of chunks

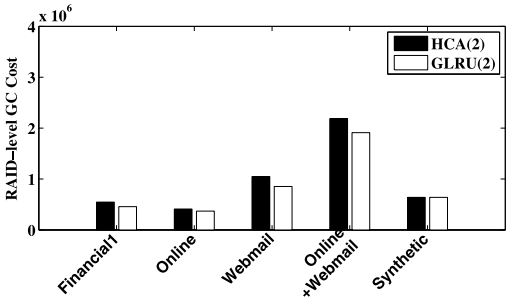


Fig. 13. Comparison of RAID-level GC cost between different hot data identification schemes.

that are assigned to group G. At last, for a new chunk C_i, it is assigned to group G_j if and only if S_{G_j} is the closest one to H_{C_i}. After assigning C_i to a group, we update the group similarity accordingly.

HCA also performs well in our evaluation, but it requires considerable memory space to maintain the metadata for every data chunk. In particular, we use 8 bytes to record the access time of a data chunk and use 2 bytes to record the write count of a data chunk, HCA will require around 2.5 MB memory space for storing the hotness information for each gigabytes of data on SSDs. With the increment of the storage capacity of SSDs, the memory cost for storing metadata becomes very large. Different from HCA, the memory cost of our new hot data identification scheme GLRU does not increase with the storage capacity of SSDs, and it is fixed as long as the number of LRU lists and the number of data items in each list are fixed. For example, according to the settings used in our experiments, our scheme requires only around 5 KB memory space regardless of the SSD capacity. Besides, GLRU performs even better than HCA, as measured by RAID-level GC cost. Fig. 13 shows the RAID-level GC cost of our WAS, but incorporating HCA and GLRU, respectively. We can easily see that GLRU can further reduce the RAID-level GC cost by up to 16.7% compared to HCA when we consider two groups in hot data identification.

We further make a discussion on the impact of number of data types used in the hot data identification scheme, and this parameter also determines the number of groups in the caching module. As we mentioned before, the more groups the caching module keeps, the higher accuracy of the hot/cold data classification we can achieve. However, caching data also consumes physical memory which is a scarce resource. From the evaluation results of RAID-level GC cost in Fig. 7 and the average response time in Fig. 8, we find that the performance improvement diminishes when the number of groups reaches a certain value, and this value may be different for different workloads, which depends on the skewness and temporal locality of the workload. However, we emphasize that separating data into two types already works well.

H. Summary

Based on the experimental results shown in the previous sections, we conclude that with hotness-aware elastic striping, our proposed RAID scheme can not only reduce the RAID-level GC cost and the number of writes to SSD RAID, it can also reduce the average I/O response time. Meanwhile, our scheme also reduces the total number of erase operations performed on SSDs and improves SSD RAID reliability. Therefore, our WAS improves both the performance and endurance of SSD RAID.

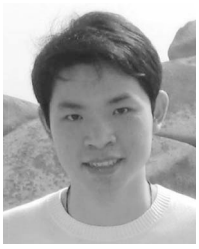
VI. Conclusion

In this paper, we proposed a WAS to take advantage of the skewness and temporal locality of workloads to enhance the performance and endurance of SSD RAID. Our scheme first classifies data chunks into different types according to their hotness and buffers them in a cache within different groups, it then writes data chunks to the underlying SSDs with a hotness-aware elastic striping approach and log-structured write policy. We developed a prototype to implement our scheme and conducted extensive experiments on commercial SSDs. Results validated that our scheme improves both the performance and endurance of SSD RAID.

References

[1] N. Agrawal *et al.*, "Design tradeoffs for SSD performance," in *Proc. USENIX ATC*, Boston, MA, USA, 2008, pp. 57–70.
[2] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 192–202, Feb. 1995.
[3] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. SIGMETRICS*, Seattle, WA, USA, 2009, pp. 181–192.
[4] C.-C. Chung and H.-H. Hsu, "Partial parity cache and data cache management method to improve the performance of an SSD-based RAID," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 7, pp. 1470–1480, Jul. 2014.
[5] P. Corbett *et al.*, "Row-diagonal parity for double disk failure correction," in *Proc. USENIX FAST*, San Francisco, CA, USA, 2004, pp. 1–14.
[6] M. E. Gomez and V. Santonja, "Characterizing temporal locality in I/O workload," in *Proc. Int. Symp. Perform. Eval. Comput. Telecommun. Syst.*, 2002.
[7] L. M. Grupp *et al.*, "Characterizing flash memory: Anomalies, observations, and applications," in *Proc. Micro*, New York, NY, USA, 2009, pp. 24–33.
[8] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of NAND flash memory," in *Proc. USENIX FAST*, San Jose, CA, USA, 2012, p. 2.
[9] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Trans. Stor.*, vol. 2, no. 1, pp. 22–40, 2006.
[10] S. Im and D. Shin, "Flash-aware RAID techniques for dependable and high-performance flash memory SSD," *IEEE Trans. Comput.*, vol. 60, no. 1, pp. 80–92, Jan. 2011.
[11] B. John, S. Jiri, S. Steve, and G. Greg. *The DiskSim Simulation Environment (v4.0)*. (Jan. 2015). [Online]. Available: http://www.pdl.cmu.edu/DiskSim/
[12] J. Kim, J. Lee, J. Choi, D. Lee, and S. H. Noh, "Improving SSD reliability with RAID via elastic striping and anywhere parity," in *Proc. IEEE/IFIP DSN*, Budapest, Hungary, 2013, pp. 1–12.
[13] H.-S. Lee, H.-S. Yun, and D.-H. Lee, "HFTL: Hybrid flash translation layer based on hot data identification for flash memory," *IEEE Trans. Consum. Electron.*, vol. 55, no. 4, pp. 2005–2011, Nov. 2009.
[14] S. Lee, B. Lee, K. Koh, and H. Bahn, "A lifespan-aware reliability scheme for RAID-based flash storage," in *Proc. ACM Symp. Appl. Comput.*, Taichung, Taiwan, 2011, pp. 374–379.
[15] S.-W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM SIGMOD*, Beijing, China, 2007, pp. 55–66.
[16] Y. Lee, S. Jung, and Y. H. Song, "FRA: A flash-aware redundancy array of flash storage devices," in *Proc. 7th IEEE/ACM Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Grenoble, France, 2009, pp. 163–172.
[17] Y. Li, H. H. W. Chan, P. P. C. Lee, and Y. Xu, "Elastic parity logging for SSD RAID arrays," in *Proc. IEEE/IFIP DSN*, Toulouse, France, 2016.
[18] Y. Li, P. P. C. Lee, and J. C. S. Lui, "Stochastic modeling of large-scale solid-state storage systems: Analysis, design tradeoffs and optimization," in *Proc. ACM SIGMETRICS*, Pittsburgh, PA, USA, 2013, pp. 179–190.

[19] Y. Li, P. P. C. Lee, J. C. Lui, and Y. Xu, "Impact of data locality on garbage collection in SSDs: A general analytical study," in *Proc. ACM/SPEC ICPE*, Austin, TX, USA, 2015, pp. 305–315.
[20] B. Mao *et al.*, "HPDA: A hybrid parity-based disk array for enhanced performance and reliability," *ACM Trans. Stor.*, vol. 8, no. 1, 2012, Art. no. 4.
[21] C. Metz. *Flash Drives Replace Disks at Amazon, Facebook, Dropbox*. (Jan. 2015). [Online]. Available: http://www.wired.com/2012/06/flash-data-centers/all/
[22] N. Mielke *et al.*, "Bit error rate in NAND flash memories," in *Proc. IEEE Int. Rel. Phys. Symp.*, Phoenix, AZ, USA, 2008, pp. 9–19.
[23] A. Miranda and T. Cortes, "CRAID: Online raid upgrades using dynamic hot data reorganization," in *Proc. USENIX FAST*, Santa Clara, CA, USA, 2014, pp. 133–146.
[24] J. Ouyang *et al.*, "SDF: Software-defined flash for Web-scale Internet storage systems," in *Proc. ASPLOS*, Salt Lake City, UT, USA, 2014, pp. 471–484.
[25] Y. Pan, Y. Li, Y. Xu, and Z. Li, "Grouping-based elastic striping with hotness awareness for improving SSD raid performance," in *Proc. IEEE/IFIP DSN*, Rio de Janeiro, Brazil, 2015, pp. 160–171.
[26] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proc. ACM SIGMOD*, Chicago, IL, USA, 1988, pp. 109–116.
[27] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[28] D. S. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proc. USENIX ATEC*, San Diego, CA, USA, 2000, p. 4.
[29] D. Stodolsky, G. Gibson, and M. Holland, "Parity logging overcoming the small write problem in redundant disk arrays," *ACM SIGARCH Comput. Architect. News*, vol. 21, no. 2, pp. 64–75, May 1993.
[30] (2002). *Storage Performance Council*. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage
[31] B. Van Houdt, "A mean field model for a class of garbage collection algorithms in flash-based solid state drives," in *Proc. ACM SIGMETRICS*, Pittsburgh, PA, USA, 2013, pp. 191–202.
[32] B. Van Houdt, "Performance of garbage collection algorithms for flash-based solid state drives with hot/cold data," *Perform. Eval.*, vol. 70, no. 10, pp. 692–703, 2013.
[33] A. Verma, R. Koller, L. Useche, and R. Rangaswami, "SRCMap: Energy proportional storage using dynamic consolidation," in *Proc. USENIX FAST*, San Jose, CA, USA, 2010, pp. 267–280.
[34] L. Xu and J. Bruck, "X-code: MDS array codes with optimal encoding," *IEEE Trans. Inf. Theory*, vol. 45, no. 1, pp. 272–276, Jan. 1999.
[35] Y. Yang and J. Zhu, "Analytical modeling of garbage collection algorithms in hotness-aware flash-based solid state drives," in *Proc. IEEE MSST*, Santa Clara, CA, USA, 2014, pp. 1–10.
[36] L. Zeng *et al.*, "HRAID6ML: A hybrid RAID6 storage architecture with mirrored logging," in *Proc. IEEE 28th Symp. MSST*, Pacific Grove, CA, USA, 2012, pp. 1–6.

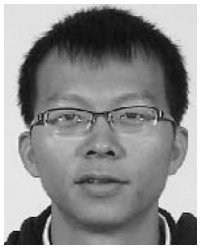


Yongkun Li received the B.Eng. degree in computer science from University of Science and Technology of China, Hefei, China, in 2008, and the Ph.D. degree in computer science and engineering from the Chinese University of Hong Kong, Hong Kong, in 2012.

He is currently an Associate Researcher with the School of Computer Science and Technology, University of Science and Technology of China. He was a Post-Doctoral Fellow with the Institute of Network Coding, the Chinese University of Hong Kong. His current research interests include performance evaluation and architectural design of networking and storage systems.

Biaobiao Shen received the bachelor's degree in computer science from the Anhui University of Technology, Ma'anshan, China, in 2013, and the master's degree from the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China, in 2016.

His current research interests include solid-state devices and distributed storage systems.



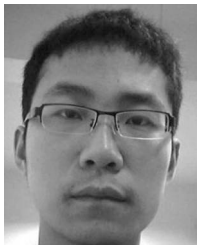
Yubiao Pan received the B.S. and Ph.D. degrees from the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China, in 2010 and 2015, respectively.

His current research interests include solid-state devices, distributed storage system, and data deduplication.

Yinlong Xu received the B.S. degree in mathematics from Peking University, Beijing, China, in 1983, and the M.S. and Ph.D. degrees in computer science from the University of Science and Technology of China (USTC), Hefei, China, in 1989 and 2004, respectively.

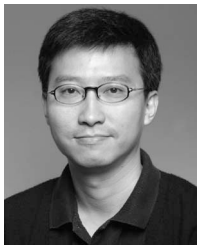
He is currently a Professor with the School of Computer Science and Technology, USTC. He served the Department of Computer Science and Technology, USTC as an Assistant Professor, a Lecturer, and an Associate Professor. He is currently leading a group of research students in doing some networking and high performance computing research. His current research interests include network coding, wireless network, combinatorial optimization, design and analysis of parallel algorithm, and parallel programming tools.

Prof. Xu was a recipient of the Excellent Ph.D. Advisor Award of the Chinese Academy of Sciences in 2006.



Zhipeng Li received the bachelor's degree from the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China, in 2010, where he is currently pursuing the Ph.D. degree.

His current research interests include storage system, especially NAND flash solid-state devices-based storage systems and lager-scale storage systems.



John C. S. Lui (F'10) received the Ph.D. degree in computer science from the University of California at Los Angeles, Los Angeles, CA, USA.

He is currently a Professor with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong. His current research interests include communication networks, network/system security, network economics, network sciences, cloud computing, large scale distributed systems, and performance evaluation theory.

Prof. Lui serves in the editorial board of the IEEE/ACM TRANSACTIONS ON NETWORKING, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the *Journal of Performance Evaluation*, and the *International Journal of Network Security*. He is an Elected Member of the IFIP WG 7.3, a fellow of ACM, a fellow of IEEE, and Croucher Senior Research Fellow.