A Stripe-Oriented Write Performance Optimization for RAID-Structured Storage Systems

Linjun Mei✉, Dan Feng, Lingfang Zeng, Jianxi Chen, Jingning Liu
School of Computer, Huazhong University of Science and Technology
Wuhan National Laboratory for Optoelectronics
✉*Corresponding author: ljmei@hust.edu.cn*
{*dfeng,lfzeng,chenjx*}*@hust.edu.cn, j.n.liu@163.com*

Abstract—**In modern RAID-structured storage systems, reliability is guaranteed by the use of parity blocks. But the parity-update overheads upon each write request have become a performance bottleneck of RAID systems. In some ways, an attached log disk is used to improve the write performance by delaying the parity blocks update. However, these methods are data-block-oriented and they need more time to rebuild or synchronize the RAID system when a data disk or the log disk fails. In this paper, we propose a novel optimization method, called SWO, which can improve RAID write performance and reconstruction performance. Moreover, when handling a write request, the SWO chooses *reconstruction-write* or *read-modify-write* combining with the log information to further minimize the number of pre-read data blocks. We have implemented the proposed SWO prototype and carried out some performance measurements using IOmeter and RAIDmeter. We have implemented the main idea of RAID6L in RAID5 and call it RAID5L. At the same time, we have evaluated the reconstruction time and the synchronization time of the SWO. Our experiments demonstrate that the SWO significantly improves write performance and saves more time than Data Logging and RAID5L when rebuilding and synchronizing.**

Keywords-**reliability; stripe-oriented; log information; write performance;**

I. Introduction

RAID (Redundant Array of Independent Disks) [1] technology is widely used in modern storage systems to meet requirements of large capacity, high performance and reliability. However, the small random write performance [2] of RAID5 is very poor due to updating the parity blocks. For example, when servicing a small write in RAID5, there will be four disk I/O operations: pre-read the old data block and the old parity block, write new data block and the new parity block. And this situation is even worse in RAID6 since more parity blocks must be updated when writing a small block.

Write performance of RAID can be improved by changing the data layout or delaying the parity updates. The methods [3] [4] [5] that delay the parity updates normally integrate a log disk into the RAID system. When servicing small write requests, the traditional RAID system reads and writes parity blocks frequently. But the RAID system with an additional log disk simply logs some related data blocks and delays the parity updates until the system is idle or lightly loaded. However, these methods are all data-block-oriented and they need more time to rebuild or synchronize the RAID system when a data disk or the log disk fails. Disk failure has obvious time locality and spatial locality. When a disk fails in the storage system, the probability of a 2nd disk failure is greatly increased [6] [7]. Carnegie Mellon University research shows that the probability of a 2nd disk failure is 0.5% when the storage system reconstruction time is one hour. If the reconstruction time is three hours, the probability of a 2nd disk failure is 1.0%. And the probability of a 2nd disk failure will be 1.4% if the reconstruction time is six hours [7]. In order to avoid data loss, the storage system has to recover data as soon as possible

Parity Logging [4], Data Logging [5] and RAID5L (we implement the main idea of RAID6L [3] in RAID5 and call it RAID5L) do not consider the log information before choosing the related data blocks to log which can affect write performance. In this paper, we propose the SWO: a novel <u>**S**</u>tripe-oriented <u>**W**</u>rite performance <u>**O**</u>ptimization for RAID-Structured storage systems which selects suitable log blocks to achieve better write performance and reconstruction performance based on the log information. The contributions of this paper are described as follows:

- We proposed a novel method, called SWO, which improves write performance and reconstruction performance.
- We observe that Parity Logging, Data Logging and RAID5L are data-block-oriented which impacts the hash table average search time when handling the stripe. SWO that is stripe-oriented can significantly reduce the reconstruction and synchronization time.
- We consider the log information when choosing the related data blocks to log. Compared with Parity Logging, Data logging and RAID5L, the write performance of SWO is better because of reducing the amount of pre-read data blocks which are written to the log disk.
- We have implemented an SWO prototype in the Linux Software RAID framework and carried out extensive performance measurements using IOmeter [8] and RAIDmeter [9]. Experimental results show that SWO

has a marked advantage over Parity Logging, Data Logging and RAID5L.

The rest of this paper is organized as follows. The background and the motivation are presented in Section II. We describe the architecture and design of SWO in Section III. The experimental results are presented in Section IV. We review the related work in Section V and conclude our paper in Section VI.

II. Background and Motivation

This section mainly focuses on the necessary background knowledge that motivates us to design the SWO.

A. The log-assisted RAID storage systems

When receiving a write request in a traditional RAID5 system, the RAID5 controller translates it into sub-write-requests which are grouped by the parity stripe. Before writing the data blocks, the controller has to compute the new parity block. There are usually two alternative methods to generate the new parity block, namely, *reconstruction-write* and *read-modify-write* respectively. 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 [3] [10]. Rmw represents the number of pre-reads when choosing the *read-modify-write* method. Rcw represents the number of pre-reads when choosing the *reconstruction-write* method. Generally, if rcw<=rmw, the controller chooses the *reconstruction-write* method which needs to pre-read the data blocks that are not to be updated. Otherwise, if rmw<rcw, it chooses the *read-modify-write* method that needs to pre-read the parity block and the data blocks that are to be updated. After the computation of the new parity block, the new parity block and new data blocks will be written to the corresponding disks.

Instead of writing the new parity block to the corresponding disk, Parity Logging [4] writes the XOR result of the data block's old and new values to the log disk sequentially. Due to this approach, Parity Logging does not reuse existing log records and the log cleaning is a relatively long operation, thus the Data Logging [5] scheme is proposed. 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 and the new value of the block to the log disk. In the case where the data block has been logged, Data Logging only writes the new value of the block to the log disk.

Observing that Parity Logging and Data Logging both choose *read-modify-write* to handle write requests, RAID5L dynamically chooses either the *read-modify-write* or the *reconstruction-write* method to boost write performance. For either the *reconstruction-write* or *read-modify-write* method, RAID5L does not need to pre-read or update the parity blocks [3].

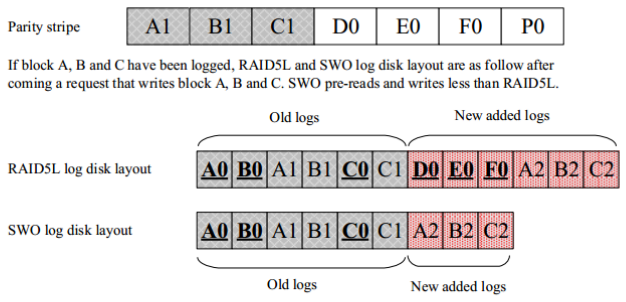

Parity stripe
A1
B1
C1
D0
E0
F0
P0
If block A, B and C have been logged, RAID5L and SWO log disk layout are as follow after coming a request that writes block A, B and C. SWO pre-reads and writes less than RAID5L.
Old logs
New added logs
RAID5L log disk layout
A0 B0 A1 B1 C0 C1 D0 E0 F0 A2 B2 C2
SWO log disk layout
A0 B0 A1 B1 C0 C1 A2 B2 C2
Old logs
New added logs

Figure 1. The Case that SWO and RAID5L Choose Different methods

B. Motivation

These three log-assisted RAID storage systems are all data-block-oriented. However, the RAID5 processing unit is a stripe. When handling the stripe for write requests or parity re-synchronization, the log-assisted RAID storage systems need to know whether each data block in the stripe has been logged or not. This process must search the hash table many times. The SWO is stripe-oriented and only needs to search the hash table once which can improve the write performance and decrease the reconstruction and log cleaning time.

In the three log-assisted schemes, only RAID5L can choose the *read-modify-write* or *reconstruction-write* method as the traditional RAID5. However, when RAID5L chooses the method, it only takes out the parity block without the data blocks that have been logged. SWO takes the log information into account when counting the number of pre-read data blocks. The main difference between SWO and RAID5L is the timing when to consider the log information. SWO considers the log information when choosing the *read-modify-write* or *reconstruction-write* method. RAID5L considers the log information after choosing the *read-modify-write* or *reconstruction-write* method. The log information is very important. SWO considers the logged data blocks which may influence the number of pre-read data blocks and the method that the controller chooses.

For example, Figure 1 is the case that SWO and RAID5L choose different methods when handling the write request. In Figure 1, block A, B and C have been logged before the write request arrives. When the write request updates data block A, B and C once again, the SWO chooses *read-modify-write*. Because SWO removes the logged data block A, B, C and parity block P, it does not need to pre-read when choosing the *read-modify-write* method and needs to pre-read 3 data blocks(D,E,F) when choosing the *reconstruction-write* method. However, RAID5L only removes parity block P, it chooses *reconstruction-write* because of rmw=3,rcw=3(rcw<=rmw). At last, RAID5L writes the pre-read data blocks D0, E0, F0 and new data blocks A2, B2 and C2 to the log disk. SWO only needs to write the new data blocks A2, B2 and C2 to the log disk.

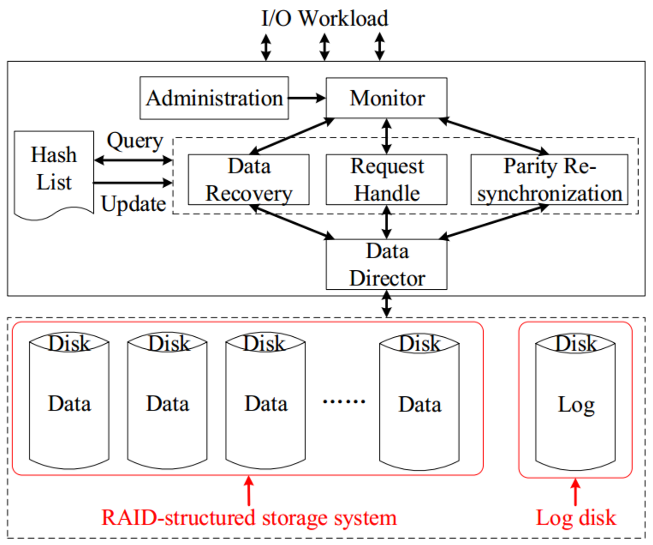


Figure 2. SWO Architecture

III. Design and Implementation

A. SWO Architecture

SWO is composed of a traditional RAID and a log disk. In this article, performance evaluations use RAID5 as the storage system. Figure 2 provides an architectural overview of the proposed SWO system. When SWO services a write request, the new data blocks are updated in-place and certain data blocks are written to the log disk sequentially to delay the parity blocks update. Writing the log disk is performed in an appending fashion. Rewriting the data block in the log disk simply invalidates the old value and writes the new value in the magnetic disc head. In addition, since serving read requests does not involve the parity block, SWO has no effect on read performance.

SWO has six key functional modules: Administration, Monitor, Request Handle, Parity Re-synchronization, Data Recovery and Data Dispatch. The Administration module sets the threshold size of the log disk for synchronizing the parity blocks and other parameters of the system. The Monitor module is responsible for monitoring the system workload and the system failure so as to decide to start the Request Handle module, Parity Re-synchronization module or Data Recovery module. The Request Handle module manages the user requests. If the system is idle or lightly loaded, the Parity Re-synchronization module is started for cleaning the log data. When the log disk fails or the size of the log disk reaches the threshold, the Parity Re-synchronization module also starts. The Data Recovery module will be triggered when a data disk fails. Meanwhile, the Request Handle module, Parity Re-synchronization module and Data Recovery module all need to query and update the hash table which is stored in memory.

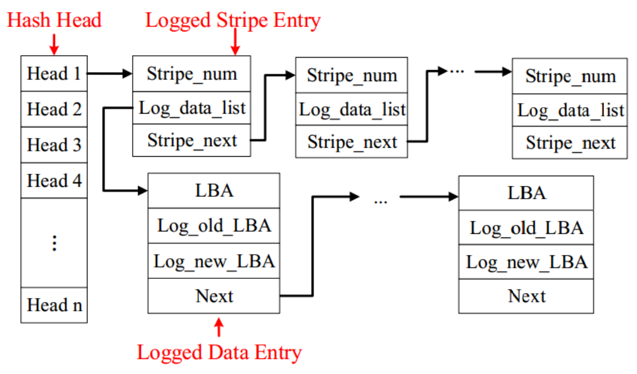


Figure 3. Key Data Structure

B. Key Data Structure

The main data structure in SWO is a hash table which is shown in Figure 3. The hash table contains lots of entries whose content is an address that points to an entry list. Each entry which is called a logged stripe entry in the list corresponds to a stripe. The logged stripe entry has an item *Log_data_list* that points to the data block list. Each data block entry that is called a logged data entry in the list is belonging to the stripe and logged in the log disk. The main variables are explained below:

- *Stripe_*num: the number of the stripe in the RAID system.
- *Log_data_list*: pointer to the data blocks that are belonging to the stripe and logged in the log disk.
- *Stripe_next*: pointer to the next stripe in which some data blocks are logged.
- *LBA*: logical block address of the data block in the RAID-structured storage system.
- *Log_old_LBA*: address of the data block's old value in the log disk.
- *Log_new_LBA*: address of the data block's new value in the log disk.
- *Next*: pointer to the next data block in the *Log_data_list*.

Since the data blocks update in-place, the parity blocks can be recomputed when the hash table is lost. So the hash table can be stored in memory. We assume that there is 1TB data in the SWO system which consists of 10 disks and 10 percent data which means 25M data blocks (the data block size is 4KB) involving about 2.5M-25M stripes when it is updated. If each item in the hash table accounts for 4 bytes, it needs 430MB-700MB memory to store the hash table. Moreover, the memory space can be further reduced by periodic parity re-synchronization during system idle or lightly loaded periods.

C. Process Flow

SWO handles the read request the same as the traditional RAID due to updating data in-place. But the write-request

```
1.  FOR EACH(parity stripe)
2.    Initialize rcw=0, rmw=0;
3.    Find the handling stripe in hash list and mark the
      logged data blocks in Logged_data_list;
4.    FOR EACH(block in the parity stripe)
5.      IF(it is to be updated and not marked)
6.        rmw++;
7.      ELSE
8.        IF(it is not parity block and not marked)
9.          rcw++;
10.       END IF
11.     END IF
12.   END FOR
13.   IF(rcw>rmw)
14.     FOR EACH(block in the parity stripe)
15.       IF(it is to be updated and marked)
16.         Write the new value of the block to the log disk and
            change the Log_new_LBA;
17.         Write the new value of the block in home location;
18.       END IF
19.       IF(it is to be updated and not marked)
20.         Pre-read the old value of the block;
21.         Create an logged data entry in the Logged_data_list
            for the block;
22.         Write the old and new value of the block to the log
            disk and change the Log_old_LBA and Log_new_LBA;
23.         Write the new value of the block in home location;
24.       END IF
25.     END FOR
26.   ELSE
27.     FOR EACH(block in the parity stripe)
28.       IF(it is not to be updated and not marked)
29.         Pre-read the old value of the block;
30.         Create an logged data entry in the Logged_data_list
            for the block;
31.         Write the old value of the block to the log  disk and
            change the Log_old_LBA;
32.       END IF
33.       IF(it is to be updated and not marked)
34.         Create an logged data entry in the Logged_data_list
            for the block;
35.         Write the new value of the block to the log disk and
            change the Log_new_LBA;
36.         Write the new value of the block in home location;
37.       END IF
38.       IF(it is to be updated and marked)
39.         Write the new value of the block to the log disk and
            change the Log_new_LBA;
40.         Write the new value of the block in home location;
41.       END IF
42.     END FOR
43.   END ELSE
44. END FOR
```

Figure 4. Write-Request Process Flow of SWO

process flow of SWO is different from the traditional RAID and RAID5L. The write-request process flow of SWO is shown in Figure 4. Firstly, SWO counts the pre-read number of *read-modify-write* and *reconstruction-write* while taking out the parity block and the data blocks that have been logged. Then, SWO chooses the method whose pre-read number is smaller. Next, SWO writes the pre-read data blocks and the new value of the data blocks that are to be updated to the log disk. At last, SWO writes the new value of the data blocks in the home location. SWO differs from RAID5L in several important ways. First, SWO is stripe-oriented and the hash table structure is different. When handling a write request, SWO finds the stripe entry from the hash table first, then marks the logged data blocks which are in the *Log_data_list*. Second, the timing when SWO and RAID5L consider the log information is different. When choosing the method to pre-read data blocks, SWO takes out the logged data blocks and parity block. But RAID5L only takes out the parity block and considers the logged data blocks after choosing the *read-modify-write* or *reconstruction-write* method. Third, SWO does not search the hash table when pre-reading the data blocks, because they are marked when choosing the *read-modify-write* or *reconstruction-write* method. However, RAID5L has to search the hash table to see whether the data block has been logged or not when pre-reading.

For example, when handling the second write request in Figure 1, RAID5L uses the *read-modify-write* method which means pre-reading block D and E. Before pre-reading block D, RAID5L has to search the hash table to ensure whether block D is logged or not. Before pre-reading block E, it has to search the hash table as well. But in the SWO system, when deciding to use the *read-modify-write* or *reconstruction-write* method, SWO first finds the handling stripe entry in the hash table. The *Log_data_list* item in the logged stripe entry points to the logged data blocks that belongs to the stripe. At this time, SWO marks block A and block B which means these two data blocks have been logged. Later in counting the pre-read data blocks number and pre-reading process, SWO does not have to search the hash table once more. Therefore, for either the *reconstruction-write* or *read-modify-write* method, SWO searches the hash table once, but the times of search operation that RAID5L needs equals the pre-read data block number.

D. Parity Re-synchronization

There are three cases in which the parity re-synchronization operations are triggered: (1) The system is idle or lightly loaded. (2) The log disk fails or the used size of log disk reaches a specific threshold. (3) The hash table in the memory is lost when the system reboots due to unplanned downtime or the system crashing. The difference between these three cases is the existence of the hash table. The hash table is not lost in the previous two cases, and SWO traverses through the hash table sequentially. For each logged stripe entry, SWO finds its corresponding parity stripe in the RAID system first, reads out all the data blocks in the parity stripe from the RAID system next, then re-computes and writes the parity block, and deletes the corresponding logged stripe entry and the logged data entries of the logged stripe at last. When the hash table is empty, the parity re-synchronization process is finished.

In the third case, the hash table is lost and SWO is in an inconsistent state because of some non-updated parity blocks. Since the hash table has been lost, the parity blocks should be re-computed as soon as possible because a disk failure during this period will cause data loss. Without the hash table assistance, SWO is not aware of which parity stripes are in an inconsistent state. Therefore, SWO re-synchronizes all the parity stripes sequentially.

For example, the blocks A, B and C have been logged in Figure 1, and assuming that a parity re-synchronization operation is needed at this time. RAID5L picks the stripe at first. In order to compute new parity block, RAID5L has to search the hash table 6 times to ensure whether A, B, C, D, E and F have corresponding entries in the hash table or not. However, SWO finds the logged stripe entry and the logged data entries A, B and C. SWO searches the hash table only one time because the *Log_data_list* item points to the logged data entries.

E. Data Recovery

When there is a disk failure, it may be a log disk or data disk. If the failed disk is a log disk, SWO only needs parity re-synchronization as above. Otherwise, the data disk recovery process flow is shown in Figure 5. For each parity stripe, if the failed block is a parity block or has no corresponding logged stripe entry in the hash table, which means all data blocks in the stripe have not yet been updated, the value of the failed block can be re-computed by the surviving blocks in the stripe.

If the failed block is a data block and has a corresponding logged stripe entry in the hash table, SWO should find the logged stripe entry and mark the logged data entries in the *Log_data_list* first. If the failed block has a corresponding logged data entry in the *Logged_data_list*, it is marked and its recovery value can be directly copied from the log disk. In more detail, if the failed block has been updated, the corresponding *Log_new_LBA* item must not be NULL, then SWO reads out the log record addressed by *Log_new_LBA* from the log disk as the recovery value of this failed block. Otherwise, if the failed block has not been updated, the corresponding *Log_old_LBA* item must not be NULL, and the recovery value can be obtained from this item. However, if the failed data block has no corresponding logged data entry under the condition that there is a corresponding logged stripe entry, it indicates that the data block has not been updated and the stripe has not used the *reconstruction-write* method to pre-read data blocks. The value of the failed block can be recovered by XORing the old value of all other surviving data blocks and the parity block in the stripe. For each surviving data block, if it has been logged, SWO reads it from the log disk. Otherwise, SWO reads it from the data disks.

```
1.  FOR EACH(parity stripe)
2.    Find the stripe in the hash list and mark the logged data
      block in the Log_data_list;
3.    IF(it has no corresponding logged stripe entry in the
         hash list or the failed block is parity block)
4.      Read out all data block in the stripe from RAID;
5.      Compute the recovery value of the failed block;
6.    ELSE
7.      IF(the failed block is marked)
          /*it has a corresponding  logged data entry*/
8.        IF(Log_new_LBA item of the entry is not NULL)
9.          Read out the new data addressed by
            Log_new_LBA from the log disk as
            the recovery value of the failed block;
10.       ELSE
            /*Log_old_LBA must not be NULL*/
11.         Read out the old data addressed by
            Log_old_LBA from the log disk as
            the recovery value of the failed block;
12.       END IF
13.     ELSE  /*the failed block must not be updated and the
                stripe has not used reconstruction-write. */
14.       FOR EACH(surviving block)
15.         IF(it is marked)
              /*the Log_old_LBA must not be NULL*/
16.           Read out the log record addressed by
              Log_old_LBA from the log disk ;
17.         ELSE
18.           Read out the old value of this block from RAID;
19.         END IF
20.       END FOR
21.       Compute the recovery value of the failed block
          using the old value of surviving blocks;
22.     END IF
23.   END IF
24. END FOR
```

Figure 5. Process Flow of SWO to Recover from Data Disk Failures

IV. Performance Evaluations

A. Experimental setup and methodology

We have implemented a SWO prototype in the Linux Software RAID (MD) framework. In order to show the performance advantages, we also implemented a Data Logging prototype and a RAID5L prototype. The performance evaluations are conducted on a platform of server-class hardware with an Intel Xeon 2.5GHz processor and 4GB DDR memory. We use a Marvel SATA controller card to carry 8 ST31000340NS 1TB SATA disks in the system. The operating system (Linux kernel 2.6.32) and other software (MD, mdadm, IOmeter and RAIDmeter) are on a separate SATA disk. The experimental setup is shown in Table I. We use two financial traces collected from OLTP (online transaction processing) applications running at a large financial institution to evaluate the IO performance, reconstruction time and parity re-synchronization time. In our experiments, RAIDmeter is used as a block-level trace replay tool to evaluate the I/O response time of the storage device.

Table I
EXPERIMENTAL SETUP & TRACE CHARACTERISTICS

Machine	Intel Xeon 2.5GHz, 4GB RAM
OS	Linux 2.6.32
Disk Driver	ST31000340NS SATA 1TB HDD
Benchmark	IOmeter Version 2006.07.27 [8]
Traces	OLTP Application I/O [11]
Trace Characteristics	Financial1.spc: Write Ratio =67.2% Average Request Size = 6.2KB Average IOPS = 69 Financial2.spc: Write Ratio =17.6% Average Request Size = 2.2KB Average IOPS = 125
Trace replay	RAIDmeter [9]

We implement the SWO module, Data Logging module and RAID5L module all by modifying the original RAID5 module in the Linux Kernel. We mainly modify the *handle_stripe5_dirtying, ops_run_prexor, ops_run_reconstruct5* and *ops_run_io* functions. The *handle_stripe5_dirtying* function counts rmw and rcw to choose *read-modify-write* or *reconstruction-write* with taking out the parity block and the data blocks that have been logged. The *ops_run_prexor* and *ops_run_reconstruct5* functions handle the pre-read data blocks and add corresponding entries into the hash table. The *ops_run_io* function sends write requests to the log disk or data disk of RAID5. In addition, we add the hash table structure in raid5.h and the related operations of the hash table function in raid5.c.

In our evaluation, the traditional RAID5 system is configured with 7 disks. The log-assisted systems are all configured with 8 disks, including a 7-disk RAID5 system and a log disk. In order to evaluate the real performance of systems, we do not add the log buffer in the systems. Each disk capacity is set to be 10GB. When the log disk utilization rate reaches 80%, the parity re-synchronization process is started. While conducting the parity re-synchronization time experiments, we assume that the log disk utilization rate does not reach 80%.

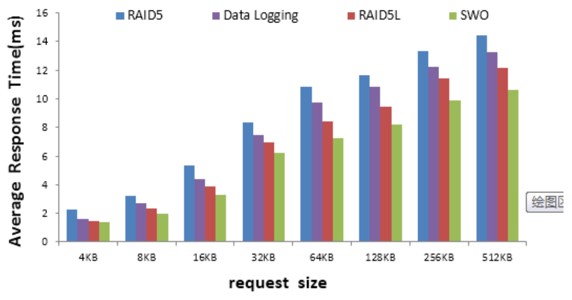
B. Performance results

1) **IOmeter results:** We first conduct an experiment on SWO, traditional RAID5, Data Logging and RAID5L using IOmeter [8] in different workloads. Different RAID architectures use the same RAID volume capacity (60GB) with a chunk size of 512KB. The IOmeter performance results with respect to different access patterns is shown in Figure 6.

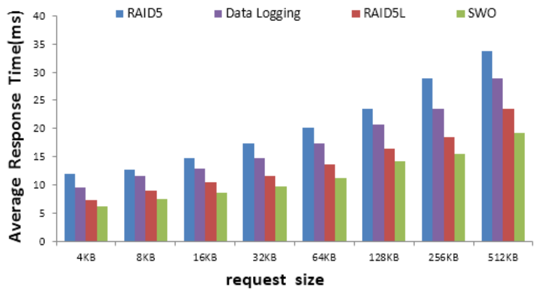
From Figure 6(a), we can see that SWO performs the best for the sequential write requests in terms of average response time. Specifically, SWO on average outperforms traditional RAID5, Data Logging and RAID5L by 32.21%, 21.86% and 12.91%, respectively. For the random write requests, SWO also performs the best in terms of average response time. As shown in Figure 6(b), SWO on average outperforms traditional RAID5, Data Logging and RAID5L by 43.36%, 33.86% and 16.34%, respectively. Since SWO, Data Logging and RAID5L delay the parity blocks update, their performance is better than the traditional RAID. Data Logging only chooses the *read-modify-write* method to handle the write request, it does not dynamically chooses either the *reconstruction-write* or *read-modify-write* method to minimize the number of pre-read operations. So its performance is worse than SWO and RAID5L. SWO takes the logged data blocks and parity block into account when choosing the *reconstruction-write* or *read-modify-write* method. RAID5L only takes the parity block into account when choosing one of the two methods. It considers the logged data blocks after choosing the method. SWO pre-reads less data blocks than RAID5L, so it can improve the write performance compared with RAID5L. Because the traditional RAID5 sequential write is very good, the performance improvement is less than that of random writes.

The default RAID chunk size is 64KB in Linux kernel 2.6.21, but it is 512KB in Linux kernel 2.6.32. In order to observe the chunk size effect on the system, we conduct another experiment on SWO and the traditional RAID5 using IOmeter in various chunk sizes with the results shown in Figure 7. SWO_4K represents the chunk size is 4KB in SWO storage system and RAID5_4K represents the chunk size is 4KB in the RAID5 storage system. With the chunk size changing from 4KB to 512KB, SWO outperforms traditional RAID5 by 18.50%, 20.76%, 22.20%, 27.46% and 32.21% on average for sequential write requests. For random write requests in terms of average response time, SWO outperforms traditional RAID5 by 24.55%, 27.86%, 39.43%, 41.10% and 43.36% on average. For both sequential and random requests, the performance improvement is highest when the chunk size is 512KB. For random requests, the SWO performance is highest when the chunk size is 64KB. This is because when the chunk size is 64KB, the traditional RAID5 performance is also highest in Figure 7(d).

2) **Benchmark trace results:** We conduct the third experiment on SWO, traditional RAID5 system, Data Logging and RAID5L with the same capacity (60GB) with a chunk size of 512KB driven by the two financial traces. The experimental chunk size is 512KB since it is the default in Linux kernel 2.6.32. We run each of the traces on the RAID architectures for an hour, and the performance results are shown in Figure 8. For the Financial1 trace, SWO reduces the average response time of the traditional RAID5 system by up to 70.11%, of Data Logging by 47.32%, and of RAID5L by 33.40%. For the Financial2 trace, the average response time of the SWO is shorter than the traditional RAID5, Data Logging and RAID5L by: 41.81%, 24.97% and 15.92% respectively. SWO requires less pre-read operations than the other two systems. The average pre-read data blocks per write request in different systems is shown in Table II.

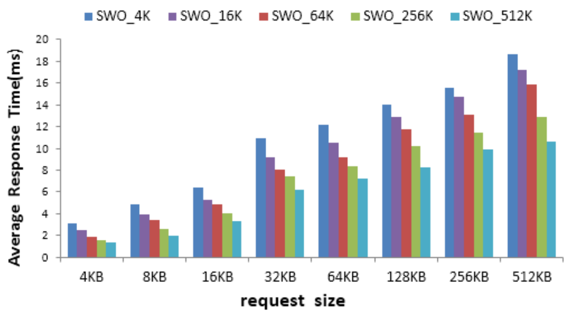


(a) Sequential Write

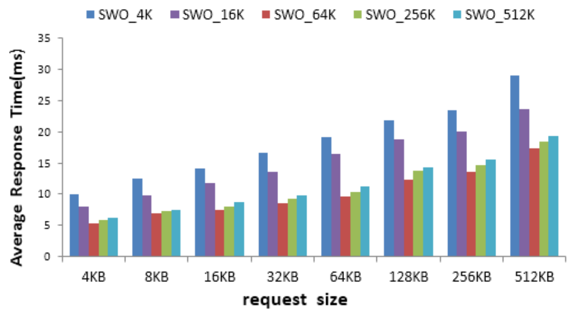


(b) Random Write

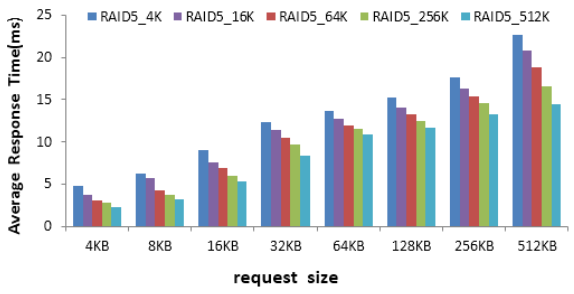
Figure 6. IOmeter Performance Results with Respect to Different Access Patterns



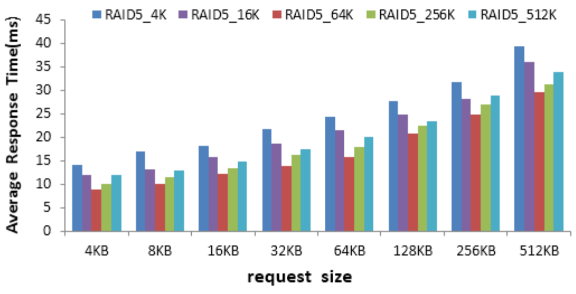
(a) Sequential Write



(b) Random Write



(c) Sequential Write



(d) Random Write

Figure 7. IOmeter Performance Results with Respect to Different Chunk Size

In particular, since the write ratio of Financial2 is smaller than that of Financial1, the performance improvement of Financial2 is not as outstanding as Financial1.

To see how effectively SWO handles disk failure recovery, we conducted experiments on the recovery process of different systems. We also run the two financial traces on the RAID architectures for an hour as initialization. Then we make a data disk of the RAID fail to measure data recovery time and the results are shown in Figure 9. The traditional RAID5 reconstruction time of the two traces is almost the same because the original data recovery process has nothing to do with the previous I/O requests. The reconstruction time of SWO, RAID5L and Data Logging are all higher than the traditional RAID5 because of the bottleneck of the log disk. However, as shown in Figure 9, SWO is shorter than Data Logging and RAID5L by:(58.52%, 21.32%) and (68.85%, 31.17%). The first parenthesized pair is the improvement achieved by SWO over Data Logging under the two traces respectively. The second parenthesized pair is the improvement achieved by SWO over RAID5L under the two traces respectively. The reason is that SWO pre-reads less data blocks than RAID5L and Data Logging, and it reads less data from the log disk when reconstructing. In addition, the hash table average search efficiency of SWO is higher than that of RAID5L and Data Logging.

We next conduct an experiment to measure the parity re-synchronization time. Both the log disk failing or the utilization rate reaching the threshold can trigger a parity

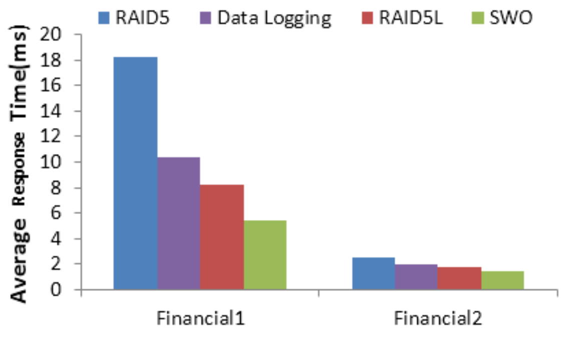


Figure 8. Comparison of Average Response Time Driven by the OLTP Financial Traces

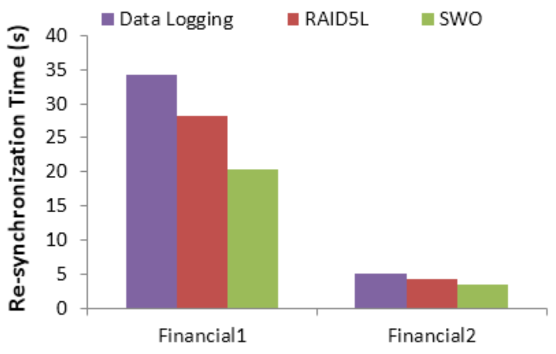


Figure 10. Comparison of Parity Re-synchronization Time

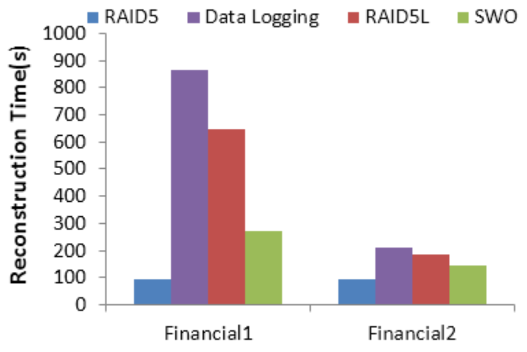


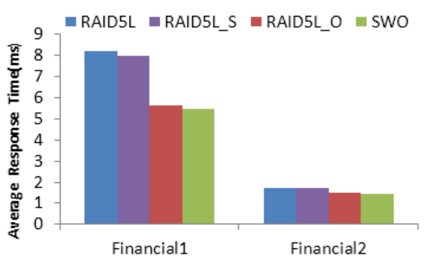
Figure 9. Data Recovery Efficiency Comparison

re-synchronization operation. In our experiment, we first run the traces on the RAID architectures for an hour, and then make the log disk fail to measure the parity re-synchronization time. Figure 10 shows the parity re-synchronization time comparison driven by the two OLTP Financial traces. Reading data blocks from the log disk is also the bottleneck of the parity re-synchronization. Because of less reading from log disk and the higher average hash table search efficiency, SWO performs the best in the three log-assist systems. For Financial1, SWO is better on average than Data Logging and RAID5L by 40.82% and 28.01%, respectively. For Financial2, SWO is better on average than Data Logging and RAID5L by 34.66% and 21.11%, respectively.

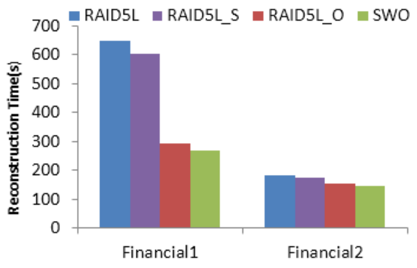
We also implement two systems: RAID5L_S and RAID5L_O by optimizing RAID5L. RAID5L_S is stripe-oriented and considers the logged data blocks after choosing the *reconstruction-write* or *read-modify-write* method. RAID5L_O is data-block-oriented and considers the logged data blocks when choosing the method. Lastly, we conduct the above three experiments on RAID5L_S and RAID5L_O. The experimental results are shown in Figure 11. For the Financial1 trace, RAID5L_S reduces the average response time, reconstruction time and re-synchronization time of the RAID5L by up to 2.59%, 7.03% and 7.14% respectively. While RAID5L_O reduces those of the RAID5L by up to 31.45%, 55.03% and 22.40% respectively. The write ratio of the Financial2 trace is not high, thus the improvements are limited. For the Financial2 trace, RAID5L_S reduces the average response time, reconstruction time and re-synchronization time of the RAID5L by up to 1.43%, 6.18% and 6.24% respectively. While RAID5L_O reduces those of the RAID5L by up to 14.47%, 15.64% and 15.74% respectively. The improvements achieved by RAID5L_O over the RAID5L are higher than RAID5L_S because RAID5L_O reduces the number of pre-read data blocks, but RAID5L_S only reduces the hash table average search time and the IO operations delay is higher than the memory operations delay. SWO improve performance by reducing the number of pre-read data blocks and the hash table average search time. Where the impact of the reduction in pre-read data blocks is bigger than the hash table average search time reduction. Table II is average pre-read data blocks per write request in different systems. For the Finacial1 trace, SWO decreases the average pre-read data blocks per write request of traditional RAID5 by up to 50.96%, that of Data Logging by 31.56%, and that of RAID5L by 14.92%. The Finacial2 trace has a more pronounced frequent update pattern and more workload locality to be exploited. As a result, SWO decreases the average pre-read data blocks per write request of traditional RAID5 by up to 61.17%, that of Data Logging by 35.96%, and that of RAID5L by 16.09%.

V. Related Work

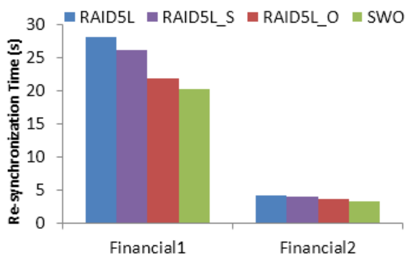
As we know, the write performance and reliability of RAID1 is higher than RAID5. Mogi et al. [12] proposed a Hot mirroring scheme, in which storage space is partitioned into two regions: mirrored region and RAID5 region. Hot data blocks are mirrored, while the cold blocks are stored in the RAID5 area. In addition, mirrored pairs and RAID5 stripes are orthogonally laid out, through which the performance degradation during rebuilding is minimized. HP AutoRAID [13] is a similar scheme as Hot mirroring, that stores write-active data in RAID1 and write-inactive data in



(a) Comparison of Average Response Time Driven by the OLTP Financial Traces



(b) Data Recovery Efficiency Comparison



(c) Comparison of Parity Re-synchronization Time

Figure 11. Different Optimization Method Performance Comparison

Table II
AVERAGE PRE-READ DATA BLOCKS PER WRITE REQUEST

	RAID5	Data logging	RAID5L_S	RAID5L_O	RAID5L	SWO	SWO Improved by
Financial1	3.14	2.25	1.81	1.54	1.81	1.54	50.96% / 31.56% / 14.92% / 0 / 14.92%
Financial2	2.08	1.44	0.87	0.65	0.87	0.65	61.17% / 35.96% / 16.09% / 0 / 16.09%

RAID5. In addition, it migrates data automatically to and from the mirror area according to the data activity.

To avoid the need for parity block updating, Mogi et al. [14] propose a dynamic parity stripe scheme with improving the write performance of RAID5. When a block is written, its old value remains in place on disk to protect the parity groups and the new value is buffered. While the buffered blocks generate a full stripe and write them back to the RAID system. In addition, it needs a garbage collection process to recover the parity blocks. The dynamic parity stripe reorganization technique solves the small-write problem at the cost of physical locality of data and the overhead of a background garbage collector. It requires either an additional storage medium to store the mapping of data blocks or a file system such as the log-structure file system (LFS) to maintain this information. However, AFRAID [15] updates the data in-place without updating the parity block but marking it as not redundant. So it needs a bitmap in NVRAM to record the parity block that is not updated. When the system is idle or the number of the parity blocks that are not updated meets the threshold, the parity blocks will be recomputed and rewritten.

HPDA [16] is a hybrid parity-based RAID architecture, that uses SSDs (data disks) and HDD (parity disk) to compose a RAID4 system. The spare parts of the HDD is a log area that can absorb the small write requests. In addition, a second HDD is provided to form a RAID1 with the log region. Note that some space of the log disk in HPDA is wasted, HerpRap [17] uses the spare space of HDDs as a log region and CDP region. HRAID6ML [18] is an extension of HPDA and HerpRap where the data disks are SSDs and two parity disks are HDDs. The free space of parity disks form a mirrored log region that not only improves small-write performance but also extends the lifetime of SSDs.

In DCD [19], the disk that is composed of the log disk and data disk buffers random small writes into a large sequential write. The write data is first sent to the NVRAM. After assembling into a large request, they are logged to the disk and then written back to the log disk when the system is idle. Inspired by this idea, PEraid [20] uses half of every primary disks to compose a RAID5 write buffer in a replication storage system. The virtual write buffer can absorb new writes improving write performance with a reliability guarantee. The new data block that does not update in-place is cached in the log area, so the read performance is degraded.

There are many studies conducted on logging techniques to improve write performance. Parity Logging [4] which logs the parity updates to overcome small-write performance issues is the first paper to introduce the logging technique used in a RAID system. Menon [21] proposed 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. Similar to Parity Logging, Logging RAID [22] is a parity-based RAID architecture that adopts data logging techniques to overcome the small-write problem. Data Logging [5] is proposed for further improving write performance in constantly active RAIDs. It logs the old and new value of the data blocks that are to be updated instead of parity update images.

Considering that Parity Logging has to read the old data block and the new data block every time it is updated, Jin et al. [3] proposed a log-assisted RAID6 architecture, called RAID6L to reduce the pre-read operations with improving write performance. RAID6L only needs to pre-read the old data block when it is updated for the first time. It can choose either the *read-modify-write* or the *reconstruction-write* method to service the write request while Parity

Logging can only choose the *read-modify-write* method.

VI. Conclusion

In this paper, we propose a write performance optimization for RAID-structured storage systems called SWO, which is stripe-oriented and can improve the write performance and decrease the reconstruction and synchronization time. SWO chooses the *reconstruction-write* or *read-modify-write* method which selects suitable log blocks to achieve better write performance based on the log information. We have implemented the SWO system and completed a series of experiments with IOmeter and RAIDmeter. Our results show that SWO is effective in improving the write performance and saving reconstruction and parity re-synchronization time. The IOmeter results show that the write performance of SWO is better on average than that of the traditional RAID, Data Logging and RAID5L by up to 32.21%, 21.86% and 12.91% respectively for sequential write requests and 43.36%, 33.86% and 16.34% respectively for random write requests. At the same time, the average response time of SWO, evaluating by the real trace, is shorter than that of the traditional RAID, Data Logging and RAID5L approaches. Moreover, when rebuilding and synchronizing, SWO is faster than Data Logging and RAID5L.

Acknowledgment

This work was supported by the National High-tech R & D Program of China (863 Program) No. 2015AA016701, No. 2015AA015301, No. 2013AA013203; NSFC No.61472153, No. 61402189; the Fundamental Research Funds for the Central Universities, HUST, under Grant 2015QN072. This work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China. We are grateful to Kenneth B. Kent and the anonymous reviewers for their valuable feedback and suggestions.

References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, *A case for redundant arrays of inexpensive disks (RAID)*. ACM, 1988, vol. 17, no. 3.

[2] M. Holland and G. A. Gibson, *Parity declustering for continuous operation in redundant disk arrays*. ACM, 1992, vol. 27, no. 9.

[3] C. Jin, D. Feng, H. Jiang, and L. Tian, “RAID6L: A log-assisted RAID6 storage architecture with improved write performance,” in *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*. IEEE, 2011, pp. 1–6.

[4] D. Stodolsky, G. Gibson, and M. Holland, “Parity logging overcoming the small write problem in redundant disk arrays,” in *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2. ACM, 1993, pp. 64–75.

[5] E. Gabber and H. F. Korth, “Data logging: A method for efficient data updates in constantly active RAIDs,” in *Data Engineering, 1998. Proceedings., 14th International Conference on*. IEEE, 1998, pp. 144–153.

[6] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, “Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics,” *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 7, 2008.

[7] G. Gibson, “Reflections on failure in post-terascale parallel computing,” in *International Conference on Parallel Processing*, 2007.

[8] IOmeter, http://sourceforge.net/projects/iometer.

[9] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song, “PRO: A Popularity-based Multi-threaded Reconstruction Optimization for RAID-Structured Storage Systems.” in *FAST*, vol. 7, 2007, pp. 301–314.

[10] C. Jin, D. Feng, H. Jiang, L. Tian, J. Liu, and X. Ge, “Trip: Temporal redundancy integrated performance booster for parity-based raid storage systems,” in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. IEEE, 2010, pp. 205–212.

[11] OLTP Application I/O, UMass Trace Repository, http://traces.cs.umass.edu/index.php/Storage/Storage.

[12] K. Mogi and M. Kitsuregawa, “Hot mirroring: a method of hiding parity update penalty and degradation during rebuilds for RAID5,” in *ACM SIGMOD Record*, vol. 25, no. 2. ACM, 1996, pp. 183–194.

[13] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, “The HP AutoRAID hierarchical storage system,” *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 108–136, 1996.

[14] K. Mogi and M. Kitsuregawa, “Dynamic parity stripe reorganizations for RAID5 disk arrays,” in *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*. IEEE, 1994, pp. 17–26.

[15] S. Savage and J. Wilkes, “AFRAID:a frequently redundant array of independent disks,” *parity*, vol. 2, p. 5, 1996.

[16] B. Mao, H. Jiang, S. Wu, L. Tian, D. Feng, J. Chen, and L. Zeng, “HPDA: A hybrid parity-based disk array for enhanced performance and reliability,” *ACM Transactions on Storage (TOS)*, vol. 8, no. 1, p. 4, 2012.

[17] L. Zeng, D. Feng, B. Mao, J. Chen, Q. Wei, and W. Liu, “HerpRap: A Hybrid Array Architecture Providing Any Point-in-Time Data Tracking for Datacenter,” in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE, 2012, pp. 311–319.

[18] L. Zeng, D. Feng, J. Chen, Q. Wei, B. Veeravalli, and W. Liu, “HRAID6ML: A hybrid RAID6 storage architecture with mirrored logging,” in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–6.

[19] Y. Hu and Q. Yang, “DCD:A new approach for boosting I/O performance,” in *ACM SIGARCH Computer Architecture News*, vol. 24, no. 2. ACM, 1996, pp. 169–178.

[20] J. Wan, C. Yin, J. Wang, and C. Xie, “A new high-performance, energy-efficient replication storage system with reliability guarantee,” in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, 2012, pp. 1 – 6.

[21] J. Menon, “A performance comparison of RAID-5 and log-structured arrays,” in *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*. IEEE, 1995, pp. 167–178.

[22] Y. Chen, W. W. Hsu, and H. C. Young, “Logging RAID:An approach to fast, reliable, and low-cost disk arrays,” in *Euro-Par 2000 Parallel Processing*. Springer, 2000, pp. 1302–1311.