Snapshots and Continuous Data Replication in Cluster Storage Environments

André Brinkmann, Sascha Effert
University of Paderborn, Germany
Email: brinkman@hni.upb.de, fermat@upb.de

Abstract

Snapshots are an elegant means to derive instant, virtual copies of storage devices. For copy-on-write snapshots, data has only to be physically copied, if either the content of the original device or the content of the snapshot device changes. The online process of copying data between the devices can have a severe impact on application performance, especially if more than a single snapshot is derived from the original volume. This performance decrease becomes even worse, if the storage device is composed from individual storage bricks, working together as peer-to-peer storage cluster. This paper presents the architecture, the used data structures, and corresponding performance results for a snapshot implementation that is able to support an arbitrary number of snapshots from an original volume in a storage cluster environment, enabling it to perform (near) continuous data protection.

1 Introduction

A snapshot volume is a virtual copy of a storage device that shares parts of its data with the original volume. The capability of making such an immediate point-in-time copy of large data sets has become an essential feature for large scale storage environments. Snapshots are required for backup purposes, recovery from failures, and to work on real data to test new applications. Depending on the application, it might suffice to have a single read-only snapshot of an original volume or it might become necessary to build snapshot hierarchies, where it is possible to write on the snapshot volumes and to derive an arbitrary number of snapshots from the original volume and the snapshots.

Besides functionality, snapshots differ with respect to their implementation and location. *Copy-on-write snapshots* have to copy data from the original volume onto the snapshot volume if either the data on the original volume or the data on the snapshot volume changes [7][4]. After a successful copy process, the new data can be written on the original volume, respectively the snapshot volume. If data is changed on the original volume in a *redirect-on-write* snapshot implementation, no data copying takes place. The snapshot manager writes the new data to a distinct location and keeps a pointer mapping to the new location. The old data is kept intact and contains the snapshot information. The actual copy process takes place during idle times of the environment. Xiao et al. present a performance evaluation of copy-on-write and redirect-on-write block level snapshots. Their implementation is carried out on a standard iSCSI target and they show that, depending on the applications and different I/O workloads, the 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 [8]. Most block level snapshot implementations only support a single accessing host. Brinkmann et al. introduce a consistency preserving protocol for distributed implementations. Furthermore, they describe snapshot implementations that provide multiple levels of data access and present benchmark results for various snapshot scenarios [2].

Basic snapshot mechanisms are able to make virtual copies at discrete time steps. The drawback of these discrete time steps is that data that has been changed between two snapshot copies cannot be recovered in case of a failure. In contrast, *continuous data protection (CDP)* systems are able to recover data from any point in time by utilizing a mixture of a snapshot and a remote mirroring facility which writes all new data to a dedicated CDP disk and annotates this data with timing information [6][5]. The drawback of real CDP schemes is that most file systems, including many log-structured file systems, require a synchronization of their metadata from memory to disk to know which data is valid and which data is not. Synchronizing the metadata for each write access significantly reduces file system performance, not synchronizing metadata endangers data consistency.

Inside this paper, we consider copy-on-write snapshots for large scale cluster storage environments. Based on possible volume sizes of several PByte, redirection tables for

0-7695-3097-4/08 $25.00 © 2008 IEEE
DOI 10.1109/SNAPI.2007.13

redirect-on-write snapshots soon become infeasible. Assuming a block size of 4 KByte, a change rate of up to 10% of the original volume and 128 Bit entries for each redirection pointer, the redirection table inside the main memory of the server for a PByte volume would have a size of more than 500 GByte.

The used model is based on virtualization environments, where both, original volumes and snapshot volumes can reside on the same set of physical disks. The address space is divided into equally sized extents, where the size of an extent can be chosen between 4 MByte and one GByte during the initialization of the storage environment. If an extent on the original volume or the snapshot volume is changed for the first time, the whole extent has to be copied from the original volume to the snapshot volume. Therefore, small extent sizes help to keep the used physical capacity and help to decrease access latencies, while big extent sizes reduce the required amount of metadata.

The outline of this paper is as follows. After presenting the used snapshot environment in section 2, the paper presents a model to create (near) continuous data protection schemes in section 3. This (near) CDP is based on a hierarchy of snapshots that uses efficient data structures which minimize the number of copy operations. Corresponding measurement results are presented in section 4. The results include the recovery from failures (rollbacks), the influence of different extent sizes, and the performance for different numbers of concurrent snapshots. While all results are based on a fully distributed implementation, the corresponding distributed measurement results will be given in an extended version of this paper.

2 Architecture

To measure the impact of different snapshot strategies, we have selected the storage management environment *V:Drive*. V:Drive is is a network-based storage virtualization solution for Linux that is based on the randomized data distribution strategy *Share* [3]. V:Drive adapts its data layout according to the underlying storage infrastructure [2]. The concept of network-based virtualization enables the user to manage many servers more easily without having to resolve resource conflicts manually. V:Drive arranges the storage subsystems into logical disk groups which hide the details about where the data is stored.

In V:Drive, physical volumes are grouped in storage pools. These storage pools are not accessed directly, but by the abstract concept of virtual volumes which are exported to the accessing servers. The properties of a virtual volume have not to be related to the properties of the underlying storage pool. It is possible to create a virtual volume with a capacity much bigger than the capacity of the storage pool, unless the used capacity of the set of virtual volumes exceeds the physically available capacity of the storage pool. Each virtual volume can be concurrently accessed by an arbitrary number of servers.

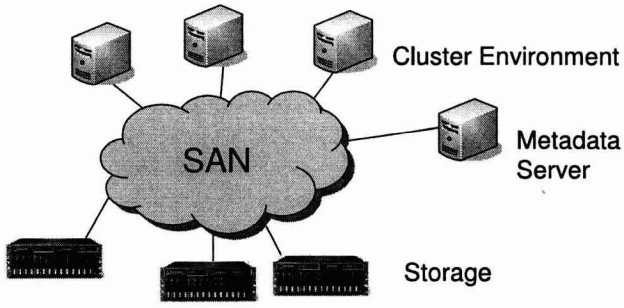


Figure 1. Distributed environment based on V:DRIVE.

The capacity of each disk in a storage pool is partitioned into minimum sized units of contiguous data blocks, so called *extents*. The extents are distributed among the storage devices according to the randomized *Share* strategy that guarantees an almost optimal distribution of the data blocks across all participating disks in a storage pool. Typical sizes of an extent vary between 4 MByte and 1 GByte.

The core component of V:Drive is a clustered *metadata appliance (MDA)* that stores information about the storage environment inside a PostgreSQL database and distributes information to the connected servers. This information includes the physical volumes, the storage pools and virtual volumes, and the set of extents which are already assigned to the different virtual volumes (see Fig. 1).

Each Linux server is running a small driver module that presents its virtual volumes to the host operating system. Each time the server accesses an address of a virtual volume that belongs to a new extent, the server has to send a request for the mapping of the extent to the MDA. The access is delayed until the reception of a valid extent location from the metadata appliance. To speed up following request to the extent, the extent location is stored inside the server.

3 Distributed Snapshot Protocols

Block level snapshots are typically handled in single server environments where the server has all information about the used and free blocks on the disks as well as about the blocks that have already been copied onto the snapshot volume. This information is not available, if multiple servers can access the same device set. In this case it is necessary to coordinate the servers. E.g., it is necessary to lock write accesses to a block while a copy-on-write is performed for this block. The coordination can either occur in

Table 1. Memory usage for extent cache and bitmap.

Extent Size vs. Volume Size		4 KB	16 KB	256 KB	4 MB	16 MB	256 MB	1 GB
1 GB	Cache	8 MB	2 MB	128 KB	8 KB	2 KB	128 Byte	32 Byte
	Bitmap	64 KB	16 KB	1 KB	64 Byte	16 Byte	1 Byte	2 Bit
64 GB	Cache	512 MB	128 MB	8 MB	512 KB	128 KB	8 KB	2 KB
	Bitmap	4 MB	1 MB	64 KB	4 KB	1 KB	64 Byte	16 Byte
1 TB	Cache	8 GB	2 GB	128 MB	8 MB	2 MB	128 KB	32 KB
	Bitmap	64 MB	16 MB	1 MB	64 KB	16 KB	1 KB	256 Byte
64 TB	Cache	512 GB	128 GB	8 GB	512 MB	128 MB	8 MB	2 MB
	Bitmap	4 GB	1 GB	64 MB	4 MB	1 MB	64 KB	16 KB
1 PB	Cache	8 TB	2 TB	128 GB	8 GB	2 GB	128 MB	32 MB
	Bitmap	64 GB	16 GB	1 GB	64 MB	16 MB	1 MB	256 KB

a totally distributed fashion or by a centralized instance. Inside this section, we present the implementation and behavior of a distributed snapshot implementation that contains a centralized coordination instance.

3.1 Metadata Handling

The proposed snapshot implementation supports storage on demand concepts, where each virtual volume and each snapshot volume only allocates the required storage capacity on the physical disks. Therefore, it is possible to create a very huge virtual volume that is mapped on much smaller back-end storage. New storage capacity has only to be integrated when the accessed capacity on the virtual volume approaches the size of the back-end storage. To map the address space of a virtual volume to physical storage, the address space is divided into extents. Inside V:Drive, each extent has got the same size and represents a continuous address space (see also section 2).

The size of an extent can be optimized in two different directions. If the size of an extent is small then each copy-on-write operation for an extent can become very fast. Based on the randomized data layout of V:Drive, smaller extents also lead to a better data distribution and load balancing. Small extents require on the other hand more metadata information to store the mapping between the virtual address space and the physical data layout. Therefore, it can become necessary to use bigger extents to reduce the traffic between the servers and the metadata appliance and to reduce the memory requirements inside each server.

In the optimal case, each server has to be able to cache the mapping of all extents of a virtual volume to their physical locations. Table 1 presents the required amount of memory inside each server for different extent sizes (from 4 KByte to 1 GByte) and different virtual volume sizes (from 1 GByte to 1 PByte). Each extent contains at least information about the extent address and the physical address as well as 16 Byte additional metadata, summing up to 32 Byte per extent (the used extent size inside V:Drive is even bigger). The required memory for the extent cache and a given virtual volume size is given in the Cache-entries of the table. Using 4 KByte sized extents and a 1 TByte virtual volume already requires 8 GByte of RAM to store all extents, which seems unfeasible. Even 1 GByte sized extents require 32 MByte of main memory to store the complete extent mapping for a 1 PByte volume. Based on the concept of locality, it is possible to build the extent mapping as a cache that only stores the location of the most recently accessed extents inside the cache. V:Drive implements such a caching concepts that restricts the size of each extent cache.

This caching concept can be in principle also applied for the copy-on-write bitmap (see bitmap row inside Table 1). Each server stores inside its copy-on-write bitmap whether an extent has already been copied from the original volume to the snapshot volume or not or whether the server does not know the current state of the extent, requiring two bits of memory for each extent. Nevertheless, the typical implementation of a copy-on-write bitmap as an array of pages, where each page stores information about thousands of extents, makes the caching of this information very impractical and all bitmaps should fit into main memory. This is putting pressure on the memory constraints for small extent sizes. Again, 4 KByte extents would require a bitmap size of 64 GByte for a 1 PByte snapshot volume, making too small extents unfeasible for big snapshot volumes.

3.2 Locking Mechanisms

Parallel accesses to a virtual volume have to be considered in distributed environments. V:Drive, like other block level virtualization solutions, does not have information about upper level file systems and is therefore not able to guarantee consistency on this level. The block level driver also does not know the content of the page cache. Therefore caching effects inside the servers can lead to an inconsistent data representations, if no parallel file system is used.

Nevertheless, a block level snapshot environment is not allowed to rely only on the consistency mechanisms of a

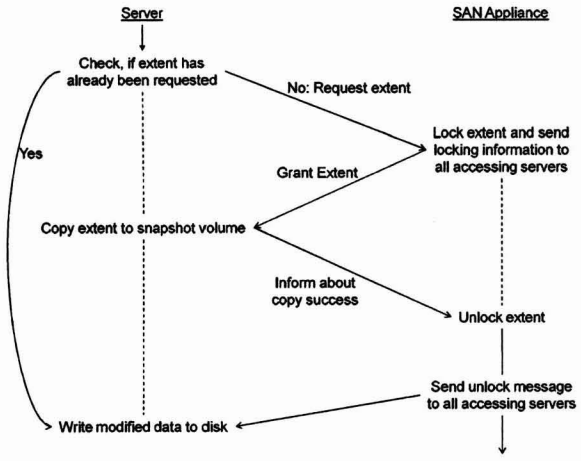


Figure 2. Distributed Locking Protocol [2].

parallel file system. The reason is that the smallest continuous unit inside the virtualization environment, which is called extent inside this paper, might span over multiple files or multiple regions of a file. If a write access is performed by a server A on a block and the corresponding extent has to be copied to the snapshot volume, another server B still might access another area of that extent.

The simplest case occurs if a write access for the same extent is performed by server B. Before copying the extent to the snapshot volume, server B has to get information from the metadata appliance concerning the location of the new extent on the snapshot volume. Instead of sending this information to server B, the metadata appliance just waits until the extent has been copied to the snapshot volume by server A and then submits the information that the extent has already been copied to server B.

A read access to the snapshot volume is more complicated. If a read access occurs for an extent on the snapshot volume and this extent has not yet been copied to the snapshot volume, the read request is normally simply redirected to the original volume. If this happens while the extent is being copied by server A, a read to a dirty area occurs. Nevertheless, this read will not produce any inconsistencies, because the information on the original volume has not yet changed. If this read occurs after the extent has been copied and after the extent has been changed by server A, inconsistencies are inevitable.

To handle read requests, we have introduce a pessimistic locking protocol in [2] (see Figure 3.2). Each time when the metadata appliance gets a request for a new extent for the snapshot, it locks the access to the corresponding address region for all servers. The information concerning the location of the new extent is sent to the requesting server A after all possibly accessing servers have confirmed the lock. No incoming I/O requests for the extent are processed during the copy process and each server buffers the requests until the extent gets unlocked by the MDA.

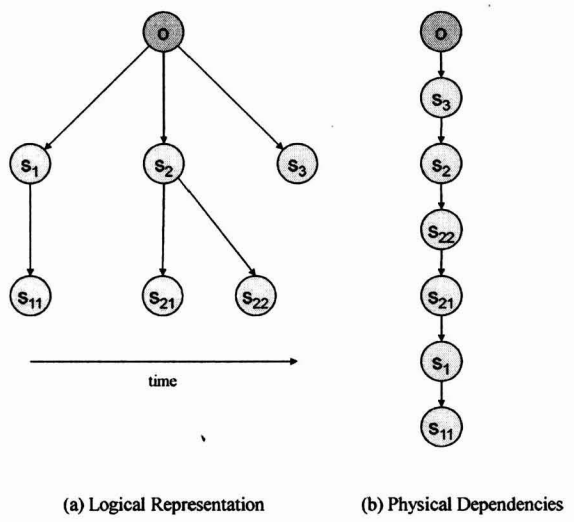
The pessimistic locking protocol produces additional latencies firstly induced by the communication overhead and secondly by blocking possible read requests. Inside this paper, we also propose a more optimistic locking protocol that does not send locking information to all servers. Instead, it simply delays write accesses of these servers to the extent by delaying the answer to the request as described above. Concerning read requests to the snapshot volume, all servers besides server A simply assume that the data on the original volume is still equal to the data on the snapshot volume and just access the original volume. After the copying process has been finished, all servers are informed that the data has been copied to the snapshot.

Inside the optimistic locking protocol it can happen that inconsistencies concerning reading data from the snapshot occur in the time span from the end of the copying process until all servers have been informed about the location of the extent on the snapshot volume. The probability of the inconsistencies is application dependent and has to be considered carefully. If the snapshot is normally not actively used, e.g. in CDP environments, these inconsistencies are very unlikely. If the snapshot is used to backup data to a tape device and the data is read very frequently, the probability of data inconsistencies increases dramatically.

An alternative of the pessimistic protocol is identical to the optimistic protocol until the end of the copy process and puts the group communication at the end of the locking process. After receiving the information about the new extent location, all servers have to acknowledge that they received the information. No server is allowed to write to the extent until the metadata appliance has received all acknowledgements and it has distributed the information that writing to the snapshot volume is allowed again. The communication overhead is similar to the pessimistic protocol, but read requests are not blocked during the copy process. On the other hand, server A has to delay its write request. A performance comparison of the different locking approaches will be given in the full version of this paper.

3.3 Snapshot Hierarchies

An important task of snapshots is to recover from accidently deleted or changed data. This recovery can, of course, only step back to the data as it existed at the creation time of the snapshot. To safe data against accidental changes, it is therefore advantageous not only to create a single snapshot once in a week, but to create new snapshots in small time intervals without removing elder snapshots too soon. Furthermore, it is often helpful to allow writing on snapshots for test purposes and to enable the creation of snapshots from snapshots.



(a) Logical Representation (b) Physical Dependencies

Figure 3. Dependencies between an original volume and derived snapshots.

The resulting logical representation of the snapshot hierarchy forms a tree-structure, where the root of the tree is the original volume and child nodes of a node represent snapshots derived from the node. Figure 3.a illustrates such a snapshot hierarchy. The first level only contains the root node, which is the original volume o. The second level contains the snapshots $\{s_1, s_2, s_3\}$, which are directly derived from the original volume o. The time line represents the relative creation time of a snapshot compared to its siblings. Therefore, s_1 has been created before s_2 and s_2 has been created before s_3. It is not possible to derive a timely relationship between s_{11} and s_{21} from the tree.

We will assume in a first step that this logical representation of the snapshot hierarchy builds the foundation of the snapshot implementation and that the snapshots in one hierarchy level are nearly independent from each other. The advantage of this assumption is that operations like *create* or *remove* of a snapshot are also independent from (most) other snapshots. When, e.g., snapshot s_2 has been created, it has only been required to build up the corresponding data structures for s_2 and link these data structures with the original volume. The same corresponds for the deletion of snapshot s_1. The snapshot can be deleted without affecting other snapshots on the same hierarchy level, while the child node s_{11} of snap s_1 vanishes together with its father. The drawback of this approach is that changing the content of an extent for the first time on the original volume o requires reading the extent from the original volume and writing it to all of its children. Assuming a huge number of snapshots from the original volume leads to a linear increase of the number of write accesses to the child nodes. This number of write accesses strictly limits the applicability of this approach.

The performance can be significantly improved, if the ordering of the snapshot hierarchy is changed slightly, as outlined in Figure 3.b. In this case, the snapshots belonging to one hierarchy level are ordered vertically and not horizontally, with the first snapshot of a hierarchy level being at the bottom of the vertical ordering. We will assume that a new snapshot s_4 is created from the original volume o. In this case, the content of o and the virtual content of s_4 do not differ at the creation time. Therefore, it is transparent for node s_3, whether it is connected to node s_4 or node o and node s_4 can become the new father of node s_3. Changes of node o have only to be forwarded to node s_4 and not to the other child nodes of the original volume, significantly reducing the number of data copies. The same is valid for snapshots derived from snapshots. It is also possible to handle these snapshots like snapshots derived from the original volume and include them into the straight line. Therefore, the number of operations that can be induced by a change to an extent of the original volume is limited to reading the extent and writing it to a single snapshot volume. Changing an extent on a snapshot volume can lead to reading it from its father volume and copying it to the snapshot volume and its child volume.

3.4 Recovering from Failures

Snapshots are often used to enable the user to experiment on its data and to recover the original volume in case of a failure to the time when the snapshot has been created. Inside this chapter, we discuss both, rollbacks for the snapshot implementation based on Fig. 3.a and based on Fig. 3.b.

The recovering seems to be simple in both cases if the father has only a single snapshot device. After the snapshot and the original volume have been unmounted on the assigned servers, all extents belonging to the snapshot replace the corresponding extents for the original device. This can be performed by a simple reassignment of the extents and

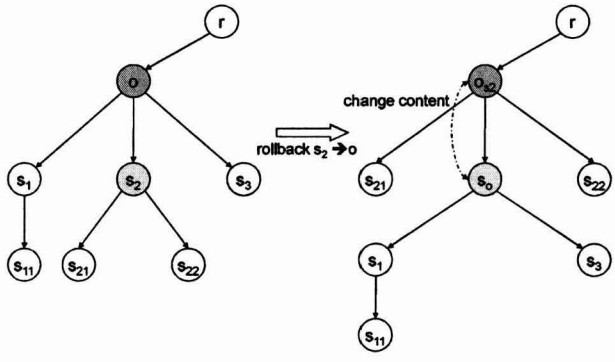


Figure 4. Recovery of a snap father volume.

without physically moving data between the two volumes. Afterwards, the snapshot is simply discarded and the original volume is reactivated (mounted again on the assigned servers).

The rollback becomes more complicated for horizontally ordered snapshots, if more than a single snapshot is part of the update process, e.g. as outlined in Fig. 3.a and the existing snapshots should be preserved. Assume that the content of o should be rolled back to the state of s_2. It is important to notice that snap volume s_3 has been created after s_2 in our example and therefore depends on changes that have been made on the original volume after the creation of s_2. If the content of s_2 just becomes the content of the original volume, s_3 becomes inconsistent immediately. In this case, s_3 and all snaps derived from s_3 have to be removed before the content of the original volume is recovered.

One solution to rollback the original volume to the state of s_2 and to keep all snapshot devices is just to rename the volumes, but to keep the relationship between all snapshots and their father volumes. The drawback of this approach is that it will never be possible to remove the former original volume, resulting in an inefficient use of disk capacity.

A more capacity efficient solutions that is able to keep all snapshot volumes is based on Figure 4. The original volume o inside the figure can be a snapshot itself, e.g. for some root volume r. The idea is as follows. We start by exchanging the content of the volume o and s_2. This can simply be done by assigning all extents assigned to s_2 to the original volume o and, in turn, assign the now superfluous extents from o to s_2. This content exchange of o and s_2 has a direct influence on the snapshots of o, resp. of s_2. All snapshots of o have now to be assigned to s_2 and vice versa. Interestingly, there have not to be done any changes concerning the former father volume r. The advantage of this solution is that the capacity of the original volume can be freed after s_1, s_{11}, and s_3 have been removed.

Concerning the rollback operation, vertically ordered snapshots based on section 3.3 behave quite different from standard snapshots. The implementation of this snapshots requires that the snapshots can be ordered in a single line. Therefore, a capacity efficient implementation of the rollback operation requires that all more recent snapshots from the same hierarchy level and their ancestors have to be deleted. Assume e.g. the snapshot ordering according to Figure 3.b and the rollback operation to recover volume o to the state of s_2. In this case, snapshot volume s_2 has firstly to be rolled back to snapshot volume s_3 and afterwards to the original volume. Therefore, the required recovery time becomes, in the worst case, linear in the number of derived snapshots and snapshot s_3 has to be removed from the snapshot list.

4 Performance Measurements

Inside this section we present various measurement results that help to understand the influence of snapshot hierarchies, extent sizes and different implementation aspects on snapshot performance. The test environment for the single server tests is based on a 2.4 GHz Intel Xeon server with 1 GB RAM. The server includes a 2 GBit QLogic 2310 FibreChannel HBA. The storage environment includes six Seagate ST373207FC hard disks bundled inside a Transtec 3000 JBOD. The server has been connected with the storage enclosure using a Brocade SilkWorm 3800 switch.

IOMeter has been used as benchmarking environment to measure the performance on the system. Analyzing the following measurement results, it has to be noticed that V:Drive supports no parallel copy-on-writes for a single virtual volume. The original volume and the snapshot volume are locked during that time frame.

4.1 Influence of the Extent Size

The influence of different extent sizes on snapshot performance and memory overhead has been theoretically discussed in section 3.1. Inside Figure 5, we present corresponding measurement results for different extent sizes. The test series is based on a single snapshot of a 10 GByte volume. The IO rate has been measured for 4 KByte random IOs and the number of outstanding IOs has been 16. The IO rate has been measured over 20 minutes for each extent size and has been divided into 30 second intervals.

The first interesting observation is that each test has got a settling time until the full IO rate can be obtained. This settling time becomes smaller for bigger extent sizes. The reason is based on the snapshot implementation. The 10 GByte virtual volume can be, e.g., divided into ten 1 GByte extents. Each time when a random IO accesses an extent for the first time, the extent has to be copied to the snapshot

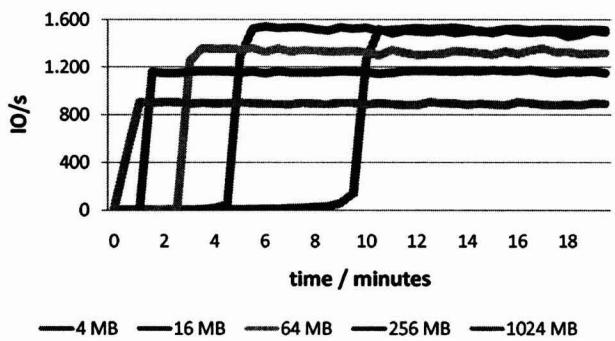


Figure 5. Influence of the extent size.

volume. During this time frame, no other IO is taking place for the original volume. On the one hand, the I/O rate is very small (even 0 for the first time frames) until all extents of the original volume have been copied to the snapshot volume. On the other hand, the snapshot implementation is able to copy the volume very efficiently to the snapshot volume.

Another extreme can be seen for 4 MByte extents. Each first hit to an extent only triggers a copy operation for 4 MByte of data. The time to complete such a miss is much smaller and the I/O rate during the settling phase is much higher than for 1 GByte extents, ranging from 4 IOs/s in the beginning up to 140 IOs/s after 9 minutes. Nevertheless, the settling phase also takes much longer in this case.

Comparing the settling times according to Figure 5 is misleading in a way that IOmeter always finishes all outstanding IOs before it returns from a test. Therefore, the first 16 random IOs for the 1 GByte extent test are only reflected by a time frame of 30 seconds inside the figure, while the real test took more than 4 minutes for these IOs. Nevertheless, when considering the real execution times, the total ordering of the settling times stays constant.

The higher IO rate for smaller extent sizes after the settling time is based on the random data distribution schemes inside V:Drive. The probability that the extents are not evenly distributed about all six disks of the test environment increases with the extent size. Therefore, some disks have to handle more IOs, which reduces the overall performance.

4.2 Snapshot Hierarchies

To show that our assumptions from section 3.3 hold, we implemented the proposed data structures and strategies inside the storage virtualization solution V:Drive [1]. First we measured a system with a growing number of snapshots ordered horizontally (as in Figure 3.a) and compared the results with vertically ordered snapshots (as in Figure 3.b). Inside this test series, we have measured the throughput while writing sequential 32 KB blocks to a 10 GByte volume. The number of outstanding IOs has been 16. Figure 6 shows the results for horizontally and vertically ordered snapshots.

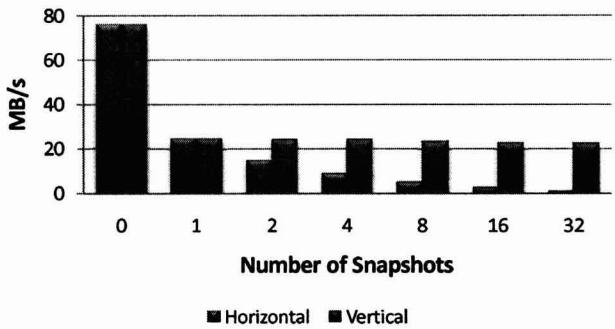


Figure 6. Measurement results for horizontally and vertically ordered snapshots.

It can be seen that the performance of the horizontally ordered snapshots decreases dramatically when taking more snapshots from the original volume. The reason is based on the large number of extent copies which have to be performed for each first access to an extent. Each extent that is changed on the original volume has to be read once from the volume and has to be copied to each snapshot. This results in a overhead proportional to the number of snapshots.

The vertically ordered snapshots show a completely different behavior. The throughput is decreasing there as well, but with a small sub-linear factor. Here only the most recent snapshots depends on the original volume, so it is only necessary to make one copy of each changed extent. The decreasing throughput is a result of a small overhead for handling the snapshots.

4.3 Recovery

The time to rollback an original volume to the state of a snapshot can be split into the time required to unmount the volumes from the servers, the time to update the data structures inside the metadata appliance and the time to mount the volumes again on the servers. Inside this section, we present measurement results concerning the update of the data structures inside the MDA. The measurements have been performed on a server with a 2.13GHz Intel Core2 6400 CPU with 2 GB RAM, SuSe Linux 10.2 with a 2.6.18.2 Kernel, and PostgreSQL 8.1.5 as database.

To rollback a virtual volume to the state of a snapshot, the extents of the snapshots have to be (virtually) moved to the virtual volume in two steps. Firstly, all extents on the virtual volume that already have copies on the snapshot have to be deleted. Afterwards all extents of the snapshot have to be moved to the virtual volume. All this can be

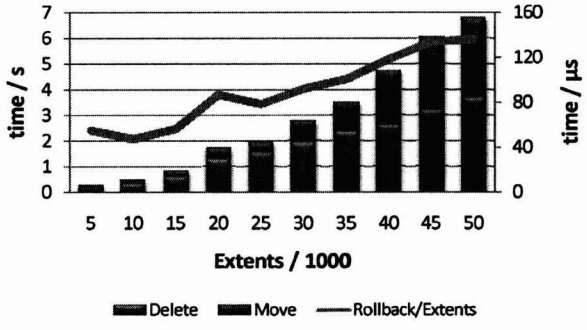


Figure 7. Recovery times.

handled inside the MDA after turning the volume and the snapshot off-line. We have measured how much time a roll-back takes for 5,000 to 50,000 extents on the virtual volume and the snapshot. Assuming an extent size of 64 MByte, the number of extents is sufficient to represent fully used snapshots with a volume size of up to 3.2 TByte.

Figure 7 shows the time in ms to delete the extents of the original volume and to move the extents. It took significant more time to delete the extents than to move them from the snapshot to the original volume, being the result of a more complex SQL-Statement. Furthermore, the time per extent is also increasing slowly for a growing number of extents. For 50,000 extents, the environment needs more than two times the time per extent as for 5,000 extents.

4.4 Near CDP

The measurement results presented in section 4.2 indicate that additional snapshots of an original volume do not influence the performance of a storage environment. Therefore, it seems to be possible to use a near infinite number of snapshots of an original device to generate near continuous data protection (CDP) schemes. Inside this section, we investigate the behavior of such snapshot series on performance. The measurement environment is identical to the environment presented at the beginning of this section. The capacity of the volume has been increased to 100 GByte.

Figure 8 presents measurement results for a raw block device and for a volume containing an EXT3 file system. For each test series, the first snapshot has been generated after one minute, afterwards, one additional snapshot of the original volume has been generated each minute. Before generating the snapshot device, V:Drive calls *freeze_bdev()* inside the Linux kernel for the original volume to lock the filesystem and to force it into a consistent state. Afterwards, V:Drive is blocking requests to the original volume until the snapshot has been started. For both test series, the generation of a snapshot leads to a small performance decrease that

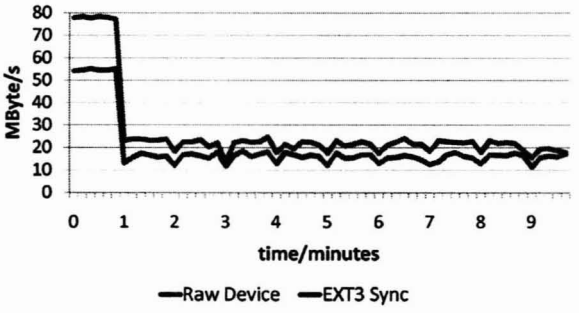


Figure 8. Influence of CDP on performance.

is based on the locking that takes between 1 and 4 seconds. The performance after the creation stays nearly independent from the number of generated snapshots.

5 Conclusions

Inside this paper, we have presented data structures, implementations, and measurement results for different snapshot schemes. While the performance of an environment is heavily influenced by the creation of the first snapshot of a volume, the proposed data structures ensure that additional snapshots have nearly no further influence on the throughput or IO rate. Inside V:Drive, the first snapshot leads to a sequential throughput of only one third of the optimal throughput (which is optimal), while the IO rate for small IOs can even become much worse (which can be significantly increased by parallel copy-on-writes for one volume). In summary, the proposed data structures lead to an efficient snapshot usage that can help to implement near continuous data protection.

References

[1] A. Brinkmann, S. Effert, M. Heidebuer, and M. Vodisek. Influence of Adaptive Data Layouts on Performance in dynamically changing Storage Environments. In *Proceedings of the 14th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, February 2006.

[2] A. Brinkmann, S. Effert, M. Heidebuer, and M. Vodisek. Realizing Multilevel Snapshots in Dynamically Changing Virtualized Storage Environments. In *Proceedings of the 5th International Conference on Networking (ICN)*, April 2006.

[3] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, Adaptive Placement Schemes for Non-Uniform Distribution Requirements. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 53–62, August 2002.

[4] EMC. EMC SnapView/IP - Fast Incremental Backup. Technical report, EMC Corporation, 2001.

[5] L. Jun, S. Ji. Wu, and X. Wei. Design and Implementation of a Continous Data Protection Systeme for a SAN environment. In *Proceedings of the 3rd International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, pages 33 – 40, September 2005.

[6] Mendocino. Mendocino Recovery One - Recovery Management for Enterprise Data Protection. Technical report, Mendocino Software, September 2005.

[7] Veritas. VERITAS FlashSnap. Technical report, Veritas, 2002.

[8] W. Xiao, Y. Liu, Qing Yang, J. Ren, and C. Xie. Implementation and Performance Evaluation of Two Snapshot Methods on iSCSI Target Storages. In *Proceedings of the 23rd IEEE Conference on Mass Storage Systems and Technologies (MSST)*, May 2006.