International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:6, No:5, 2012

File System-Based Data Protection Approach

Jaechun No

Abstract**—**As data to be stored in storage subsystems tremendously increases, data protection techniques have become more important than ever, to provide data availability and reliability. In this paper, we present the file system-based data protection (WOWSnap) that has been implemented using WORM (Write-Once-Read-Many) scheme. In the WOWSnap, once WORM files have been created, only the privileged read requests to them are allowed to protect data against any intentional/accidental intrusions. Furthermore, all WORM files are related to their protection cycle that is a time period during which WORM files should securely be protected. Once their protection cycle is expired, the WORM files are automatically moved to the general-purpose data section without any user interference. This prevents the WORM data section from being consumed by unnecessary files. We evaluated the performance of WOWSnap on Linux cluster.

Keywords**—**Data protection, Protection cycle, WORM

I. Introduction

As data to be stored in storage subsystems tremendously increases, data protection techniques have become more important than ever, to provide data availability and reliability. A simple way to protect data securely is to store them in a read-only CD (Compact Disk) or OD (Optical Disk). However, those storage media can be lost and even be broken, thus storing sensitive data in such devices cannot guarantee data integrity and availability. Another way is to have a separate data storage device where only those sensitive data are stored and then to permit only privileged read requests. Unfortunately, this method causes high price to provide such storage devices.

To overcome these problems, some file systems use WORM [1-3,15] to integrate data protection with file operations. In the WORM method, once files have been created, those are used for read requests, rejecting the requests for data modification or deletion. However, the existing WORM method suffers from three major drawbacks.

The first problem is that it requires to maintain a separate disk device to enforce data security, resulting in the burden of buying additional disk devices. The second problem is that, once they have been created, the WORM files are considered to be protected permanently. However, a large amount of WORM files have their own WORM protection cycle that is a time period during which the WORM file should securely be protected. When the protection cycle is expired, the WORM files should be treated as an ordinary file, by releasing the associated WORM characteristics.

Jaechun No is with the College of Electronics and Information Engineering, Sejong University, 98 Gunja-dong, Gwangjin-gu, Seoul, Korea (phone: +82-2-3408-3747; fax: +82-2-3408-4321; e-mail: jano@ sejong.ac.kr).

In this paper, we present file system-based data protection mechanism, which is called WOWSnap (WORM-based Snapshot). One of our objectives in developing the WOWSnap is to provide a secure access process for WORM files. Moreover, we attempt to reduce the cost of data protection, by not using complicated cryptographic operations. Our second objective is to optimize the disk space by storing both the WORM files and the general-purpose files together, even though in the different disk section. In the WOWSnap, the disk partition to be allocated for file system is divided into two sections: one for storing WORM files, and the other for storing general-purpose files. The files to be allocated in the WORM disk section are written only once at the file creation time. Once the WORM files have been created, only the privileged read requests to them are allowed, to protect data against any malicious intrusions.

All WORM files in WOWSnap are associated with their protection cycle. The protection cycle would be a finite value if the associated WORM file needs to be protected for a certain time. To optimize the disk space for the WORM files, the WORM daemon periodically checks files to see if there may exist the WORM files whose protection cycle has already been expired. If so, the daemon moves those files to the general-purpose disk partition, to reuse the occupied WORM disk space.

In the remaining part of this paper, we describe our snapshot mechanism integrated with the WOWSnap. There are several file systems supporting journaling [12-14] or snapshot [10,11], to enhance reliability and availability. For example, WAFL[9] provides a read-only snapshot to protect data blocks. In WAFL, all data blocks are organized in a tree, and snapshot is performed by duplicating the root node. Since WAFL snapshot was implemented based on cow(copy-on-write), update operations cause the modified block and its parents to be copied to a backup area. Another example is Ext3cow[6,7,8] that has been implemented on top of Ext3, to support file system snapshot and individual file versioning by using time-shifting interface. Like WAFL, the data modification in Ext3cow is performed based on cow, allocating new data blocks to hold the modified data while preserving a copy of old block.

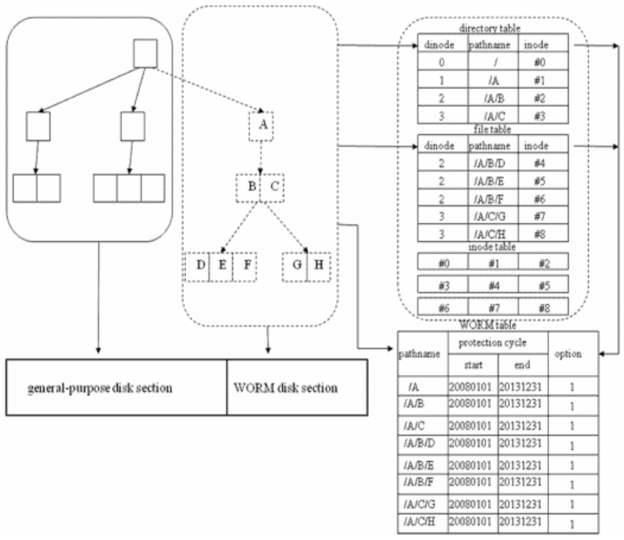
Unlike WAFL or Ext3cow, the snapshot of WOWSnap has been built based on row(redirect-on-write) because row-based snapshot does not incur the copy cost to preserve the old data. Besides, by pre-allocating inode and data blocks for the following snapshot image, we attempted to minimize I/O processing overhead to take a picture. Furthermore, by using the snapshot spatial algorithm implemented in the WOWSnap, we tried to maintain the disk space allocated for snapshot images as small as possible.

The rest of this paper is organized as follows: Section II and III describe the implementation detail of the WORM and snapshot structures of the WOWSnap, respectively. In Section IV, the performance evaluations obtained on the Linux cluster are shown. We conclude in Section V.

II. WORM Structure

A. Overview

Fig. 1 illustrates the WORM structure of the WOWSnap. The disk space where the WOWSnap is mounted is divided into two sections: general-purpose disk section and WORM disk section. User could either install both sections on the different partition of the same disk, or could use different disks for two sections. As a future work, we plan to use non-volatile memory, such as SSD (Solid-State Disk), for the WORM section because we expect that the high-speed read performance of SSD may contribute to provide a fast service for accessing the WORM files. In the WOWSnap, besides creating WORM files to protect data, users can also protect general files by converting them to WORM data. Once they are converted to WORM, these files are hidden to unauthorized users by make them invisible. This is performed by skipping the WORM directory entry traversal when authorized users execute the pathname lookup operations. The WORM file creation is performed by calling *worm_commit*. This system call is also used to convert general directories or files to WORMs. In case of converting a directory to WORM, all the directories and files in the sub-level are also converted to WORMs and are moved to the WORM disk section. In such a case, the corresponding data blocks are eliminated from the general disk section, but the hierarchical pathname in the general disk section is stored in database. This information is used to restore the data to the general disk section, in case of expiration of the WORM protection cycle.



directory table

dinode	pathname	inode
0	/	#0
1	/A	#1
2	/A/B	#2
3	/A/C	#3

file table

dinode	pathname	inode
2	/A/B/D	#4
2	/A/B/E	#5
2	/A/B/F	#6
3	/A/C/G	#7
3	/A/C/H	#8

inode table

#0	#1	#2
#3	#4	#5
#6	#7	#8

WORM table

pathname	protection cycle		option
	start	end	
/A	20080101	20131231	1
/A/B	20080101	20131231	1
/A/C	20080101	20131231	1
/A/B/D	20080101	20131231	1
/A/B/E	20080101	20131231	1
/A/B/F	20080101	20131231	1
/A/C/G	20080101	20131231	1
/A/C/H	20080101	20131231	1

Fig. 1 WORM structure

Fig. 1 shows the directory table, file table, inode table and WORM table to maintain the hierarchy. All the attributes related to WORM are inserted in the WORM table. The WORM protection cycle is a protection time period of the WORM file. If the WORM file is supposed to be protected permanently, then the protection cycle is set to zero. The option flag is used to represent how to handle the associated WORM file, in case of the expiration of its protection cycle. If the value is set to 1, then the associated WORM file becomes a general file, while being moved to the general disk section by using the hierarchical pathname in database. Otherwise, the file will either be permanently deleted with the value of 2, or will be backed up with the value of 3. When a directory located at the middle level is converted to WORM, this directory is re-installed in the root directory of the WORM section because its ancestors should not be moved to the WORM section.

On the expiration of the protection cycle, a WORM file or directory is moved to the general disk section, to save the costly WORM disk space. In such a case, all the files and directories in the sub-level are also moved to the general disk section. Also, by using the hierarchical pathname stored in database, its original hierarchy in the general section will be restored. If a WORM file does not have the original hierarchical pathname in the general section because it has been created as WORM, then it is moved to the root directory of the general disk section.

B. Protection Cycle

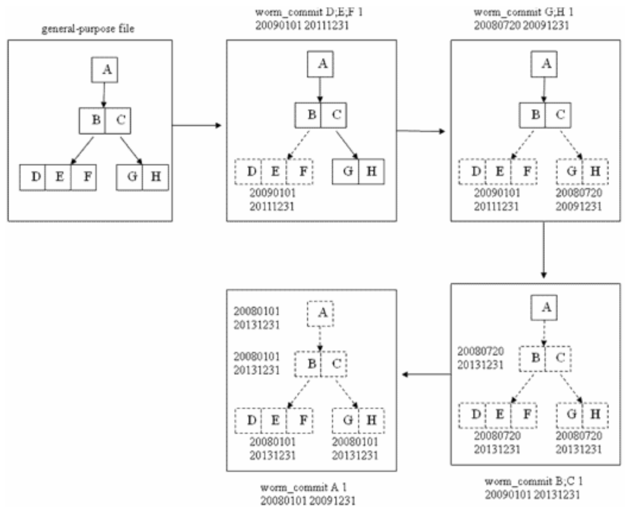


Fig. 2 WORM protection cycle

The WORM protection cycle is given as input when performing *worm_commit*. The cycle value of a directory affects the cycle value of its successors. Each file in a directory can be given the different cycle value. When a certain cycle value is given to its parent, the WOWSnap first compares the new cycle value with all cycle values of its child files. The earliest starting date and the latest ending date among the parent and its descendents are chosen and assigned to them as the new protection cycle. By doing this way, a directory and its descendents in the hierarchy can have the same WORM characteristics, and thus the cost of managing the WORM structure can significantly be minimized.

Fig. 2 illustrates an example of propagating the protection cycle. When the first *worm_commit* is performed with the cycle value, the files D, E and F take the arguments and the WORM attributes are inserted into database. With the second *worm_commit* call, the files G and H also take the input arguments without any modification. However, when the third *worm_commit* is called to convert B and C to WORMs, with the cycle value of 20090101 and 20131231, the WOWSnap chooses the earliest starting date and the latest ending date among the input and the cycle value of descendents. Then the chosen value, 20080720 and 20131231, are assigned as the new protection cycle and the WORM attributes in database are modified to reflect this change.

III. Snapshot Structure

A. Overview

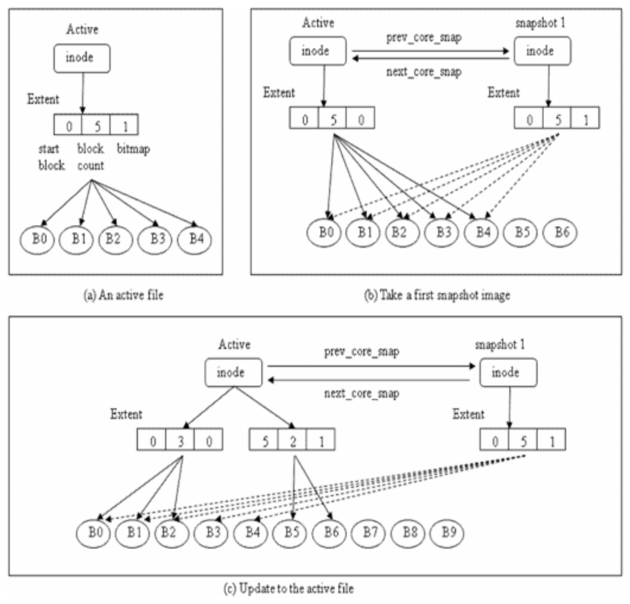


Fig. 3 An overview of snapshot structure

We implemented the snapshot structure in the WOWSnap, to protect files against data crash. Since the WOWSnap divides the disk space into two sections based on the protection level, the snapshot policy for two sections can be setup differently. For example, if one of user's goals is to save the WORM section, then user can limit the number of snapshot images for each WORM file to be one, while the number of snapshots for the general files to be a larger value than one.

To group snapshot images for an active file, the WOWSnap uses two pointers, *prev_core_snap* and *next_core_snap*. By using these two pointers, the snapshot daemon can traverse the history link to verify snapshot integrity. The file linked by the *prev_core_snap* is a preceding snapshot image, and the file linked with the *next_core_snap* is either a following snapshot image or an active file.

When the inode of an active file is created, an additional inode is also pre-allocated and is linked with the active inode using *prev_snap_ino*. If the next snapshot is taken, then there is no need to allocate and to replicate the inode of the active file because duplicating the inode attributes has already been performed when the extra inode has been pre-allocated. The only thing to be performed at that moment is to adjust two pointers, *prev_core_snap* and *next_core_snap*, and to setup the bitmap value of the extent to indicate the sharing of data blocks. In the WOWSnap, the snapshot daemon periodically issues a snapshot call to check snapshot integrity.

The WOWSnap assigns a bitmap value to each extent structure to indicate the sharing of data blocks, as shown in Fig. 3(a). If the bitmap value is zero, it then means that the data blocks of the extent cannot be modified because those blocks must be shared with other files. Otherwise, the blocks of the extent can be modified. Fig. 3(a) shows an example of the snapshot structure. The file is composed of five data blocks, *B*0 through *B*4, and its extent includes three components: the starting block number, block count, and bitmap value. The initial value of the bitmap value is one, meaning that no other file currently shares the data blocks belonging to this file.

Fig. 3(b) illustrates the steps involved in taking the first snapshot. The file located at the left side denotes an active file and the file at the right side denotes its point-in-time snapshot image. As can be seen in the figure, the bitmap value of the active file is changed from one to zero because the data blocks of the active file are shared with the first snapshot.

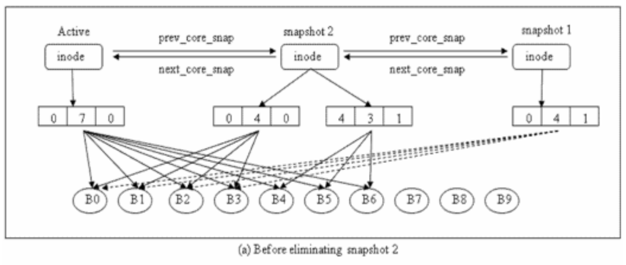
Changing the bitmap value has significant performance impact on data modification and deletion, because by checking the current bitmap value, the WOWSnap can easily determine whether the data blocks can be updated, or not. If the WOWSnap finds out that the data blocks cannot be modified due to the sharing among multiple files, then it allocates new data blocks to receive the up-to-date value. Also, the extent associated with the data blocks is split to separate the new blocks. Fig. 3(c) shows an example where update process is occurred after the first snapshot has been taken.

In Fig. 3(c), the update requires two blocks, *B*3 and *B*4, to be modified. Since these two blocks are shared between the active file and the first snapshot, two new blocks, *B*5 and *B*6, are allocated and the update process is performed on these two blocks. Also, to reflect the new block allocation for update, an additional extent is created and its bitmap value is initially set to one, because no other file currently shares these new blocks with the active file.

B. Snapshot Spatial Algorithm

In the WOWSnap, we provide a snapshot spatial algorithm for consistently maintaining the snapshot history. The spatial algorithm is periodically performed by the snapshot daemon that is responsible for deleting snapshot images that have been corrupted or that have been backup-ed to other disk. Fig. 4(a) and (b) show the steps involved in the snapshot spatial algorithm. Fig. 4(a) shows a snapshot overview before eliminating *snapshot 2*.

In the figure, the first snapshot image, *snapshot 1*, includes an extent indicating that four data blocks, *B*0 to *B*3, are allocated to this image. The second snapshot image, *snapshot 2*, takes over four blocks from *snapshot 1*, while changing the bitmap value from one to zero. It is noted that when *snapshot 2* was the active file, there existed a write operation requesting three new data blocks, *B*4 to *B*6, to be allocated. At that moment, the WOWSnap allocated a new extent for these blocks and assigned the bitmap value to one because no block sharing happened yet.



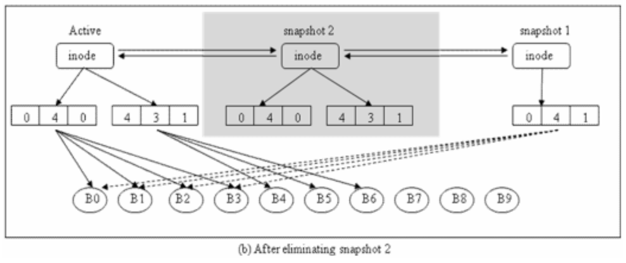

Active
snapshot 2
snapshot 1
inode
0 4 0
4 3 1
0 4 0
4 3 1
0 4 1
B0 B1 B2 B3 B4 B5 B6 B7 B8 B9
(b) After eliminating snapshot 2

Fig. 4 Snapshot spatial algorithm

Suppose that the snapshot daemon finds out that the second snapshot image, *snapshot 2*, has been corrupted. The elimination process of the spatial algorithm requires the daemon to traverse *prev_core_snap* and *next_core_snap*, to determine if the data blocks belonging to *snapshot 2* can be de-allocated. With the bitmap value of zero, the blocks belonging to this extent cannot be modified since these blocks are shared with other files linked by the *prev_core_snap*. On the other hand, with the bitmap value of one, the blocks belonging to this extent are not shared with the preceding snapshots. However, these data blocks might be shared with other following snapshots linked to the *next_core_snap*. Therefore, the snapshot images linked to the *next_core_snap* should be checked before de-allocating the data blocks belonging to *snapshot 2*.

In Fig. 4(b), *snapshot 2* has two extents to manage the associated data blocks. Since the bitmap value of the first extent is zero, the corresponding data blocks cannot be de-allocated, thus the spatial algorithm simply unlinks the pointers from those data blocks. The bitmap value of the second extent is one, meaning that the preceding snapshot connected by the *prev_core_snap* has not shared the data blocks with *snapshot 2*. However, since these blocks might be shared with the active file, before de-allocating the blocks, the spatial algorithm should check the bitmap value of the active file.

Because the active file includes the data blocks being shared with *snapshot 2*, the spatial algorithm splits the extent into two parts, to separate those blocks. The bitmap value of the first split extent is set to zero because the associated data blocks, *B*0 to *B*3, are shared with the first snapshot, *snapshot 1*. On the other hand, the bitmap value of the second split extent is set to one, because, after eliminating *snapshot 2*, no other file shares the associated data blocks, *B*4 to *B*6, with the active file.

IV. Performance Evaluation

A. WORM Structure

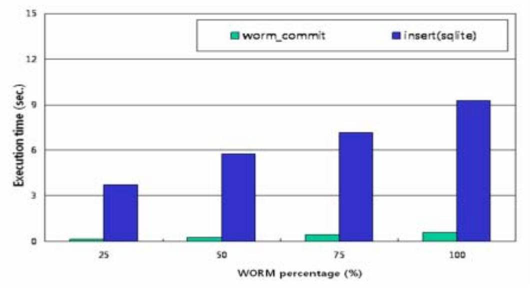


Fig. 5 WORM overhead

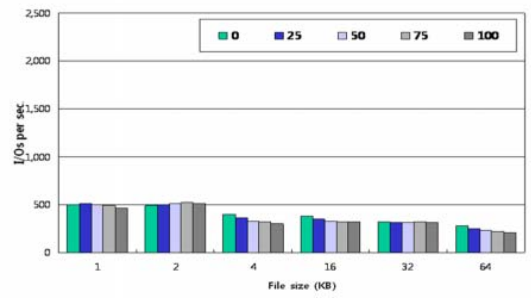


Fig. 6 Create performance as a function of WORM files

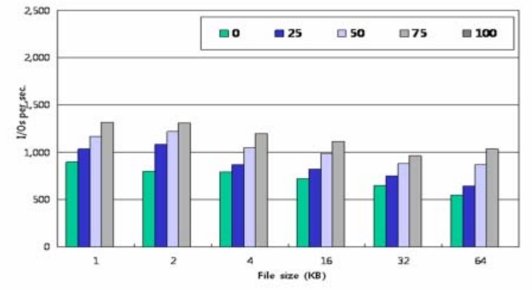


Fig. 7 Write performance as a function of WORM files

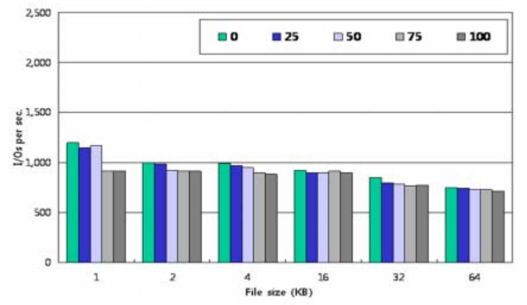


Fig. 8 Read performances as a function of WORM files

For the WORM evaluation, we equally divided HDD into two partitions and used one of them as the WORM partition. We used SQLite to store the WORM attributes. Fig. 5 shows the WORM overhead for calling *worm_commit* and inserting WORM attributes to database(sqlite), while changing the number of WORM files from 25,000 to 100,000. As can be seen, the overhead for calling *worm_commit* is very small, compared to the overhead for inserting the WORM attributes. However, most of execution time is consumed by IO. This fact is clear in Fig. 6 to 8. Fig. 6 to 8 show I/O performance which obtained using 100,000 files, with a size range of 1Kbytes to 64Kbytes. Also, we changed the number of WORM files among them to be 25%, 50%, 75%, and 100% of total number of files.

In case of performing create operations, as shown in Fig. 6, increasing the number of WORM files from 0 to 100,000 slightly affects I/O performance, because creating WORM files causes the access to database to insert the associated WORM attributes. However, this WORM overhead does not significantly lower the create bandwidth. On the other hand, in Fig. 7, as the percentage of WORM files increases, we can see that the write bandwidth becomes large, because once the WORM files have been created, they are not allowed to be modified. Therefore, with 25% of WORM frequency, only 75,000 of general files are written. The extreme case is with 100% of WORM frequency in which no write operation is executed.

In case of read operations, as can be shown in Fig. 8, changing the number of WORM files has little effect on I/O performance. This is because the WORM attributes are rarely used in the read operation, except for checking the authorization to the WORM file.

B. Snapshot Structure

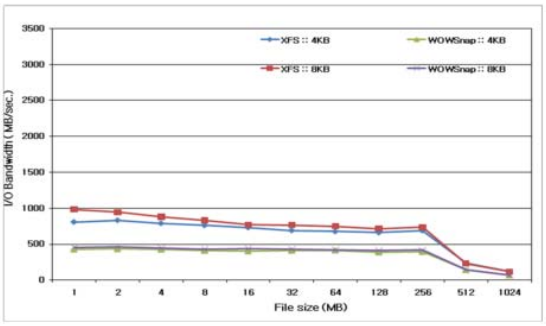


Fig. 9 Write performance compared to XFS without snapshot

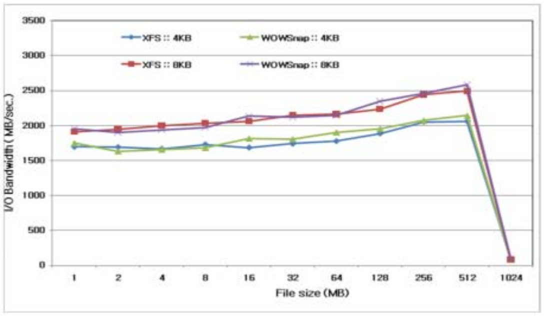


Fig. 10 Read performances compared to XFS without snapshot

Fig. 9 to 10 show the WOWSnap I/O performance with snapshot capability. Also, we compared the performance with XFS[4,5] that does not have snapshot, to find out the effect of snapshot overhead. It is noted that there exists a tradeoff between performance and reliability in using snapshot. We used IOzone benchmark to measure I/O performance.

Fig. 9 shows the performance of the WOWSnap write operation, while varying the record unit for each file size. Also, we configured the WOWSnap to take a snapshot every one second. In this case, changing the record size for each file size does not lower the write performance as much as we expected. This is because, at each snapshot checkpoint, only the file metadata would be duplicated. Fig. 9 also shows the effect of snapshot overhead in the WOWSnap write operations, by comparing I/O performance to that of XFS without snapshot. In case of write operations, the impact of the overhead is obvious, especially in file sizes smaller than 256Mbytes. With small file-sizes, the larger number of snapshot-related metadata operations, such as pre-allocating snapshot inode, would occur than with large file-sizes, thus causing the performance decrement.

Fig. 10 shows the performance of the WOWSnap read operations, with the impact of memory cache. Unlike in write operations, we can see that using large record size in read operations generate better I/O performance, due to the data coalesce. Also, memory shortage might cause the sharp performance downgrade, with file sizes larger than 512Mbytes. When comparing to XFS read performance, we could see little impact of snapshot overhead.

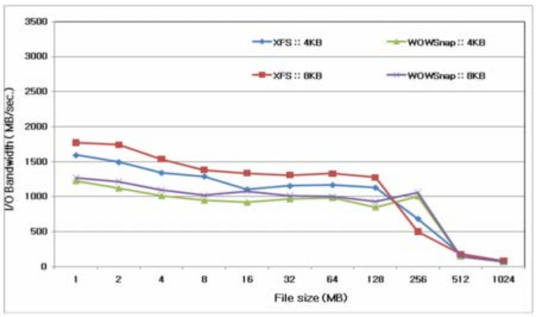


Fig. 11 Rewrite performance compared to XFS without snapshot

Fig. 11 shows the performance of the WOWSnap re-write operations combined with snapshot. In this measurement, after creating each file, the data of record size are re-written in the file. Since the metadata operations for writing files, such as bitmap and block allocation, have been performed before the measurement, the performance of re-write operations is higher than that of write operations in most cases. Also, based on the Figure, we can see that varying the record size does not much affect the performance.

In order to see the snapshot overhead in modifying data between checkpoints, we compared the WOWSnap re-write performance to XFS. In re-write operations, snapshot incurs the related metadata operations, such as inode pre-allocation, snapshot-associated pointer setup, and new block allocation if data modification occurs within the checkpoint. As shown in write operations, with the small file-size, the overhead to create snapshots overwhelms the entire re-write performance because of the large number of snapshots being taken. However, with the file size larger than 128Mbytes, we can see that such a snapshot-related overhead becomes small, thereby the WOWSnap produces better performance than XFS. In this case, even if new block allocations are needed due to data modification, it does not lower the WOWSnap performance.

V. Conclusion

We presented the WORM and snapshot structures of the WOWSnap. These structures have been implemented to enhance data availability and reliability. Unlike other WORM structures, the WOWSnap is capable of protecting WORM files for a finite time period, by assigning the protection cycle to each file. Instead of unnecessarily taking space of the costly WORM disk section, as soon as the protection cycle is expired, the associated WORM file can automatically be moved to general disk section without user interference. We also described the snapshot structure of the WOWSnap that has been implemented based on file system-based method and row approach. The snapshot performance results show that the impact of the snapshot overhead is obvious with the small file-size because the larger number of snapshot-related metadata operations would occur than with large file-sizes. Also, the WORM performance results show that the overhead to issue the necessary system call and to insert WORM attributes to database is very small. The dominated performance factor is I/O performance of the underlying file system.

Acknowledgment

This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0025935).

References

[1] A. Buldas, P. Laud, H. Lipmaa and J. Villemson, “Time-Stamping with Binary Link-ing Schemes,” *In Advances on Cryptology (CRYPTO’98)*, LNCS 1462, 1998, pp.486-501.

[2] S. L. Garfinkel and J.S. Love, “A File System for Write-Once Media,” *MIT Technical report*, 1986.

[3] A. Apvrille and J. Hughes, “A Time Stamped Virtual WORM System,” *Securite des Communications sur Internet (SECI02)*, 2002.

[4] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto and G. Peck, “Scalability in the XFS File system,” *USENIX 1996 Annual Technical Conference*, 1996.

[5] J. Mostek, W. Earl, and D. Koren, “Porting the SGI XFS File system,” *Linux 6th Linux Kongress: The Linux Storage Management Workshop (LSMW)*, 1999.

[6] Z.N.J. Peterson and R.C. Burns, “Ext3cow: The design, implementation, and analysis of metadata for a time shifting file system,” *Technical report*, Department of Computer Science, The Johns Hopkins University, 2003.

[7] Z.N.J. Peterson and R.C. Burns, “Ext3cow: A Time-Shifting File System for Regulatory Compliance,” *ACM Transactions on Storage,* vol. 1, no. 2, 2005, pp.190-212.

[8] S. Shim, W. Lee and C. Park, “An Efficient Snapshot Technique for Ext3 File System in Linux 2.6,” *Technical report,* Pohang University of Science and Technology, 2005.

[9] American Megatrends. Inc., “AMI Snapshot Technology,” *Technical report*, 2005.

[10] B. Mary and D. Peterson, “Integrating Network Appliance Snapshot and SnapRestore with Veritas Netbackup in an Oracle Backup Environment,” *Technical report 3394*, Network Appliance, 2006.

[11] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman and S. Owara, “SnapMirror: File System based Asynchronous Mirroring for Disaster Recovery,” *Proc. of the FAST’02 Conference on File and Storage Technologies*, 2002.

[12] J. Piernas, T. Cortes and J. Garcia, “DualFS: A New Journaling File System without Meta-Data Duplication,” *Proc. of the 2002 International Conference on Supercomputing*, 2002.

[13] D. Santry, M. Feeley, N. Hutchinson and A. Veitch, “Elephant: The File System that Never Forgets,” *Proc. of IEEE Hot Topics in Operating Systems*, 1999.

[14] D. Santry, M. Feeley, N. Hutchinson, R. Veitch, R. Carton and J. Ofir, “Deciding when to forget in the Elephant file system,” *Proc. of 17th ACM Symposium on Operating Systems Principles (SOSP’99),* 34(5),1999, pp.110-123.

[15] S. Quinlan, “A Cached WORM File System,” *Software Practice and Experience,* Vol. 21, No. 12, 1991, pp.1289-1299.