RAID6L: A Log-Assisted RAID6 Storage Architecture with Improved Write Performance

Chao Jin*, Dan Feng*✉, Hong Jiang†, Lei Tian*†
*School of Computer, Huazhong University of Science and Technology
*Wuhan National Lab for Optoelectronics
†University of Nebraska-Lincoln
✉Corresponding author: dfeng@hust.edu.cn
chjinhust@gmail.com, jiang@cse.unl.edu, ltian@hust.edu.cn

***Abstract*—The RAID6 architecture is playing an increasingly important role in modern storage systems due to its provision of very high reliability. However, its high write penalty, because of the double-parity-update overheads upon each write operation, has been a persistent performance bottleneck of the RAID6 systems. In this paper, we propose a log-assisted RAID6 architecture, called RAID6L, to boost the write performance of the RAID6 systems. RAID6L integrates a log disk into the traditional RAID6 architecture, and alleviates its write penalty by simplifying the processing steps to service a write request. On the other hand, RAID6L also guarantees that the accelerated RAID6 systems can still recover from double disk failures. The Parity Logging scheme was originally proposed to accelerate the XOR based RAID5, and we propose a method to make it also applicable to the Reed-Solomon based RAID6. We present a detailed comparison between RAID6L and Parity Logging, and show that RAID6L has several advantages over Parity Logging. Experimental results show that RAID6L significantly increases the data transfer rate and decreases the request response time when compared with the traditional RAID6 and Parity Logging systems.**

Keywords—RAID6; log; write performance; parity logging

I. INTRODUCTION

The RAID6 architecture outperforms the other RAID levels in fault tolerance due to its ability to recover from arbitrary two concurrent disk failures in the array. However, the reliability benefit of RAID6 comes at the expense of its write performance degradation. Take the typical Reed-Solomon coded RAID6 [1] for example, there are two parity blocks (i.e., parities P and Q) in each parity stripe, and when one data block is updated, the two parity blocks must be updated synchronously to guarantee parity consistency. Thus, writing a RAID6 data block entails the following three phases: pre-read the old values of the data and parity blocks, compute the new values of the data and parity blocks, and write the new values to the disks. This process incurs as many as six disk operations [2]. Obviously, the write penalty of the RAID6 architecture is quite high, which is the main reason why its usage in practical storage systems is still limited.

In this paper, we propose a log-assisted RAID6 architecture, called RAID6L. RAID6L integrates a log disk into the traditional RAID6 array, and greatly boosts the write performance of the RAID6 array with a minimal reliability loss. The traditional RAID6 array updates the data blocks and parity blocks simultaneously. On the contrary, RAID6L simply logs the old or new values of the related data blocks when serving a write request, and delays the parity updates until the system is idle or lightly loaded. The log records are first accumulated in a log buffer, and flushed to the log disk when the buffer is full or reaches a utilization threshold. The periodical large sequential flush operations incur very little overhead to the system, allowing the write penalty of RAID6L to be greatly reduced compared with the traditional RAID6 array. On the other hand, the log disk guarantees that the RAID6 array can still recover from arbitrary two concurrent disk failures even when its parities are no longer consistent with the data.

The Parity Logging scheme [3] is one of the most representative schemes that are proposed to boost the write performance of the XOR parity based RAID systems. However, it cannot be applied to the Reed-Solomon coded RAID6 directly, since the Q parity of the Reed-Solomon coded RAID6 is not computed by XOR parity but the finite field arithmetic. In this paper, we propose a general way to make Parity Logging also applicable to the Reed-Solomon coded RAID6. Then, we present a detailed comparison between RAID6L and Parity Logging, and show that RAID6L has several advantages over Parity Logging.

We implement a RAID6L prototype and a Parity Logging prototype in the Linux software RAID framework, and evaluate their practical performance through trace-driven experiments. Experimental results show that RAID6L significantly increases the data transfer rate and decreases the user request response time when compared with the traditional RAID6 and Parity Logging systems.

II. DESIGN AND IMPLEMENTATION OF RAID6L

A. RAID6L Architecture

RAID6L is composed of a traditional RAID6 array and a log disk. Figure 1 provides an architectural overview of the RAID6L system. Generally, RAID6L may operate in any of the following three states: the accelerating state when RAID6L delays parity updates to accelerate the write speed, the transitional state when RAID6L re-synchronizes the parities between the data disks and the parity disks, and the normal state when the parity is consistent with the data. In the normal state, RAID6L acts in exactly the same way as a traditional RAID6 array. However, in the accelerating state or the transitional state, RAID6L handles the write requests differently from the traditional RAID6 array. We will illustrate the write-request process flow of RAID6L in details in the following sections. The key in-memory metadata structure of RAID6L is a hash list. Each entry in the hash list corresponds to a data block, and stores the log addresses of the data block's original and latest values in the log disk.

978-1-4577-0428-4/11/$26.00 ©2011 IEEE

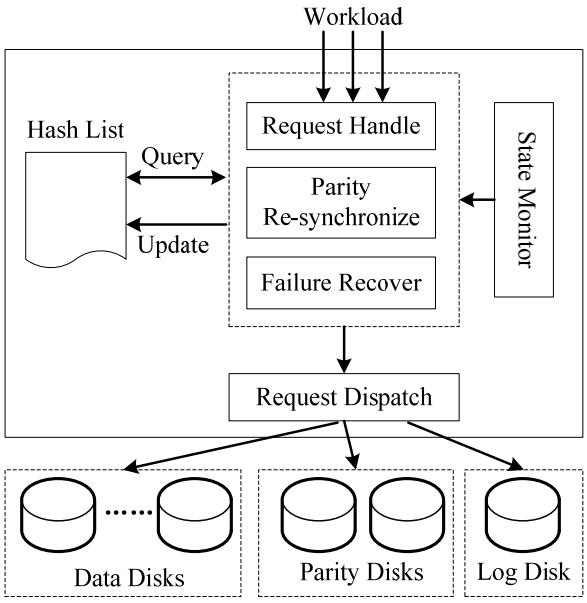

Figure 1 RAID6L Architecture

B. Hash List Structure

The structure of the Hash list is shown in Figure 2. The hash list is composed of a table head array and several entry lists. Each element of the table head array is called a hash slot, and its content is an address that points to an entry list. Each entry in the entry list corresponds to a data block in the RAID6 array. Each entry has the following items:

LBA: logical block address of the data block in the array.

Ori_Ad: address of the data block's original value in the log disk.

Cur_Ad: address of the data block's current (i.e., latest) value in the log disk.

Next: pointer to the next item in the entry list.

To search the corresponding hash entry of a data block, the system first finds the corresponding hash slot by hashing the logical block address of that data block, and then searches sequentially the entry list in the slot. 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 is just the corresponding entry of the data block; otherwise, if no such entry is found, the data block does not have a corresponding entry in the hash list. Similarly, to insert a hash entry into the hash list, the system first finds the corresponding hash slot by hashing its *LBA* item, and then inserts the entry into the entry list in this slot.

C. Process Flow of Write Requests

First we review how a traditional RAID6 array processes the write requests. On receiving an upper-level write request on the RAID device, the RAID6 controller first translates it into the requests on the component disks. These requests are grouped by the parity stripes. Before writing the new data to the disks, RAID6 needs to compute the new parity blocks for each parity stripe. There are usually two alternative methods to do this, namely, *reconstruction-write* and *read-modify-write* respectively. The main difference between the two methods

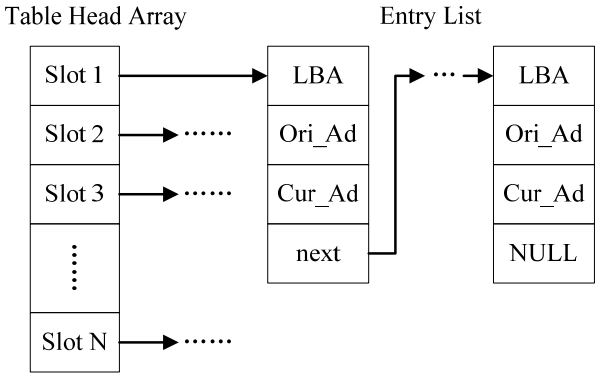

Figure 2 Hash List Structure

lies in the data blocks that must be pre-read for the computation of the new parity blocks [4]. Generally, the RAID6 controller dynamically chooses one of the two methods for each parity stripe to minimize the number of pre-read operations. After the new parity blocks are computed, the new values of the data and parity blocks are written to the disks and the write request completes.

The write-request process flow of RAID6L in the accelerating state is shown in Figure 3. RAID6L differs from the traditional RAID6 array in several important ways. First, before pre-reading a data block, RAID6L queries the hash list to see if the block has a corresponding entry in the hash list, and only executes the pre-read operation if it does not have a corresponding hash entry. Second, when a data block is pre-read or updated, RAID6L logs the pre-read or new value of the data block into the log disk, and updates the corresponding hash entry of the data block. Third, for either the *reconstruction-write* or *read-modify-write* method, RAID6L does not need to pre-read or update the parity blocks.

As mentioned before, the transitional state of RAID6L describes the parity re-synchronizing process. In the transitional state, some of the parity stripes have returned to the consistent state through re-synchronization, while the others still remain in the inconsistent state. Before writing a parity stripe, RAID6L checks if the data blocks of this parity stripe have corresponding entries in the hash list. If none of the data blocks has a corresponding hash entry, the parity stripe must be in the consistent state; otherwise, it must be in the inconsistent state. Writing a consistent parity stripe follows the exact process flow of a traditional RAID6 array, and either the *reconstruction-write* or *read-modify-write* method can be chosen to compute the new parity blocks. On the other hand, writing an inconsistent parity stripe also follows the process flow of a traditional RAID6 array, with the exception that the *reconstruction-write* method must be selected to compute the new parity blocks. After the writing process completes, the parity blocks of the inconsistent parity stripe are updated, thus the re-synchronization of this parity stripe is completed at the same time. Then, all the corresponding hash entries of the data blocks in this parity stripe are deleted from the hash list, since the parity stripe has already returned to the consistent state.

D. Parity Re-synchornization

In a RAID6L system, the parity re-synchronization operations may be triggered in the following four situations.

```
1.  Translate the up-level update request into parity stripes
    according to the structure of the RAID6 array.
2.  FOR EACH (parity stripe)
3.      IF (reconstruction - write style is used)
4.          FOR EACH (data block not to be updated)
5.              IF (it has no corresponding entry in the hash list)
6.                  Pre-read the original value of the block, and
                    store it in the log disk;
7.                  Create an entry in the hash list for the block;
8.                  Record the log address of the original value
                    in the hash entry.
9.              END IF
10.         END FOR
11.     ELSE
12.         FOR EACH (data block to be updated)
13.             IF (it has no corresponding entry in the hash list)
14.                 Pre-read the original value of the block, and
                    store it in the log disk;
15.                 Create an entry in the hash list for the block;
16.                 Record the log address of the original value
                    in the hash entry.
17.             END IF
18.         END FOR
19.     END IF
20.     FOR EACH (data block to be updated)
21.         IF (it has no corresponding entry in the hash list)
22.             Create an entry in the hash list for the block.
23.         END IF
24.         Store the new value of the data block in the log disk;
25.         Record the log address of the new value in the hash
            entry of the data block.
26.         Write the new value of the data block to the RAID disk.
27.     END FOR
28. END FOR
```

Figure 3 Write-Request Process Flow of RAID6L in the Accelerating State

First, the system enters the transitional state. In the transitional state, RAID6L traverses through the hash list sequentially, and for each hash entry, RAID6L finds its corresponding data block in the RAID6 array, reads out all the data blocks in the same parity stripe, and then re-computes the parity blocks in the parity stripe. After that, RAID6L deletes all the corresponding hash entries of the data blocks in the parity stripe, and the re-synchronization of this parity stripe completes. When there is no hash entry left in the hash list, the re-synchronization of the entire system completes. Then, RAID6L frees all the space in the log disk, and switches to the normal state.

Second, the system crashes when RAID6L is in the accelerating or transitional state. After the system reboots, the hash list in the memory has been lost. At this time, RAID6L must re-synchronize the parity stripes as soon as possible, for otherwise a disk failure may cause permanent data loss. However, without the hash list, RAID6L is not aware of which parity stripes have been updated. Thus, RAID6L needs to re-synchronize all the parity stripes sequentially in the RAID6 array. This process is similar to the initial synchronization process when the RAID6 array is created. To protect the hash list from power failures, Non-Volatile RAM may be used to store the hash list.

```
1.  FOR EACH (parity stripe)
2.      FOR EACH (failed data block)
3.          IF (it has a corresponding entry in the hash list)
4.              IF (the Cur_Ad item of the entry is not NULL)
5.                  Read out the log record addressed by Cur_Ad
                    from the log disk as the recovery value
                    of this data block.
6.              ELSE
7.                  Read out the log record addressed by Ori_Ad
                    from the log disk as the recovery value
                    of this data block.
8.              END IF
9.          ELSE
10.             FOR EACH (surviving data block)
11.                 IF(it has a corresponding entry in the hash list)
12.                     Read out the log record addressed by
                        Ori_Ad from the log disk as the
                        original value of this block.
13.                 ELSE
14.                     Read out the original values of this data block
                        from the RAID disk.
15.                 END IF
16.             END FOR
17.             Read out the original values of the parity blocks
                from the RAID disks.
18.             Compute the original value of the failed data block
                using the RAID6 recovery algorithm, and set the
                original value as the recovery value of this block.
19.         END IF
20.     END FOR
21. END FOR
```

Figure 4 Process Flow of RAID6L to Recover From Two Data Disk Failures

Third, if the log disk fails when RAID6L is in the accelerating or transitional state, the unsynchronized parity stripes lose the protection of the log disk, thus RAID6L must also start to re-synchronize the parity stripes immediately. Since the hash list is not affected, RAID6L is aware of which parity stripes have been updated and only needs to re-synchronize these parity stripes.

Fourth, the log disk is filled up when RAID6L is in the accelerating state. At this time, RAID6L is still able to tolerate double disk failures, since the hash list and the log records in the log disk are complete. Thus, RAID6L can choose not to start the parity re-synchronization operations so as not to compete with the application I/O operations. However, the process flow in the transitional state rather than the accelerating state must be used to handle the write requests, since RAID6L can not add new log record into the log disk. When the application workload becomes idle, RAID6L enters the transitional state and start to re-synchronize the unsynchronized parity stripes.

E. Data Recovery

If RAID6L is in the normal state when disk failures occur, the recovery strategy is the same as a traditional RAID6 array. Otherwise, no matter whether RAID6L is in the accelerating or transitional state, it needs to process the data recovery with the help of the log disk, and the recovery strategy is divided into three cases as follows.

Case 1: Two parity disks fail.

In this case, the parity disks can be directly reconstructed by the data disks, since all the data disks are complete. When the parity disks are reconstructed, the parity re-synchronization is done at the same time.

Case 2: Two data disks fail.

In this case, each parity stripe in the RAID6 array loses two data blocks. The data recovery process flow is shown in Figure 4. For each failed data block, if it has a corresponding hash entry in the hash list, its current value can be directly copied from the log disk. In particular, if the failed data block has been updated in the accelerating state, the *Cur_Ad* item of its hash entry must not be *NULL*, and its current value is addressed by the *Cur_Ad* item in the log disk; otherwise, if the failed data block has not been updated in the accelerating state, the *Ori_Ad* item of its hash entry must not be *NULL*, and its original value addressed by the *Ori_Ad* item in the log disk is just its current value.

On the other hand, if the failed data block does not have a corresponding hash entry in the hash list, it must not have been updated in the accelerating state. For each of the surviving data block in the parity stripe, its original value can be read out either from the log disk (addressed by the *Ori_Ad* item) if it has a corresponding hash entry in the hash list, or from the RAID disk if it does not have a corresponding hash entry. Since the parity stripe is originally in the consistent state, the original value of the failed data block can be re-computed by the original value of all the surviving data blocks and the parity blocks through the RAID6 parity algorithm. Since the failed data block has not been updated, its original value is just its current value.

Case 3: A data disk and a parity disk fail.

In this case, each parity stripe loses one data block and one parity block. The data recovery process is similar to the second case. If the failed data block has a corresponding hash entry, it can be directly recovered from the log disk; otherwise, it can be recovered through the RAID6 recovery algorithm. After the failed data blocks are recovered, RAID6L starts the parity re-synchronization operations, and the RAID6 array returns to the consistent state.

III. Comparsion with Parity Logging

A. Applying Parity Logging to the Reed-Solomon Coded RAID6

Parity Logging was originally applied to the RAID5 array. When a data block in the RAID5 parity stripe is updated, Parity Logging creates a log record that contains the XOR result of the data block's old and new values in the log disk. The log record is actually the update image of the parity block in that parity stripe. In the parity re-synchronization process, each parity block is updated to its latest value by XORing with all its corresponding update images. When one-disk failure occurs in the RAID5 array, the system first updates all the parity blocks to their latest values using the log records in the log disk, and then recovers the lost data through the RAID5 recovery algorithm.

Different from RAID5, there are two parity blocks in each Reed-Solomon coded RAID6 parity stripe, namely, P parity block and Q parity block respectively. The P parity block is generated by simple XOR operations like RAID5, however, the Q parity block is generated by the more complicated finite field arithmetic, and it can not be updated to its latest value by XORing its original value with its corresponding update images. Thus, the Parity Logging scheme can not be applied to the RAID6 array directly. Inspired by [5], we think of a method to solve this problem. In fact, if the log records are regarded as the update images of the data blocks rather than the parity blocks, Parity Logging can also be applied to the RAID6 array. It must be noted that, for each data block, its latest value can be computed by XORing its original value with its corresponding update image, and conversely, its original value can also be computed by XORing its current value with its corresponding update image. Suppose that two data disks fail in the RAID6 array. First, the original value of the surviving data blocks can be computed through their latest values and corresponding update images. Then, the original value of the failed data blocks can be computed by the RAID6 parity algorithm, since the RAID6 array is originally in the consistent state. Given the original values of the failed data blocks recovered, their latest values can be computed through their corresponding update images.

B. Comparsion between Parity Logging and RAID6L

Compared with Parity Logging, RAID6L has the following advantages.

First, RAID6L takes advantage of spatial locality that commonly seen in enterprise workloads [6] to reduce the pre-read operations when serving the write requests. Parity Logging must pre-read the old value of a data block every time it is updated. On the other hand, RAID6L only needs to pre-read the old value of a data block when it is updated for the very first time. RAID6L creates a hash entry in the hash list for each updated data block. When updating a data block, RAID6L skips the pre-read operation if the data block has a corresponding hash entry in the hash list. Compared with Parity Logging, RAID6L saves the disk I/O operations when serving the write requests. The stronger the workload locality is, the more disk operations RAID6L saves.

Second, RAID6L can choose either the *read-modify-write* or the *reconstruction-write* method to serve write requests, while Parity Logging can only choose the *read-modify-write* method. For the RAID6L scheme, if the *read-modify-write* method is used, the system pre-reads the data blocks that must be updated in the parity stripe; on the other hand, if the *reconstruction-write* method is used, the system pre-reads the data blocks that should not be updated in the parity stripe. RAID6L dynamically chooses one method for each parity stripe to minimize the pre-read operations. However, for the Parity Logging scheme, the system always needs to pre-read the data blocks that must be updated.

Third, in case of disk failures, the data recovery process of RAID6L is faster than Parity Logging. To recover the failed blocks in a parity stripe, Parity Logging must first compute the original values of the surviving data blocks in the parity stripe, and then compute the original values of the failed blocks, and finally compute the latest values of the failed blocks. In the data recovery process, all the log records in the log disk will be used. On the other hand, under the RAID6L scheme, a failed data block can be directly recovered from the log disk if it has a corresponding entry in the hash list. Otherwise, if the failed data block does not have a corresponding entry (indicat-

ing that it has not been updated), RAID6L can quickly locate the original values of the other blocks in the same parity stripe by querying the hash list, and recover the failed data block through the RAID6 recovery algorithm. In this process, only a small part of the log records will be used. Compared with Parity Logging, RAID6L effectively reduces data recovery time, thus decreases the risk of data loss, and increases the reliability of the RAID6 array.

Additionally, Parity Logging must perform the XOR operations every time it creates a log record, while RAID6L directly logs the data value in the log disk. Thus, RAID6L saves the CPU resource and time overhead of the XOR computation. RAID6L may need more log space than Parity Logging. This, however, is not a major problem, since modern disk capacity is growing steadily and the space of the log disk is periodically freed and reused.

IV. Performance Evaluation

A. Prototype Implementation

In order to evaluate the practical performance of RAID6L, we have implemented a RAID6L prototype in the Linux software RAID framework. We have also implemented a Parity Logging prototype for comparison. We conduct the experiments on a server-class hardware platform with an Intel Xeon 3.0GHz processor and 1GB DDR memory. A Marvel SATA controller card is used to carry 8 SATA disks. A separate IDE disk is used to house the operating system (Linux Kernel 2.6.21.1) and other software (*MD* and *mdadm*).

We implement the RAID6L module by modifying the original RAID6 module in the Linux Kernel. We mainly modify the *handle_stripe6* function and add the hash list structure in *raid5.c*. Some other changes are also made to *md.c*, *md_k.h*, and *raid5.h*. The total amount of modification is about 500 lines of C code.

The implementation of Parity Logging is a little different from that of RAID6L. It also sets up a hash list in the memory. Each updated data block has a corresponding hash entry in the hash list, and the hash entry stores the address of the latest log record for that data block. Each log record has a log head that contains the *LBA* of the data block and the address of the previous log record of the data block. Thus, the log records of the same data block are chained together, and we can traverse them one by one starting from the one addressed by the hash entry. It must be noted that, we cannot store the addresses of all the log records in the hash list, for otherwise the memory usage could be extremely high.

In our evaluation, we conduct the experiments on the traditional RAID6 array, RAID6L, and Parity Logging respectively. The traditional RAID6 array is configured with 7 disks. Each of RAID6L and Parity Logging is configured with 8 disks, including a 7-disk RAID6 array and a log disk. The log buffer for RAID6L or Parity Logging is set up as a two-dimensional byte array in the memory. The capacity of the log buffer is set to be 10MB, and its content is flushed to the log disk when its utilization rate reaches 80%.

B. Trace-Driven Performance Evaluation

We conduct the performance experiment on the three RAID architectures by replaying traces collected from the real-world environment. The characteristics of the traces [9] are shown in Table 1. We observe that workload locality widely exists in these traces, and 13-40% of the requested addresses are updated more than once. The trace replay tool is RAIDmeter [10] that can replay block-level traces and evaluate the practical performance of storage systems.

We run each of the traces on the RAID architectures for an hour, and the evaluation result is shown in Table 2. For the Finacial1 trace, RAID6L reduces the average response time and the average number of pre-read data blocks of the traditional RAID6 array by up to 45% and 48% respectively, and those of Parity Logging by 27% and 17% respectively. The Finacial2 trace has a more pronounced frequent update pattern, thus exposing more workload locality to be exploited. As a result, RAID6L improves on the average number of pre-read data blocks per write request over the traditional RAID6 array by 66%. However, the write ratio of the financial2 trace is not high, thus the improvements of the average response time achieved by RAID6L over the traditional RAID6 array is also limited, by about 21%. Meanwhile, Parity Logging does not have the ability to exploit the workload locality, thus RAID6L also outperforms Parity Logging in the average response time and the average number of pre-read data blocks per write request by 12% and 31% respectively. The Exchange trace also has strong repeated update pattern, moreover, it has bigger request sizes and relatively more sequential I/O requests, increasing the possibility for RAID6L to use the *reconstruction-write* method to further reduce the number of pre-read data blocks for each write request. RAID6L decreases the average response time and the average number of pre-read data blocks per write request by 30% and 44% when compared with the traditional RAID6 array, and by 17% and 22% when compared with Parity Logging. The Build trace is typically composed of small random write requests. It exhibits much less of a frequent-update pattern than the Finacial2 and Exchange traces, lowering RAID6L's advantage over Parity Logging in these two measures to just up to 7% and 12% respectively. However, RAID6L performs much better than the traditional RAID6 array, and achieves the improvements in these two measures by 63% and 65% over the traditional RAID6 array.

C. Evaluation of Data Recovery Efficiency

We conduct another experiment on the three RAID architectures to evaluate their data recovery efficiency in case of double disk failures. Since the data recovery times of RAID6L and Parity Logging are affected by factors such as the number of log records in the log disk, thus, before evaluating the data recovery times of RAID6L and Parity Logging, we first replay the Financial1 trace on each of them for half an hour to warm them up and stabilize their recovery times.

Figure 5 plots the measured data recovery time for the three RAID architectures. The *Replay Once* case refers to the experiment where the trace is replayed for only once, and the *Replay Twice* case refers to the experiment where the trace is replayed for a second time before evaluating the data recovery times of the RAID architectures. In our experiments, individual disk capacity is set to be 10GB, and in both cases, the traditional RAID6 takes about five minutes to recover from double disk failures. Although RAID6L may have a simpler data recovery process than the traditional RAID6 (e.g., the data block that has a corresponding hash entry can be directly

Table I Characteristics of the Traces

Trace	avg. req. size	write/read Ratio	IOPS	addresses updated more than once
Finacial1 [7]	6.2KB	3.32	69	30%
Finacial2 [7]	2.2KB	0.22	125	40%
Exchange [8]	12.5KB	1.53	611	35%
Build [8]	4.1KB	41	1980	13%

Table II Evaluation Results of Replaying the Traces to the RAID Architectures

	average response time (ms)				average pre-read data blocks per write request			
	RAID6	Parity Logging	RAID6L	RAID6L Improved by	RAID6	Parity Logging	RAID6L	RAID6L Improved by
Finacial1	2.31	1.76	1.28	45% / 27%	3.29	2.05	1.71	48% / 17%
Finacial2	1.16	1.05	0.92	21% / 12%	2.26	1.10	0.76	66% / 31%
Exchange	4.05	3.40	2.83	30% / 17%	4.17	3.01	2.35	44% / 22%
Build	15.98	6.55	6.06	63% / 7%	2.96	1.17	1.03	65% / 12%

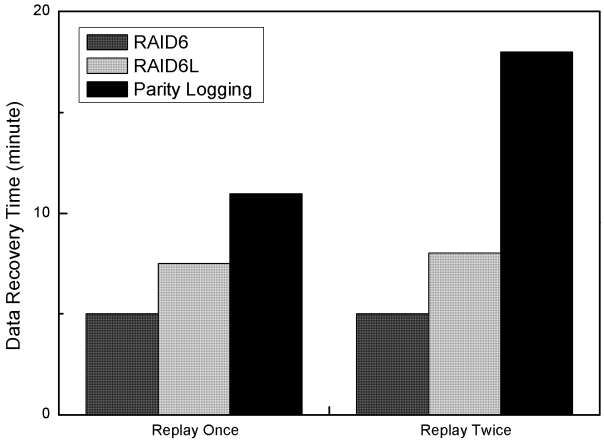

Figure 5 Data Recovery Efficiency Comparison

recovered by copying its latest value from the log disk), the measured data recovery time of RAID6L is somewhat longer than the traditional RAID6. This is because the traditional RAID6 can handle the parity stripes sequentially and efficiently, while RAID6L needs to access the log disk frequently. Replaying the trace for a second time creates few new hash entries for RAID6L, thus the data recovery time of RAID6L for the *Replay Twice* case does not change much. As for Parity Logging, its measured data recovery time is longer than RAID6L. This is partly due to our implementation of Parity Logging in which only the address of the latest log record of a data block is stored in the memory, and the address of a previous log record of the same data block must be read from the head of the current log record. It must be noted that, even though the memory is big enough to hold the addresses of all the log records for Parity Logging, its data recovery time should still be longer than RAID6L, since it must use all the log records in the recovery process. On the other hand, replaying the trace for a second time incurs a longer data recovery time for Parity Logging, since the number of log records increases.

V. Conclusion

We have addressed the write performance problem of the RAID6 architecture in this paper. Our main contribution is twofold. First, we observe that the representative Parity Logging scheme can not be used directly to boost the write performance of the Reed-Solomon coded RAID6 array, and we propose a method to generalize the Parity Logging scheme, making it applicable to the Reed-Solomon coded RAID6 array. Second, we propose a log-assisted RAID6 architecture, RAID6L. RAID6L greatly improves the write performance of the RAID6 array, at the expense of minimal reliability losses. Moreover, through detailed comparisons, we show that RAID6L is more advantageous over Parity Logging.

Acknowledgment

This work is supported by the National Basic Research 973 Program of China under Grant No. 2011CB302301; 863 Project 2009AA01A401 and 2009AA01A402; NSFC No. 61025008, 60933002, 60873028, 60703046; Changjiang innovative group of Education of China No. IRT0725; the Fundamental Research Funds for the Central Universities, HUST, under Grant 2010MS043; and the US NSF Grant IIS-0916859, CCF-0937993, CNS-1016609.

References

[1] Plank J S. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. Software Practice and Experience, 1997, 27(9):995-1012.

[2] Jin C, Jiang H, Feng D, Tian L. P-Code: A New RAID6 Code with Optimal Properties. In Proc. of ICS'09, 2009.

[3] Stodolsky D, Gibson G, Holland M. Parity Logging: Overcoming the Small Write Problem in Redundant Disk Arrays. In Proc. of ISCA'93, 1993.

[4] Jin C, Feng D, Jiang H, Tian L, Liu J, Ge X. TRIP: Tempormal Redundancy Integrated Performance Booster for Parity Based RAID Systems. In Proc. of ICPADS'10. 2010.

[5] Yang Q, Xiao W, Jin R. TRAP-Array: A Disk Array Architecture Providing Timely Recovery to Any Point-in time. In Proc. of ISCA'06, 2006.

[6] Narayanan D, Donnelly A, Rowstron A. Write Off-Loading: Practical Power Management for Enterprise Storage. In Proc. of FAST'08, 2008.

[7] UMass Trace Repository. *http://traces.cs.umass.edu/index.php*.

[8] S. I. Repository. *http://iotta.snia.org/traces/*.

[9] Hu J, Jiang H, Tian L, Xu L. PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD. In Proc. of MASCOTS'10, 2010.

[10] Tian L, Feng D, Jiang H, Zhou K, et al. PRO: A Popularity-based Multi-threaded Reconstruction Optimization for RAID-Structured Storage Systems. In Proc. of FAST'07, 2007.