Design and Analysis of Block-Level Snapshots for Data Protection and Recovery

Weijun Xiao, *Member*, *IEEE*, Qing Yang, *Senior Member*, *IEEE*,
Jin Ren, Changsheng Xie, and Huaiyang Li

Abstract—This paper presents a comprehensive study on implementations and performance evaluations of two snapshot techniques: copy-on-write snapshot and redirect-on-write snapshot. We develop a simple Markov process model to analyze data block behavior and its impact on application performance, while the snapshot operation is underway at the block-level storage. We have implemented the two snapshots techniques on both Windows and Linux operating systems. Based on our analytical model and our implementation, we carry out quantitative performance evaluations and comparisons of the two snapshot techniques using IoMeter, PostMark, TPC-C, and TPC-W 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.

Index Terms—Data storage, data protection, snapshot, copy-on-write, redirect-on-write.

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 simplify backup and recovery process. 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 [7] and redirect-on-write [34].

While snapshot technologies have been widely used in various storage products for the purpose of data backup and data protections, little is known in the open literature about the principle of snapshots, data recoverability of various snapshot techniques, and their impacts on application performance. Because of such lack of understanding, a large percentage of data recoveries based on snapshots failed in the real world [4]. While this fact is well known, there has been no research study on why this is the case except for our recent analytical study that uncovered several important unknown issues [5]. Therefore, we believe that it is very important to understand how snapshot techniques work and how they protect and recover data.

In addition to the understanding of how snapshots work and how to recover data, it is also very important to understand their performance impacts because snapshots are runtime operations. There has been no performance evaluation of snapshot techniques in the research literature except for some scattered product information from storage vendors. For example, Microsoft suggests that users should not create snapshots more frequently than once per hour with the default configuration being two snapshots per day (Microsoft's snapshot is done in Virtual Shadow Copy Service). Otherwise, performance impact would be significant [6]. We believe that it is desirable and important to have a clear understanding of the performance characteristics of various snapshot techniques independent of specific vendor products. Such clear understanding will benefit storage designers in making design decisions and playing trade-offs between the performance and cost. It will also benefit storage users in their storage configuration and planning for data protection and recovery.

We present in this paper a simple mathematical analysis of the snapshot techniques using a Markov process. Our model provides insightful details helping storage researchers and designers in understanding how snapshots work and how they impact application performance at runtime. In addition to the analytical modeling, we have developed and implemented the two snapshot techniques on Windows operating system as well as Linux operating system. Our implementations accurately characterize the performance of different snapshot methods independent of other storage optimization techniques. We have tested our implementations with many applications such as MySQL database, Postgres database, NTFS, Tomcat 4.1, and more to show

- *W. Xiao, Q. Yang, and J. Ren are with the Department of Electrical, Computer, and Biomedical Engineering, The University of Rhode Island, 4 East Alumni Ave, Kingston, RI 02881. E-mail: {wjxiao, qyang, rjin}@ele.uri.edu.*
- *C. Xie and H. Li are with the Department of Computer Engineering, National Laboratory for Data Storage Systems, Huazhong University of Science and Technology, Wuhan, Hubei 430074, P.R. China. E-mail: cs_xie@mail.hust.edu.cn, lhyclh@sohu.com.*

Manuscript received 5 Feb. 2008; revised 21 Nov. 2008; accepted 11 Mar. 2009; published online 23 July 2009.
Recommended for acceptance by A. George.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-02-0056.
Digital Object Identifier no. 10.1109/TC.2009.107.

that they are fairly robust. We carry out measurement experiments on the snapshot implementations to validate our analytical model, and then, to evaluate and compare the performance impacts of the snapshots techniques on application performance using standard benchmarks including IoMeter, PostMark, TPC-C, and TPC-W.

Our analysis and 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 the background and related work. Section 3 gives a Markov process model for analyzing the behavior of data blocks when snapshots start. The detailed design and implementations are discussed in Section 4. Section 5 describes our experimental settings for our performance evaluations. Numerical results and discussions are given in Section 6. We conclude our paper in Section 7.

2 Background and Related Work

A snapshot creates a storage image at a point in time for the purpose of data backup and protection. One can create a copy of entire storage, called cloning, as a snapshot, or record only the changed blocks, called differential snapshots, to save storage space. There are basically two types of differential snapshots: copy-on-write (COW) snapshot and redirect-on-write (ROW) snapshot.

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 are 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 are 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 three I/O operations upon the first write to a block [7]: 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.

It should be mentioned that COW copies only the original data to the snapshot volume upon the first write to a data block at production time, while ROW redirects all writes to the snapshot. From the performance point of view, there is a trade-off between COW and ROW. This trade-off is similar to the trade-off between buying and renting. COW (analogous to buying) pays a higher upfront cost but is profitable if there are lots of subsequent accesses. ROW (analogous to renting) pays low costs upfront but the costs are incurred on an ongoing basis. The background costs come from the time necessary to identify the data blocks and possibly merge data from the source volume and snapshot volume.

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 [7]. 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 to indexes. For example, Peterson and Burns [8] designed a versioning file system named as 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 refers to the original block that is unmodified on the disk [9]. Clotho implements remap-on-write to provide transparent data versioning as a Linux device driver [10]. 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 [11], VMS [12], Elephant [13], and CVFS [14] that make use of copy-on-write snapshot.

For data backup and recovery, Plan 9 [15], Petal [16], Microsoft Volume Shadow Copy Service (VSS) [17], and Spiralog [18] 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], [15]. Petal creates a virtual disk backup using tar command through snapshots [16]. VSS provides a backup infrastructure for Microsoft Windows XP and Microsoft Windows Server 2003 operating systems, as well as a mechanism for creating consistent

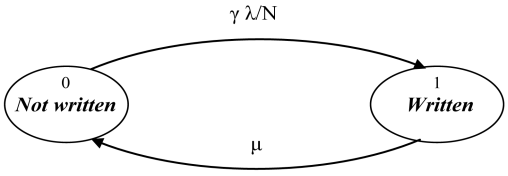


Fig. 1. State diagram of the Markov process.

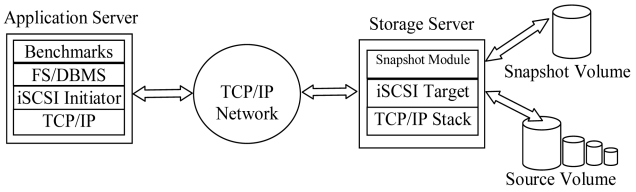


Fig. 2. Software stack of the iSCSI implementation.

point-in-time copies [17]. Spiralog provides online backup of a log-structured file system (LFS) [19].

At the block level, there are many storage products using snapshot technologies. Typical products include EMC's TimeFinder/Snap [20], HDS's copy-on-write Snapshot [21], Microsoft's VSS, and NetApp's Snapshot [22]. Most of these products use copy-on-write method [7] with the exception of NetApp that uses a method similar to the redirect-on-write described in this paper.

3 A Simple Analytical Model

Consider a block storage system with N active data blocks that are uniformly accessed by applications during a time period. These N data blocks can be viewed as the working set of upper layer applications during this time period. Let us assume that the applications issue λ I/O requests per second and the write ratio of the I/O request is , i.e., the probability of a given I/O being a write is γ and the probability of being a read is $1-\gamma$. We further assume that the I/O requests to follow a Poisson distribution. Suppose that the interval between two consecutive snapshots is an exponentially distributed random variable with mean $1/\mu$. Whenever a new COW snapshot starts, the storage system will make a copy of a block upon the first write. Subsequent writes to the same block are done as usual without making copies. Therefore, the write I/O time depends on whether the block has been written before or not. In order to analyze the I/O time, we first determine the state of the block to be written. We use a two-state Markov process to model the state of each data block, as shown in Fig. 1. State 0 means that the block has not been written since the new snapshot started, while state 1 means that the block has been written at least once since the snapshot started. In this Markov model, we have assumed that an I/O request goes to any one of N data blocks equally likely, i.e., all the blocks are uniformly accessed. Let P_0 and P_1 be the steady-state probabilities that the block is in state 0 and state 1, respectively. Solving the Markov chain model, we derive these steady-state probabilities as follows:

$$P_0 = \frac{\mu}{XX}, \quad P_1 = \frac{\gamma}{XX}.$$

In case of COW snapshot, the first write to a block requires three I/O operations: read old data, write old data to snapshot volume, and write the new data in place. Subsequent writes to the same block and all read operations are performed as usual without additional I/O operations. Therefore, the average I/O response time is given by

$$T_{COW} = (1-\gamma) \times D + \gamma \times (P_1 \times D + P_0 \times 3D), \quad (1)$$

where D is the average disk access time.

In case of ROW snapshot, on the other hand, a write operation will not incur any additional I/Os because all writes are redirected to the snapshot volume. However, read operations require determination of which volume will supply the data. More importantly, all changed data blocks since the snapshot started need to be written in place from the snapshot volume to the source volume. This update can be done before the start of another snapshot or offline when storage is not actively serving applications. If this write-back operation is done at runtime before next snapshot starts, the average I/O response time of ROW snapshot is given by

$$T_{ROW} = (1-\gamma) \times (D+\sigma) + \gamma \times D + N \times P_1 \times 2D \times \mu, \quad (2)$$

where σ is the time it takes to identify and merge read data that can potentially come from one of or both source volume and snapshot volume. Compared to the disk access time, σ is usually a small quantity since it involves only simple computation that does not need additional disk operations. The last term in the above equation takes into account the overhead of writing back the latest data in the snapshot volume to the source volume in place. If this write back operation can be done offline, while storage is not serving application request, then the overhead can be ignored. In this case, the average I/O response time of ROW is given by

$$T_{ROW} = (1-\gamma) \times (D+\sigma) + \gamma \times D. \quad (3)$$

It should be mentioned that our analytical model is meant to provide a quick and rough estimate of snapshot performance. A more comprehensive performance model can be found in [23] for general storage systems.

4 Design and Implementation

In this section, we present our implementations of the two snapshot techniques on Windows OS and Linux OS. On Windows system, we make use of the standard iSCSI initiator available in Windows system. We have designed and implemented a complete block-level storage target using the iSCSI protocol. Our implementation of the iSCSI target is similar to UNH iSCSI implementation [24] built on top of the TCP/IP stack, as shown in Fig. 2. There has been research in the literature on the iSCSI protocol including storage implementations [24], [25], [26], [27], performance evaluations using simulations [33], [28] and measurements [29], [30], [31]. It has been shown in these studies that iSCSI performs very well as a block-level data storage.

Our iSCSI target conforms to the IPS draft (20) [32] and runs on the Windows machine as a user mode program. It

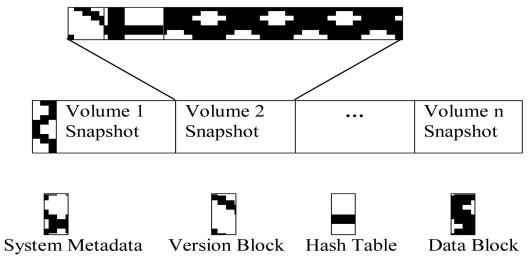


Fig. 3. Data organization of snapshot volume.

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. The iSCSI target includes four modules: user interface, basic I/O module, disk and volume manager, and iSCSI protocol module. 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 [33] 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 percent 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 unit referred as *snap_block*. This snap_block size is a user configurable parameter ranging from 512 B to 64 KB. Using a fixed data unit simplifies the indexing structure and recovery process. However, it may suffer from a performance penalty when actual I/O request sizes differ greatly from the snap_block size in the snapshot volume. For example, the request size can be much larger than the snap_block size and the starting LBA address might be in the middle of a snap_block. This 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. For simplicity, in this paper, we only consider the fixed snap_block size implementation.

Besides the implementation of two snapshot technologies on Windows OS, we have also designed and implemented them on Linux OS as a block device driver. This block device driver runs on top of physical hard disk drivers to manage the source and snapshot volumes. As shown in Fig. 3, we allocate a 512-byte data block at the beginning of the snapshot volume to store system metadata followed by the independent volume snapshots. The system metadata include snapshot flag, created time, and beginning and ending addresses of volume snapshots on disk. For each volume snapshot, storage space is organized as snapshot segments. A snapshot segment has three types of blocks on disk: version block, hash table block, and physical data blocks. The version block stores the created time of snapshot, starting and ending offset of hash table, and corresponding source volume flag. The hash table is the index of physical data blocks. The size of hash table and snapshot data blocks can be customized and adjusted dynamically. The advantage of aforementioned data organization is that a snapshot volume can be used to store the snapshots for multiple source volumes.

Because the snap_block is user-configurable, we use a hash table and simple MOD hash function to manage LBA requests in our implementation. This hash table can be loaded from and saved to physical storages. It should be noted that we can also use other data structures to handle LBA requests such as bitmaps, linked lists, and so on. Although these data structures may save some memory space, they involve complicated address computations and table lookups.

4.1 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 write I/O 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.

4.2 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 I/O 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, the original data are 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 [34].

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 into 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 that the fresh data to this block are in the snapshot volume. We read the data block from the snapshot volume. Otherwise, the data are 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 Bloom filter technique to quickly determine which volume we will read data from [35].

4.3 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. Because any I/O request could be started from any valid LBA address, the fragmentation algorithm may deal with partial data of a block. In our current implementation, we simplify this process by aligning the LBA address 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.

5 Experimental Methodology

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

5.1 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 Fig. 2. Two PCs are interconnected using the Intel's NetStructure 10/100/1,000 Mbps 470 T switch. One of the PCs acts as an application server running benchmarks with the 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.

TABLE 1
Hardware and Software Environments

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 4 (Linux Kernel 2.6.9)
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)

In order to test our snapshot implementations 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 4 (Linux Kernel 2.6.9) and Microsoft Windows XP Professional on the PCs. On the Linux machine, the UNH iSCSI initiator [24] is installed. On the Windows machine, the Microsoft iSCSI initiator [36] is installed.

On top of 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 4. 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 4.

5.2 Workload Characteristics

Workloads that drive the performance evaluation are very important to provide meaningful and accurate performance results. We choose a set of standard benchmarks that are widely used in the research community and industry. The first benchmark we select is IoMeter [37] that was originally developed by the Intel Corporation and latter maintained and further developed by the Open Source Development Lab (OSDL). It has been registered at SourceForge.net since 2001. IoMeter is a flexible and configurable benchmark tool that is widely used in industry and the research community. It can be used to measure and characterize the performance of a mounted file system or a block device. As indicated by the original document, IoMeter does for a computer's I/O subsystem what a dynamometer does for an engine. It is particularly suitable to our evaluation here because it measures performance under a controlled load and can be configured to emulate the disk or network I/O load, or can

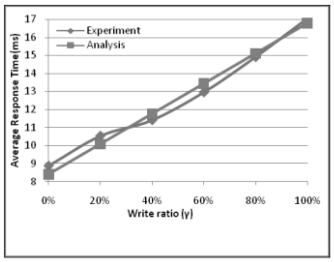


Fig. 4. Response time validation for COW.

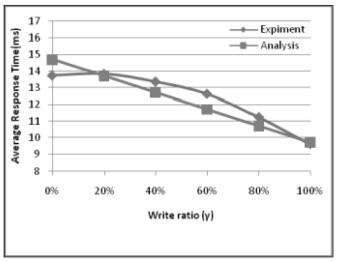


Fig. 5. Response time validation for ROW.

be used to generate entirely synthetic I/O loads. Specifically, we can change write ratios, the proportion of write IOs, address distribution of IOs, and block sizes, which allow us to provide a comprehensive evaluation of COW and ROW performances.

Another popular standard benchmark is PostMark that is widely used as file system benchmark tool written by NetApp, Inc., [38]. It measures the 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 4 KB.

To evaluate how the two snapshot techniques affect database performance, we have also chosen two typical benchmarks that run on databases. TPC-C is a well-known benchmark used to model the operational end of businesses where real-time transactions are processed [39]. TPC-C simulates the execution of a set of distributed and online transactions (OLTP) for a period of 2-8 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. We set up the Postgres database based on the implementation from TPCC-UVA [40]. Five warehouses with 50 users are built on Postgres database taking 2 GB storage space. Details regarding TPC-C workloads specification can be found in [39].

Our second database benchmark, TPC-W, is a transactional Web benchmark that models an online bookstore. The benchmark comprises a set of operations on a Web server and a back-end database system. It simulates a typical online/e-commerce application environment. Typical operations include Web browsing, shopping, and order processing. We use the Java TPC-W implementation of the University of Wisconsin-Madison [41] and build an experimental environment. This implementation uses Tomcat 4.1 as an application server and MySQL 5.0 as a back-end database. The configured workload includes 30 emulated browsers and 10,000 items in the item table.

6 Numerical Results and Discussions

Using our implementations and the experimental settings described in the previous sections, we carried out experiments to measure snapshot performance. 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.

6.1 Validation of the Analytical Model

Our first experiment is to verify and validate our theoretical analysis in Section 3. We ran IoMeter benchmark on a 4 GB disk partition with NTFS installed for 20 minutes and measured the results in terms of average I/O response time. The buffer size is 4 KB and all I/O operations are randomly generated. We compared our measured results with the analytical results from (1) and (2) by substituting D and σ with the values of 8.4 and 4.95 ms, respectively. The values of D and substituting D and σ are taken from the average values of 100 I/O experiments. Fig. 4 shows such comparison and validation for COW snapshot with different write ratios. As shown in this figure, our analysis matches very well with experiments. The two curves show exactly the same trend while changing the write ratio. The maximum error is less than 6 percent indicating high accuracy of our analytical model. Fig. 5 shows the same validation of ROW snapshot. The results are similar to the COW case implying high accuracy of the model.

6.2 Performance Evaluation

Our second experiment is to study how write ratios affect the performances of two snapshot technologies. We ran IoMeter benchmark with six different write ratios ranging from 0 to 100 percent for three snap_block sizes of 1, 2, and 4 KB. Fig. 6 shows the results in terms of average I/O response time for 1 KB snap_block size. As shown in this figure, the performance trends of COW and ROW are consistent with the results by using our analytical model, as shown in Figs. 4 and 5. That is, the response time of COW increases and the one of ROW decreases as write ratio increases. This fact demonstrates that our analytical model can quickly and correctly predict the relative performance of ROW and COW. Specifically, the performance of COW decreases as the write ratio increases. This is because higher write ratio will incur more additional I/O operations for COW as we mentioned before. On the contrary, the performance of ROW increases as the write ratio increases. In other words, ROW performs better when write ratio is high because ROW does not have the extra I/O operations of COW and redirects all

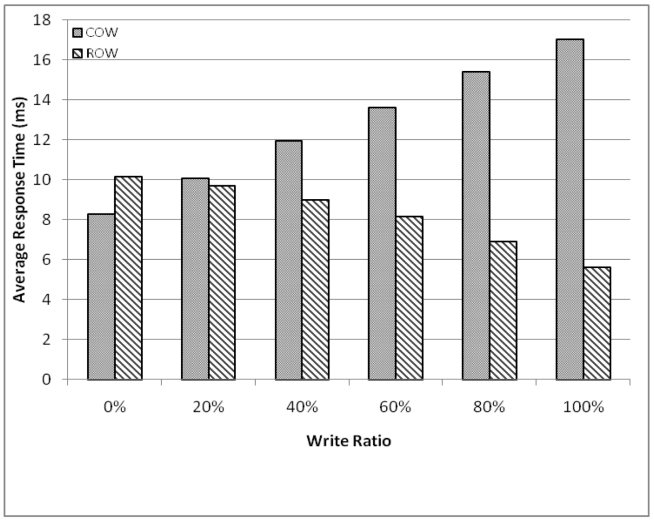


Fig. 6. Average I/O response time comparison for 1KB snap_block size of IoMeter benchmark.

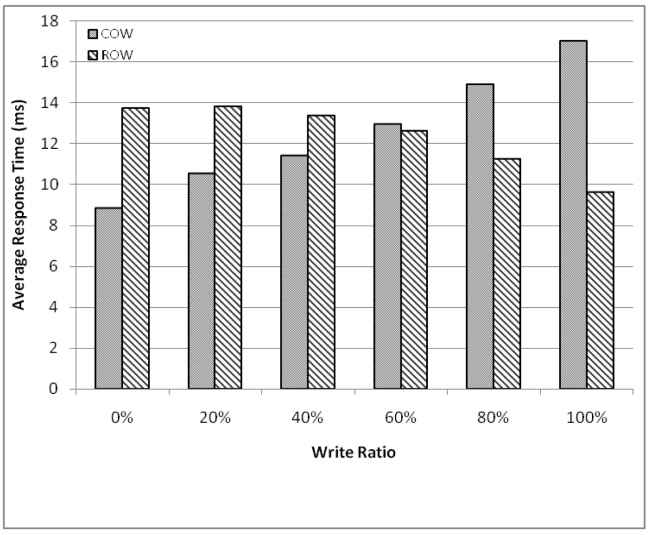


Fig. 8. Average I/O response time comparison for 4KB snap_block size of IoMeter benchmark.

new write IOs to the snapshot volume. In addition, we also observed that ROW has better performance than COW when write ratio is greater than 20 percent which is the threshold of performance turnover between COW and ROW.

Figs. 7 and 8 show the similar results for 2 and 4 KB snap_block sizes, respectively. The only difference is that the threshold of performance turnover for 2 KB snap_block is 40 percent and the one for 4 KB snap_block is 60 percent. In other words, large block sizes make COW perform better than ROW for a large range of write ratios. This result is well under our expectation. Recall that COW copies the original data block upon the first write to the block, which is the major overhead of COW snapshot. When block size is large, the chance that subsequent write operations addressed to the same block becomes high. Consider a 4 KB block as an example. When a write operation changes the first 1 KB of the block, COW makes a copy of the original block (4 KB) to the snapshot volume. Subsequent writes to the second KB, the third KB, and the fourth KB of the same block can be done in place with no copying necessary. As a result, the overhead of COW is amortized by large block sizes and subsequent overwrite to the block.

In order to further validate our analysis above, we plot overwrite ratios for different snap_block sizes with 40 percent random writes of IoMeter. Fig. 9 shows the overwrite ratio of COW as a function of block sizes. We observed that increasing snap_block size will increase the overwrite ratio. As we discussed in Section 2, overwrite I/Os to a block do not need to copy the original data to the snapshot volume. As a result, the threshold of performance turnover between COW and ROW increases as the block size increases.

In the third experiment, our objective is to demonstrate the performance characteristics of ROW and COW for different snap_block sizes. We measured the performance

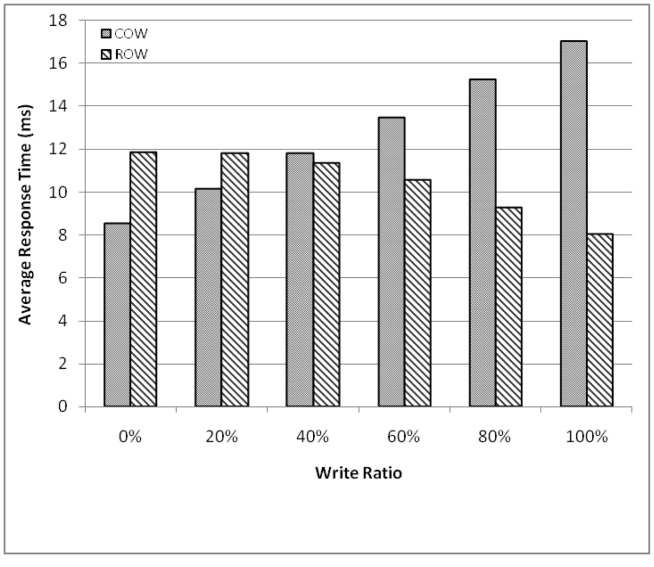


Fig. 7. Average I/O response time comparison for 2KB snap_block size of IoMeter benchmark.

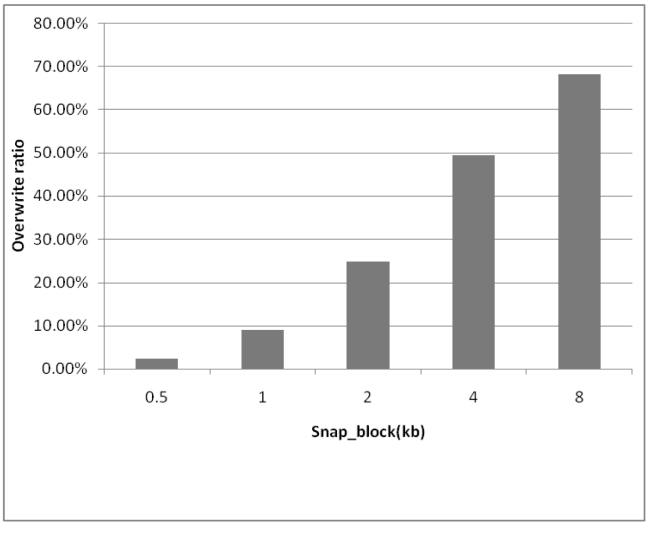


Fig. 9. Overwrite ratio of COW as a functionn of snap_block sizes for IoMeter benchmark.

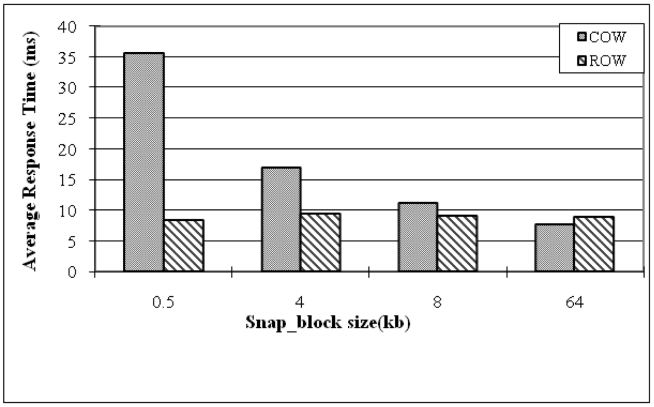


Fig. 10. Average I/O response time comparison for 100 percent random write of IoMeter benchmark.

results for different benchmarks in descending order of write ratio.

We began with 100 percent random writes for IoMeter benchmark. Fig. 10 shows measured results of the average I/O response time for four different snap_block sizes. For such a write-intensive workload, we observed that ROW performs better than copy-on-write for all snap_block sizes except for 64 KB when over write ratio becomes high. Furthermore, internal fragmentations and LBA alignments become excessive when block size is very large. For the snap_block size of 512 B, the redirect-on-write snapshot implementation performs four times better than the copy-on-write implementation. For snap_block size of 4 KB, the performance difference is about 40 percent. 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 three 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 offline and not during production time, one can benefit from such deferring of data copies. Again, as block size increases, the performance difference between the two decreases because of high chance of overwrites to the same block.

In addition, one observation in Fig. 10 is that the performance of 512 B snap_block size is not as good as other block sizes, as shown in Fig. 10. 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 64 KB data in a buffer to a disk using different block sizes at the block device. 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 8, 16, and 64 KB are not significant. Noticeable longer time is observed when the block size changes from 8 to 4 KB. There is a dramatic increase in time for the block size of 512 B. This result explains again why 512 B snap_block performs poorly in the benchmark studied.

TABLE 2
I/O Time Measurements with Different Snap_Block Sizes

snap_block size	WriteTime(ms)	ReadTime(ms)
64K	8.33	1.9
16K	8.36	2.3
8K	8.48	2.6
4K	12	3.6
0.5K	39	10.2

Measurement for PostMark has been done with 100,000 transactions on 50,000 files. For this experimental setting, the average proportion of write IOs is 99 percent. Fig. 11 shows the measured results in terms of total running time. As shown in this figure, we observed that ROW has better performance than COW which is in agreement with our expectation because ROW benefits from write-intensive benchmark. Once again, it has also been demonstrated that the performance difference of two snapshot technologies decreases as the block size increases because of the high probability of overwrites.

For the TPC-C benchmark, we measured the throughputs running on Postgres database using our iSCSI target as the block-level storage. We measured the write ratio of this benchmark to be 90 percent. Fig. 12 shows the measured results based on Windows implementation in terms of tpmC that is the number of transactions finished per minute. For the snap_block size of 512 B, 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 512 B to 8 KB. 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 8 KB, performance drops

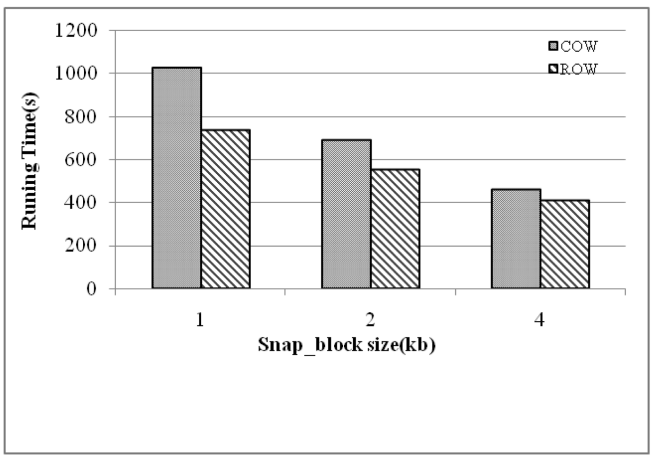


Fig. 11. Running time comparison for PostMark benchmark.

Fig. 12. Measured throughtputs comparison for TPC-C benchmark (Windows implementation).

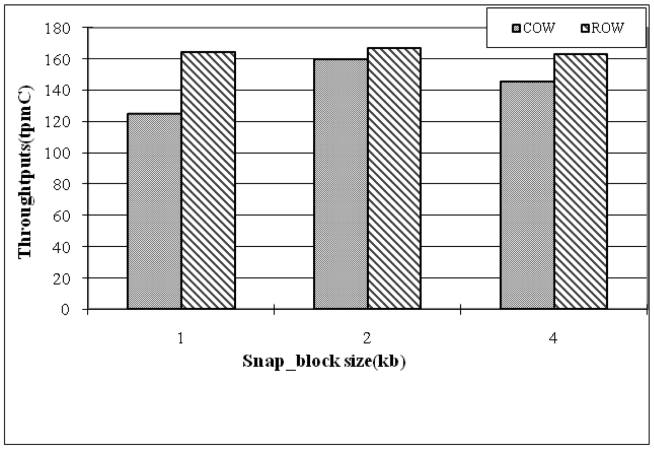


Fig. 13. Measured throughtputs comparison for TPC-C benchmark (Linux implementation).

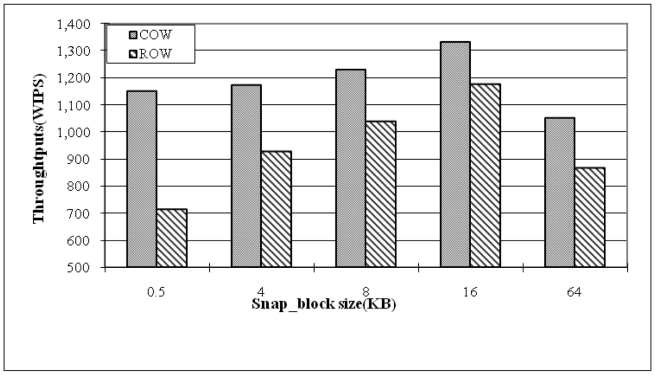


Fig. 14. Measured throughtputs comparison for TPC-W benchmark.

because of excessive internal fragmentations. Fig. 13 shows the similar results for Linux implementation.

Throughput results for TPC-W are shown in Fig. 14. We ran 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 results of previous benchmarks. For all the snap_block sizes, copy-on-write method performs much better than redirect-on-write for TPC-W benchmark, as shown in Fig. 14. The reason for this phenomenon is that the proportion of write IOs of TPC-W is about 39 percent, whereas the proportions of write IOs for the previous three

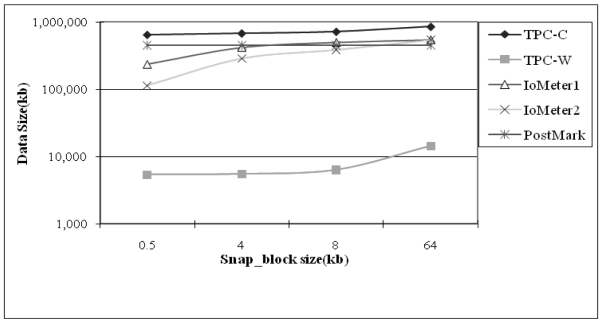


Fig. 15. Space usage of snapshot volume (in logarithmic scale).

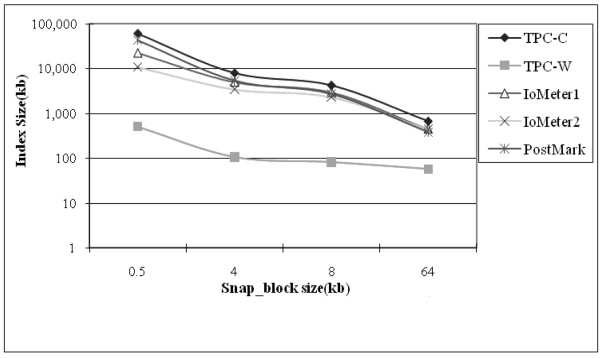


Fig. 16. Space required for index structure (in logarithmic scale).

benchmarks are all greater than 90 percent. 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. Redirect-on-write, on the other hands, suffers from performance penalty because of read merging. In addition, we also observed from Fig. 14 that the throughputs for both copy-on-write and redirect-on-write increase as the snap_block size increases except for the case of 64 KB. This observation is consistent with our discussions above: larger block sizes take shorter time to access data than smaller block sizes. For the large snap_block size of 64 KB, the performance of COW and ROW drops down because of excessive fragmentation and LBA alignment.

6.3 Storage Space Analysis

Small block sizes not only slow down I/O operations but also require large index data structure for hashing. Figs. 15 and 16 show the space used for the snapshot volume and the sizes of the index data structure for different block sizes. For 512 B block size, the index structure takes about 10 percent of the snapshot volume size, whereas for 8 KB block size, the index structure takes about half of a percent of the snapshot volume. For 64 KB block size, the index structure is less than 0.08 percent 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 IO operations in the snapshot volume. To quantitatively observe internal fragmentations, we measured

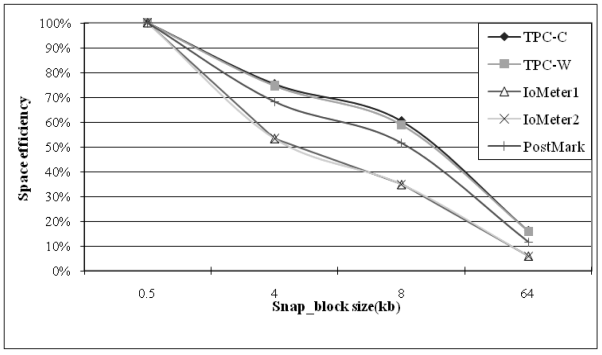


Fig. 17. Space efficiency as a function of snap_block size.

the *space efficiency* defined as the average ratio between the sizes of the write IOs coming from the host and the actual data size written in the snapshot volume because of the write IOs. The space efficiency is an indicator of the degree of internal fragmentations. The efficiency of 100 percent means that the data size written in the snapshot volume is exactly the same as the write IOs data size from the host without storage waste. A smaller efficiency implies a large internal fragmentation. To see how the internal fragmentation occurs, consider the following example. Suppose two consecutive 16 KB snap_-blocks with the LBAs of ***A*** and ***A+32***, respectively. If the host issues a write I/O of size 2 KB with starting LBA of ***A+30***, the write I/O will result in changes in both of the two snap_-blocks. The first 1 KB is written at the end of the first snap_block with the LBA of A and the other 1 KB at the beginning the second snap_block with the LBA of ***A+32***. The total internal fragmentation is 30 KB.

Fig. 17 shows the space efficiency of the two snapshot methods for different benchmarks. Note that the two snapshot methods use the same amount of storage space in our implementation. As shown in the figure, the efficiency for the snap_block size of 512 B is 100 percent without storage waste. The space efficiency drops rapidly as block size increases implying large internal fragmentations. For 64 KB block size, the efficiency drops below 20 percent. 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 (Figs. 10, 11, 12, 13, and 14) and decreasing block size for better space efficiency (Fig. 17). Therefore, there is a trade-off 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 8 KB or 16 KB block sizes depending on applications.

7 Conclusions

In this paper, we have presented a comprehensive study on implementations and performance evaluations of two differential snapshot methods: copy-on-write and redirect-on-write. A simple Markov model has been developed for quick performance estimates of the two snapshot technologies. To provide detailed and meaningful performance evaluation and comparison of the two snapshots, we have designed and implemented the two snapshot methods on Windows and Linux platforms. Extensive experiments have been carried out to measure the performance impacts of the two snapshot methods. We use standard benchmarks such as IoMeter, Postmark, TPC-C, and TPC-W to measure and compared the performances. Our numerical results uncover many important performance characteristics that were unknown before. In general, copy-on-write snapshot performs well for read-intensive workloads, while redirect-on-write snapshot performs well for write-intensive workloads. There are many trade-offs in terms of performance and cost depending on the workload characteristics of applications. Our experimental results can provide a useful guide to storage designers in making their design decisions and to storage users in planning their data protection and recovery strategies.

Acknowledgments

This research is sponsored in part by the National Science Foundation under grants CCR-0312613, CCF-0610538, and CCF-0811333. This work is also supported in part by the Natural Science Foundation of China under grant number NSFC-60736013 and Chinese 973 grant number 2004CB318203. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The authors would like to thank the anonymous reviewers for the valuable comments that improve the quality of this paper.

References

[1] A.L. Chervenak, V. Vellanki, and Z. Kurmas, "Protecting File Systems: A Survey of Backup Techniques," *Proc. Joint NASA and IEEE Mass Storage Conf.*, Mar. 1998.

[2] M. Ji, A. Veitch, and J. Wilkes, "Seneca: Remote Mirroring Done Write," *Proc. 2003 USENIX Ann. Technical Conf.*, pp. 253-268, 2003.

[3] M. Zhang, Y. Liu, and Q. Yang, "Cost- Effective Remote Mirroring Using the iSCSI Protocol," *Proc. 21st IEEE Conf. Mass Storage Systems and Technologies*, pp. 385-398, Apr. 2004.

[4] The 451 Group, "Total Recall: Challenges and Opportunities for the Data Protection Industry," http://www.the451group.com/reports/executive_summary.php?id=218, May 2006.

[5] W. Xiao and Q. Yang, "Can We Really Recover Data If Storage Subsystem Fails?" *Proc. 28th Int'l Conf. Distributed Computing Systems (ICDCS '08)*, June 2008.

[6] Novastor Corporation, "Microsoft Shadow-Copy Service and Its Role in an Organization's Total Backup Strategy," http://www.novastor.com/graphics/VSS_White_Paper.pdf, 2009.

[7] G. Duzy, "Match Snaps to Apps," *Storage*, Special Issue on Managing the Information That Drives the Enterprise, pp. 46-52, Dec. 2004.

[8] Z. Peterson and R.C. Burns, "Ext3cow: A Time-Shifting File System for Regulatory Compliance," *ACM Trans. Storage*, vol. 1, no. 2, pp. 190-212, 2005.

[9] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," *Proc. USENIX Winter Technical Conf.*, pp. 235-245, 1994.

[10] M.D. Flouris and A. Bilas, "Clotho: Transparent Data Versioning at the Block I/O Level," *Proc. 12th NASA/IEEE Conf. Mass Storage Systems and Technologies (MSST '04)*, Apr. 2004.

[11] L. Moses, "An Introductory Guide to TOPS-20," Technical Report TM-82-22, USC/Information Sciences Inst., 1982.

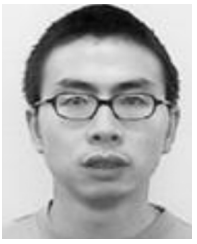
[12] K. McCoy, *VMS File System Internals.* Digital Press, 1990.

[13] 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," *Proc. 17th ACM Symp. Operating System Principles*, pp. 110-123, Dec. 1999.

[14] C.A.N. Soules, G.R. Goodson, J.D. Strunk, and G.R. Ganger, "Metadata Efficiency in Versioning File Systems," *Proc. Second USENIX Conf. File and Storage Technologies*, pp. 43-58, Mar. 2003.

[15] R. Pike et al., "Plan 9 for Bell Labs," http://plan9.bell-labs.com/sys/doc/, 2009.

[16] E.K. Lee and C.A. Thekkath, "Petal: Distributed Virtual Disks," *Proc. Seventh Int'l Conf. Architecture Support for Programming Languages an Operating Systems (ASPLOS-7),* 1996.
[17] A. Sankaran, K. Guinn, and D. Nguyen, "Volume Shadow Copy Service," http://www.microsoft.com, Mar. 2004.
[18] R. Green, A. Baird, and C. Davies, "Designing a Fast, On-Line Backup System for a Log-Structured File System," *Digital Technical J.,* vol. 8, no. 2, pp. 32-45, Oct. 1996.
[19] M. Rosenblum and J. Ousterhout, "Log-Structured File System," *Proc. 13th ACM Symp. Operating Systems Principles,* pp. 1-15, June 1991.
[20] EMC Corporation, "EMC TimeFinder Family," http://www.emc.com/products/software/timefinder.jsp, 2009.
[21] Hitachi, Ltd., "Hitachi ShadowImage Implementation Service," http://www.hds.com/copy_on_write_snapshot_467_02.pdf, June 2001.
[22] NetAppliance Corporation, "Snapshot Technology," http://www.netapp.com/products/snapshot.html, 2009.
[23] M. Mesnier, M. Wachs, R. Sambasivan, A. Zheng, and G. Ganger, "Modeling the Relative Fitness of Storage," *ACM SIGMETRICS Performance Evaluation Rev.,* vol. 35, no. 1, pp. 37-48, June 2007.
[24] UNH, "iSCSI Reference Implementation," http://unh-iscsi.sourceforge.net., 2005.
[25] H. Xiong, R. Kanagavelu, Y. Zhu, and K.L. Yong, "An iSCSI Design and Implementation," *Proc. 12th NASA Goddard/21st IEEE Conf. Mass Storage Systems and Technologies (NASA/IEEE MSST '04),* 2004.
[26] Intel Co., "Intel iSCSI Reference Implementation," http://sourceforge.net/projects/intel-iscsi, 2009.
[27] Cisco, "Linux-iSCSI Project," http://linux-iscsi.sourceforge.net/, 2008.
[28] Y. Lu, F. Noman, and D.H.C. Du, "Simulation Study of iSCSI-Based Storage System," *Proc. 12th NASA Goddard/21st IEEE Conf. Mass Storage Systems and Technologies (NASA/IEEE MSST '04),* pp. 399-408, 2004.
[29] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy, "A Performance Comparison of NFS and iSCSI for IP-Network Storage," *Proc. Third USENIX Conf. File and Storage Technologies (FAST),* 2004.
[30] S. Aiken, D. Grunwald, A.R. Pleszkun, and J. Willeke, "A Performance Analysis of the iSCSI Protocol," *Proc. 20th IEEE/11th NASA Goddard Conf. Mass Storage Systems and Technologies (MSS '03),* Apr. 2003.
[31] I. Dalgic, K. Ozdemir, R. Velpuri, J. Weber, U. Kukreja, Atrica, H. Chen, and U. Kukreja, "Comparative Performance Evaluation of iSCSI Protocol over Metro, Local, and Wide Area Networks," *Proc. 12th NASA Goddard/21st IEEE Conf. Mass Storage Systems and Technologies (NASA/IEEE MSST '04),* 2004.
[32] 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.
[33] X. He, Q. Yang, and M. Zhang, "A Caching Strategy to Improve iSCSI Performance," *Proc. IEEE Ann. Conf. Local Computer Networks,* Nov. 2002.
[34] H. Simitci, "Backups Using Snapshots," *Storage Network Performance Analysis,* pp. 280-282, Wiley Publishing, Inc., 2003.
[35] B. Bloom, "Space/Time Trade-Offs in Hashing Coding with Allowable Errors," *Comm. ACM,* vol. 13, no. 7, pp. 422-426, July 1970.
[36] Microsoft Corp., "Microsoft iSCSI Software Initiator Version 2.0," http://www.microsoft.com/windowsserversystem/storage/default.mspx, 2005.
[37] Intel, "IoMeter: Performance Analysis Tool," http://www.iometer.org/, 2009.
[38] J. Katcher, "PostMark: A New File System Bench-Mark," Technical Report 3022, Network Appliance, 1997.
[39] Transaction Processing Performance Council, "TPC BenchmarkTM C Standard Specification," http://www.tpc.org/tpcc, 2005.
[40] J. Piernas, T. Cortes, and J.M. García, "TPCC- UVA: A Free, Open-Source Implementation of the TPC-C Benchmark," http://www.infor.uva.es/~diego/tpcc-uva.html, 2005.
[41] H.W. Cain, R. Rajwar, M. Marden, and M.H. Lipasti, "An Architectural Evaluation of Java TPC-W," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture (HPCA '01),* Jan. 2001.



Weijun Xiao received the bachelor's and master's degrees in computer science from Huazhong University of Science and Technology, China, in 1995 and 1998, respectively. He is currently working toward the PhD degree in electrical, computer, and biomedical engineering at The University of Rhode Island. He holds a faculty position in the School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests include computer architecture, networked storage system, embedded system, and performance evaluation. He is a student member of the IEEE and a student member of the IEEE Computer Society.

Qing Yang received the BSc degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree in electrical engineering from the University of Toronto, Canada, in 1985, and the PhD degree in computer engineering from the Center for Advanced Computer Studies, University of Louisiana, Lafayette, in 1988. Presently, he is a distinguished engineering professor in the Department of Electrical and Computer Engineering at The University of Rhode Island, where he has been a faculty member since 1988. His research interests include computer architectures, memory systems, disk I/O systems, data storages, parallel and distributed computing, performance evaluation, and local area networks. He is a senior member of the IEEE, a senior member of the IEEE Computer Society, and a member of the SIGARCH of the ACM.

Jin Ren received the bachelor's degree in material science and engineering and the master's degree in computer science from Huazhong University of Science and Technology, China, in 1999 and 2002, respectively. He is currently working toward the PhD degree in electrical, computer, and biomedical engineering at The University of Rhode Island. His research interests include computer architecture, networked storage system, and performance evaluation.

Changsheng Xie received the BS and MS degrees in computer science from Huazhong University of Science and Technology (HUST), China, in 1982 and 1988, respectively. Presently, he is a professor in the Department of Computer Engineering at Huazhong University of Science and Technology. He is also the director of the Data Storage Systems Laboratory of HUST and the deputy director of the Wuhan National Laboratory for Optoelectronics. His research interests include computer architecture, disk I/O system, networked data storage system, and digital media technology. He is the vice chair of the expert committee of Storage Networking Industry Association (SNIA), China.

Huaiyang Li received the PhD degree in computer architecture from Huazhong University of Science and Technology (HUST), Wuhan, China, in 2006, the MS degree in computer engineering from Air Force University of Engineering, Xi'an, China, in 1999, and the associate college degree in electron information engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, 1991.