Implementation and Performance Evaluation of Two Snapshot Methods on iSCSI Target Storages

Weijun Xiao, Yinan Liu, and Qing (Ken) Yang
Dept. of Electrical and Computer Engineering
University of Rhode Island, Kingston RI 02881
Tel: (401) 874-5880, Fax: (401) 782-6422
Email: {wjxiao,yinan,qyang}@ele.uri.edu

Jin Ren and Changsheng Xie
National Laboratory for Data Storage Systems
Huazhong University of Science and Technology
Wuhan, Hubei, P. R. China

Abstract

While snapshots have been commonly used in data storages for backup and data protections, little is known in the open literature how such snapshots impact application performance. This paper presents an implementation and performance evaluation of two snapshot techniques: copy-on-write snapshot and redirect-on-write snapshot. Our implementation is carried out at block level on a standard iSCSI target. We carry out quantitative performance evaluations and comparisons of the two snapshot implementations using TPC-C, TPC-W, IoMeter, and PostMark benchmarks. Our measurements reveal many interesting observations regarding the performance characteristics of the two snapshot techniques. Depending on the applications and different I/O workloads, the two snapshot techniques perform quite differently. In general, copy-on-write performs well on read-intensive applications while redirect-on-write performs well on write-intensive applications.

1. Introduction

As organizations and businesses depend more and more on digital information, data protection and disaster recovery have become the top challenge for data storage designers and administrators. In most storage systems, data protection relies on periodic backup [1] and remote replications [2,3]. Both backup and replication often make use of snapshot technologies to enhance and simplify recovery process by reducing recovery time and providing more recovery points. A snapshot creates a point-in-time image of a data storage volume by making a full copy (clone) or a differential copy of the volume. The differential copy snapshot improves space

efficiency upon full copy snapshot because only changes to the volume are stored after the snapshot. There are basically two types of differential snapshots: copy-on-write [16] and redirect-on-write [8].

Copy-on-write snapshot: At the time when the snapshot is created, a small volume is allocated as a snapshot volume with respect to the source volume. Upon the first write to a data block after the snapshot, the original data of the block is copied from the source volume to the snapshot volume. After copying, the write operation is performed on the block in the source volume. As a result, the data image at the time of the snapshot is preserved. The combination of the source volume and the snapshot volume presents the point-in-time image of the data. After the snapshot is created, all subsequent read I/Os are performed on the source volume. Write I/Os after the first change to a block is also performed on the source volume, i.e. only the first write to a block copies the original data to the snapshot volume.

Redirect-on-write snapshot: Copy-on-write requires 3 I/O operations upon the first write to a block [16]: (1) read the original block from the source volume, (2) write the original block to the snapshot volume, and (3) write the new data in the source volume. These I/O operations are done at production time, which may negatively impact application performance. To overcome this, one can do redirect-on-write that leaves the original block in the source volume intact and the new write operation is performed on the snapshot volume. This eliminates the extra I/O operations of the copy-on-write method. After the snapshot, all subsequent write I/Os are performed on the snapshot volume while read I/Os may be from source volume or

snapshot volume depending on whether the block has been changed since the snapshot. The point-in-time image of the data at the time of a snapshot is the source volume itself since the source volume has been read-only since the snapshot time. The source volume will be updated at a later time, hopefully not in production time, by copying data from the snapshot volume.

Clearly, the two different snapshot methods described above have different performance characteristics. While they have been used in various storage products, little is known in the open literature about their impact on application performance except for some scattered product information from vendors. For example, Microsoft suggests that users should not create shadow copies more frequently than once per hour with the default configuration being two shadow copies per day (Microsoft's snapshot is done in Virtual Shadow Copy Service). Otherwise, performance impact would be significant [4]. We believe that it is desirable and important to have a clear understanding of the performance characteristics of various snapshot technologies independent of specific vendor products. Such clear understanding will benefit storage designers in making design decisions and in playing tradeoffs between performance and cost. It will also benefit storage users in their storage configuration and planning for data protection and recovery. We therefore present in this paper an implementation and quantitative performance evaluation of the two snapshot methods.

While there are existing snapshot implementations on various storage products, direct measurements on these products may not provide exact performance characteristics of the different snapshot methods because of variety of storage optimizations built in each storage product. In order to accurately characterize performance of different snapshot methods independent of other storage optimization techniques, we have developed and implemented an iSCSI target software. Our iSCSI target implementation is a user level program running on Windows platform. The target program communicates directly with the standard iSCSI initiator available on Linux and Windows. We have tested our target program for many applications such as MySQL database, Postgres database, NTFS, Tomcat 4.1, MS Office, VC++6.0, gcc, VMWare, RedHat installation, Windows XP installation, and more to show that it is fairly robust and we plan to make the program available to the research community online.

Based on this iSCSI target program, we implemented the two snapshot methods: copy-on-write

and redirect-on-write. Databases, file systems and benchmarks are set up on the machines with a standard iSCSI initiator. We then carry out our performance evaluations on the implementations of the two snapshot methods with all other storage configurations being the same. We use the real world benchmarks including TPC-C, TPC-W, IoMeter, and PostMark to drive our tests. Our measurements allow us to make several interesting observations on the two snapshot techniques. For example, for applications with large proportion of write I/Os, redirect-on-write performs better than copy-on-write snapshot for small block sizes. As the block size increases, such difference diminishes. For read-intensive applications, the results are quite different. There are many factors affecting the snapshot performance including basic hashing unit for doing the snapshot, write frequency, I/O request sizes, and overwrite rate etc. We use our measurement results to analyze in detail how these factors affect storage performance.

The paper is organized as follows. In the next section, we present in detail our design and implementation of the iSCSI target program and the two snapshot methods. Section 3 describes our experimental settings for our performance evaluation. Numerical results and discussions are given in Section 4. Section 5 discusses related research work followed by our conclusions in Section 6.

2. System Design and Implementation

To enable quantitative performance evaluation, we have designed and implemented a complete block level storage target using iSCSI protocol. Our implementation of the iSCSI target is on top of the TCP/IP stack, as shown in Figure 1. In the iSCSI protocol, there are two communication parties, namely iSCSI initiator and iSCSI target [5]. An iSCSI initiator runs under the file system or database system as a device driver. When I/O operations come from the host, the initiator generates I/O requests using SCSI commands encapsulated inside the TCP/IP packets that are sent to the designated iSCSI target. The iSCSI target unwraps the TCP/IP packets to obtain the SCSI commands and data. It then finishes the requested I/O operations on the target side.

Our iSCSI target conforms to the IPS draft (20) [5] and runs on the Windows machine as a user mode program. It can export any disk file, disk volume or the whole disk as a device to provide block level services to the iSCSI initiator. User authentication is based on the IP address of the machine running the iSCSI initiator.
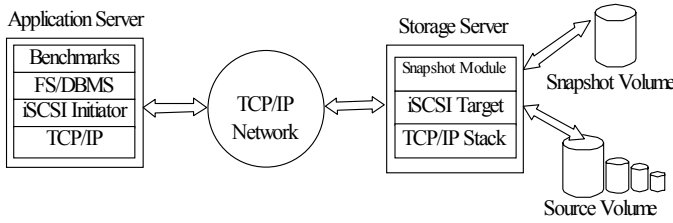
Figure 1. Software Stack of the iSCSI Implementation

The iSCSI target includes four modules: User Interface (UI), basic I/O module, disk and volume manager, and iSCSI protocol module. The UI module deals with user interactions and setting up configuration parameters. Configuration parameters include target device designation, user access authorization, and status monitoring. We can easily designate any disk file, disk volume and the whole disk as a serving device in the iSCSI target for an initiator. The basic I/O module provides transparent functions for the designated device to deal with basic I/O operations. The basic I/O operations include device open, close, read and write. Disk and volume manager is responsible for disk and volume start, close, deletion, designation and status message collection. The iSCSI protocol module includes front-end target layer and STML (SCSI Target Mid-level) layer similar to the UNH iSCSI implementation [6]. The entire target is implemented using MS *Visual C++* 6.0 and has been tested extensively to show that it is fairly robust and performs well. We are currently trying to integrate iCache mechanism [7] to improve the performance further.

Based on our iSCSI target implementation, we have designed and implemented the two snapshot methods, copy-on-write and redirect-on-write. The snapshots are implemented as an independent module, called *snapshot module*, embedded in the iSCSI target. Upon receiving a snapshot request from the host, the snapshot module allocates a small volume as the snapshot volume. The size of snapshot volume is determined by the size of the source volume and the change rate of the source volume. This size can be configurable and dynamically changeable. Currently we allocate 10% of the space of the source volume as the size of the snapshot volume. To simplify our implementation, the snapshot volume is managed using a fixed block size similar to the paging mechanism. That is, all accesses to the data in the snapshot volume are done using the fixed data units referred as *snap_block*. This snap_block size is a user configurable parameter ranging from 512B to 64KB. Using fixed data unit simplifies the indexing structure

and recovery process. However, it may suffer from performance penalty when actual I/O request sizes differ greatly from the snap_block size in the snapshot volume. The penalty comes from frequent fragmentations of the I/O request data to fit the snap_block size. Alternatively, one can manage the snapshot volume using variable block sizes to optimize performance with the extra cost of complicated indexing structure and recovery process. Because of the time limit, in this paper we only report the fixed snap_block size implementation.

With fixed snap_block size, we designed a hash table to store the metadata about the snapshot volume. The hash table uses an LBA as the key. The hash structure is as follows:

```
typedef struct    _HASH_ITEM{
    unsigned long lba; //lba address
    __int64 data_offset; //offset for snapshot volume
    unsigned int read_count, write_count;// counters
}HASH_ITEM;
typedef struct    _HASH_T{
    HASH_ITEM *bucket; //basic hash table
    int collisions;
    int insertions;
    int n;              //length of basic table
    __int64 data_len;
    HASH_ITEM *ext; //extend hash table
}HASH_T, *PHASH_T;
```

2.1 Copy-on-write Snapshot Implementation

For the copy-on-write implementation, a write I/O request goes through the process of determining whether or not it is the first write to the block after the snapshot. This process involves the hash table lookup using the LBA of the write I/O. Depending on the snap_block size and the write I/O size, LBA alignment and data fragmentation may need to be done. The details of alignment and fragmentation will be discussed shortly. If the write I/O goes across snap_block boundaries either because the data size is larger than the snap_block size or the LBA of the I/O is not aligned with the snap_block, the write I/O is decomposed into several small writes of the snap_block size. For every small write, we use its LBA as the key to look up the hash table. If the LBA cannot be found in the hash table, this indicates that this write is the first time to this block. The original data block is copied from the source volume to the snapshot volume. In addition, a new hash entry with this LBA is inserted into the hash table. On the other hand, if the LBA is found in the hash table, this shows that this write is not the first time to this

block, nothing needs to be done on the snapshot volume for this snap_block. After copying all data blocks pertaining to this I/O write from the source volume to the snapshot volume, the write I/O is performed on the source volume.

For read I/Os, there is no need to access the hash table. Our snapshot module will forward read I/Os directly to the source volume. The read operations are performed as usual disk operations in the source volume.

2.2 Redirect-on-write Snapshot Implementation

For the redirect-on-write implementation, a write I/O request goes through the similar process of the hash table lookup, LBA alignment and fragmentation. The difference is that if the LBA is found in the hash table, an overwrite operation is performed on the snapshot volume. No write operation is performed on the source volume. If the LBA of the write is not found in the hash table, a new entry with the LBA is inserted into the hash table and a new write is performed on the snapshot volume. Redirect-on-write leaves the source volume intact. As a result, original data is preserved in the source volume and all changes happen in the snapshot volume. The point-in-time snapshot image is completely contained in the source volume. The source volume will be updated afterward when backup is done or another snapshot is created. Therefore, redirect-on-write snapshot does not eliminate copying but defer it to a later time and hopefully not in the production time [8].

Because the latest changed data are in the snapshot volume and unchanged data in the source volume, read I/Os need to merge data from the two volumes. When a read I/O request comes, the read request is fragmented to one or several requests based on the snap_block size and the LBA. For every fragmented read request, we use its LBA as the key to look up the hash table. If the LBA is found, it indicates the fresh data to this block is in the snapshot volume. We read the data block from the snapshot volume. Otherwise the data is from the source volume. When all the fragmented reads are done, we merge all required data blocks to the read buffer and send the read response to the requestor. Several optimizations are possible for read I/Os. One straightforward optimization is using Bloomfilter technique to quickly determine which volume we will read data from [9]. Because the data size of the snapshot volume in our implementation is limited, the simple hash table performs fairly well. We are currently trying to incorporate various optimizations in our

implementation but not yet reported in this paper because of time constraint.

2.3. Fragmentation and Alignment

For both copy-on-write and redirect-on-write, fragmentations and alignments are necessary. Fragmentation divides a request into several small requests. The LBA of an I/O request needs to be aligned with an LBA of a snap_block since an I/O request can start from any address that might be in the middle of a snap_block. Suppose the starting LBA of a I/O is A, the snap_block size is B, and the data size of the I/O request is L. Assume that an LBA is the logical sector address and a sector has 512 bytes. The fragmentation and alignment are done as follows:

Fragmentation
  Remain = A & (B/512-1);
  If (Remain>0)
  {
        The starting LBA of the first fragmented request is A-Remain;
  }
  if (L<=(B-Remain)*512)
  {
        This Fragmentation only generates one fragmented request;
    Exit Fragmentation;
  }
  Count = (L-(B-Remain)*512)/B;
  Leftsize = (L-(B-Remain)*512) MOD B;
  Generate Count fragmented requests;
  If (Leftsize>0)
  {
        Generate the last fragmented requests with starting LBA as A-Remain+B/512+Count*B/512;
  }

From the above algorithm, one can see that the first fragmented request and the last fragmented request may deal with partial data of a block. In our current implementation, we simplify this process by aligning the LBA address to *A-Remain* and fill up the rest of data from the source volume for the first and the last block fragments. The fact that a snap_block is filled with partial data is known as *internal fragmentation*. Such internal fragmentations cause performance loss because an internal fragmentation not only takes additional space in the snapshot volume but also involves additional I/O operations. Several optimizations are possible to avoid this additional cost such as using variable block sizes.

But these optimizations generally require additional data structure in the hash table. This will make the hash table complicated and the effectiveness remains to be seen. Our current implementation uses the fixed snap_block size that is user configurable.

3. Experimental Methodology

This section presents experimental methodology and the test-bed that we use to study quantitatively the performance of the two different snapshot technologies.

3.1 Experiment Setup

Using our implementation described in the last section, we installed our prototype software on a PC serving as a storage server, as shown in Figure 1. Two PCs are interconnected using the Intel's NetStructure 10/100/1000Mbps 470T switch. One of the PCs acts as an application server running benchmarks with iSCSI initiator installed and the other acts as the storage server with our iSCSI target installed. The hardware characteristics of the PCs are shown in Table 1.

In order to test our iSCSI target and snapshot module under different applications and different software environments, we set up both Linux and Windows operating systems in our experiments. The software environments on these PCs are listed in Table 1. We install Fedora 2 (Linux Kernel 2.4.20) and Microsoft Windows XP Professional on the PCs. On the Linux machine, the UNH iSCSI initiator [6] is installed. On the Windows machines the Microsoft iSCSI initiator [10] is installed.

On top of the iSCSI target and the snapshot module, we set up two different types of databases and two types of file systems. Postgres Database 7.1.3 is installed on Fedora 2. MySQL 5.0 database is set up on Windows. To be able to run real world web applications, we install Tomcat 4.1 application server for processing web application requests issued by benchmarks. For File system benchmarks, IoMeter runs on Windows and PostMark runs on Fedora 2.

3.2 Workload Characteristics

The first benchmark, TPC-C, is a well-known benchmark used to model the operational end of businesses where real-time transactions are processed [11]. TPC-C simulates the execution of a set of distributed and on-line transactions (OLTP) for a period of two to eight hours. It is set in the context of a wholesale supplier operating on a number of warehouses and their associated sales districts. TPC-C incorporates five types of transactions with different complexity for online and deferred execution on a database system. These transactions perform the basic operations on databases such as inserts, deletes, updates and so on. From data storage point of view, these transactions will generate reads and writes that will change data blocks on disks. For Postgres Database, we use the implementation from TPCC-UVA [12]. 5 warehouses with 50 users are built on Postgres database taking 2GB storage space. Details regarding TPC-C workloads specification can be found in [11].

PC 1	P4 2.8GHz/256M RAM/80G+10G Hard Disks
PC 2	P4 2.4GHz/2GB RAM/200G+10G Hard Disks
OS	Windows XP Professional SP2
	Fedora 2 (Linux Kernel 2.4.20)
Database	Postgres 7.1.3 for Linux
	MySQL 5.0 for Microsoft Windows
iSCSI	UNH iSCSI Initiator 1.6
	Microsoft iSCSI Initiator 2.0
Benchmark	TPC-C for Postgres(TPCC-UVA)
	TPC-W Java Implementation
	IoMeter
	PostMark
Network	Intel NetStructure 470T Switch
	Intel PRO/1000 XT Server Adapter (NIC)

Table 1. Hardware and Software Environments

Our second benchmark, TPC-W, is a transactional web benchmark developed by Transaction Processing Performance Council that models an on-line bookstore. The benchmark comprises a set of operations on a web server and a backend database system. It simulates a typical on-line/E-commerce application environment. Typical operations include web browsing, shopping, and order processing. We use the Java TPC-W implementation of University of Wisconsin-Madison [13] and build an experimental environment. This implementation uses Tomcat 4.1 as an application server and MySQL 5.0 as a backend database. The configured workload includes 30 emulated browsers and 10,000 items in the ITEM TABLE.

Besides benchmarks running on databases, we have also run two file system benchmarks PostMark and IoMeter. PostMark is a widely used file system
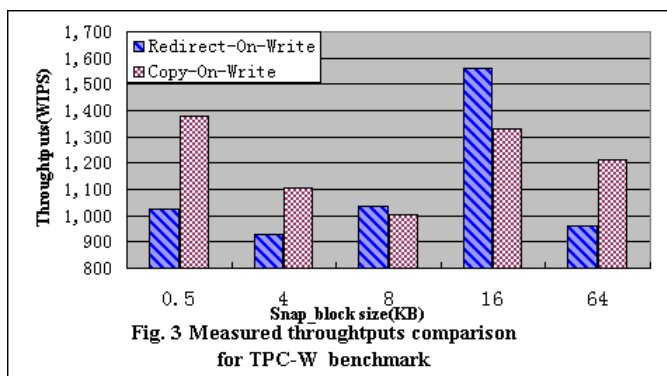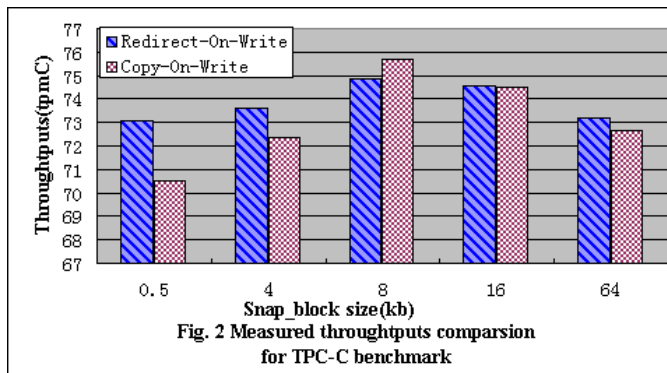
benchmark tool written by Network Appliance, Inc [14]. It measures performance in terms of transaction rates in an ephemeral small-file environment by creating a large pool of continually changing files. Once the pool has been created, a specified number of transactions occur. Each transaction consists of a pair of smaller transactions, i.e. Create file/Delete file and Read file/Append file. Each transaction's type and files it affected are chosen randomly. The read and write block size can be tuned. In our experiments, we set PostMark workload to include 50,000 files and to perform 100,000 transactions. Read and Write buffer sizes are set to 4KB. IoMeter is another flexible and configurable benchmark tool that is also widely used in industries and the research community [15]. It can be used to measure the performance of a mounted file system or a block device. We run the IoMeter on NTFS with 4K-block size for two types of workloads: 100% random writes, and 50% writes and 50% reads.

4.   Numerical Results and Discussions

Using our implementations and the experimental settings described in the previous sections, we carried out extensive experiments to measure snapshot performances. In order to isolate the effects of various file systems, we use two raw partitions for the source volume and the snapshot volume in our experiments. All results reported here are measured using the two raw partitions. We consider 5 different snap_block sizes for TPC-C and TPC-W: 512B, 4KB, 8KB, 16KB, and 64KB. For IoMeter and PostMark, we run our experiments for snap_block sizes of 512B, 4KB, 8KB, and 64KB.

Our first experiment is to measure the throughputs of TPC-C benchmark running on Postgres database using our iSCSI target as the block level storage with each of the two different snapshots enabled. Figure 2 shows the measured results in terms of tpmC that is the number of transactions finished per minute. For the snap_block size of 512B, we observed noticeable difference between copy-on-write and redirect-on-write. As the snap_block size increases, the performance difference reduces. It is interesting to note that the performance of both snapshot methods increases as we increase the snap_block size from 512B to 8KB. As discussed before, large snap_block sizes increase the chance of internal fragmentations and LBA alignments, giving rise to performance penalties. However, our experiments show that this penalty is compensated by large and integrated I/O operations on the snapshot volume. But if we increase the snap_block size further

beyond 8KB, performance drops because of excessive internal fragmentations.

Fig. 2 Measured throughputs comparsion for TPC-C benchmark

Fig. 3 Measured throughputs comparison for TPC-W benchmark

Throughput results for TPC-W are shown in Figure 3. We run the TPC-W benchmark on MySQL database to measure the throughputs in terms of WIPS that is the web interactions finished per second. The TPC-W results are quite different from the TPC-C results. For the snap_block size of 512B, copy-on-write method performs much better than redirect-on-write for TPC-W benchmark as shown in Figure 3. This is in a quite contrast to TPC-C. There are two major reasons for this phenomenon. First, the ratio between read I/Os and write I/Os in TPC-C is about 1:9 whereas the ratio in TPC-W is 3:2. With large proportion of read I/Os in the TPC-W benchmark, copy-on-write snapshot shows better performance because read I/Os are not affected by the snapshot, while redirect-on-write suffers from performance penalty because of read merging. Secondly and more importantly, we noticed in TPC-W that the average write size is about 11KB whereas the average read size is about 16KB. For the small snap_block size of 512B, consecutive data blocks may be scattered in the snapshot volume. As a result, merging small blocks scattered on a disk volume takes a lot of slow I/O operations, giving rise to large I/O response time. Our analysis is further proved by the fact the

6

redirect-on-write performs very well and better than copy-on-write for the snap_block size of 16KB as shown in the figure. In this case, both reads and writes are performed sequentially with the size matching the average I/O size.
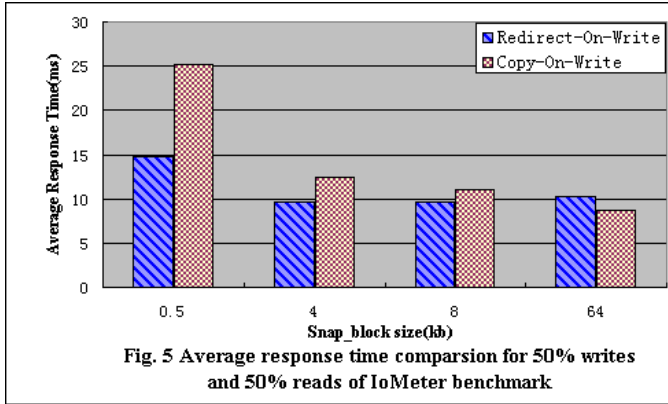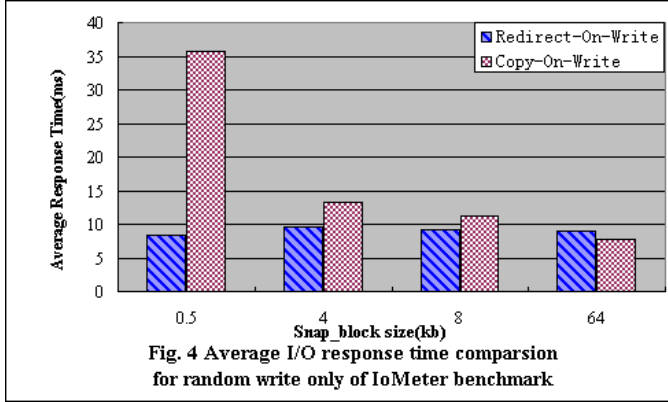

Fig. 4 Average I/O response time comparsion for random write only of IoMeter benchmark

Fig. 5 Average response time comparsion for 50% writes and 50% reads of IoMeter benchmark
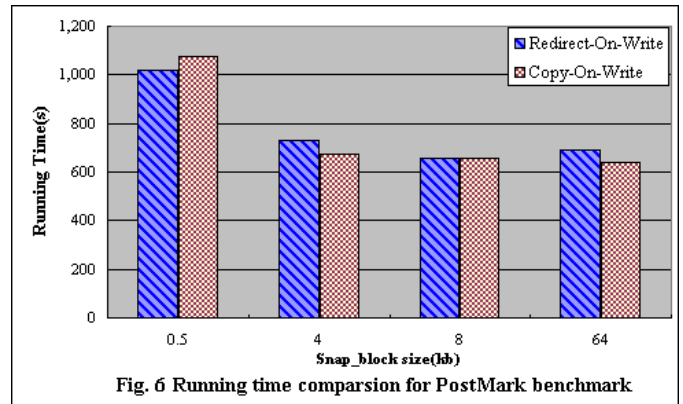
Figure 4 shows the measured results in terms of average I/O response time for IOMeter benchmark with 100% random write I/Os. For such write-intensive benchmarks, we observed the similar performance characteristics to that of TPC-C benchmark. Redirect-on-write performs better than copy-on-write for all snap_block sizes except for 64KB when internal fragmentations and LAB alignments become excessive. For the snap_block size of 512B, the redirect-on-write snapshot implementation performs 4 times better than the copy-on-write implementation. For snap_block size of 4KB, the performance difference is about 40%. The performance difference can mainly be attributed to the reduced I/O operations of the redirect-on-write compared to the copy-on-write. Recall that 3 I/Os are needed for the first write to each data block after the snapshot. Note that the redirect-on-write snapshot does not eliminate the copy operations but defer them to a later time. If the copy operations can be done off line

and not during production time, one can benefit from such deferring of data copies.

In order to observe how the two snapshots impact application performances with mixed read and write I/Os for the IOMeter benchmark, we measured again the IOMeter performance with 50% random reads and 50% random writes. The average I/O response times are shown in Figure 5. Similar performance results to that of Figure 4 are observed except for smaller differences between the two snapshot methods. The performance difference is small because read operations of the copy-on-write perform better than redirect-on-write. This observation suggests that there is a room for performance optimization of the redirect-on-write implementation. We are currently working on various optimization techniques as discussed in Section 2. Notice that in both Figures 4 and 5 the average read and write I/O sizes are about 4KB.
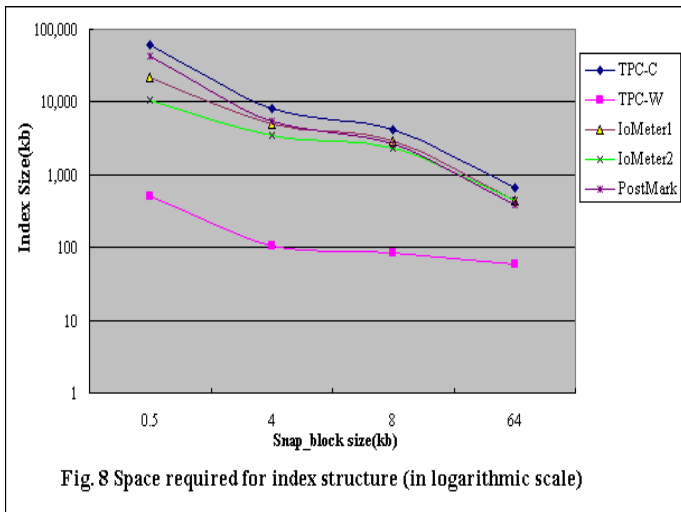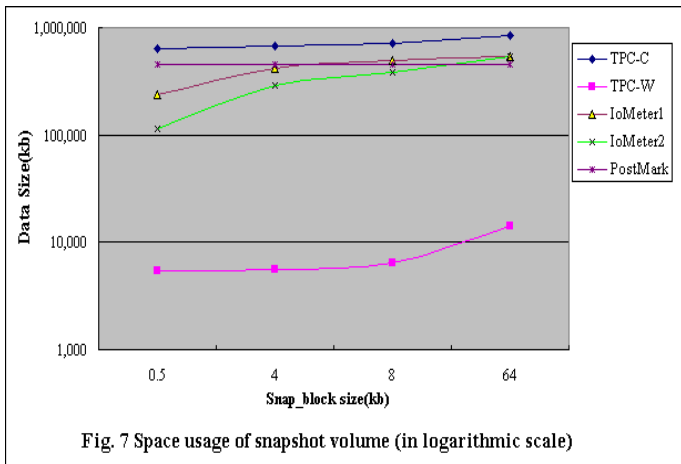

Fig. 6 Running time comparsion for PostMark benchmark

snap_block size	WriteTime(ms)	ReadTime(ms)
64K	8.328	1.906
16K	8.359	2.344
8K	8.484	2.593
4K	12.516	3.594
0.5K	39.562	10.219

Table 2   I/O Time Measurements with different snap_block sizes

PostMark results are shown in Figure 6 in terms of total running time for 100,000 transactions on 50,000 files. For this benchmark, it seems that the two snapshot methods show similar performance across all block sizes considered with the difference less than a few percents. One observation that is consistent with all other benchmarks is that the performance of 512B snap_block size is not as good as other block sizes. This observation suggests that using sector size to do snapshot is not an

optimal solution even though it does not incur any internal fragmentation. To further clarify this observation, we carried out a small experiment of reading and writing a 64KB data in a buffer to a disk using different block sizes at block device level. We measured the read and write I/O times in the experiment. The results are listed in Table 2. As shown in Table 2, larger block sizes take shorter time to write than smaller block sizes. However, the time differences for the block sizes of 8KB, 16KB, and 64KB are not significant. Noticeable longer time is observed when the block size changes from 8KB to 4KB. There is a dramatic increase in time for the block size of 512B. This result explains again why 512B snap_block performs poorly in all the benchmarks studied.

Fig. 7 Space usage of snapshot volume (in logarithmic scale)

Fig. 8 Space required for index structure (in logarithmic scale)

Small block sizes not only slow down I/O operations but also require large index data structure for hashing. Figures 7 and 8 show the space used for the snapshot volume and the sizes of the index data structure for different block sizes. For 512B block size, the index

structure takes about 10% of the snapshot volume size whereas for 8KB block size the index structure takes about half of a percent of the snapshot volume. For 64KB block size, the index structure is less than 0.08% of the snapshot volume. These two figures clearly show that the larger the snap_block size is, the smaller the index structure will be. Therefore, to limit the overhead in the index data structure, one would like to use large block sizes.

On the other hand, large block sizes incur internal fragmentations as discussed previously. The internal fragmentation not only wastes storage space but also add more unnecessary I/O operations in the snapshot volume. To quantitatively observe internal fragmentations, we measured the *space efficiency* defined as the average ratio between the size of the write I/O coming from the host and the actual data size written in the snapshot volume because of the write I/O. The space efficiency is an indicator of the degree of internal fragmentations. The efficiency of 100% means that the data size written in the snapshot volume is exactly the same as the write I/O data size from the host with no storage waste. A smaller efficiency implies a large internal fragmentation. To see how the internal fragmentation occurs, consider the following example. Suppose two consecutive 16KB snap_blocks with the LBAs of *A* and *A + 32*, respectively. If the host issues a write I/O of size 2KB with starting LBA of *A + 30*, the write I/O will result in changes in both of the two snap_blocks. 1KB is written at the end of the first snap_block with the LBA of *A* and the other 1KB at the beginning the second snap_block with the LBA of **A+32**. The total internal fragmentation is 30KB.

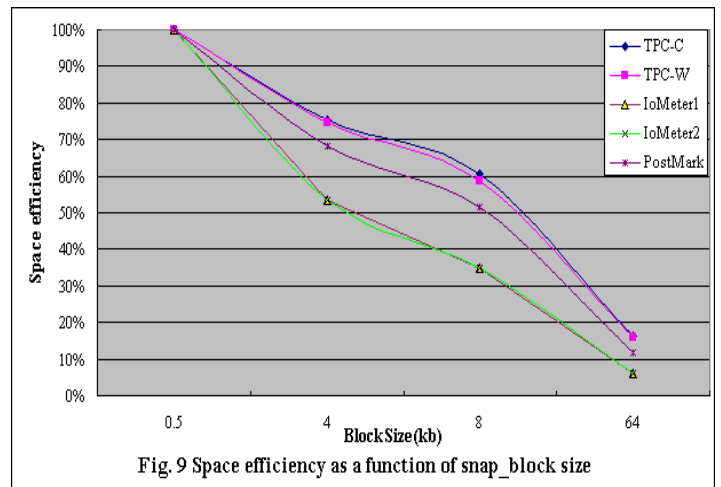Fig. 9 Space efficiency as a function of snap_block size

Figure 9 shows the space efficiency of the two

snapshot methods for different benchmark runs. Note that the two snapshot methods use the same amount of storage space in our implementation. As can be seen in the figure, the efficiency for the block size of 512B is 100% with no storage waste. The space efficiency drops rapidly as block size increases implying large internal fragmentations. For 64KB block size, the efficiency drops below 20%. Therefore, to minimize internal fragmentations, one would like to use small block sizes.

It is very interesting to observe the two contradicting objectives: increasing block size for better performance (Figures 1 through 8) and decreasing block size for better space efficiency (Figure 9). Therefore, there is a tradeoff between performance and space efficiency in selecting the snap_block size in designing a snapshot implementation. Clearly, our experiments suggest against sector size and favor 8KB or 16KB block sizes depending on applications.

5. Related Work

Snapshot has been widely used in the storage industry for data protection and data recovery. A good summary of various snapshot methods can be found in [16]. In general, a snapshot can be used in a file system for versioning or it can be used in a block level device for backup and recovery of a data volume.

For file versioning, a snapshot can be implemented efficiently with the availability of file system intelligence and access of indexes. For example, Peterson and Burns [17] recently designed a versioning file system named Ext3cow that uses snapshot functionality. Although the snapshot is called copy-on-write, the actual implementation allocates a new block for a new write and preserves a copy of the old block in the old version. The pointer in the I-node will be updated to reflect different versions of the file. Similarly, NetApp's WAFL (Write Anywhere File Layout) writes a new data block to another place on the disk, and changes the I-node to point to the new block. The point-in-time snapshot image still references to the original block that is unmodified on the disk [18]. From performance point of view, these file system based snapshots should be similar to the redirect-on-write described in this paper. There are many versioning file systems such as Tops-20 [19], VMS [20], Elephant [21], and CVFS [22] that make use of copy-on-write snapshot.

For data backup and recovery, Plan 9 [23], Petal [24], Microsoft Volume Shadow Copy Service (VSS) [25], and Spiralog [26] backup systems use copy-on-write to create snapshots. Plan 9 backups data daily by creating snapshots of the file system. When creating a snapshot, it freezes the state of the file system and makes subsequent modifications to a copy of the frozen data [1,23]. Petal creates a virtual disk backup using tar command through snapshots [24]. VSS provides a backup infrastructure for Microsoft Windows XP and Microsoft Windows Server 2003 operating systems, as well as a mechanism for creating consistent point-in-time copies [25]. Spiralog provides on-line backup of a log-structured file system (LFS) [27].

At block device level, there are many storage products using snapshot technologies. Typical products include EMC's TimeFinder/Snap [28], HDS's copy-on-write Snapshot [29], Microsoft's VSS, and NetApp's Snapshot [30]. Most of these products use copy-on-write method [16] with the exception of NetApp that uses a method similar to the redirect-on-write described in this paper.

Although snapshots have been implemented in many file systems and storage products, there has been no quantitative performance evaluation of different snapshot methods at block device level. To the best of our knowledge, we are the first one to implement two different snapshot methods on the same storage target and to accurately compare the performances of the two snapshot methods.

Extensive research has been reported in the literature on iSCSI protocol including storage implementations [6, 31, 32, 33], and performance evaluations using simulations [7,34] and measurements [35,36,37]. It has been shown in these studies that iSCSI performs very well as a block level data storage. Radkov et al [35] have shown that iSCSI outperforms NFS by a factor of 2 or more for meta-data intensive workloads. Most of the iSCSI target implementations reported in the literature are on Linux system. Few Windows based target implementations are reported in the open literature except for one or two commercial products. Furthermore, our primary purpose here is the quantitative evaluation of the two snapshot techniques that we have implemented in the iSCSI target.

6. Conclusions

In this paper, we have presented an implementation and performance evaluation of two differential snapshot methods: copy-on-write and redirect-on-write. Our implementation is based on the standard iSCSI protocol. A robust iSCSI target program for Windows has been developed and tested that works smoothly with two publicly available initiators: Windows' initiator and

Linux initiator. The two snapshot methods are implemented as an independent program module embedded in the iSCSI target. Extensive experiments have been carried out to measure the performance impacts of the two snapshot methods. We use real world benchmarks such as TPC-C, TPC-W, IOMeter, and Postmark to measure the performances. Our numerical results uncover many important performance characteristics that were unknown before. Our analysis can provide a useful guide to storage designers in making their design decisions and to storage users in planning their data protection and recovery. We plan to make our implementation program available to the research community online.

As a future research work, we plan to optimize our iSCSI target program as well as the snapshot implementations. Possible optimizations include proper caching at the iSCSI target, different hashing functions, efficient merging of read I/Os on redirect-on-write snapshot, variable snap_block sizes, and so forth.

Acknowledgments

References

[1] A.L. Chervenak, V. Vellanki, and Z. Kurmas, "Protecting file systems: A survey of backup techniques," In *Proc. of Joint NASA and IEEE Mass Storage Conference*, College Park, MD, March 1998.

[2] Minwen Ji, Alistair Veitch, and John Wilkes, "Seneca: remote mirroring done write," In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, pp. 253-268

[3] Ming Zhang, Yinan Liu, and Qing Yang, "Cost- Effective Remote Mirroring Using the iSCSI Protocol," In *21st IEEE Conference on Mass Storage Systems and Technologies*, April, 2004, pp.385-398.

[4] Novastor Corporation, "Microsoft Shadow-Copy Service and its Role in a Organization's Total Backup Strategy," http://www.novastor.com/graphics/VSS_White_Paper.pdf.

[5] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, "iSCSI draft standard," http://www.ietf.org/internet-drafts/draftietf-ips-iscsi-20.txt, Jan. 2003.

[6] UNH, "iSCSI reference implementation," 2005, http://unh-iscsi.sourceforge.net.

[7] Xubin He, Qing Yang, and Ming Zhang, "A Caching Strategy to Improve iSCSI Performance," in *Proceedings of IEEE Annual Conference on Local Computer Networks*, Nov. 6-8,2002.

[8] H. Simitci, "Backups using snapshots," In *Storage Network Performance Analysis*, Wiley Publishing, Inc., 2003, pp. 280-282

[9] B. Bloom, "Space/time trade-offs in hashing coding with allowable errors, " *Communication of the ACM*, Vol.13 (7), pp. 422-426, July 1970.

[10] Microsoft Corp., "Microsoft iSCSI Software Initiator Version 2.0," 2005, http://www.microsoft.com/windowsserversystem/storage/default.mspx.

[11] Transaction Processing Performance Council, "TPC BenchmarkTM C Standard Specification," 2005, http://www.tpc.org/tpcc.

[12] J. Piernas, T. Cortes and J. M. García, "TPCC- UVA: A free, open-source implementation of the TPC-C Benchmark," 2005, http://www.infor.uva.es/~diego/tpcc-uva.html.

[13] H.W. Cain, R. Rajwar, M. Marden and M.H. Lipasti, "An Architectural Evaluation of Java TPC-W," *HPCA 2001*, Nuevo Leone, Mexico, Jan. 2001.

[14] J. Katcher, "PostMark: A new file system bench -mark," *Network Appliance*, Tech. Rep. 3022, 1997.

[15] Intel, "IoMeter: Performance Analysis Tool," http://www.iometer.org/.

[16] G. Duzy, "Match snaps to apps," *Storage, Special Issue on Managing the information that drives the enterprise*, pp. 46-52, Sept. 2005.

[17] Z. Peterson and R. C. Burns, "Ext3cow: A Time-Shifting File System for Regulatory Compliance", *ACM Transactions on Storage*, Vol.1, No.2, pp. 190-212, 2005.

[18] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," In Proc. of the USENIX Winter Technical Conference, San Francisco, CA, 1994, pp. 235-245.

[19] L. Moses, "An introductory guide to TOPS-20," *Tech. Report TM-82-22*, USC/Information Sciences Institutes, 1982.

[20] K. McCoy, "*VMS File System Internals*," Digital Press, 1990.

[21] D. S. Santry, M.J. Feeley, N.C. Hutchinson, A.C. Veitch, R.W. Carton, and J. Ofir, "Deciding when to forget in the Elephant file system," In *Proc. of 17th ACM Symposium on Operating System Principles*, Charleston, SC, Dec. 1999, pp. 110-123.

[22] C.A.N. Soules, G. R. Goodson, J. D. Strunk, and G.R. Ganger, "Metadata efficieny in versioning file systems," In *Proc. of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2003, pp. 43-58.

[23] R. Pike, D. Presotto, K. Thompson, and et al, "Plan 9 for Bell Labs," http://plan9.bell-labs.com /sys/doc/

[24] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," In *Proc. of the 7th International Conference on Architecture Support for Programming Languages an Operating Systems (ASPLOS-7)*, Cambridge, MA, 1996.

[25] A. Sankaran, K. Guinn, and D. Nguyen, "Volume Shadow Copy Service," March 2004, http://www.microsoft.com.

[26] R. Green, A. Baird, and C. Davies, "Designing a Fast, On-line Backup System for a Log-structured File System," *Digital Technical Journal*, Oct. 1996.

[27] M. Rosenblum and J. Ousterhout, "Log-Structured File System," In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, June 1991, pp. 1-15.

[28] EMC Corporation, "EMC TimeFinder Family," http://www.emc.com/products/software/timefinder.jsp

[29] Hitachi Ltd., "Hitachi ShadowImage implementation service," June 2001,http://www.hds.com /copy_on_ write_snapshot_467_02.pdf.

[30] NetAppliance Corporation, "Snapshot Technology," http://www.netapp.com/products/snapshot.html.

[31] Hui Xiong, Renuga Kanagavelu, Yaolong Zhu, Khai Leong Yong, "An iSCSI Design and Implementation," *in Proc. of the Twelfth NASA Goddard / Twenty-First IEEE Conference on Mass Storage Systems and Technologies NASA / IEEE MSST2004*

[32] Intel Co. "Intel iSCSI Reference Implementation," *http://sourceforge.net/projects/intel-iscsi.*

[33] *Cisco, "Linux-iSCSI Project," http://linux-iscsi.source forge.net/.*

[34] Yingping Lu, Farrukh Noman, and David H.C. Du, "Simulation Study of iSCSI-based Storage System," *in Proc. of The Twelfth NASA Goddard /Twenty-First IEEE Conference on Mass Storage Systems and Technologies NASA / IEEE MSST2004, pp. 399-408*

[35] P. Radkov, Li Yin, P. Goyal, P. Sarkar, and P. Shenoy, "A performance comparison of NFS and iSCSI for IP-Network Storage," *Proceedings of FAST 2004.*

[36] Stephen Aiken , Dirk Grunwald , Andrew R. Pleszkun , Jesse Willeke, "A Performance Analysis of the iSCSI Protocol," *In Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03), April 07-10, 2003.*

[37] Ismail Dalgic, Kadir Ozdemir, Rajkumar Velpuri, Jason Weber, Umesh Kukreja, Atrica, and Helen Chen, and Umesh Kukreja, "Comparative Performance Evaluation of iSCSI Protocol over Metro, Local, and Wide Area Networks," *In Proc. of Twelfth NASA Goddard / Twenty-First IEEE Conference on Mass Storage Systems and Technologies NASA / IEEE MSST2004.*