Front. Comput. Sci., 2019, 13(5): 913–928
https://doi.org/10.1007/s11704-018-7009-0

RESEARCH ARTICLE

Exploiting flash memory characteristics to improve performance of RAIS storage systems

Linjun MEI, Dan FENG (✉), Lingfang ZENG, Jianxi CHEN, Jingning LIU

Wuhan National Laboratory for Optoelectronics, School of Computer, Huazhong University of Science and Technology, Wuhan 430074, China

Abstract Redundant array of independent SSDs (RAIS) is generally based on the traditional RAID design and implementation. The random small write problem is a serious challenge of RAIS. Random small writes in parity-based RAIS systems generate significantly more pre-reads and writes which can degrade RAIS performance and shorten SSD lifetime. In order to overcome the well-known write-penalty problem in the parity-based RAID5 storage systems, several logging techniques such as Parity Logging and Data Logging have been put forward. However, these techniques are originally based on mechanical characteristics of the HDDs, which ignore the properties of the flash memory.

In this article, we firstly propose RAISL, a flash-aware logging method that improves the small write performance of RAIS storage systems. RAISL writes new data instead of new data and pre-read data to the log SSD by making full use of the invalid pages on the SSD of RAIS. RAISL does not need to perform the pre-read operations so that the original characteristics of workloads are kept. Secondly, we propose AGCRL on the basis of RAISL to further boost performance. AGCRL combines RAISL with access characteristic to guide read and write cost regulation to improve the performance of RAIS storage systems. Our experiments demonstrate that the RAISL significantly improves write performance and AGCRL improves both of write performance and read performance. AGCRL on average outperforms RAIS5 and RAISL by 39.15% and 16.59% respectively.

Received January 9, 2017; accepted June 12, 2018

E-mail: dfeng@hust.edu.cn

Keywords solid state drives, RAIS, properties, performance

1 Introduction

Due to mechanical properties, the speed of HDD already cannot satisfy the needs of users. SSD is increasingly deployed to construct RAID [1, 2] in enterprise environments with its price dropping and technology maturing [3–6]. The design of SSD is completely different from HDD. SSD has high performance and low power consumption because it does not seek and rotate when reading and writing [7, 8]. However, RAIS is generally based on the traditional redundant array of independent disks design and implementation [9, 10]. When handling random writes, the parity-based RAIS has to read and write the parity blocks frequently which degrades the performance and shortens SSD lifetime.

Numerous studies are conducted on logging techniques to improve the traditional RAID write performance. In order to overcome small-write problem, Parity Logging [11] delays parity update by writing the parity updates to the log disk sequentially. Menon [12] propose a log-structure array (LSA) that combines LFS, RAID5, and a non-volatile cache. Instead of writing in-place, LSA writes the updated data into a new disk to improve the write performance of RAID5. Unfortunately, these techniques were devised for HDD, which cannot fully exploit the characteristics of SSD.

In addition to out-of-place update, SSD has a write and read cost characteristic as well. The write operation of flash memory is performed by the incremental-step pulse programming (ISPP) [13] scheme, which uses a small verification voltage to reliably program flash cells to their specified voltages. The level of the verification voltage will affect the write and read cost of the SSD. In brief, with a large step size in the ISPP process, the write cost is reduced, but the read cost increases. With finer step size in the ISPP process, the write cost increases, but the read cost decreases. Li et al. [14] propose AGCR, an access characteristic guided cost regulation scheme, which exploits the above tradeoff to improve flash performance. Based on workload characteristics, logical pages receiving more reads will be written using a finer step size so that their read cost will be reduced. Similarly, logical pages receiving more writes will be written using a coarser step size so that their write cost will be reduced.

AGCR presents a study on the access characteristics of several workloads; and the performance improvement of AGCR is based on this access characteristics. However, when workloads are handled by RAID controller, there are many pre-read operations, which can change the access characteristics of the workloads. Thus, AGCR cannot be used in the RAIS storage systems directly. But the original access characteristics of the workloads can be guaranteed with the help of RAISL, which removes the pre-read operations. AGCRL combines RAISL and AGCR to further improve the RAIS storage systems performance. The contributions of this article are described as follows:

- We present the study on the access characteristics of some workloads before and after RAID controller handling. Most of the requests access read-only or write-only pages before handling by RAID controller. However, most of the requests access interleaved-access pages after handling by RAID controller.
- We propose RAISL, a flash-aware logging technique to improve the small write performance of RAIS storage systems by delaying parity update. RAISL makes full use of the invalid pages to remove the pre-read operations.
- We propose AGCRL on the basis of RAISL to further boost performance of RAIS storage systems.
- We prototype and evaluate the proposed RAISL and AGCRL. For performance comparison, we also implement the Parity Logging scheme and Data Logging [15] scheme in the RAIS5, which are called RAIS_PL and RAIS_DL, respectively.

The rest of this paper is organized as follows. We present the background and the motivation in Section 2. The design and implementation of RASL and AGCRL are described in Section 3. The experimental results are presented in Section 4. We review the related work in Section 5 and conclude our paper in Section 6.

2 Background and motivation

2.1 SSD and FTL

Most modern SSDs are constructed by using NAND flash memory. Generally, NAND flash memory can be divided into two categories: single-level cell (SLC) and multi-level cell (MLC). A SLC flash memory cell stores one bit and a MLC flash memory cell can store two bits or more. SSD has several special features. First, SSD cannot update in-place. A page can be written only after it is erased. The unit of read and write operations is a page, whereas the unit of erase operation is a block. Each block is composed of 64 to 256 pages. The size of a page is 2KB or 4KB. Second, the erase times of the flash memory cell is limited. A SLC flash memory has around 100,000 erase cycles and a MLC flash memory has only around 10,000 erase cycles or less. Third, SSD has a write and read cost characteristic. If the write cost of a page is high, then the read cost is low; if the write cost of a page is low, the read cost is high.

Flash translation layer (FTL) is a software layer of SSD, which mainly contains three modules: address mapping, garbage collection, and wear-leveling [16, 17]. When rewriting a logical page, SSD controller assigns a new page for writing and the old page is marked as an invalid. The address mapping module manages a mapping table in the process. When there is not enough free space in the SSD, the garbage collection module reclaims the invalid pages. The wear-leveling module ensures that the erase times of each physical block are almost the same.

2.2 Bio

Bio is the main unit of I/O for the block layer and lower layers. It represents a read or write request from the upper level. Each bio in Linux kernel mainly has the following items:

- *bi_bdev*: block device descriptor of the request.
- *bi_sector*: the start disk sector number to transfer.
- *bi_size*: residual I/O count.
- *bi_rw*: I/O operation flag (bottom bits represent read or write, top bits represent priority).
- *bi_flags*: status of the request.

- *bi_io_vec*: segment descriptor array.
- *bi_next*: pointer to the next bio that belongs to the same request queue.

In Linux kernel, each bio defines 12 kinds of states as shown below:

- *BIO_UPTODATE*: I/O has completed without error.
- *BIO_RW_BLOCK*: I/O would block.
- *BIO_EOF*: out-out-bounds error.
- *BIO_SEG_VALID*: bi_phys_segments valid.
- *BIO_CLONED*: bio does not own data.
- *BIO_BOUNCED*: bio is a bounce bio.
- *BIO_USER_MAPPED*: bio contains user pages.
- *BIO_EOPNOTSUPP*: not supported.
- *BIO_CPU_AFFINE*: complete bio on same CPU as submitted.
- *BIO_NULL_MAPPED*: bio contains invalid user pages.
- *BIO_FS_INTEGRITY*: filesystem owns integrity data.
- *BIO_QUIET*: make bio quiet.

2.3 Access characteristics of workloads

Li et al. [14] defined three types of data accesses: read-only, write-only, and interleaved-access. If almost all the accesses (>95%) to a page are read requests, this page is characterized as read-only. If almost all the accesses (>95%) to a page are write requests, this page is characterized as write-only. If the accesses to a data page are interleaved with reads and writes, this page is characterized as interleaved-access. They observed that most read requests access read-only pages, most write requests access write-only pages, and only a small part of requests access interleaved-access pages. The main idea of the access characteristic guided read and write cost regulation method is to apply low-cost writes for write-only pages, low-cost reads (enabled by applying high-cost write), for read-only pages, and medium-cost accesses for interleaved-access pages.

Table 1 shows the basic characteristics of 10 representative workloads from Microsoft Research (MSR) Cambridge [18]. All workloads are converted to replay in a simulator. The logical block address (LBA) and length of each request are mod 8. We observe that most of the workloads are mainly consist of small write requests. When RAID controller handles write request, it divides the request into sub-requests and adds them on the corresponding stripes, then computes the new parity blocks of each stripes, and writes the new data and parity blocks at last.

Table 1 Trace information

Trace File	Total request number	Write request ratio	Average size/KB
mds_0	91,021	90.49%	7.27
rsrch_0	102,037	90.76%	8.88
rsrch_2	17,573	52.87%	4.13
src2_0	91,139	87.91%	6.86
stg_0	129,590	90.72%	9.30
usr_0	168,422	60.41%	18.66
wdev_0	80,147	78.10%	9.09
hm_0	244,890	68.68%	8.51
proj_0	164,971	64.41%	16.37
proj_3	135,150	1.45%	11.13

There are usually two alternative methods to generate the new parity block, namely, *reconstruction-write* and *read-modify-write* respectively [19–21]. The main difference between the two methods lies in the data blocks that must be pre-read for the computation of the new parity block [22, 23]. *Reconstruction-write* method needs to pre-read the data blocks that are not to be updated. *Read-modify-write* method needs to pre-read the parity block and the data blocks that are to be updated. The RAID controller dynamically chooses the method that leads to less pre-read operations and it will choose *reconstruction-write* when the pre-read operations are the same.

When handling the small write dominated workloads, the controller chooses the *read-modify-write* and generates many pre-read operations that read the pages to be written. Figure 1 shows the total access pages number before and after RAID controller handling. The result shows that the total access pages numbers of the workloads after RAID controller handling are much more than that before handling. The increased access pages numbers are almost three times of the corresponding write access pages number. Because one small

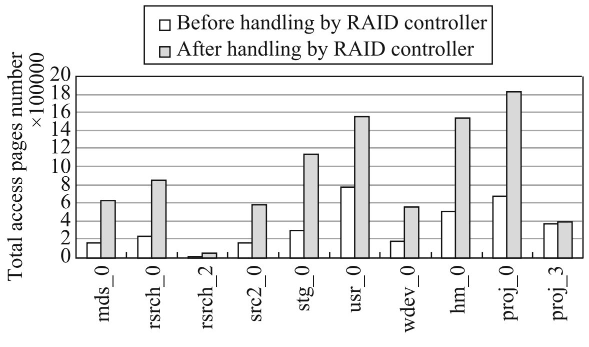


Fig. 1 Total access pages number before and after handling by RAID controller

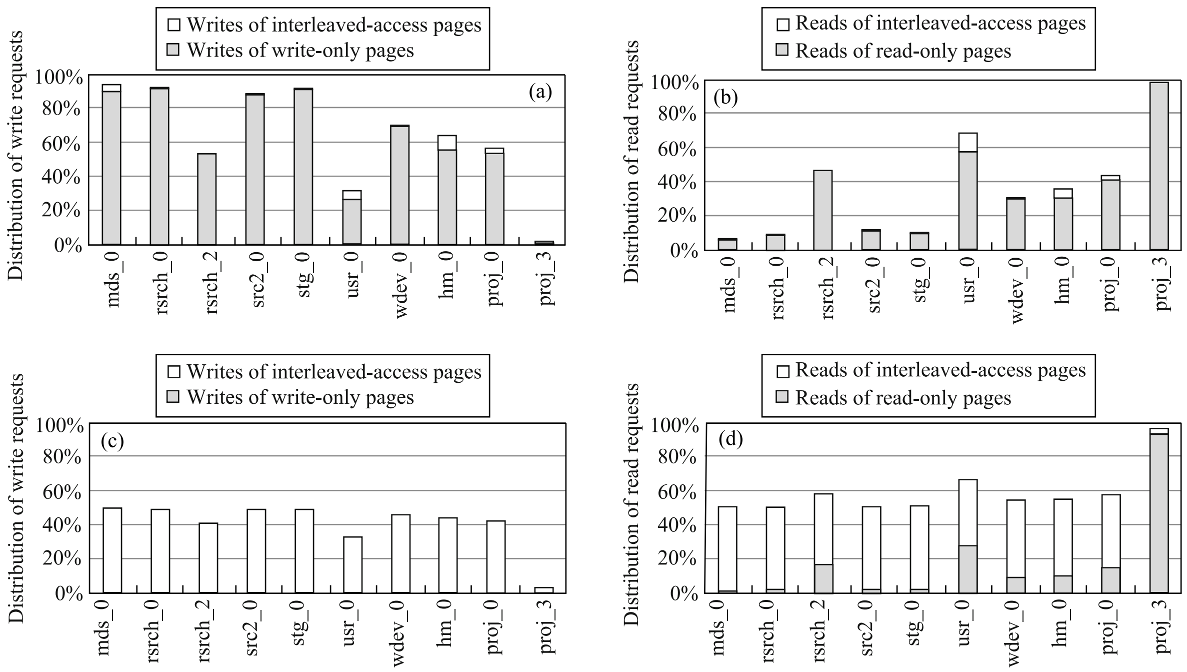


Fig. 2 Distribution of read and write requests on the three access characteristics. (a) Distribution of write requests before handling by RAID controller; (b) distribution of read requests before handling by RAID controller; (c) distribution of write requests after handling by RAID controller; (d) distribution of read requests after handling by RAID controller

write request leads to two pre-reads and two writes. The pre-read operations corrupt the original characteristic of the workloads. The pre-read operation reads the page that will be written later, which makes the page change from write-only to interleaved-access.

Figure 2 shows the distribution of read and write requests on the three access characteristics before and after RAID controller handling. Most of the requests access read-only or write-only pages before handling by RAID controller. However, most of the requests access interleaved-access pages except proj_3 which mainly consists of read requests after handling by RAID controller. The pages that most write requests access change from write-only to interleaved-access after handling by RAID controller. The ratio of the write requests is higher, the ratio of the requests access to interleaved-access pages is larger after handling by RAID controller. Thus, the access characteristic guided read and write cost regulation method [14] cannot be used directly in the RAIS5 storage systems when the workloads consist mostly of write requests.

We measure four combinations of read and write costs in RAIS5 storage systems: 1) All the writes are performed with a high cost, followed by low-cost reads (HWLR); 2) All the writes are performed with a medium cost, followed by medium-cost reads (MWMR); 3) All the writes are performed with a low cost, followed by high-cost reads (LWHR); 4) All the writes are performed with a low cost and all the reads are performed with a low cost (LWLR). The detailed settings for the regulator can be found in Section 4. Figure 3 presents the comparison of access latency for different workloads from MSR. Compared to the HWLR, MWMR, and LWHR, LWLR improves overall performance by 35.21%, 31.75%, and 36.96%, on average. For random write dominated workloads, the overall performance of LWHR is better than that of HWLR and MWMR. For read dominated workloads, the overall performance of HWLR is better than that of LWMR and MWMR. The results show that if page is written and read with low cost, the storage systems can get significant performance improvement.

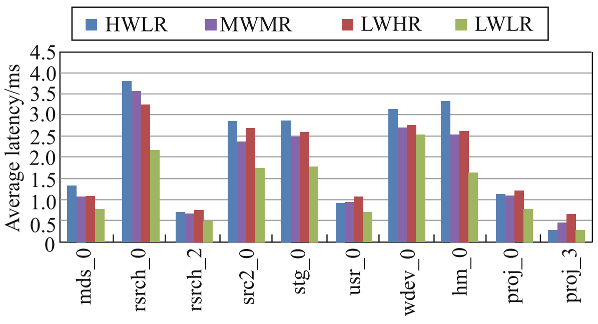


Fig. 3 Overall performance comparison with different access costs

2.4 Logging techniques

As we know, most of the above workloads are small write request dominated. RAID5 has to read the old data and the old parity, and write the new data and the new parity to service the small write request. One small write request results in four I/O operations. Consequently, the small write performance of RAID5 is very bad. There are many logging techniques that delay the parity update to improve the small write performance. Parity Logging scheme writes the XOR result of the old data and the new data to the log disk sequentially. Parity Logging delays the parity block update and transforms random writes into sequential writes to improve performance.

However, the existing log records cannot be reused in the Parity Logging scheme. Thus, the Data Logging scheme is proposed. Instead of writing the XOR result of the old data and the new data, Data Logging scheme writes the old data and the new data to the log disk sequentially. When writing a data block, Data Logging must find whether the data block has been logged or not. If the data block has not been logged, Data Logging needs to write the old data and the new data to the log disk.If the data block has been logged, Data Logging only writes the new data to the log disk.

2.5 Motivation

There are two problems when all the workloads in Table 1 are replayed in RAIS5 systems. The first is the small write problem. The second is that the original characteristic is corruption as described in Section 2.2. Because of parity blocks updating frequently in RAIS5 systems, the small write request not only affects the overall performance, but also affects the lifetime of each SSD. Although the traditional logging techniques can improve the small write performance, they do not take the features of SSD into account. We propose a novel flash-aware logging technique, called RAISL, to improve small write performance. There are many invalid pages in SSD. RAISL makes full use of the invalid pages to removes the pre-read operations so that the original characteristics of workloads are kept. Additionally, combined RAISL with access characteristic guided read and write cost regulation, AGCRL can further improve the performance of RAIS storage systems.

3 Design and implementation

3.1 RAISL

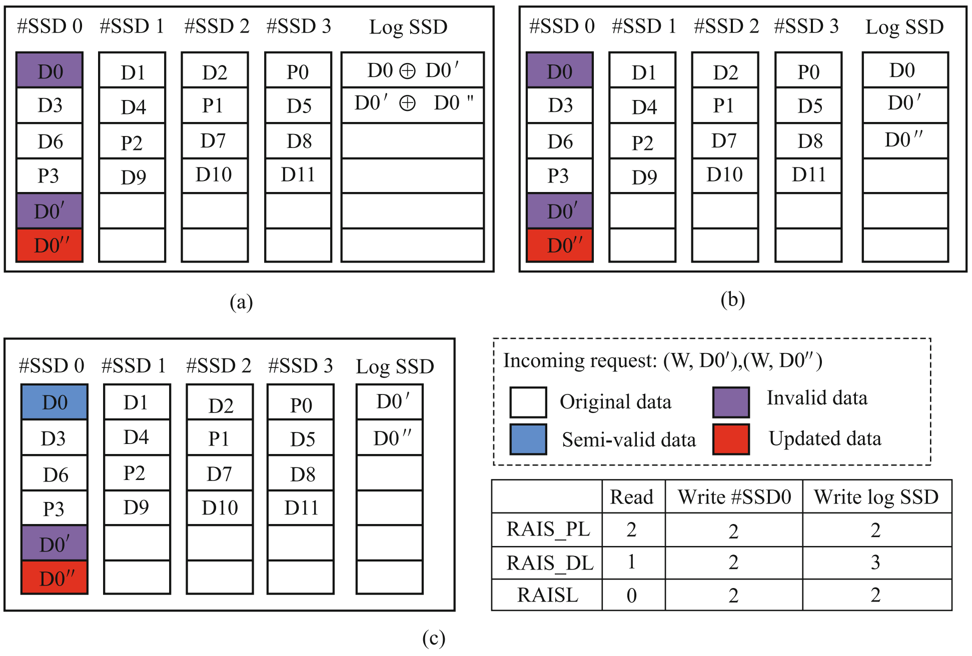
When handling write request, RAISL writes the new data in home location and writes the new data to the log SSD sequentially without updating parity block and pre-reading any other data blocks. Figure 4 shows an illustration of Parity Logging, Data Logging,and the proposed RAISL schemes. We denote each request as a pair (M, N), where M denotes the access mode (Read/Write), and N is the block to be accessed. When D0 is first updated, Parity Logging writes XOR result of D0 and D0′ to the log SSD, and Data Logging writes D0 and D0′ to the log SSD. When D0′ is updated, Parity Logging writes XOR result of D0′ and D0″ to the log SSD. But, Data Logging only writes D0″ to the log SSD. During this process, Parity Logging reads D0 and D0′, and Data Logging reads D0, and write the data block or the XOR result to the log SSD in order to recovery data when there is a failure in other SSDs. For example, when #SSD1 fails, data block D1 needs D0 to recover. Data is updated in-place in the traditional disk. So the traditional logging techniques need to read and write certain data blocks to ensure the reliability when delaying parity update. However, data is updated out-of-place in SSD. D0 and D0′ are still in existence before they are erased by GC operation. Therefore, RAISL only needs to write the new data of the updated block to the log SSD, and does not need to pre-read any data block by recording the physical page number of D0.

To the best of our knowledge, RAISL is the first log technique to delay parity blocks update in RAIS5 storage systems. Parity Logging and Data Logging are the logging techniques that are designed based on the HDDs mechanical characteristic. Compared to Parity Logging and Data Logging, RAISL improves the performance by reducing the number of pre-read operations. As shown in Fig. 4, Parity Logging needs 2 pre-read operations, Data Logging needs 1 pre-read operation, and RAISL needs 0 pre-read operation. In addition, RAISL writes 2 data blocks to the log SSD while Data Logging writes 3 data blocks.

RAISL must record some invalid pages by extending the mapping table when updating data blocks. These pages are valuable before the corresponding parity blocks are updated. RAISL marks these pages as semi-valid state. A semi-valid page becomes invalid when the corresponding parity block is updated. In Fig. 4, when D0 is first updated, D0 is marked as a semi-valid page. When D0′ is updated, D0′ is marked as an invalid page and the state of D0 is still semi-valid.

3.2 Key data structures

There are two key data structures: hash list and extended mapping table as shown in Fig. 5. The hash list, which is composed of a table head array and several entry lists, is used to record the updated data. The table head array contains some hash slots and each slot points to an entry list. Each entry



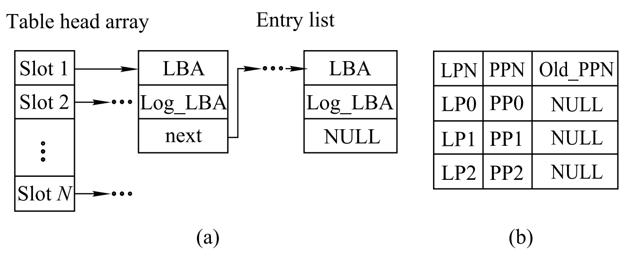
	Read	Write #SSD0	Write log SSD
RAIS_PL	2	2	2
RAIS_DL	1	2	3
RAISL	0	2	2

First request: (W,D0′)
RAIS_PL: (1) read D0 ; (2) write D0⊕D0′ to the Log SSD; (3) write D0 to #SSD0
RAIS_DL: (1) read D0 ; (2) write D0,D0′ to the Log SSD; (3) write D0 to #SSD0
RAISL: (1) write D0′ to the Log SSD; (2) write D0′ to #SSD0
Second request: (W, D0″)
RAIS_PL: (1) read D0′; (2) write D0′⊕D0″ to the Log SSD; (3) write D0″ to #SSD0
RAIS_DL: (1) write D0″ to the Log SSD ; (2) write D0″ to #SSD0
RAISL: (2) write D0″ to the Log SSD ; (2) write D0″ to #SSD0

Fig. 4 Illustration of Parity Logging in RAIS5 (RAIS_PL), Data Logging in RAIS5 (RAIS_DL), and the proposed RAISL schemes. (a) Parity logging in RAIS5 (RAIS_PL); (b) data logging in RAIS5 (RAIS_DL); (c) RAISL

in the entry list corresponds to an updated data block. Each entry has the following items:

- *LBA*: logical block address of the data block in the RAIS storage system.
- *Log_LBA*: logical block address of the new data in the log SSD.
- *Next*: pointer to the next entry.



LPN	PPN	Old_PPN
LP0	PP0	NULL
LP1	PP1	NULL
LP2	PP2	NULL

Fig. 5 Key data structures. (a) Hash list; (b) extended mapping table

To search a data block in the hash list, RAISL first finds the corresponding hash slot by hashing the logical block address of the data block, and then finds the corresponding entry in the entry list. If there is an entry in the entry list satisfying the condition that its *LBA* item equals the logical block address of the data block, this entry corresponds to the data block; otherwise, the data block has not been updated and does not have a corresponding entry in the hash list.

The extended mapping table adds an *Old_PPN* entry to record the old data of the updated blocks. The *Old_PPN* with NULL represents that the data block has not been updated since the parity block is last updated. The main variables are explained below:

- *LPN*: logical page number in SSD.
- *PPN*: physical page number in SSD.
- *Old_PPN*: physical number of semi-valid page in SSD.

Figure 6 shows how the value of structure entry changes when updating data block D0 twice as Fig. 4. We assume that the size of data block D0 is the same as a page, the LBA of D0 is 0, and the LPN of D0 in #SSD0 is 0. When data block D0 is first updated, the new data is written to the physical page number 4, and it is also written to the logical block address 0

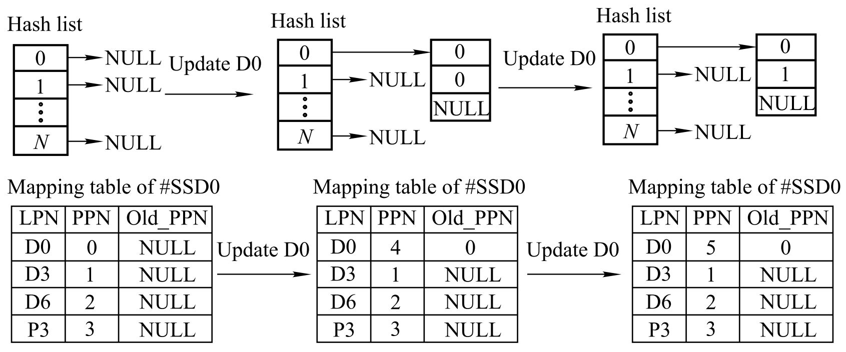


Fig. 6 How the value of structure entry changes when updating data block

in the log SSD. The page in PPN 0 is a semi-valid page, so the value of *Old_PPN* entry is 0. When D0 is updated once again, the new data is written to PPN 5, and it is also written to LBA 1 in the log SSD. The page in PPN 4 is an invalid page, and the value of *Old_PPN* entry does not change.

3.3 Data recovery from disk failures

Disk failures can occur either in the SSDs of RAIS or in the log SSD. If the log SSD fails, the parity re-synchronization operation is initiated. For each stripe that contains updated data blocks, the parity block needs to be re-computed.

If the failed disk is a SSD of RAIS, the data recovery process is triggered. The data SSD recovery process flow is shown in Algorithm 1. For each parity stripe, if the failed block is a parity block, the failed data can be re-computed by the latest surviving data in the stripe. If the failed block is a data block that has been updated, its recovery data can be directly copied from the log SSD. In more detail, RAISL reads out the log record addressed by *Log_LBA* from the log SSD as the recovery data. However, if the failed block is a data block that has not been updated, the failed data can be recovered by XORing all other surviving old data and the old parity in the stripe. For each surviving data block, if it has been updated, RAISL reads the data addressed by *Old_PPN*. Otherwise, RAISL reads the data addressed by *PPN*.

For example, if #SSD3 fails after D0 is updated twice in Fig. 4, the failed block P0 is a parity block. RAISL uses the XOR result of D0″, D1 and D2 as the recovery data. The failed block D5 is an non-updated block and all the data blocks in its corresponding stripe are not updated. RAISL uses the XOR result of D3, D4 and P1 as the recovery data. If the failed disk is #SSD0, the failed block D0 is an updated data block. RAISL reads D0″ from the log SSD as the recovery data. If the failed disk is #SSD1, D1 is an non-updated block but its corresponding stripe contains updated data block, RAISL uses the XOR result of D0, D2 and P0 as the recovery data. The address of D0 is recorded by *Old_PPN*.

Algorithm 1 Process flow of RAISL to recover from data SSD failures

```
for each parity stripe do
  if the failed block is a parity block then
    The failed data can be re-computed by the latest surviving data in the stripe.
  else
    if the failed block has been updated then
      Copy the new data from the log SSD.
    else
      for surviving block do
        if it is updated then
          Read out the data addressed by Old_PPN.
        else
          Read out the data addressed by PPN.
        end
      end
      The failed data can be recovered by XORing all other surviving old data and the old parity in the stripe.
    end
  end
end
```

3.4 AGCRL

Since RAISL keeps the original features of workloads, the access characteristic guided read and write cost regulation method can be used on the basis of RAISL. AGCRL combines RAISL with the access characteristic guided read and write cost regulation method. Figure 7 provides an architectural overview of the proposed AGCRL system. When AGCRL services a write request, the new data blocks are updated in home location and written to the log SSD sequentially to delay the parity update.

AGCRL has seven key functional modules: Administration, Monitor, Classifier, Request Handle, Parity Resynchronization, Data Recovery, and Data Director. The Administration module sets the threshold size of the log SSD and other parameters of the system. The Monitor module is responsible for monitoring the system failure. The Classifier module is responsible for identifying the access characteristic of a page. The Request Handle module manages the user requests. When the log SSD fails or the number of blocks used by the logging data reaches the threshold or the system is idle, the Parity Re-synchronization module is started for cleaning the log data. The Data Recovery module will be triggered when a data SSD fails. Meanwhile, the Request Handle module, Parity Re-synchronization module, and Data Recovery module all need to query and update the hash list which is stored in memory.

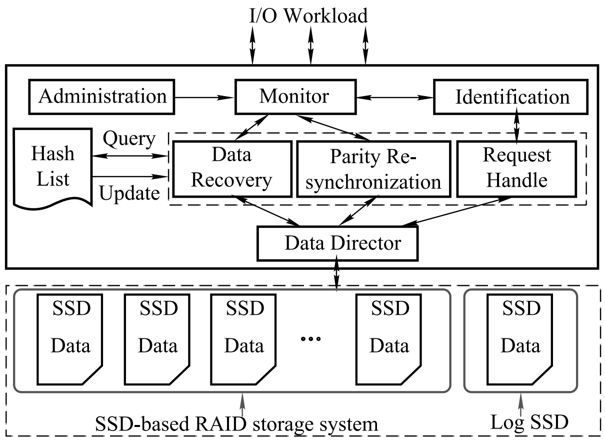


Fig. 7 AGCRL architecture

The difference between AGCRL and RAISL is the existence of the Classifier module. RAISL does not need this module to identify the access characteristic of a page. AGCRL identifies the access characteristic of a page by using the history window to record the access request types of the page. The size of history window, which affects the storage overhead as well as the identification accuracy, is an important parameters in Classifier module. To understand the effect of the window size on identification accuracy, we use 4 different sizes in the identification method, and the identification accuracies are shown in Fig. 8. The results show that a larger window results in a higher accuracy. When the history window size is set to 1, 2, 3, and 4, respectively, the average identification accuracies are 95.91%, 97.18%, 97.67%, and 97.90%. First, the increase in accuracy decreases as the size increasing. Second, the average identification accuracies are all between 97% and 98% when the history window size is set to 2, 3, and 4, respectively. Thus, AGCRL sets the length of the history window to 2 to trade-off the storage overhead and identification accuracy. When the history window size is 2, it means to use the most recent request and the upcoming request to identify the access characteristic of a page. If the most recent request and the upcoming request are both writes, the page is marked as a write-only page. If the most recent request and the upcoming request are both reads, the page is marked as a read-only page. Otherwise, the page is marked as an interleaved-access page.

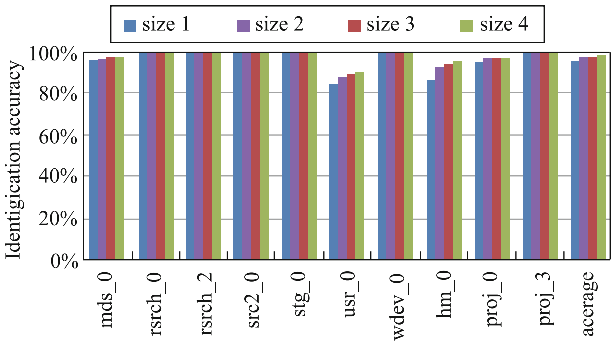


Fig. 8 Window size impact on identification accuracy

Except identification, the process flow of AGCRL is the same as RAISL on the RAID level. However, when request is handling in data SSD, AGCRL has to regulate the cost of the upcoming request based on the identified access characteristics of the page. When writing a write-only page, AGCRL applies a low-cost write, as shown in Fig. 9(a). When writing an interleaved-access page, AGCRL applies a medium-cost write, as shown in Figs. 9(c) and 9(d). For a read-only page with low-cost, nothing should be done, as shown in Fig. 9(b). When reading an interleaved-access page, AGCRL does not regulate the read cost, as shown in Figs. 9(c) and 9(d). If a read-only page was not written with a high cost before, its corresponding read cost is high or medium (denoted with **H** or **M** in Figs. 9(c)and 9(d)), and it needs to be re-write with a high-cost write during idle time. The re-write operation must be completed before upcoming reads of the page.

3.5 Global garbage collection

When the space of RAIS is not enough, local garbage collection is performed in each SSD of RAIS. The local garbage collection leads to performance degradation. In the RAIS storage system, uncoordinated garbage collection amplifies these performance degradation. AGCRL and RAISL perform two types of garbage collections: a local garbage collection, and a global garbage collection. The local garbage collection reclaims invalid pages. The global garbage collection reclaims semi-valid pages.

Similar to that in [24], AGCRL and RAISL trigger the local garbage collection on different SSDs synchronously in

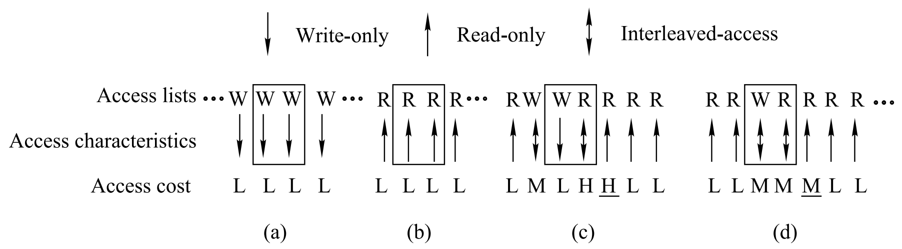


Fig. 9 Example on the access characteristic identification and cost regulation: access cost, L - low-cost, M - medium-cost, and H - high-cost. (a) Case 1; (b) Case 2; (c) Case 3; (d) Case 4

order to minimize the impact on RAIS performance. If the free space is still not sufficient, the global garbage collection is triggered. The global garbage collection finds the stripes whose parity blocks are delayed updating, and prioritizes the stripe that has more semi-valid pages. After choosing one stripe, the global garbage collection computes and updates the parity block, then marks the corresponding semi-valid pages as invalid state. The invalid pages are reclaimed at last.

3.6 Overhead analysis

AGCRL has time overhead and space overhead. The results in Li et al. [14] show that the percentage of re-write operations is negligible, and no more than 1% of all accesses issued by the host. Thus, the time overhead, which is caused by the re-write operations, is negligible. The performance of SSD is many orders of magnitude slower than memory. Thus, the time overhead, which is caused by the hash list and the extended mapping table, is also negligible. The space overhead is due to the hash list and the extended mapping table. The hash list is stored in memory and the extended mapping table is stored in the internal memory of SSD. The total hash list size is $n \times c/r$, where n is the number of bytes per hash list entry, c is total updated data size, and r is the data block size. The total size of the extended mapping table is $m \times c/p$, where m is the number of bytes that are allocated for each mapping relationship, c is total updated data size, and p is the page size. We assume that there is 1TB data in the AGCRL system and 10 percent data is updated. If the data block size is 4KB and each item in the hash list accounts for 4 bytes, it needs 300MB memory to store the hash list. If the AGCRL consists of 10 SSDs, the page size is 4KB and each mapping relationship accounts for 4 bytes, each SSD needs extra 10MB memory.

3.7 System reliability

Due to the parity blocks, RAID5 storage systems can tolerate one disk failure. However, parity blocks are delayed to update in AGCRL storage system. As described in Section 3.3, when the log SSD fails, the parity re-synchronization operation must be done to guarantee system reliability. When one SSD of RAIS5 fails, AGCRL can recover the failed data according to the Algorithm 1. Therefore, AGCRL also can tolerate one SSD failure. If the extended mapping table is lost, it can be regarded as the corresponding SSD fails.

If there is no SSD failure but the hash list is lost because of sudden power outage, AGCRL can do parity re-synchronization to guarantee reliability. When the hash list is lost, the parity blocks that have been delayed to update should be re-computed as soon as possible. Because an SSD failure during this period will cause data loss. AGCRL cannot know which stripes are in an inconsistent state without the hash list. Thus, for each stripe, AGCRL computes a new parity block by XORing the data blocks, then compares the new parity block with the old parity block. If the two parity blocks are different, AGCRL writes the new parity block to the corresponding location and marks the corresponding semi-valid pages as invalid state; otherwise, the stripe is in a consistent state and AGCRL skips the stripe.

However, if the hash list is lost during the AGCRL reconstruction period, the data will be lost. To prevent the loss of the hash list because of sudden power outage or a system crash, AGCRL can store the hash list in a nonvolatile RAM (NVRAM).

3.8 Customization on SSDs and RAID controller

To implement the RAISL and AGCRL, we should customize the bio, RAIS controller, and SSDs.

1) Bio. When there is an SSD fails, RAIS controller may need to read the semi-valid pages. Thus, there is information interaction between RAIS controller and SSDs. SSDs need to know whether a read request is to read normal valid pages or corresponding semi-valid pages. When an SSD writes a page, if the value of the corresponding *Old_PPN* is NULL, the SSD change it to the value of *PPN*. However, SSD internal erase operations also generate write requests. To deal with the internal generation write requests, the SSD does not

have to consider the *Old_PPN*. Therefore, SSDs also need to know whether a write request comes from the RAIS controller or themselves. Besides, when RAIS controller sends requests to the SSDs, the SSDs needs to know which latency is deployed to service the requests. These information can be recorded in bio. In the bio structure, there is a *bi_flags* field that represents the state of the bio. In Linux kernel, each bio defines 12 kinds of states. *Bi_flags* is a short integer variable with 16 bits. 12 kinds of states only require 4 bits to indicate. We can use another 4 bits of *bi_flags*, which have not been used, to transfer information between the RAID controller and SSDs. We use *w_flags* and *req_flags* to represent the first 2 bits in the 4 bits, and use *cost_flags* to represent the last 2 bits. If *w_flags* is set to "1", it indicates that the read request needs to read the semi_valid pages. If *w_flags* is set to "0" and the request is read, it indicates that the read request needs to read the normal pages. If *req_flags* is set to "1", it indicates that the request comes from the RAIS controller. If *req_flags* is set to "0", it indicates that the request comes from the SSDs. If *cost_flags* is set to "00", it indicates that the SSDs do not need to adjust latency. If *cost_flags* is set to "01", it indicates that the SSDs can use low-cost to service the request. If *cost_flags* is set to "10", it indicates that the SSDs can use medium-cost to service the request. If *cost_flags* is set to "11", it indicates that the SSDs can use high-cost to service the request. The RAIS controller sets the values of *w_flags*, *req_flags*, and *cost_flags*, and the SSDs perform the corresponding processing according to the values of these bits.

2) RAIS controller. As shown in Fig. 7, there are seven key functional modules and a Hash list in AGCRL. Administration, Monitor, Request Handle, Parity Re-synchronization, Data Recovery, and Data Director are included in the traditional RAID controller. We just need a few modifications on these functions to implement AGCRL. In addition, RAIS controller has to add a Classifier module and a hash list. The hash list is used to record the data blocks that have been written to the log SSD. How AGCRL identifies the access characteristic of a page has been described in Section 3.4. After identifying the type of a page, RAIS controller needs to set the *cost_flags*. When RAIS controller writes a write-only page, the *cost_flags* is set to "01". When RAIS controller writes an interleaved-access page, the *cost_flags* is set to "10". When RAIS controller reads a page, the *cost_flags* is set to "00". If a read-only page was not written with a high cost before, RAIS controller needs to generate a high-cost write (i.e., the *cost_flags* is set to "11") to re-write the page during idle time. It is easy to modify these functions in the Linux kernel MD module.

3) SSDs. To implement the AGCRL, the SSDs must have three capabilities: a) record the semi-valid pages; b) be able to recognize the semantic information in *bi_flags* that are transmitted by the RAIS controller; c) dynamically regulate read and write cost. As shown in Fig. 5(b), the SSDs can record the corresponding semi-valid pages by adding an *Old_PPN* entry to the mapping table. When the SSD handles a read request and checks that the *w_flags* is set to "1", it reads the page that the corresponding *Old_PPN* points to. Otherwise, it reads the page that the corresponding PPN points to. When the SSD handles a write request and checks that the *req_flags* is set to "1", it need to change the value of the corresponding *Old_PPN* if it is NULL. Otherwise, it does not need to consider the *Old_PPN*. If a flash page is written with a higher cost by using a finer step size during the incremental-step pulse programming (ISPP) process, it can be read with a relatively low cost due to the time saved in sensing and transferring, and vice versa [14]. By adjusting the step size, SSDs can dynamically regulate read and write cost. To achieve the aforementioned three capabilities, we must modify the internal drivers and FTL. Thus, we have to work with manufacturers to customize the SSDs.

4 Performance evaluations

4.1 Experimental setup

We implemented the RAISL and AGCRL scheme using the SSD extension on Disksim developed by Microsoft research [25]. We select 10 representative traces from MSR to evaluate the proposed schemes. We choose the first 12 hours of the workloads; and the basic characteristics of the workloads are shown in Table 1.

The parameters of the simulator are shown in Table 2. The top part describes the parameters of SSD and the bottom part shows the parameters of RAlS storage systems. We have implemented RAIS5, RAIS_PL, RAIS_DL, RAISL and AGCRL based on the simulator. RAIS_PL is a log-assisted RAIS5 storage system that uses Parity Logging scheme to improve small write performance. RAIS_DL is a log-assisted RAIS5 storage system that uses Data Logging scheme to improve small write performance. The simulated storage systems are configured with eight chips. Each chip has eight planes. Each plane has 2,048 blocks. Each block has 256 pages and the page size is 4 KB. The read and write cost has three different types.

4.2 Performance results

In this section, read, write, and overall performance of

Table 2 Parameters of the simulator

SSD configuration			
Total capacity	128 GB		
Reserved free blocks	15%		
# of chips per device	8		
# of planes per chip	8		
# of blocks per plane	2,048		
# of pages per block	256		
Page size	4 KB		
Page read latency	Low	Medium	High
	0.07 ms	0.17 ms	0.31 ms
Page write latency	Low	Medium	High
	0.8 ms	0.6 ms	0.45ms
Block erase latency	3 ms		
RAID configuration			
# of SSD	6		
Chunk size	64K		
RAID level	RAID5		

RAISL and AGCRL are evaluated and compared with RAIS5, RAIS_PL, and RAIS_DL. For RAIS5,RAIS_PL, RAIS_DL, and RAISL, all writes are performed with low-cost, followed by high-cost reads. Figures 10 and 11 show the access latency compared to RAIS5, RAIS_PL and RAIS_DL. From Fig. 10, we can see that AGCRL performs the best for all workloads in terms of overall average latency. Specifically, AGCRL, on the average, outperforms RAIS5, RAIS_PL, RAIS_DL, and RAISL by 39.15%, 36.77%, 22.89%, and 16.59%, respectively. RAISL performs better than RAIS5, RAIS_PL, and RAIS_DL for all workloads in terms of overall average latency. RAISL, on the average, outperforms RAIS5, RAIS_PL, and RAIS_DL by 26.03%, 22.29%, and 7.40%, respectively.

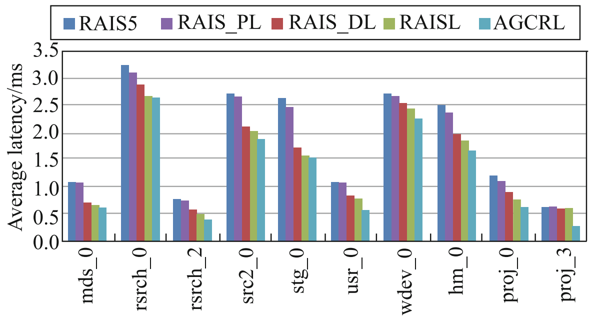


Fig. 10 Overall performance comparison

As shown in Figs. 11(a) and 11(b), AGCRL reduces the latency by (35.52%, 55.65%), (31.36%, 55.47%), and (11.89%, 55.44%) on the average, compared to RAIS5, RAIS_PL, and RAIS_DL. The first parenthesized pair is the improvement achieved by AGCRL over RAIS5 for the average write and read performance. The second parenthesized pair is the improvement achieved by AGCRL over RAIS_PL for the average write and read performance. The third parenthesized pair is the improvement achieved by AGCRL over RAIS_DL for the average write and read performance. The experiment results show that the average write performance of RAISL is almost the same as AGCRL, and the average read performance of RAISL is almost the same as RAIS5, RAIS_PL, and RAIS_DL. The reason is that RAISL, RAIS_PL, and RAIS_DL are all using the logging technique to improve the small write performance of RAIS, and they have no effect on read performance. All writes are performed with low-cost, and all reads are performed with high-cost in RAISL. While most writes and reads are performed with low-cost in AGCRL; thus, compared to RAISL, AGCRL mainly decreases the average read latency. AGCRL reduces read latency more than 55%, on the average.

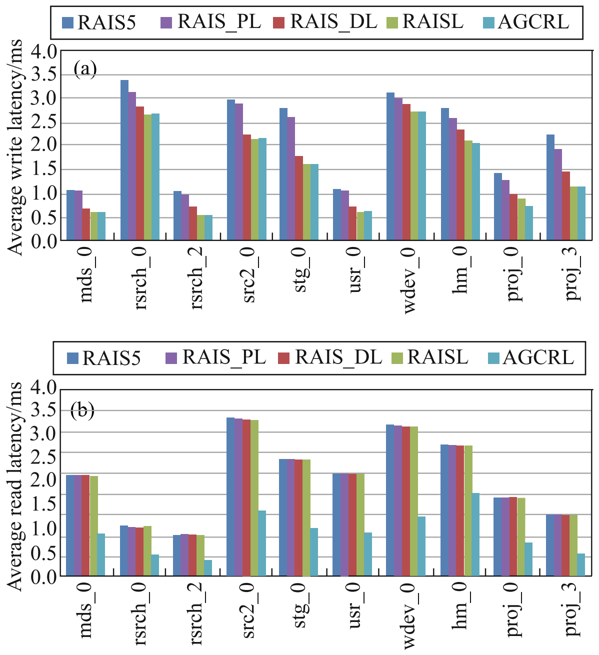


Fig. 11 Access latency comparison. (a) IO latency for write requests; (b) IO latency for read requests

To evaluate the space cost of the embedding data structures like the hash list and the extended mapping table, we counted the number of the updated pages, the size of the hash list, and the size of extra extended mapping table. As shown in Table 3, for all workloads, the memory that the hash list needs is no more than 1MB. The memory that the extra extended mapping table needs is no more than 0.3MB. The results show that AGCRL can improve performance efficiency at the cost of much less amount of space overhead.

Figures 12 and 13 show the effect of the write cost on RAISL. All writes are performed with high-cost and all reads are performed with low-cost in RAISL_HWLR. All writes and reads are performed with medium-cost in RAISL_MWMR. All writes are performed with low-cost and all reads are performed with high-cost in RAISL_LWHR. The experiment results show that AGCRL outperformed RAISL_HWLR, RAISL_MWMR, and RAISL_LWHR. As shown in Fig. 13(a), the average write latency of AGCRL and RAISL_LWHR are the best. As shown in Fig. 13(b), the average read latency of AGCRL and RAISL_HWLR are the best. As the write-cost decreasing in RAISL, the average write latency decreases, but the average read latency increases. For random write dominated workloads, low-cost write can result in a satisfactory overall performance in RAISL. For read dominated workloads, high-cost write can result in a satisfactory overall performance in RAISL. No matter what the write ratio is and what the write cost is in RAISL, AGCRL outperforms RAISL for overall performance.

Table 3 Space cost

Trace file	Number of updated pages	Hash list size/MB	Extra mapping table size/MB
mds_0	33,004	0.38	0.13
rsrch_0	34,252	0.39	0.13
rsrch_2	3,907	0.04	0.01
src2_0	26,293	0.30	0.10
stg_0	31,458	0.36	0.12
usr_0	54,936	0.63	0.21
wdev_0	32,991	0.38	0.13
hm_0	63,603	0.73	0.24
proj_0	41,502	0.47	0.16
proj_3	5,895	0.07	0.02

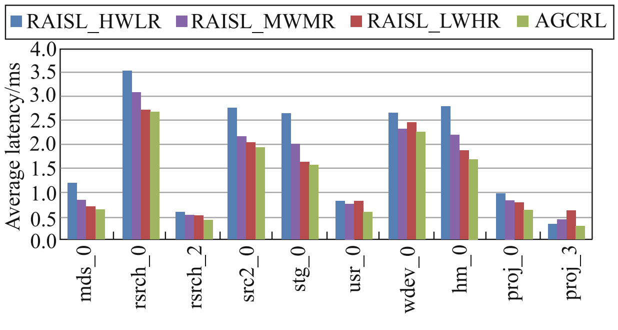


Fig. 12 Overall performance comparison

The average write handling overhead is mainly determined by the pre-read operation and the pages that are written to the log SSD. Figure 14 shows the normalized pre-read number. Figure 15 shows the normalized number of pages that are written to the log SSD. RAIS_PL scheme should generate one read operation and one write operation additionally to delay parity updating for each page write request. Thus, in the RAIS_PL system, the pre-read number and the number of pages that are written to the log SSD are both the same as the updated pages number. Since RAIS_DL scheme writes the pre-read data blocks and the updated data blocks to the log SSD, the number of pages that are written to the log SSD in RAIS_DL is more than that in RAIS_PL. If the data block to be updated has been updated before, RAIS_DL does not need any pre-read operation. The total pre-read number of RAIS_DL is much smaller than that of RAIS_PL scheme thanks to local principle. RAISL and AGCRL do not have any pre-read operation as shown in Fig. 14. Due to that RAISL and AGCRL only write the updated data blocks to the log SSD, the numbers of pages that are written to the log SSD in RAISL and AGCRL are the same as that in RAIS_PL.

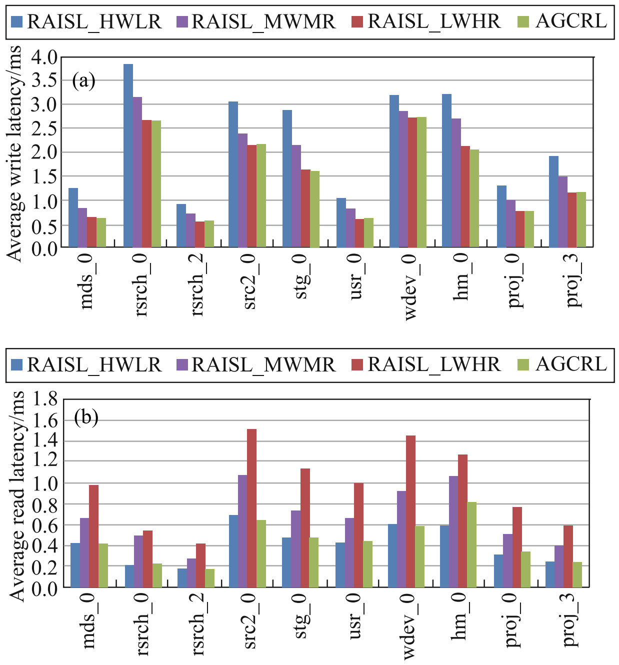


Fig. 13 Access latency comparison. (a) IO latency for write requests; (b) IO latency for read requests

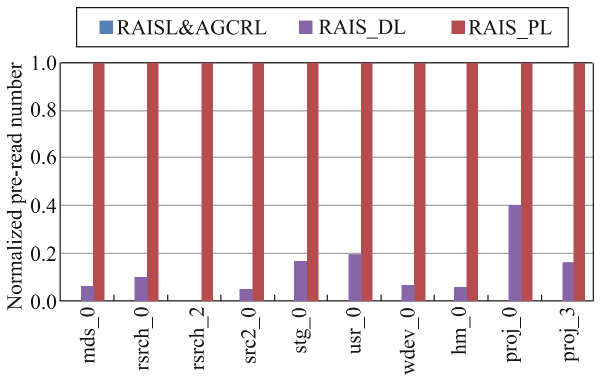


Fig. 14 Normalized pre-read number

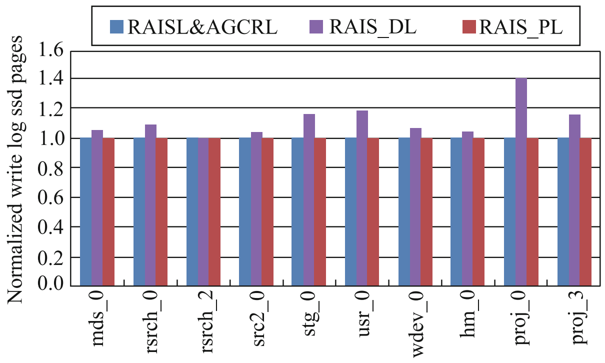


Fig. 15 Normalized number of pages that are written to the log SSD

5 Related work

There are a lot of logging technologies used in the traditional RAID storage systems to improve write performance. Parity Logging [11], Data Logging [15], and RAID6L [22] are all using logging technology to delay parity update for performance improvement which transform random write into sequential write. Due to high performance and low power consumption, SSD gradually replaces the HDD to construct array systems. In order to reduce the time of programming in SSD, HPDA [3] and HerpRap [26] both use several SSDs and one HDD to form a RAID4 where the data disks are SSDs and the parity disk is HDD. Since the capacity of HDD is usually much larger than SSD, the free space of the parity disk is used as a log region to absorb the small random write requests. To enhance energy efficiency, performance, and reliability, Zeng et al. [27] propose a new logging architecture for RAID6 systems which is called HRAID6ML. In HRAID6ML, several SSDs and two HDDs form a RAID6 where the data disks are SSDs and the parity disks are HDDs. The free space of the two parity disks is also used as mirrored log region for the whole system to absorb the small random writes requests.

In the RAID5 systems, the load of each disk is almost the same, because parity blocks are evenly placed on all disks. Thus, all SSDs in the RAIS5 systems are almost wear out at same time. When one SSD in RAIS5 fails towards the end of its lifetime, the risk of data loss is higher than the traditional RAID5. Balakrishnan et al. [7] propose Diff-RAID, a parity-based redundancy solution that creates an age differential in an array of SSDs. Diff-RAID distributes the parity blocks unevenly across the array, leveraging their higher update rate to age devices at different rates. SSDs can exhibit significant performance degradations when garbage collection conflicts with an ongoing I/O request stream. In SSD-based RAID systems, the lack of coordination of the local GC processes amplifies these performance degradations. In order to coordinate GC cycles across each SSD in the RAIS, Kim et al. [24] propose a global garbage collection (GGC) mechanism. When one SSD in RAIS needs to do GC, this mechanism forces all SSDs in the RAIS to do GC at the same time.

Flash memory has many characteristics, such as out-of-place update, write and read cost interconnections, and so on. By taking full advantage of the update characteristic, CD-RAIS [9] combines static striping and dynamic striping to form a novel striping strategy, which can place the unnecessarily consecutive logical blocks that comes from different SSDs in one stripe. CD-RAIS alleviates the write request increase due to parity block update and thus improves the system response time and extends the SSD lifetime. With SSD capacity increasing, the performance can be improved by making use of parallel structure, but reliability is a serious challenge. For both high performance and reliability, Im and Shin [28] propose PPC, a novel flash-aware RAID technique for flash memory SSD. PPC delays the parity update to reduce the parity update cost. In addition, PPC uses the partial parity technique to reduce the number of read operations required to calculate a parity by exploiting the characteristics of flash memory. Chung and Hsu [29] combine the PPC and data cache management to form a new method that can improve the RAIS performance. The proposed method adds a new cache to keep the original data which can reduce read operations when updating the partial parity. When write requests hit the cache, the method can reduce writing SSD and thus extends the SSD lifetime.

Greenan et al. [8] propose a new SSD-based RAID4 architecture, which adds a NVRAM. In order not to update the parity blocks frequently, the NVRAM is used to store the local parity blocks. The parity block is updated until all the data blocks in the corresponding stripe are updated. The FRA scheme [30] also can reduce the parity write frequency for multiple write requests to the same area by delaying parity update. Instead of using a NVRAM to store the parity block, FRA uses a dual-mapping table in FTL to identify which parity has been delayed.

Several strategies have been recently proposed for exploiting the write and read cost characteristics of flash memory to regulate read and write costs for performance improvement on flash memory. Li et al. [31] propose a novel method to reduce the latency of IO requests by reducing access conflict latency on flash memory storage system. This method uses low-cost writes for write conflict reduction, and high-cost writes for read conflict reduction. Pan et al. [32] and Liu et al. [33] reduce write costs by relaxing the retention time requirement of the programmed pages, which are motivated by the significantly short lifetime of most data in several workloads compared to the predefined retention time. Li et al. [14] propose a comprehensive approach called AGCR, which exploits the access characteristics of workloads for cost regulation to improve flash memory performance. AGCR observes that most accesses from the host are performed on either read-only or write-only pages. For read-only page, low-cost read is preferred and for write-only page, low-cost write is preferred. Thus, read and write costs can be regulated to significantly improve the overall performance.

6 Conclusion

In this article, we first propose RAISL, a flash-aware logging technique to improve the write performance of RAIS. RAISL makes full use of the invalid pages in SSD and only needs to write the new data to the log SSD. RAISL needs not to perform the pre-read operations so that the original characteristics of workloads are kept. Secondly, we propose AGCRL that combines RAISL with access characteristic guided read and write cost regulation to further improve the performance of RAIS storage systems. Our simulation results show that AGCRL is effective, reducing I/O latency by as much as 39.15%, 36.77%, 22.89% and 16.59%, compared to RAIS5, RAIS_PL, RAIS_DL and RAISL.

Acknowledgements This work was supported by the National High Technology Research and Development Program (863 Program) (2015AA016701, 2015AA015301); NSFC (61772222, U1705261, 61472153, 61402189, 61303046); State Key Laboratory of Computer Architecture (CARCH201505); Wuhan Applied Basic Research Project (2015010101010004); This work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China. We are grateful to the anonymous reviewers for their valuable feedback and suggestions.

References

1. Patterson D A, Gibson G, Katz R H. A case for redundant arrays of inexpensive disks (RAID). In: Proceedings of the International Conference on Management of Data. 1988, 109–116
2. Katz R H, Gibson G A, Patterson D A. Disk system architectures for high performance computing. Proceedings of the IEEE, 1989, 77(12): 1842–1858
3. Mao B, Jiang H, Feng D, Wu S, Chen J, Zeng L, Tian L. HPDA: a hybrid parity-based disk array for enhanced performance and reliability. In: Proceedings of IEEE International Symposium on Parallel & Distributed Processing. 2010, 1–12
4. Narayanan D, Thereska E, Donnelly A, Elnikety S, Rowstron A. Migrating server storage to SSDs: analysis of tradeoffs. In: Proceedings of the 4th ACM European Conference on Computer Systems. 2009, 145–158
5. Dirik C, Jacob B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. ACM SIGARCH Computer Architecture News, 2009, 37(3): 279–289
6. Chen F, Koufaty D A, Zhang X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. ACM SIGMETRICS Performance Evaluation Review, 2009, 37(1): 181–192
7. Balakrishnan M, Kadav A, Prabhakaran V, Malkhi D. Differential RAID: rethinking raid for ssd reliability. ACM Transactions on Storage, 2010, 44(1): 55–59
8. Greenan K M, Long D D E, Miller E L, Schwarz T J E, Wildani A. Building flexible, fault-tolerant flash-based storage systems. In: Proceedings of Workshop on Hot Topics in Dependability. 2009, 1–6
9. Du Y, Zhang Y, Xiao N, Liu F. CD-RAIS: constrained dynamic striping in redundant array of independent SSDs. In: Proceedings of IEEE International Conference on Cluster Computing. 2014, 212–220
10. Wu S, Yang W, Mao B, Lin Y. MC-RAIS: multi-chunk redundant array of independent SSDs with improved performance. In: Proceedings of International Conference on Algorithms and Architectures for Parallel Processing. 2015, 18–32
11. Stodolsky D, Gibson G, Holland M. Parity logging overcoming the small write problem in redundant disk arrays. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. 1993, 64–75
12. Menon J. A performance comparison of RAID-5 and log-structured arrays. In: Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing. 1995, 167–178
13. Suh K D, Suh B H, Lim Y H, Kim J K, Choi Y J, Koh Y N, Lee S S, Kwon S C, Choi B S, Yum J S. A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme. IEEE Journal of Solid-State Circuits, 1995, 30(11): 1149–1156
14. Li Q, Shi L, Xue C J, Wu K, Ji C, Zhuge Q, Sha E H M. Access characteristic guided read and write cost regulation for performance improvement on flash memory. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies. 2016, 125–132
15. Gabber E, Korth H F. Data logging: a method for efficient data updates in constantly active RAIDs. In: Proceedings of the 14th International Conference on Data Engineering. 1998, 144–153
16. Hu Y, Jiang H, Feng D, Tian L, Luo H, Ren C. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. IEEE Transactions on Computers, 2013, 62(6): 1141–1155
17. Hu Y, Jiang H, Feng D, Zhang S, Liu J, Tong W, Qin Y, Wang L Z. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies. 2010, 1–12
18. Wu S, Mao B, Chen X, Jiang H. LDM: log disk mirroring with improved performance and reliability for SSD-based disk arrays. ACM Transactions on Storage, 2016, 12(4): 22
19. Mei L, Feng D, Zeng L, Chen J, Liu J. A stripe-oriented write performance optimization for RAID-structured storage systems. In: Proceedings of IEEE International Conference on Networking, Architecture and Storage. 2016, 1–10
20. Li Y, Shen B, Pan Y, Xu Y, Li Z, Lui J C S. Workload-aware elastic striping with hot data identification for SSD RAID arrays. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2017, 36(5): 815–828
21. Li C, Feng D, Hua Y, Wang F. Improving raid performance using an

endurable SSD cache. In: Proceedings of the International Conference on Parallel Processing. 2016, 396–405

22. Jin C, Feng D, Jiang H, Tian L. RAID6L: a log-assisted RAID6 storage architecture with improved write performance. In: Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies. 2011, 1–6
23. Jin C, Feng D, Jiang H, Tian L, Liu J, Ge X. Trip: temporal redundancy integrated performance booster for parity-based RAID storage systems. In: Proceedings of the 16th IEEE International Conference on Parallel and Distributed Systems. 2010, 205–212
24. Kim Y, Oral S, Shipman G M, Lee J, Dillow D A, Wang F. Harmonia: a globally coordinated garbage collector for arrays of solid-state drives. In: Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies. 2011, 1–12
25. Agrawal N, Prabhakaran V, Wobber T, Davis J D, Manasse M, Panigrahy R. Design tradeoffs for SSD performance. In: Proceedings of USENIX Annual Technical Conference. 2008, 57–70
26. Zeng L, Feng D, Mao B, Chen J, Wei Q, Liu W. HerpRap: a hybrid array architecture providing any point-in-time data tracking for datacenter. In: Proceedings of IEEE International Conference on Cluster Computing. 2012, 311–319
27. Zeng L, Feng D, Chen J, Wei Q. HRAID6ML: a hybrid RAID6 storage architecture with mirrored logging. In: Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies. 2012, 1–6
28. Im S, Shin D. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. IEEE Transactions on Computers, 2011, 60(1): 80–92
29. Chung C C, Hsu H H. Partial parity cache and data cache management method to improve the performance of an SSD-based RAID. IEEE Transactions on Very Large Scale Integration Systems, 2014, 22(7): 1470–1480
30. Lee Y, Jung S, Song Y H. FRA: a flash-aware redundancy array of flash storage devices. In: Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis. 2009, 163–172
31. Li Q, Shi L, Gao C, Wu K, Xue C J, Zhuge Q, Sha H M. Maximizing IO performance via conflict reduction for flash memory storage systems. In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition. 2015, 904–907
32. Pan Y, Dong G, Wu Q, Zhang T. Quasi-nonvolatile SSD: trading flash memory nonvolatility to improve storage system performance for enterprise applications. In: Proceedings of the 18th IEEE International Symposium on High-Performance Computer Architecture. 2012, 1–10
33. Liu R S, Yang C L, Wu W. Optimizing NAND flash-based SSDs via retention relaxation. In: Proceedings of the 10th Usenix Conference on File and Storage Technologies. 2012, 11–24

Linjun Mei received his BE degrees in Computer Science and Technology from Huazhong University of Science and Technology, China in 2007. He is currently pursuing the PhD degree in the School of Compute from the Huazhong University of Science and Technology, China. His current research interests include hybrid storage systems, redundant array of independent disks, and SSD-based RAID.

Dan Feng received her BE, ME and PhD degrees in Computer Science and Technology from Huazhong University of Science and Technology, China in 1991, 1994 and 1997 respectively. She is a professor and Vice Dean of the School of Computer, China. Her research interests include computer architecture, massive storage systems and parallel file systems. She has over 100 publications in journals and international conferences, including FAST, ICDCS, HPDC, SC, ICS and ICPP. Dr. Feng is a member of IEEE and a member of ACM.

Lingfang Zeng received his BE in Applied Computer from Huazhong University of Science and Technology (HUST), China in 2000, MS in Applied Computer from China University of Geoscience, China in 2003 and PhD in Computer Architecture, from HUST, China in 2006. He was research fellow for four years in Department of Electrical and Computer Engineering, National University of Singapore, Singapore, during 2007–2008 and 2010–2013. He is currently with Wuhan National Lab for Optoelectronics, and School of Computer, HUST, as an associate professor. He publishes more than 40 papers in major journals and conferences, including ACM Transactions on Storage, IEEE Transactions on Magnetics, Journal of Parallel and Distributed Computing, FAST, SC, MSST, and CLUSTER, and serves for multiple international journals and conferences. He is a member of IEEE.

Jianxi Chen received his BE degrees in Nanjing University, China in 1999, MS in Computer Architecture from Huazhong University of Science and Technology, China in 2002 and PhD in Computer Architecture, from HUST, China in 2006. He is currently with Wuhan National Lab for Optoelectronics, and School of Computer, as an Associate Professor. He publishes more than 20 papers in major journals and conferences. He is a member of CCF and IEEE.

Jingning Liu received her BE degrees in Department of Computer Peripheral Equipment from Huazhong University of Science and Technology, China in 1982. She is currently with Wuhan National Lab for Optoelectronics, and School of Computer, as a professor. She publishes more than 30 papers in major journals and conferences.