SNPdisk: An Efficient Para-Virtualization Snapshot Mechanism for Virtual Disks in Private Clouds

Lei Yu, Chuliang Weng, Minglu Li, and Yuan Luo, Shanghai Jiao Tong University

Abstract

Using virtualization techniques, a cloud administrator can consolidate, distribute, and manage computing resources through a network from a central console. Existing open source virtual machine monitors, such as Xen, do not support virtual disk snapshot and rollback, which is important for clouds, especially for virtual cloud storage and computing resource management. Although traditional snapshot technologies may be directly applied to virtual cloud storage, they are inefficient and not fully reliable. Most seriously, the techniques impose limitations on the cloud environments. In this article, we present SNPdisk (snapshot disk), a para-virtualization snapshot mechanism, which uses a novel sparse tree scheme to implement reliable and efficient snapshots. Experiments with a prototype system demonstrate its high performance; it is especially useful for private clouds.

In private clouds, the virtual disk (VD) is one of the primary subsystems. It is exposed to the cloud services and virtual machines (VMs) as a block-level disk. The VD is important for computing resources managements as well as virtual cloud storage.

As an important component of cloud computing, VM techniques have been widely used to distribute and manage computing resources, especially in private clouds. Open-source VM techniques, such as Xen, have received extensive attention in many academic and commercial clouds. From another point of view, each VM in a cloud can be regarded as a software container that bundles a set of virtual hardware resources, an operating system (OS) and its applications/cloud services, into one or more files and VDs called *VM images*. Encapsulation of VMs as VDs/files makes it easy to manage cloud services and computing resources. A cloud administrator can clone, snapshot, modify, and rollback the VMs, which contain various cloud services, just like the common files via networks.

The development of clouds imposes higher requirements on this manageability of computing resources. Unlike traditional Network Computing Architecture [1] (NCA), in a cloud, the client's computer resources have been further concentrated in the cloud platform [2]. People have even become accustomed to using clouds' services, data, and computing resources just form "thin" devices [3], such as on netbooks, tablet computers (e.g. iPad) and cell phones. Those clients have only minimal computing resources, for example: storage capacities of iPhone4 and iPad just are 16-64GB, whereas a standard PC has 500GB-2TB. A cloud platform must efficiently provide services and computing resources on demand to users, via a network.

Point-in-time snapshots have become essential for VDs, because they are widely used for cloud service management, VM management, data protection, failure recovery, and even data sharing. In clouds, a large number of VMs may run the same OS and cloud services, which can be effectively deduplicated by the snapshot mechanism of VDs. Unfortunately, non-commercial versions of virtual machines, such as open source Xen, do not support copy-on-write (CoW) cloning, increment snapshot, or rollback. Administrators can snapshot the states of CPU and memory of VMs, but not the state of the file system.

Techniques such as LVM, Ventana [4], and Parallax [5, 6] can take snapshots of VDs, but there are performance problems. Using those outside techniques also introduces more limitations on storage environments and even on the guest OS. Moreover, some of these mechanisms are not fully reliable, such as LVM.

In this article, we present SNPdisk, a novel para-virtualization snapshot mechanism for VDs in private clouds. To improve the performance and save mapping tree space, we propose a special sparse tree to implement an efficient CoW snapshot in limited space. We constructed the prototype system based on Xen. Experiments demonstrate its advantages in input/output (I/O) performance, and theoretical analysis shows its integrity and validity.

In the next section, we discuss related work. The architecture of SNPdisk is then introduced. Next, the detailed design and implementations are described. The article gives a theoretical analysis of SNPdisk's integrity and its validity. Then we provide our experimental setting, workload characteristics, results analysis, and experience discussions. Finally, we conclude our article.

Related Work

Citrix [7] proposed Logic Volume Management (LVM) as the snapshot infrastructure of Xen; this is a traditional snapshot mechanism for UNIX. Reference [4] presented a new approach, named Ventana and used in VMware, based on the

0890-8044/11/$25.00 © 2011 IEEE

Network File System (NFS); it takes snapshots on the file system level. Parallax [5, 6] provides a novel outside snapshot mechanism based on *blktap*, a device component of Xen. Tape libraries and virtual tape libraries (VTLs) can also support snapshots and incremental backup of VD, but they do not support CoW.

LVM

A particularly appealing method is to employ LVM volumes as the storage snapshot mechanism. LVM provides dynamic CoW snapshots of logical volume [8]. The big advantage is that this method greatly reduces the amount of time services such as databases are down during backups because a snapshot is usually created in a fraction of a second.

However, there are problems. First, it puts extra restrictions on bottom-level storage systems, which must support LVM. Second, LVM is inefficient and uses lots of VMM memory for the mapping data. More important, it has been proven that under some conditions the data cannot be correctly recovered after a failure [9].

Conversely, SNPdisk works inside Xen, and is suitable for all environments Xen supports. Moreover, SNPdisk employs a sparse tree scheme to implement the CoW backup, the advantages of which have been demonstrated by tests. Finally, SNPdisk is more reliable and secure.

Ventana

As an NFS variant, Ventana provides version control at the granularity of files [4].

Ventana is essentially a file system, quite different from block-level storage. It just supports snapshot and rollback of files. It also sacrifices the generality of cloud platforms, because some OSs, such as Windows, do not support NFS boot.

Ventana's versioning information and metadata are both stored in a traditional database. As the number of snapshots increases, its performance will inevitably degrade. In contrast, SNPdisk employs a sparse tree structure, and we demonstrate with experiments that this is efficient.

Parallax

Parallax, a virtual storage system, was specially designed for Xen. It creates a big storage pool across multiple physical machines, and then carves it up into several VDs. Those VDs are exposed to VMs as *blktap* devices, a kind of virtual disk device in Xen. The mappings from virtual address in the I/O request from VM to physical address are stored in a three-level tree. This is a full tree, so the amount of index data is large and difficult to handle in memory, so there is extra I/O to read and write index data. Its full tree architecture also reduces the performance advantages of the storage system on sequential I/O, important on both solid-state disks (SSDs) [10] and common hard disk drives (HDDs). The snapshot data is fragmented, and is not regrouped as sequential segments for I/O requests.

Most important, Parallax is essentially also an outside mechanism. The versioning mechanism for *blktap* devices runs on the *privileged VM*, and the I/O requests are not directly identified and distributed on the *standard VM* end.

Parallax employs a radix tree to map between virtual addresses and snapshot file addresses, while SNPdisk employs a sparse tree. Tests demonstrated SNPdisk is more efficient in both speed and space. Moreover, SNPdisk is an inside snapshot mechanism of Xen.

System Architecture

This section introduces the background and design of SNPdisk's architecture.

Overall Architecture of Xen

Our prototype system is designed inside the open source Xen. Xen has been widely used to construct the infrastructure of private cloud platforms, such as Amazon's EC2 and Eucalyptus.

An open source Xen virtual system consists of several items that work together to manage computing resources of clouds: the VM manager (VMM), a privileged VM, standard VMs, and domain management control.

The VMM, or *hypervisor*, is the basic abstraction layer of software that sits directly on the hardware below any OSs. It is responsible for CPU scheduling and memory partitioning of the various virtual machines running on the hardware device. It also provides inter-VM *memory sharing*, and the *XenBus* and *virtual interrup*t mechanisms for virtual I/O devices (also used by SNPdisk). Their functions are similar to mechanisms in a traditional OS, but virtualized, and work between VMs.

The privileged VM, also called *Domain0*, is a modified Linux kernel, and is a unique VM that has special rights to access physical I/O resources as well as interact with the other VMs running on the same physical machine. It can also access the network storage devices via the iSCSI protocol and distribute them to other VMs.

A standard VM, also called *DomainU*, running on a Xen VMM is similar to a traditional physical server. In private clouds, the OS and services as well as the configuration (files) of distributed computing resources are entirely bundled into standard VMs.

Finally, *domain management control* supports the overall management and control of the virtualization environment and exists within the privileged VM. Although open source Xen itself does not provide a network central console for the management, there are many third-party open source tools that can consolidate them via the network conveniently.

Para-Virtualization Architecture

Currently, there are several ways to implement disk virtualization in private clouds. Two leading approaches are full virtualization and para-virtualization. The former is designed to provide total abstraction of the underlying physical system, and creates a complete virtual computer system for the guest OS, with virtual BIOS, virtual memory space, and virtual devices for each VM. This incurs performance penalties.

In contrast, para-virtualization uses a split driver architecture, which enables the guest OS to directly access various parts of the raw hardware.

As a para-virtualization mechanism, SNPdisk deploys a front-end driver in the guest OS and a back-end module on the privileged VM, which does dynamic I/O management and address mapping.

SNPdisk Structure

If we employ a traditional snapshot storage system, the front-end driver of Xen first groups the I/O requests into a single request as much as possible. However, the grouped address region may not be as contiguous as it seems for an outside snapshot storage system, so the "optimization" may be invalid and result in extra overhead.

In SNPdisk, the front-end directly transports the I/O requests from the file system of the VM without "optimization." Its back-end driver consists of three modules: the back-end receiver, snapshot *mapping module*, and physical driver as shown in Fig. 1.

In Fig. 1, when the back-end receiver gets a request from the front-end, it calls other components to read/write physical disks and sends the results back. The front-end of SNPdisk first creates a memory unit to store the I/O request and its I/O data or data space into the *shared ring*, an inter-VM I/O sharing and

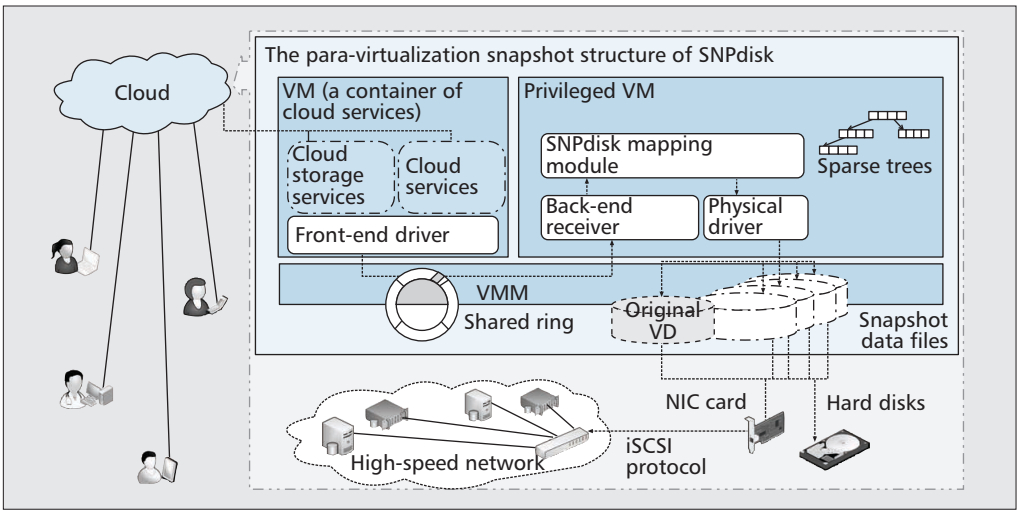

Figure 1. *The para-virtualization snapshot structure of SNPdisk.*

controlling mechanism of Xen, and informs the receiver via *virtual interrupt* and *Xenbus*. The back-end receiver examines the memory unit and passes the requests to the *mapping module*. That module translates the requests into a series of I/O instructions for the physical disk according to the mapping rules, which are stored as a sparse tree structure. Those instructions call the physical driver to read/write on various resources, such as an original VD (an online image file, a partition, an iSCSI disk, or a physical disk) and its snapshot files.

Design and Implementation

In the para-virtualization snapshot architecture of SNPdisk, the primary responsibility of the *mapping module* is to translate virtual addresses from a VM into the addresses of the original VD and snapshot files. The mapping data are stored in a sparse tree structure.

We have implemented the snapshot mechanism in Xen 3.4.1. Because it does not place extra requirements on the storage environment, the SNPdisk system is suitable for any environments Xen supports.

Index Tree

For a CoW snapshot mechanism, mapping data can be difficult to manage. Even if it employs a tree structure, a large index is still problematic. Tree nodes should be cached in memory as much as possible, especially high-level nodes. In Parallax, the full tree of a snapshot of a 2-Tbyte VD requires 8 Gbytes of space. A large snapshot requires either a lot of I/O or a lot of memory, but memory is limited in kernel space.

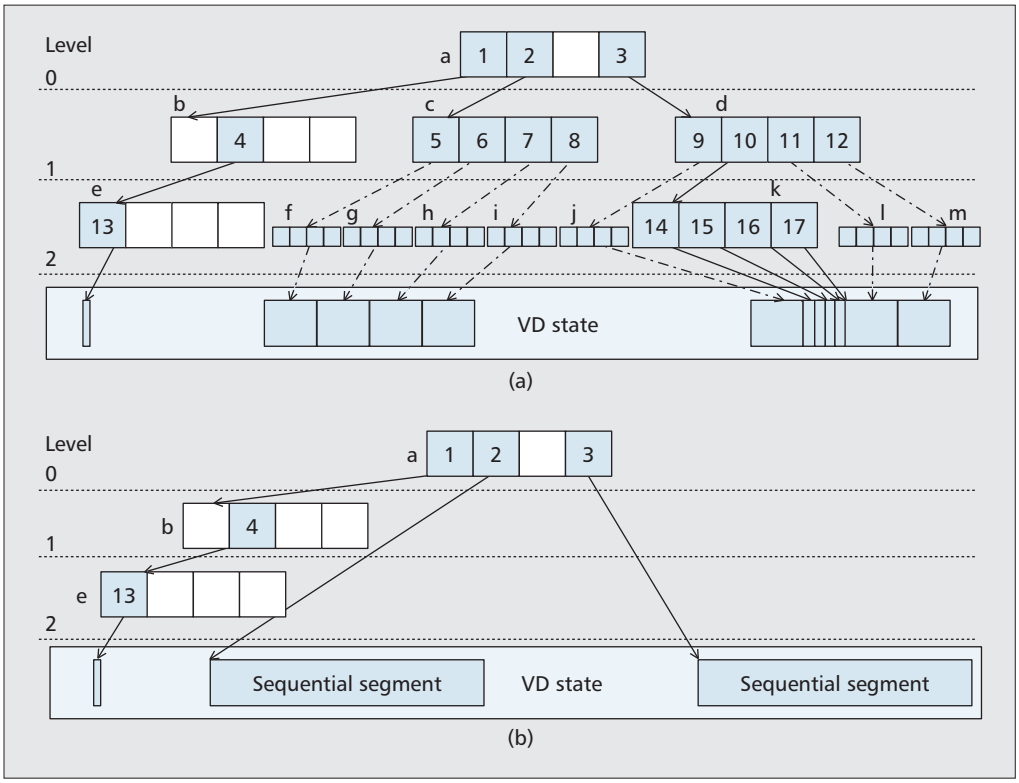

Figure 2. *A simplified example of a three-level sparse tree. In a), index units 2 and 3 point to full child trees, which can be regrouped as shown in b).*

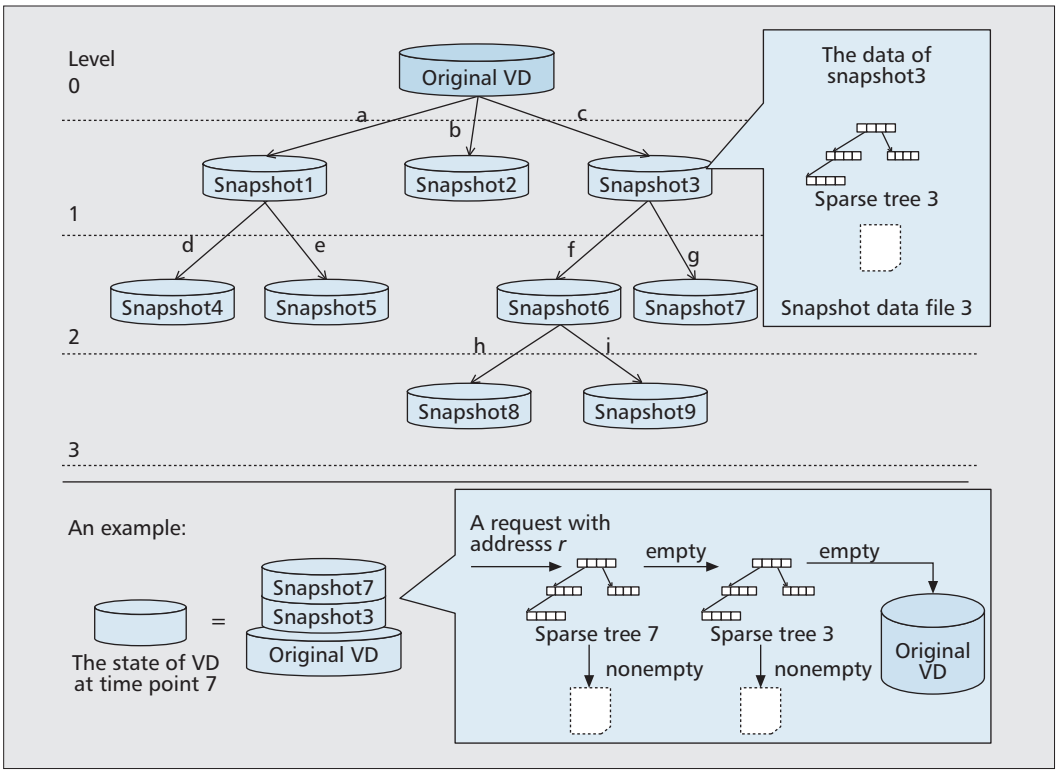

Figure 3. *The hierarchy structure of multilevel snapshots. The data of each snapshot involve a sparse tree and a snapshot data file.*

In addition, the traditional index tree reduces the performance advantages of sequential I/O, because the data in snapshot files are not regrouped to be contiguous.

The mapping data of a SNPdisk's snapshot are stored in a multilevel index tree. Each node of the tree is 4 kbytes to match the block size and memory page size. Each node contains 128 units of size 32 bits, and each unit holds a 31-bit address and a sign bit, which indicates whether this address is a real address of the snapshot data file or a child node address of the tree. We call these *index units* with sign bit 0 and *data units* with 1.

Figure 2a shows a three-level four-node index tree. Although it is simplified, the figure is still quite large, so we omit nodes f, g, h, i, j, l, and m, which are full stored nodes like node k. Each blue unit is an index unit or data unit, while each white unit is empty, which means the data at this address is not modified after the snapshot. Units 1, 2, 3, 4, and 10 are index units; units 5–9 and 11–13 are data units. In this tree, each unit represents a virtual address that is used in I/O requests from the VM, while the data stored in data units are real addresses of the snapshot data file.

Sparse Tree

We assume the *mapping module* uses the simplified index tree in Fig. 2a, where the address space of the VD is 0–47 and the size of the VD is 48 blocks. When it receives a read request for a block with address 4, it calculates a path on the tree, which is 1st-2nd-1st (the first unit labeled 1 of the node on level 1, the second unit labeled 4 of the node on level 2, etc.). If a unit on this path is empty, it will read directly from the original VD. But in Fig. 2b, it gets the unit labeled 13, which points to a 4-kbyte block of the snapshot data file. A four-level tree has ability to store the mapping of a terabyte VD, with a space overhead of 1 Gbyte (2^7 addresses of a 4-kbyte space is stored per tree node, four levels, so 2^{7*4} *4 kbytes = 1 Tbyte). Even if the tree is full, the space overhead is just 32 bits * 1 Tbyte/4 kbytes = 1 Gbyte. We can also use a five-level tree with 63-bit addresses for a 128-Tbyte VD), whereas that in Parallax is 4 Gbytes.

To further reduce the storage overhead, we present a novel sparse tree structure to store the mapping data. This method also avoids the degradation of the performance on sequential I/O. In our design, as the snapshot data increase, when an index unit points to a full child tree, the data in the snapshot file mapped by this child tree will be regrouped and sequentially stored in the original order, and the child tree will be released as shown in Fig. 2b.

In Fig. 2b, each data unit on level 0 points to a 64-kbyte continuous storage space in the snapshot data file; those on level 1 point to a 16-kbyte one; on level 2, to a 4-kbyte one. On receiving an I/O request from the VM OS, the *mapping module* will parse out an address r from the request. We assume the first block address of the VD is 0 and $r = 52$. It will calculate a path on the index tree quickly, which is 4th-2nd-1st (the units: 3-10–14). In Fig. 2b, the *mapping module* first visits the 4th unit of the root node a, and finds it is a data unit, so it must point to a 64-kbyte continuous storage space (at the address r' of the snapshot data file). The *mapping module* calculates the offset o based on the address r', where $o = 4$. If the I/O request is to read 16 kbytes of data, the request will be grouped and translated into a new request on the snapshot data file, where the read address is $r' + o$ and read size is 16 kbytes. So four blocks of data (16 kbytes) will be read sequentially at once.

Standard SNPdisk utilizes a four-level sparse tree, and the data units on levels 0, 1, 2, and 3 point to 8 Gbytes, 64 Mbytes, 512 kbytes, and 4 kbytes continuous storage space of the snapshot data file, respectively.

Multilevel Snapshot

In SNPdisk, an unlimited number of snapshots can be taken for a VD. It utilizes a hierarchy in which the states of VD are derived from the original VD (root node).

In Fig. 3, to back up a state of the original VD, a logic rela-

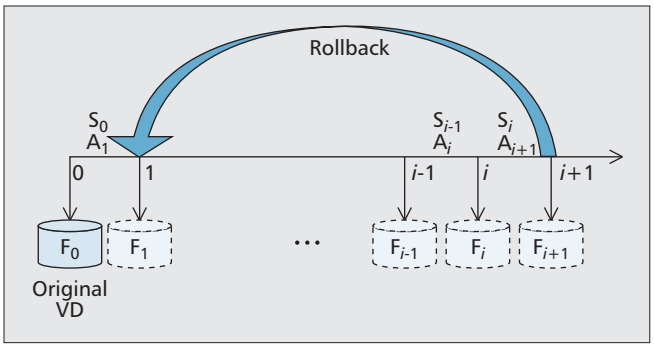

Figure 4. *The states of a virtual disk on the time axis.*

tionship (snapshot1 and line a), branch, directly from the original VD (root node) is first created at time point 1, and the original VD is set to "read-only." Subsequently, a new relationship, such as branch b, is created directly from the original VD when the original VD has been cloned or shared between two VMs. In another case, the VM is rolled back to its original state and creates a new node (snapshot3), and this is followed by some branches (e.g., branches f, g, h, and i). Every snapshot stores a point-in-time state of the VD that can be rolled back or shared. It results in a hierarchical structure of snapshots as shown in Fig. 3.

In Fig. 3, each snapshot node represents a state of the VD that can be rolled back or shared, but except for the root node, it is just an incremental backup and cannot independently constitute a complete VD. For instance, snapshot7 represents the VD state at time point 7, which is derived from the original VD and snapshot3, but the data of snapshot7 just involves sparse tree 7 and data file 7 rather than a copy of the original VD. When the VD runs at state "snapshot7," the *mapping module* will first search for a virtual address for each request in sparse tree 7; If the corresponding unit is nonemp- will search for it in sparse tree 3. If that is also empty, the module will do it directly from the original VD.

Theoretical Analysis

This section presents an analysis of the validity and integrity of the snapshot and rollback mechanisms of SNPdisk.

In Fig. 4, each point on the time axis represents a state of the VD, which can be backed up: a snapshot. The state can be described by two sets, the block address set A and the corresponding data value set V. The VD state at time point i is denoted F_i, which is a *function* relation between A and V. In addition, F_i is a *discrete function*, so it can also be considered as an *ordered-pair set* $F_i = \{(a_1, v_1), \ldots, (a_n, v_n)\}$ (for n block addresses of the VD).

The Snapshot Mechanism of SNPdisk

Unlike LVM, what is stored in an SNPdisk snapshot is newly written content rather than a backup of original data.

We denote a block address *set* as A_i, which includes the block addresses overwritten between time point i-1 and i. Correspondingly, the block values are denoted by *set* V_i.

SNPdisk's snapshot data, stored in a sparse tree and a snapshot data file, at time point i is defined as $S_i = F_{i+1} \uparrow A_{i+1}$, where $F_{i+1} \uparrow A_{i+1}$ means a *restriction* of function F_{i+1} to A_{i+1}. It records the contents that are written between time point i and $i + 1$. Obviously, S_i is also a discrete function, so it can also be considered as an ordered-pair set. The data values (*set* V_i.) are stored in a snapshot data file, while the mapping addresses A_i are stored in a sparse tree structure.

The state of the VD at time point i+1 can be generated by

$$F_{i+1} = (F_i - F_i \uparrow A_{i+1}) \cup S_i, \quad (1)$$

if and only if we have its previous state F_i and current snapshot data S_i. In particular, when the state of original VD is denoted by F_0, then

$$F_1 = (F_0 - F_0 \uparrow A_1) \cup S_0. \quad (2)$$

We assume that the VD's working state is switched (rolled back or data-shared) to time point i in Fig. 4;, then a new state F_{i+1} will be created (where F_i is a pervious snapshot, and it had been set to "read-only"). When the *mapping module* receives a write request with address r from VM, r is added into set A_{i+1}, and the corresponding data value v_r is written into V_i. When the *mapping module* receives a read request with address r, it will, in turn, search *set* $A_{i+1}, A_i, \ldots, A_1$ for r. The search method has been introduced earlier. If $r = r^e$ is found in $A_e (1 \le e \le i + 1)$, it also gets (r^e, v_r^e) from the snapshot data S_{e-1}, and then v_r^e is read and resent back. If r has not been found in the sets, it is directly read from the original VD, state F_0.

Thus, the historical states of the VD do not depend on the current state. Each state of the VD at any time point can be generated from all its previous snapshot data and original VD state. For instance, state F_i can be generated if and only if we have $S_0, \ldots, S_{i-1}$ and F_0. Even if a system crash, hardware failure, or power interruption results in an abnormal state of F_{i+1} (the current state), the historical states can be securely recovered and create a new working state without any data errors.

On the other hand, [11] presented a similar method to demonstrate that the LVM's CoW snapshot is not secure in some cases. For LVM's CoW snapshot, an abnormal state of F_{i+1} may cause the historical states to be irretrievable.

The Analysis of Multiple-Level Snapshots of SNPdisk

Essentially, the time axis in Fig. 4 should use logical time rather than absolute time. We redefine a state of the VD at logical time point h as F_h^ℓ.

For the example in Fig. 3, we denote $F_0^\ell = F_0$, $F_1^\ell = F_3$, and $F_2^\ell = F_7$. For convenience, the snapshot data on the logical time axis also are denoted $S_0^\ell = S_{0,3}$, $S_1^\ell = S_{3,7}$, where $S_{i,j}$ represents the snapshot data that record the content written in a "same" VD between absolute time point i and j.

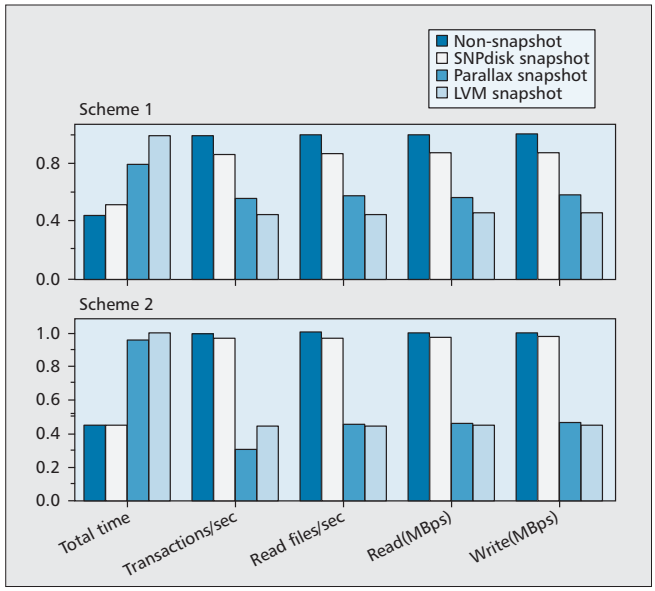

Figure 5. *The normalized experiment results of schemes 1 and 2.*

	Snapshot	Total Time(s)	Trans(ps)	Read (files)	Read (Mbytes/s)	Write (Mbytes/s)
Experiment scheme 1	Non-snapshot	19	2631	1318	45.89	47.73
	SNPdisk snapshot	22	2272	1138	39.64	41.22
	Parallax snapshot	34	1470	736	25.65	26.67
	LVM snapshot	43	1162	582	20.28	21.09
Experiment scheme 2	Non-snapshot	234	219	110	3.58	5.03
	SNPdisk snapshot	240	213	107	3.49	4.91
	Parallax snapshot	512	68	50	1.64	2.3
	LVM snapshot	532	97	49	1.57	2.21

Table 1. *The results of Postmark benchmark for the performance evaluation.*

In Fig. 3, if we have original VD state F_0' and all previous snapshot data on a "one-way" path from root node to the target node, we can securely recover the states at any time point in the multilevel snapshot tree. This demonstrates the validity and integrity of the multilevel snapshot mechanism in SNPdisk.

Evaluations and Discussion

This section presents the methodology we use to quantitatively study the I/O performance of SNPdisk compared to Parallax and LVM.

Experimental Setup

Our benchmark was run on a Dell Poweredge 1900 with dual 1.6 GHz quad-core CPUs and 2 Gbytes of RAM. The infrastructure of the private cloud was Xen 3.4.1; the OS in both privileged VM and VMs was Ubuntu 8.10 with kernel 2.6.18. The hard disk was a 10000 RPM 145-Gbyte SCSI disk with an Ext2 file system.

To isolate other effects from the OS or other applications, we used two raw partitions in our experiments. All results reported here were measured using those partitions. The test results of multilevel snapshot are not included, but do not significantly differ from the results shown.

I/O Performance

Postmark [12] is an I/O intensive benchmark that can simulate the common operations of private clouds, such as email server and online transaction processing (OLTP). It measures transaction rates by creating a large pool of continually changing files. The read and write file size can be configured, and the number of transactions can be specified. We used two experimental schemes. One's workload was set to 1000 files and 5000 transactions, and the other was set to 10,000 files and 50,000 transactions. File sizes ranged from 1 to 50 kbytes. Each test was run at least five times, and the averages are shown.

Table 1 presents the results, while Fig. 5 shows the normalized results for schemes 1 and 2. They demonstrate the excellent performance of SNPdisk, especially for the intensive I/O. The I/O performance of SNPdisk is almost double that of LVM and 1.5 times that of Parallax. The sparse tree of SNPdisk saves ; thus, more mapping can be completed in memory without extra I/O. Since Postmark spent almost all its time reading and writing small files, where mapping is the main load for CoW snapshot systems, the result is not surprising.

Discussion

Before SNPdisk, we also tried to construct a sparse tree mapping mechanism based on open source Cleversafe, a well-known cloud storage system. In Cleversafe, data dispersal is a key feature; the data of each "disk" are dispersed to multiple servers for "high" performance and security.

However, we found that it had low read and write performance at smaller request sizes, even without CoW snapshots. When request size is 4–8 kbytes, the performance on gigabit Ethernet is less than 60 percent of the standard iSCSI storage under heavy workloads. We think the net communication and data synchronization inevitably add to the time overheads. Even on a high-speed network, this data dispersal architecture is not feasible for constructing a high-performance snapshot mechanism. So we employed the architecture inside Xen rather than data dispersal.

Moreover, although SNPdisk has good performance, if the requirements of snapshot, data sharing, and data protection are not imposed, such as for some free cloud storage services, SNPdisk is not recommended, because there are some simpler methods that provide cloud storage services directly to VMs and even user clients (e.g., NFS).

Conclusions

In this article, we present a novel para-virtualization snapshot mechanism for VDs in private clouds. We have implemented a prototype based on Xen. In order to reduce the CoW mapping time, a special sparse tree is proposed for the address mapping. It considerably reduces the amount of index data and extra I/O. Experimental results show that it has good performance, and is especially feasible for online transaction processing applications, a common workload for private clouds.

We also give some mathematical analyses of the validity and integrity on SNPdisk, and use it to demonstrate the validity of the multi-level snapshot mechanism in SNPdisk.

Acknowledgment

Dr. Chuliang Weng, the corresponding author of the article, would like to acknowledge the support from National Key Basic Research and Development Plan (973 Plan) of China (No. 2007CB310900) and National Natural Science Foundation of China (Nos. 90612018, 60970008, and 90715030). This research was also partially supported by the Research Fund for the Doctoral Program of Higher Education of China (No. 20100073110016) and the Shanghai Jiao Tong University Innovation Fund for Postgraduates.

References

[1] L. Zahn, T. Dineen, and P. Leach, *Network Computing Architecture*, Prentice-Hall, Inc., 1990.
[2] B. L. Haibo, L. K. Sohraby, and C. Wang, "Future Internet Services and Applications," *IEEE Network*, vol. 24, no. 4, July-Aug. 2010, pp. 4–5.
[3] T. M. Chen, "Life in the Digital Cloud," *IEEE Network*, vol. 25, no. 1, Jan.–Feb. 2011, pp. 2.
[4] B. Pfaff, T. Garfinkel, and M. Rosenblum, "Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks," *Proc. USENIX Symp. Networked Systems Design & Implementation*, May 2006, pp. 353–66.
[5] D. Meyer *et al.*, "A. Parallax: Virtual Disks for Virtual Machines," *Proc. ACM SIGOPS/EuroSys Euro. Conf. Comp. Sys. (EuroSys* , May 2008, pp. 41–54.
[6] A. Warfield *et al.*, " Parallax: Managing Storage for A Million Machines," *Proc. USENIX Hot Topics in Op. Sys.*, June 2005, pp. 1–11.
[7] Citrix Systems Inc., "Xen 3.0 Virtualization User Guide," http://bits.xensource.com/Xen/docs/user.pdf.
[8] Red Hat, Inc., " LVM Architectural Overview," http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual Cluster LogicalVolume_Manager/LVM_definition.html.
[9] W. Xiao and Q. Yang, " Can We Really Recover Data If Storage Subsystem Fails?," *Proc. Int'l. Conf. Distrib. Comp. Sys.*, June 2008, pp. 597–604.
[10] S. Dongjun, "Issues on Using SSD Under Linux," http://www.usenix.org/event/lsf08/tech/shin_SSD.pdf
[11] W. Xiao, J. Ren, and Q. Yang, "A Case for Continuous Data Protection at Block Level in Disk Array Storages," *IEEE Trans. Parallel and Distrib. Sys.*, vol. 20, no. 6, Jun. 2009, pp. 898–911.
[12] J. Katcher, "Postmark: A New System Benchmark," tech. rep. 3022, Network Appliance, Inc., Oct. 1997.

Biographies

LEI YU (anycom@sjtu.edu.cn) received his B.E degree from Northwest Institute of Light Industry, China, and his M.E. degree from Nanjing University of Technology, China. He is currently a Ph.D. candidate in Shanghai Jiao Tong University's Department of Computer Science and Engineering. His research interests are in the areas of virtualization/cloud and network storage security.

CHUNJIANG WENG (clweng@sjtu.edu.cn) is an associate professor in Shanghai Jiao Tong University's Department of Computer Science and Engineering. He has a Ph.D. in computer science from Shanghai Jiao Tong University. Currently, he leads a research group, Advanced System Laboratory, in Shanghai Jiao Tong University, and his research interests include parallel and distributed systems, system virtualization/cloud, operating systems, and system security.

MINGLU LI (mlli@sjtu.edu.cn) is a professor in Shanghai Jiao Tong University's Department of Computer Science and Engineering. He has a Ph.D. in computer science from Shanghai Jiao Tong University. He is vice dean of the School of Electronic Information and Electrical Engineering at Shanghai Jiao Tong University. His research interests include wireless sensor networks, grid computing, cloud computing, and services computing.

YUAN LUO (yuanluo@sjtu.edu.cn) is a professor in Shanghai Jiao Tong University's Department of Computer Science and Engineering. He has a Ph.D. in probability and mathematical statistics from Nankai University. His research interests include information theory, network coding, and computer security.yu