Proceedings of the 2007 IEEE
International Conference on Integration Technology
March 20 - 24, 2007, Shenzhen, China

# ESnap – A Cached Dependent Snapshot System

Guangjun Xie, Genxi Fu, Yusen Li, Gang Wang, XiaoGuang liu and Jing Liu

*Department of Computer Science*
*Nankai University*
*Tianjin, China*

{xieguangjun1980,fgx_4321, li.yusen, wgzwp}@163.com liuxg74@hotmai.com jingliu@nankai.edu.cn

***Abstract* - Snapshot technology is becoming prevalent to perform data protection and other tasks such as data mining and data cloning. To improve the performance and reliability of the traditional Linux LVM snapshot, we propose a novel cached dependent snapshot system, Esnap. Esnap decreases the total amount of data copy effectively using the data dependency among snapshot volumes. A new snapshot metadata organization scheme is designed to support massive snapshot volumes and the corresponding read / write algorithms are put forward. Also the automatic extending of snapshot volumes is implemented in Esnap to avoid the failure of the whole dependent snapshot chain due to space overflow of one snapshot. Experimental results show that Esnap has higher performance, reliability and the resource utilization rate than traditional Linux LVM snapshot system.**

***Index Terms* - *dependent snapshot, LVM, storage virtualization, automatic extending*.**

## I. INTRODUCTION

Snapshot is the instantaneous image of storage system in specific time. LVM (Logical Volume Manager) builds a virtual view of physical storage devices. It supports snapshot of LV, which is used to record the data view of LV at given time. The main idea of LVM snapshot is COW (Copy-on-Write), which means that the data are only copied to snapshot before they are updated for the first time. However, there are some problems in LVM snapshot, which depress the availability and performance of systems.

First, in Linux LVM, it consumes many resources when the snapshot is very large because the whole mapping information is kept in memory. Second, if an origin has more than one continuous snapshot, a write operation will cause more than one COW to every snapshot. It's possible to depress the performance sharply. At last, snapshot may overflow if there are many COW operations to origin.

In this paper an enhance LVM snapshot, called ESnap, is presented. In order to reduce the memory occupation of snapshot, a new organization scheme of metadata is designed. According to this scheme, all metadata are stored on disks initially. A dynamic schedule algorithm is used to load metadata into memory if needed. To improve write performance, we present a new concept – "dependent snapshot", which chains all snapshots for the same original volume according their time sequence. The COW is only performed on the recent snapshot. ESnap also has a monitor to watch the use rate of snapshot. If use rate of snapshot exceeds a threshold, an expanding process will be run to expand the size of snapshot.

## II. GENERAL SNAPSHOT TECHNOLOGIES

### A. Snapshot in Linux LVM

There are four main snapshot technologies: split-mirror, copy on demand, virtual view and incremental snapshot. The split-mirror technology [1][3] can get full copy of the original volume and has very low snapshot creation latency, but has very poor flexibility. The copy on demand technology [5] can create snapshot at any time. The virtual view technology [2][4] doesn't create full copy of the original volume. When the original volume update was dispatched, the COW (copy on write) is performed before the update operation to save the old content, the metadata of the snapshot is also updated to maintain the mapping between the snapshot volume address and the original volume address of COW chunks. The incremental snapshot technology [8][9] aims for efficient supporting for the sequential points snapshot of the same original volume. The duplicate block copying is avoided and correct snapshot content is guaranteed through an extra incremental bitmap.

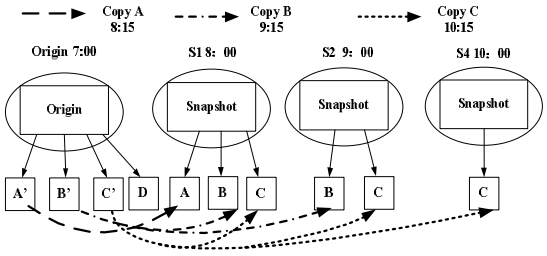


Fig. 1 Linux LVM snapshot technology.

Linux LVM[6][7] uses virtual view technology. When create snapshot, Linux LVM only allocate snapshot storage space and initialize snapshot metadata. When LVM receives an original volume update request, it checks whether the block is updated for the first time. If the block has been updated, LVM performs the update directly. Otherwise, LVM does COW first - copies old content of the block to the snapshot space and records address pair (snap addr, org addr) in mapping table, and then perform the update operation. Figure 1 gives an example. The original volume ORG has 4 blocks: A, B, C and D. The snapshot volume S1 is created at 8:00am. The user updates A at 8:15am, then LVM copies A to S1 and writes new value A' to ORG. The snapshot volume S2 is created at 9:00am. When an update request for B arrives at 9:15am, LVM copies B to S1 and S2, and write B' to ORG. At 10:00am, another snapshot volume S3 is created. At 10:15am, the user updates C, LVM copies C to S1, S2 and S3, and then write C' to ORG.

This paper is sponsored by NSF of China (No.90612001), Science and Technology Development Plan of Tianjin (043185111-14), Foundation of Tianjin Education Committee Research (No.20061016) and Nankai University Innovation Fund and ISC

### B. LVM1 Metadata Organization

The snapshot is a container of COW chunks, but it must show users a "virtual view" with the same size as the original volume. How can we distinguish between modified and unmodified data to decide read the original data from the original volume or from the "snapshot container", and where can we find the original data in the snapshot? So the metadata maintenance is a key problem.

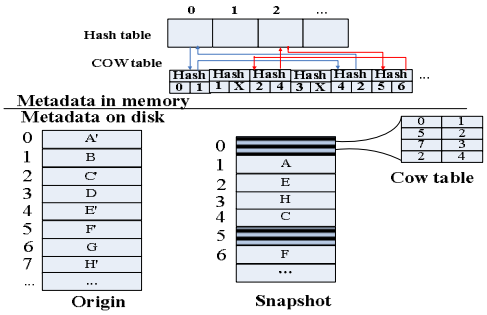


Fig. 2 Linux LVM1 snapshot metadata organization.

Linux LVM version 1 (LVM1) organizes the snapshot metadata into an "exception table" which resides in the main memory. Each exception table entry gives the original volume address and the snapshot volume location of a COW chunk. The entries are also organized into a chained hash table to speed COW chunk search. The mapping entries are also stored in the disk as a "COW table". The COW table is split into many parts, and each part is stored in the first chunk of a certain snapshot PE and records the mapping entries of the COW chunks stored in the rest chunks of the same PE. The figure 2 illustrates the memory layout and the disk layout of LVM1 snapshot metadata.

### C. LVM1 Snapshot Algorithms

The original volume write algorithm and the snapshot volume read algorithm are described below.(The snapshot volume write algorithm is implemented in LVM 2.)

```
Procedure: write_origin(buf, origin , addr)
    for each snapshot of origin do
        hash_table.search(add, &exception);
        if fail then
            call cow_data(origin, snapshot, addr);
        end if
        endfor
    write(buf, origin, addr, chunk_size);
end procedure

Procedure:cow_data(origin, snapshot, addr)
      exception.origin_addr = to_phy(addr);
      exception.snap_addr = snapshot.current_addr;
      add exception to hash table;
      write exception to disk COW table;
      read(buf, origin, exception.origin_addr, chunk_size);
      write(buf, snapshot, exception.snap_addr, chunk_size);
      snapshot.current_addr += chunk_size;
end procedure

Procedure: read_snapshot(buf, snapshot, addr)
      hash_table.search(addr, &exception);
      if succ then
read(buf, snapshot, exception.snap_addr, chunk_size);
      else
            read(buf, snapshot.origin, addr, chunk_size);
    end if
end procedure

Procedure: write_snapshot(buf, snapshot, addr)//LVM2
      origin = snapshot.origin;
      hash_table.search(addr, &exception);
      if fail then
            call cow_data(origin, snapshot, addr);
            hash_table.search(addr, &exception);
      end if
      write(buf, snapshot, exception.snap_addr, chunk_size);
end procedure
```

It is easy to see some defects of LVM1 from these algorithms.

First, because LVM1 places the whole exception table in the main memory, and the size of the exception table is linear with the amount of the updated original volume chunks, thus large volumes with heavy write workload may exhaust memory quickly.

Second, the time complexity of the function COW data is linear with the number of the snapshots of the original volume. For the original volume with a great lot of snapshots, the performance is poor.

Third, as the more chunks of the original volume are modified, the more COW chunks are stored in the snapshot volume. So deciding the initial size of the snapshot volume is great dilemma: if create a small snapshot, it maybe overflows quickly; if create a snapshot with the same size or larger than that of the original volume, most space maybe is wasted.

## III. THE DESIGN OF ESNAP

### A. New Metadata Organization

We redesigned metadata organization of LVM1 to reduce memory consumption. The main idea is that not place all the metadata in the main memory, but use cache mechanism like file systems. The detailed ameliorations are introduced below.

First, we change the metadata disk layout. The COW table entries are not arranged according to the snapshot volume address, but the original volume chunk number. The original volume chunk number needn't to be stored in the COW table any longer, because it can be derived from the table index of the entry. The figure 3 gives an illustration of the new metadata layout. We describe the metadata layout both in memory and on disk.

Second, the structure of the exception table is changed. We split the COW table into fixed length (such as 4KB) segments, and treat them as cache blocks. The in memory metadata is still organized into a hash table and each hash entry contains one COW table segment. Since not all the metadata resides in the main memory, the search algorithm must be modified. Given an original volume chunk number, we can easily compute which COW table segment it belongs to. If the segment is in the hash table, we can check whether the entry corresponding to the chunk is -1 to determine the search fails or succeeds. But if the segment is not in the hash table, we must go to check the disk COW table, return fail if the entry is -1, and return success otherwise. Whether

succeeds or not, the segment need to be read from disk to main memory and to be linked into the hash table. If the search fails, we perhaps need to do COW and update the segment. The cache replacement strategy is described in detail in subsection C.

Third, we do lazy initializing strategy through a fixed length (such as 64KB) bitmap called initialization bitmap. Each bit of the initialization bitmap corresponds to some continuous segments of the COW table, and denotes whether theses segments are initialized. So at the time snapshot initializing, we need only zero the initialization bitmap and write it to the disk. When we do metadata search, we check the initialization bitmap first. If the bit corresponding to the metadata segment is clear, we return fails directly and initialize all the segments the bit corresponds to (write -1 to disk).

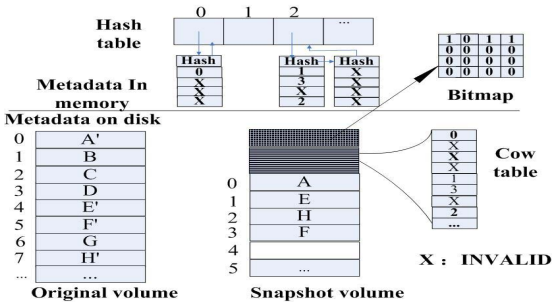


Fig. 3 ESnap snapshot metadata organization.

## B. Dependent Snapshot

As described above, the performance of the original volume write will be reduced the seriously if there are a lot of active snapshots of the original volume. Moreover, a mass of snapshot volume space is wasted because each snapshot records the same COW chunk. ESnap provides a new kind of snapshot called dependent snapshot, which only copies the old data chunk to the recently created snapshot for multi-snapshots based on the same original volume. So one original volume write operation can only cause at most one COW operation. The ESnap organizes all the snapshot volumes of one original volume into a chain, each dependent snapshot volume will be added to the tail of the chain when created.

Figure 4 shows an example of dependent snapshot, the sequence of snapshot creations and data modifies are same as Figure 1.

The dependent snapshot technology optimizes time and space complexity of original volume write operations greatly, but makes the snapshot volume read, write and delete algorithms sophisticated. We can't go to read the original volume directly if searching snapshot volume S metadata fails, we should go on searching metadata of other snapshots along the dependent chain. Only if all the snapshots follow S (include S) don't contain the required COW chunk, Enap reads the old content of the chunk from the original volume. Otherwise, Enap reads the old content from S or one of its dependent snapshots.

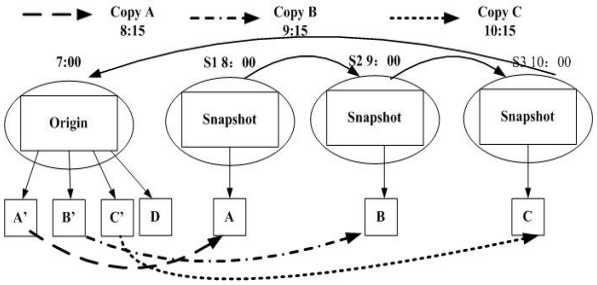


Fig. 4 dependent snapshots.

The snapshot volume write algorithm also becomes very complicated. The key is that updating snapshot S can't destroy the correct views of the snapshots which depend on S. When the user writes S, ESanp tries to found the COW chunk from S to the end of the dependent chain first. If found, ESnap perhaps need copy the COW chunk to S's predecessor T (if T doesn't contain a COW chunk with same original address), and then write the new data to S. Otherwise, ESnap do COW to T (similarly, if T doesn't contain a COW chunk with same original address), and then do COW to S and perform write request on S.

## C. Cache Replacement Strategy

When a metadata segment is loaded into the main memory, the total amount of memory consumed by all the cached segments perhaps reaches the threshold. Under this circumstance, we must select one segment which is not being used and replace it by the new segment. We implement three different metadata cache replacement strategies: round robin, FIFO and LRU. The round robin strategy maintain a counter c, when cache overflow happens, the cth hash table entry is replaced by the new segment. The FIFO strategy replaces the oldest cache segment. The LRU strategy selects the "Least Recently Used" segment like many file systems do. The performance of different strategies is determined by the storage system access pattern and metadata size.

Nowadays, there are many storage subsystem products have non-volatile memory. We can use it to optimize the performance of ESnap. The simplest idea is postponing metadata update. We needn't write the metadata segment to snapshot every time it is modified. We can write it back to disk only when it is to be replaced. Even if the system broke down before the metadata is written back to disk, we can get it from the non-volatile memory after the system restarted. This optimization will improve the original volume write performance effectively.

## D. Snapshot Automatic Extending

In order to save storage space, system administrators usually create snapshots smaller than their origin. After a great lot of COW operations are performed on a snapshot, it may overflow. Although system administrators can use command lvextend to extend the snapshot volume space, it is hard to do this timely. The case is more serious for dependent snapshot. If one dependent snapshot fails, the whole dependent chain will be invalid. Therefore we designed and implemented the automatic extending mechanism in ESnap.

When the cow_data function detects that the free space of the snapshot becomes smaller than the threshold user set, an automatic extending thread is waken up. The automatic extending thread tries to allocate storage space from VG and extends the snapshot volume, and then the cow_data function resumes doing COW.

### *E. ESnap Algorithms*

```
Procedure: find_exception(snapshot, addr, exception)
    if !test_bit(init_bitmap, addr_to_segno(addr)) then
        initialize corresponding metadata segments
        read metadata segment or do replacement
        link the segment into hash table
        return fail
   else
        hash_table.search(addr, &exception);
        if fail then
            read metadata segment or do replacement
            link the segment into hash table
            hash_table.search(addr, &exception);
       endif
       if exception.addr = INVALID then
            return fail;
       else exception.tag = COW then
           return succ;
      else
           return WRI;
     endif
   endif
end procedure

Procedure: write_origin(buf, origin , addr)
    for each snapshot of origin do
        if snapshot.type = dependent and snapshot is not the newest dependent
snapshot then
            continue;
       endif
       find_exception(snapshot, addr, &exception
       if fail then
            exception.tag = COW
           call cow_data(origin, snapshot, addr, exception);
      endif
   endfor
   write(buf, origin, addr, chunk_size);
end procedure

Procedure :cow_data(origin ,snapshot, addr, exception)
      exception.addr = snapshot.current_addr;
      write segment of exception to disk COW table;
      read(buf, origin, addr, chunk_size);
      write(buf, snapshot, exception.addr, chunk_size);
      snapshot.current_addr += chunk_size;
end procedure

Procedure: read_snapshot(buf, snapshot, addr)
    snap = snapshot;
    do_COW = find_exception(snap, addr, &exception);
    if snapshot.type = dependent then
           while do_COW <> succ and snap.next <> NULL do
                  snap = snap.next;
                  COW=find_exception(snap,addr,&exception);
           enddo
    endif
    if do_COW = succ then
            read(buf, snap, exception.addr, chunk_size);
    else
            read(buf, snapshot.origin, addr, chunk_size);
    endif
end procedure

Procedure: write_snapshot(buf, snapshot, addr)
    origin = snapshot.origin;
    snap = snapshot;
    do_COW = find_exception(snap, addr, &exception);
    while do_COW <> succ and snap.next <> NULL do
          snap = snap.next;
          do_COW = find_exception(snap, addr, &exception);
    enddo
    snap1 = snapshot.prev;
    if snap1 <> NULL then
          do_COW1=find_exception(snap1,addr, &exception);
          if do_COW1 = fail then
                if do_COW = succ then
                     copy COW chunk from snap to snap1;
                else
                     find exception and do COW for snap1;
                endif
          endif
    endif
    do_COW = find_exception(snapshot, addr, &exception);
    if do_COW = fail then
          exception.tag = UPDATED;
          all cow_data(origin, snapshot, addr, &excpetion);
          find_exception(snapshot, addr, &exception);
    endif
    write(buf, snapshot, exception.addr, chunk_size);
end procedure
```

## IV. IMPLEMENTATION

Because Linux LVM2 is unstable, we implemented ESnap based on Linux LVM1. We first migrated LVM1 from RedHat Enterprise AS 3.0 (Kernel 2.4) to RedHat Enterprise AS 4.0 (kernel 2.6), and then implemented our new technologies described in section III based on the migrated version. Some implement problems are described below in brief.

Linux LVM1 uses a data structure named lv_t to represent logical volumes. We add a list_head field to lv_t to organize all the dependent snapshots of an original volume into the dependent chain. The creation timestamp is used as the unique ID of the dependent snapshot. So finding dependent snapshot can be implemented by traversing the dependent chain forwards, and deleting dependent snapshot need traverse the chain backwards. We also add a boolean field to lv_t to distinguish between traditional snapshots and dependent snapshots. For traditional snapshots, ESnap uses old version algorithms.

LVM1 does most pv/vg/lv management including volume extending in user level, while ESnap starts the dependent snapshot automatic extending in kernel. This hot potato motivates us migrate LVM1's user level codes into kernel. The migrated functions including:

1. PV/VG/LV creation and extending, the main work of these codes is allocating storage space.
2. VGDA(Volume Group Descriptor Area) management, VGDA is the metadata of LVM.
3. We designed an array chain structure. ESnap doesn't destroy old mapping array, but add a new array to the chain.

Based on these works, the automatic extending thread can call kernel extending function to do extending. The experiment shows that, the snapshot overflow doesn't happen even if workload is heavy.

We haven't implemented snapshot volume write function till now.

## V. EXPERIMENTS

We tested ESnap on a virtual machine environment. The hardware platform is a PC with an Intel Pentium D 820 CPU, 1GB main memory and a WD1600JS-75NCB3 hard disk. The virtual machine software is VMWARE v5.5, the main memory size of the virtual machine is set as 256MB, the guest operating system is RedHat Linux AS 4.0, and the benchmark tool is iozone. In all experiments we create a 24GB original volume and a same size snapshot volume except the dependent snapshot experiment, and the chunk size is 64KB. The metadata memory occupation threshold is set as 8KB / 1GB original volume space and the metadata segment size is set as 4KB. We tested four snapshot systems: the original LVM1, ESnap with round robin cache replacement strategy, ESnap with LRU strategy and ESnap with non-volatile memory optimization (LRU strategy) which are represented by "LVM1", "RR", "LRU" and "NV" respectively.

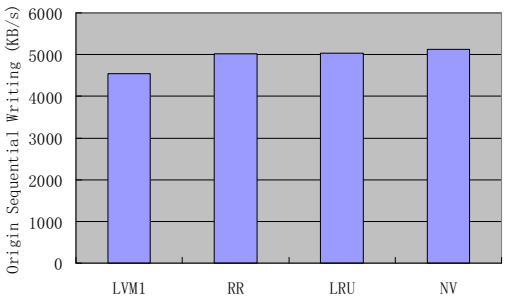


Fig.5 original volume sequential writing performance.

We first tested origin sequential write performance by creating an original volume and a snapshot volume for it, and then writing 1GB data to the original volume sequentially. Figure 5 shows the result of throughput. We can see that RR, LRU and NV have comparative performance. This result is comprehensible. In our case, 1GB / 64KB = 16K COW entries were read and then modified, and they are all continuous. So 16K / 1K (4KB / 4B) = 16 cache misses happened. ESnap did about 16 metadata initializing, 16 metadata segment reading, and 16K metadata segment writing (LVM1 did about 16K metadata block writing). There is no difference among RR, LRU and NV.

After this, we went on to do snapshots sequential reading test. As we calculate, all the metadata of the 1GB data are all in memory, metadata updating counts and extra metadata operation counts of RR, LRU and NV are all zero. Figure 6 shows the snapshot sequential reading performance. We can see that LVM1's performance is higher than others, and the performance of RR, LRU and NV is almost same. The reason is similar to the first experiment.

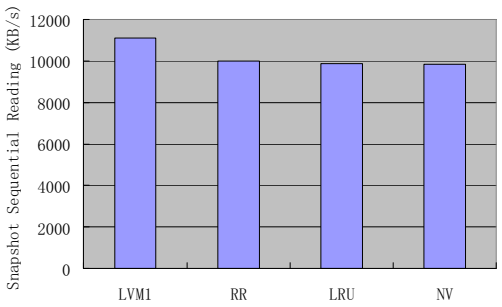


Fig. 6 snapshot sequential reading performance.

Figure 7 shows the result of original volume random writing test. LRU and RR have almost same performance because we did complete random writing. But even if the requests have high locality, the LRU is not superior to RR. The reason is that the metadata has coarse grain, the local requests usually fall the same metadata segment, so the two replacement strategy are same here. Data locality isn't equal to metadata locality. LVM1 has the best performance because it has no extra metadata operations. NV can decrease the number of metadata updating, figure 7 shows that its performance is higher than RR and LRU, and is close to LVM1 as we expected.

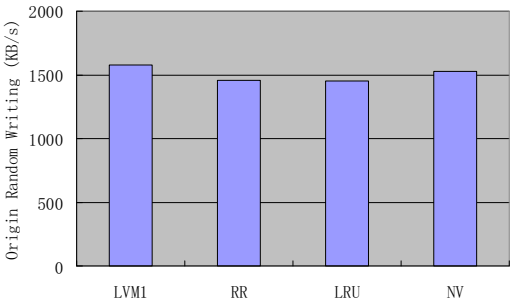


Fig. 7 original volume random writing performance.

We also tested random reading on empty snapshots. LVM1 still exhibited the best performance as figure 8 shows because its metadata all resides in memory. We closed initialization bitmap to test the impact of cache misses, so RR, LRU, NV have almost same extra metadata operation counts the slight difference is cause by the random number generator.

Because we haven't implement snapshot write till now, we didn't test snapshot writing performance.

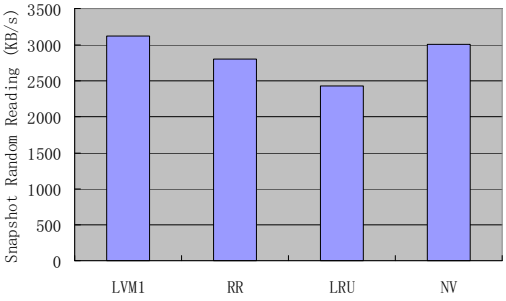


Fig. 8 snapshot random reading performance.

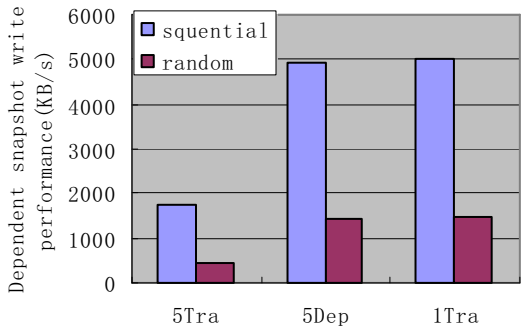


Fig.9 original volume (with dependent snapshots) sequential writing performance.

Finally, we tested writing performance of original volume with dependent snapshots. we created 5 traditional snapshots, 5 dependent snapshots and 1 traditional snapshot respectively. All three cases are based on ESnap with LRU strategy. Figure 9 show the result. Apparently, dependent snapshot technology improves the performance greatly. Of course, the performance is not improved as theoretic analysis because of the impact of the hard disk cache and the virtual machine.

## VI. CONCLUSION

In this paper, we introduced the design and implement of ESnap, a cached dependent snapshot system which can improve the performance and reliability of Linux LVM1 snapshot module. ESnap reduces the amount of data copy greatly using dependent snapshot technology. A new snapshot metadata organization scheme is designed to support massive snapshots and the corresponding read / write algorithms are presented. We also implemented snapshot automatic extending function through migrating metadata management code to kernel and enhance the reliability of ESnap. Experimental results show that the write performance of the dependent snapshots is greatly improved compared with the traditional LVM1 snapshots. The new metadata organization improves the scalability and availability greatly and pays a little in read/write performance.

Shah proposed an optimization of LVM snapshot [10], which can improve performance 18% - 40% compared with the traditional method. But if users want to extend the original volume or snapshots, the method will be disabled completely. The Blue Whale system implements iterative snapshot mechanism[11]. ESnap can achieve the same object by creating two simultaneous dependent snapshots simply and setting the older one as a read-only volume.

Future work will focus on then implement of snapshot write function, the further improvement of COW performance though multi-thread copy technique, performance optimization though non-volatile memory (such as asynchronous COW, virtual pointer strategy, and so on), performance optimization though snapshot volume layout and the optimization of VGDA data layout.

## REFERENCES

[1] EMC Time Finder[M].EMCorporation. http://www.emc.com/products/product_pdfs/ds/timefinder_1700-4.pdf,2000
[2] RAMAC Virtual Array. http://www.redbooks.ibm.com/redbooks/pdfs/sg244951.pdf
[3] Hitachi ShadowImage. http://www.hds.com/pdf/shadowimageR6.pdf, 2001-06
[4] StorageTek(tm)Snapshot Software. http://www.storagetek.com
[5] HP Storageworks Business Copy EVA, http://h18006.www1.hp.com/
[6] Hasentein M.LVM Whitepaper.SuSE Inc.http://www.sistina.com,2001
[7] AJ Lewis.LVM HOWTO. Sistina Software, Inc,2002-2003, Red Hat, Inc,2004-2005
[8] Xu Guangping, Wang Gang and Liu Jing," "Design of repetitious points incremental snapshots based on same snapshot volume", Computer engineering and applications, vol,no3, pp 413,113-115, January 2005.
[9] Li Zhong, Wang Gang and Liu Jing, "A Technology of Implementing Sequential Points Snapshot in the Storage Subsystem," Computer engineering and applications, vol. 40, no. 9, pp. 18-20, 32, March 2004.
[10]Bhavana Shah, "Disk Performance of Copy-On-Write Snapshot Logical Volumes" master degree THESIS, The University Of British Columbia, 2006.
[11]Liu Zhenjun, Xu Lu ,Feng Shuo and Yin Yang, "The Design and Implementation of an Iterative Snapshot System," Jisuanji Gongcheng Yu Yingyong, vol. 42, no.14, pp. 11-15, May, 2006.