LETTER

Parity Data De-Duplication in All Flash Array-Based OpenStack Cloud Block Storage*

Huiseong HEO[†a)], ***Member***, **Cheongjin AHN**[†b)], ***and*** **Deok-Hwan KIM**[†c)], ***Nonmembers***

SUMMARY In recent years, the need to build solid state drive (SSD)-based cloud storage systems has been increasing in order to process the big data generated by lots of Internet of Things devices and Internet users. Because these kinds of cloud systems require high performance and reliable storage, the use of flash-based Redundant Array of Independent Disks (RAID) will increase. But in flash-based RAID storage, parity data must be updated with every data write operation, which can more quickly overwhelm SSD's lifespan. To solve this problem, this letter proposes parity data deduplication for OpenStack cloud storage systems using an all flash array. Unlike the traditional data deduplication method, it only removes parity data, which will be stored in the parity disks of the all flash array. Experiments show that the proposed parity data deduplication method can efficiently reduce the number of parity data write operations, compared to the traditional data deduplication method.
key words: *parity data deduplication, RAID, OpenStack, cloud storage, solid state drives*

1. Introduction

Recently, in order to store and process big data generated from lots of Internet of Things devices and Internet users, the need to build cloud storage with high-performance computing power has been increasing [1]. One way to enhance I/O performance and reliability of the storage system is with Redundant Array of Independent Disks (RAID) [2]. To build high-performance cloud storage, NAND flash memory-based solid state drives (SSDs) are mainly used [3]. An SSD can handle data much faster than a hard disk drive (HDD), although it incurs greater cost per unit of capacity than an HDD and can not be used when it reaches its program/erase (P/E) cycle limit [4]. These problems may deteriorate both I/O performance and the reliability of SSD RAID-based cloud storage systems, and if there are many small write operations in SSD-based RAID, data and parity must be updated continuously, the SSDs will wear out more quickly. To solve this problem and so increase SSD lipespan, we can employ data deduplication, which reduces the number of write operations by eliminating duplicated data. But when we apply data deduplication to the entire RAID disk, we must make fingerprints of all input data. This causes high CPU overhead and requires greater DRAM capacity for hash tables.

In this letter, we propose a new parity data deduplication process for OpenStack cloud block storage, Cinder, which is composed of an all flash array. We show that the proposed method can efficiently reduce the number of parity data write operations and DRAM usage for hash tables better than the existing alternative.

2. Background Knowledge

2.1 OpenStack Cloud Block Storage

OpenStack is composed of Nova, Glance, Swift, Cinder and some other components [5]. To provide cloud services, these components interact with each other through the management network. Figure 1 shows OpenStack framework. In this figure, Nova can determine which host will launch the virtual machine, and then the selected host creates a virtual machine. After that, Nova processes computing tasks for virtualized resources like a CPU, DRAM and storage.

Cinder, which provides block storage services, combines many disks into a volume group by means of logical volume manager (LVM) and then uses it as a large block disk. If a virtual machine needs additional block storage, a cloud manager can assign it to the virtual machine by using Cinder.

2.2 Data Deduplication

Data deduplication methods generate a hash value for every input chunk and try to find the same hash value in the

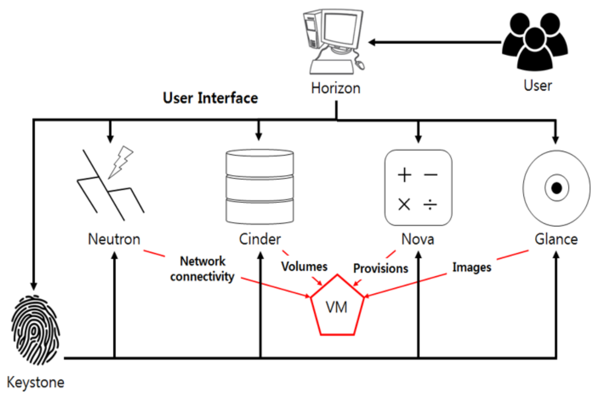

Fig. 1 OpenStack framework

Manuscript received January 8, 2016.
Manuscript publicized February 2, 2016.
[†]The authors are with the Department of Electronic Engineering, Inha University, Korea.
*This study was supported in part by Inha University Research Grant.
a) E-mail: hhs89@iesl.inha.ac.kr
b) E-mail: acj88@iesl.inha.ac.kr
c) E-mail: deokhwan@inha.ac.kr (Corresponding author)
DOI: 10.1587/transinf.2016EDL8006

hash table. If the same hash value is found in the table, redundant copies of the chunk are replaced with a reference to a single copy to save disk space. This data deduplication method can be classified as inline data deduplication or post-process data deduplication according to the period when deduplication happens [6]. For all flash array storage systems, an inline data deduplication method can be used to reduce wear-out by decreasing the number of write operations.

3. Parity Data Deduplication in All Flash Array Based Cloud Storage

In this letter, we propose a new parity data deduplication method in OpenStack cloud block storage using an all flash array.

3.1 Parity Data Deduplication

When write operations are generated in a RAID disk, data is saved to data disks and then parity is stored in the parity disk. Parity data are generated as a result of read-modify-write operations because parity data should be updated for every small data write operation, and the parity disk can be a bottleneck in RAID storage.

Existing RAID uses distributed parity, as shown in Fig. 2. It prevents RAID from converging intensive write operations into two parity disks due to continuous parity updates [7]. To reduce the number of write operations into disk, data deduplication can be applied. But in existing RAID, each disk saves data and parity data together, so data deduplication should be applied to all RAID disks. In this case, the data deduplication technique needs to process a number of CPU operations for hashing and also needs greater DRAM space to keep the hash table. Furthermore, it requires CPU and DRAM resources in proportion to the number of disk. So, these problems especially stand out in a cloud system because a cloud system should have a large storage cluster to provide the storage users might request.

So, to efficiently reduce the number of write operations to RAID storage, we suggest a new parity data deduplication method.

Table 1 and Fig. 3 show the pseudo-code of the parity data deduplication algorithm and the parity data deduplication scheme, respectively. In Table 1, Generate_parity() is the function that generates parity data using Reed Solomon code, even-odd code, etc. And Generate_hash_value() is the function that generates a hash value using SHA-1, MD5, etc.

Because intensive parity data writing can cause a bottleneck for an all flash array disk, we need to remove the duplicated parity data by using inline deduplication. When we apply the proposed parity data deduplication to an all flash array, it can efficiently reduce hashing overhead and DRAM usage, in comparison with the existing data deduplication because it only calculates a hashkey for parity data that may cause a bottleneck, not for the overall RAID storage. In addition, by minimizing the size of the parity chunk for hashing, rather than the RAID chunk size, it is possible to minimize parity data that is actually stored in the par-

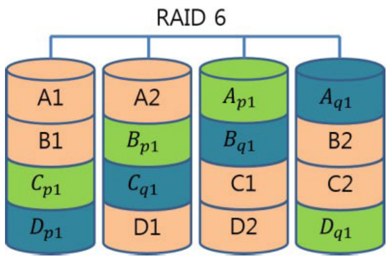

Fig. 2 RAID-6 Storage using distributed parity

Table 1 Pseudo-code of the parity data deduplication algorithm

```
k : The number of Data disks
m : The number of Parity disks
F : Workload size
n : RAID chunk size
c : Chunk size for generating hash value
β = [F / (k * n)] + 1 : The number of iteration
CH_new : New data array
CH_old : Old data array
Parity : Parity data array
                    /*   Read-Modify operation   */
1.  for (j=1; j ≤ β; j++)
2.      for (i=1; i ≤ k; i++)
3.          read    New data from host and allocate it to CH_new[j][i]
4.          read    Old data from disks and allocate it to CH_old[j][i]
5.      end for
6.      for (i=1; i ≤ m; i++)
7.          read    Parity data from disks and allocate it to Parity[j][i]
8.          Parity[j][i] = Generate_parity(CH_new[j][i], CH_old[j][i],
                                                      Parity[j][i])
9.      end for
                    /*   Write new data to data disks   */
10.     for (i=1; i ≤ k; i++)
11.         write    CH_new[j][i]    into    data disks
12.     end for
                    /*   Parity data deduplication   */
13.     for (i=1; i≤m; i++)
14.         for (l=1; l≤ n/c; l++)
15.             hashkey =Generate_hash_key(Parity[j][i][l])
16.             if ( hashkey does not exist in hashtable )
17.                 add      hashkey into hashtable
18.                 write    Parity[j][i][l] into parity disk
19.             else if ( hashkey exist in hashtable )
20.                 write    Reference pointer for original chunk to
                                                      parity disk
21.             end if
22.         end for
23.     end for
24. end for
```

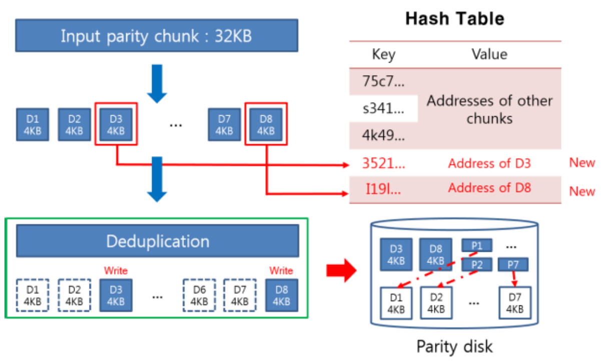

Fig. 3 Parity data deduplication scheme

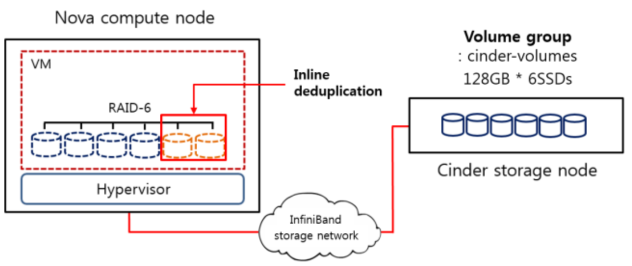

Fig. 4 Parity data deduplication in cloud block storage

ity disk in cases where it is rarely changed. And the proposed method does not decrease reliability of all flash array disk. Although our proposed deduplication scheme removes duplicated parity chunks, it creates small reference pointers pointing original chunks (*e.g.*, P1, P2, . . . , P7 in Fig. 3) and in case of one or two disk failure, these pointers help the storage system recover data by referring the original chunks.

3.2 Parity Data Deduplication in Cloud Block Storage

OpenStack Cinder provides virtualized block storage to the virtual machine generated by OpenStack Nova. When we apply the proposed parity data deduplication to the OpenStack block storage system, we can reduce the number of write operations for the parity disk, and this may help to increase the lifespan of an all flash array.

Figure 4 illustrates an OpenStack cloud block storage system applying the proposed parity data deduplication. When there are write operations to an all flash array, only parity data will be removed by deduplication.

4. Experimental Result

4.1 Experimental Environment

In this letter, we assigned a total of six virtualized SSDs from a block storage node to the virtual machine and simulated a RAID-6 disk using Jerasure code software [8]. For detailed experiments, we used Samsung 840 PRO 128GB SSDs, and the OpenStack Juno version to build the cloud storage service, and the hypervisor kernel-based virtual machine (KVM) to virtualize servers, networks and storage. Table 2 shows the specifications for two nodes of nova and cinder, the virtual machine generated in Nova. We repeated five times for measurement, but only show the average of the results.

We compared write performance, the number of write operations, decoding time and DRAM usage results of the proposed method with those of the traditional RAID-6 and of traditional RAID-6 using data deduplication. In all cases, Reed Solomon code and SPC were used for RAID-6 parity computation. As a workload for the experiment, a compressed 2.19GB tar file comprising of Linux kernel version 2.6.0~2.6.39 files was downloaded from a web site [9] and executed. The chunk size for the hashkey was set to 4KB.

Table 2 Experimental environment

	Nova / Cinder Node	Virtual Machine
OS	Ubuntu Server 14.10 LTS	Ubuntu Server 14.04 LTS
CPU	Intel Xeon E5-2620@2.0GHz	4 vCPUs
RAM	DDR3 32GB	DDR3 8GB
OpenStack	Juno version	-

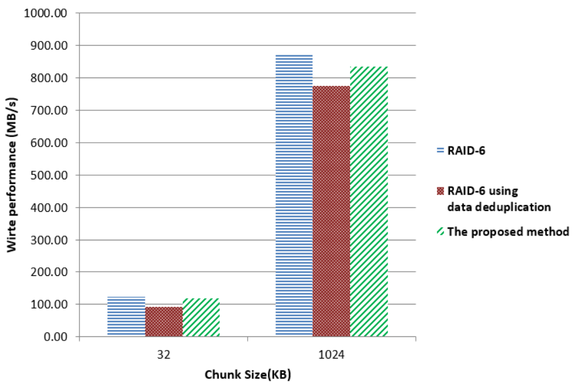

Fig. 5 Write performance

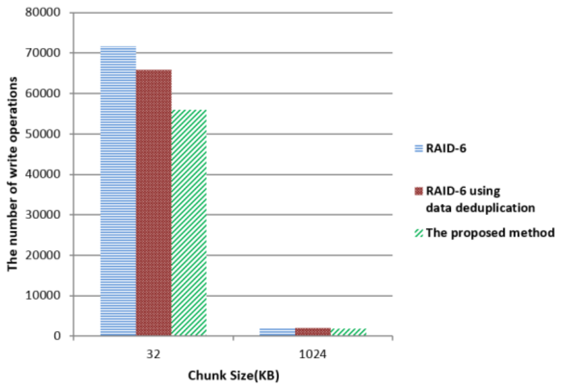

Fig. 6 The number of write operations to the parity disk

4.2 Experimental Result and Discussion

Figure 5 shows the write performance of the traditional RAID-6, traditional RAID-6 using data deduplication and the proposed parity data deduplication. For each experiment, the RAID chunk size is set to 32KB and 1024KB.

Figure 6 shows the results for the number of write operations for the three cases. Compared to RAID-6, RAID-6 using data deduplication shows 75% and 89% of the write performance when the chunk size is 32KB and 1024KB, respectively. But the proposed parity deduplication method shows 95% and 96% of the write performance in the same environment. This is because the traditional RAID-6 using data deduplication must process more hash operations than the proposed method.

In terms of the number of write operations to the parity

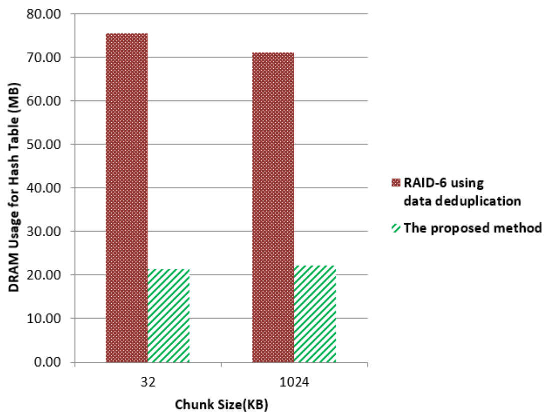

Fig. 7 DRAM usage for hash table

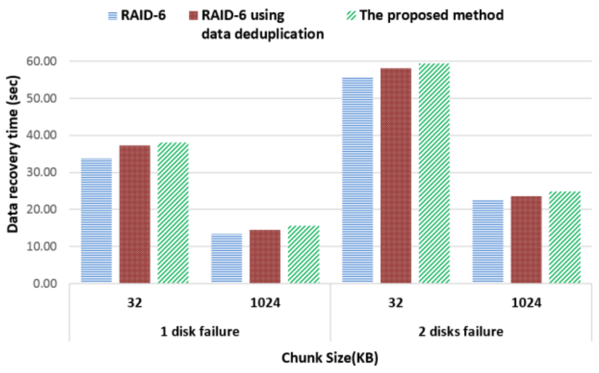

Fig. 8 Data recovery time

disk, for the same chunk sizes, RAID-6 using data deduplication shows 8% and 13% fewer write operations, respectively, than RAID-6, whereas the proposed method shows 22% and 19% fewer write operations than RAID-6. That is because the proposed method only removes duplicated parity chunks that are frequently updated, whereas traditional RAID-6 using data deduplication saves data and parity data over many disks, which decreases the efficiency of deduplication per disk.

Figure 7 shows DRAM usage for hash tables and Fig. 8 shows decoding time for restoring data.

In the proposed method, DRAM usage for hash tables are 72% and 69% less than RAID-6 using data deduplication. This is because the proposed method only keeps hash values for parity chunks in the hash table, but the traditional method keeps hash values of all input data chunks and parity chunks. We found that RAID-6 using data deduplication uses DRAM capacity as much as 3.5% of the workload size, whereas the proposed method uses 1% of the workload size.

In Fig. 8, traditional RAID-6 consumes the smallest decoding time in all cases. The result shows that in case of 1 disk failure, RAID-6 using data deduplication spent 10% and 8% more time than RAID-6, respectively. While the proposed method spent 13%, 16% more time than existing RAID-6. And in case of two disks failure, RAID-6 using data deduplication spent 4% and 5% more time than RAID-6, respectively. Whereas the proposed method spent 7%, 11% more time than traditional RAID-6.

The reason for the gap between RAID-6 and the proposed method is because the number of chunks that should be referred in the case of parity deduplication is increased. Because the proposed parity data deduplication removes more chunks than RAID-6 using data deduplication, it needs more time to refer the original chunks. But surely, we could recover all original data without any loss.

5. Conclusion

In this letter, we proposed a novel parity data deduplication method for cloud block storage systems. The proposed method can efficiently reduce the number of write operations, requiring less DRAM usage than the traditional method, and also minimizes deterioration of write performance by reducing the number of hash operations. Our experiments showed that the proposed method reduces the number of parity data write operations by 22% and 19% than RAID-6, in case for 32KB and 1024KB chunk sizes, respectively.

References

[1] J. Wu, L. Ping, X. Ge, Y. Wang, and J. Fu, “Cloud Storage as the Infrastructure of Cloud Computing,” 2010 International Conference on Intelligent Computing and Cognitive Informatics, pp.380–383, 2010.
[2] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson, “RAID: High-performance, reliable secondary storage,” ACM Computing Surveys(CSUR), vol.26, no.2, pp.145–185, June 1994.
[3] M. Pirahandeh and D.-H. Kim, “Energy-aware erasure codes using XOR Reference Matrix for SSD based RAID systems,” 2014 International Conference on Big Data and Smart Computing (BIGCOMP), pp.121–122, Bangkok, 2014.
[4] N. Agrawal et al., “Design tradeoffs for SSD performance,” USENIX Annual Technical Conference, pp.57–70, 2008.
[5] “OpenStack,” http://www.openstack.org/
[6] Q. He, Z. Li, and X. Zhang, “Data deduplication techniques,” 2010 International Conference on Future Information Technology and Management Engineering, pp.430–433, Oct. 2010.
[7] A. Merchant and P.S. Yu, “Analytic modeling of clustered RAID with mapping based on nearly random permutation,” IEEE Trans. Comput., vol.45, no.3, pp.367–373, March 1996.
[8] J.S. Plank, S. Simmerman, and C.D. Schuman, “Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2,” Technical Report CS-08-627, University of Tennessee, Knoxville, USA, Aug. 2008.
[9] “The Linux Kernel Archives,” https://www.kernel.org/