Implementation of snapshot capability within LVM Radix Targets

Anirban Sinha
Department of Computer Science
University of British Columbia, Canada
anirbans @ cs.ubc.ca

ABSTRACT

Logical volume management (LVM) provides a higher-level view of the disk storage on a computer system than the traditional view of disks and partitions. This gives the system administrator much more flexibility in allocating storage to applications and users. Storage volumes created under the control of the logical volume manager can be resized and moved around almost at will, although this may need some upgrading of file system tools. In order to map a device into a blockstore, currently Logical Volume Manager within Linux supports three different types of targets, namely, linear, stripped and error. Abhishek[*] has designed a radix tree target for LVM with which we can pull in existing devices into the blockstore thus creating a virtual disk image (VDI) for the actual physical device. However, prior to this work there was no facility to create snapshots of existing Virtual Disk Image devices in the target. The primary work of this paper is to implement snapshot capability within the radix target so that we can duplicate read-only & read write snapshots for existing VDI's.

C.R Categories: D.4.1 [Operating Systems]: Process Management; D.4.2 [Operating Systems]: Storage Management; H.3.2 [Information Storage]: File Organization

Keywords: Logical volume management, snapshots, radix targets, parallax.

1 INTRODUCTION

There has been a tremendous interest in virtual machine technology lately. With modern hardware supporting powerful processors with high processing speeds coupled with availability of large primary memory & huge amount of secondary storage, we can now imagine running of multiple operating systems in parallel on a single hardware using virtual machines which is reminiscent of mainframe architectures. However, from a file system's perspective one major challenge is to use disk space efficiently, especially when large number of VM instances is forked. Each VM's require at least one fixed sized disk partition for utilization. Further, each of these VM does require identical software images to be available on their allocated disk space.

Linux LVMs with support for snapshots are an important concept in this direction. It allows the administrator to create a new block device which presents an exact copy of a logical volume, frozen at some point in time. Linux LVM1 has read-only snapshots. Read-only snapshots work by creating an exception table, which is used to keep track of which blocks have been changed. If a block is to be changed on the origin, it is first copied to the snapshot, marked as copied in the exception table, and then the new data is written to the original volume.

The current version of LVM, LVM2, supports snapshots are read/write by default. Read/write snapshots work like read-only snapshots, with the additional feature that if data is written to the snapshot, that block is marked in the exception table as used, and never gets copied from the original volume. It is extremely useful in the context of XEN, a virtual machine monitor. One can create a disk image, then snapshot it and modify the snapshot for a particular domU instance. One can then create another snapshot of the original volume, and modify that one for a different domU instance.

However, one major concern of LVM's is that before a write is performed on the original disk image, original data has to be copied to all the existing snapshots which overtly taxes systems running large multiple VM's at the same time. Also, LVM does not support creation of recursive snapshots.

To overcome these problems of LVM, Parallax [1], a distributed storage system, uses a radix tree based user level block storage manager to support rapid creation of snapshots & use copy-on-write mechanism for writable snapshots. It was in the lights of this idea that Abhishek designed a radix tree target in LVM similar to Parallax that effectively addresses the drawbacks related to current LVM implementation. However, prior to this work, no snapshot capability existed within his designed system. The primary concern of this paper is the design & implementation of read/write snapshot capabilities within the radix target in LVM similar to the work described in Parallax.

In this work, the author has continuously collaborated with Abhishek for fulfilling the target objectives.

The rest of the paper is organized as follows. Section 2 describes some related work in the area. Section 3 describes the blockstore architecture in detail. Section 4 describes the radix tree implementation & mapping mechanism. Section 5

[*] Abhishek Gupta (agupta@cs.ubc.ca), a current second year masters student in the DSG group.

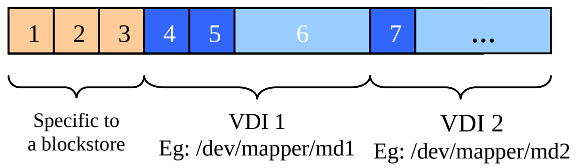
goes into describing the actual snapshot mechanism. Section 6 suggests some future work in the direction & draws the final conclusion.

2 RELATED WORK

Warfield et al describes Parallax[1], a distributed storage system that used radix tree like architecture. The same authors also describe XEN, a popular virtual machine monitor in an earlier paper[2]. Samuel T. King et al [8] describes virtual machines & a novel way to debug operating systems through snapshots & check pointing.

3 BLOCKSTORE ARCHITECTURE

The current blockstore architecture for radix implementation segments the entire blockstore into several data structures. Figure 1 describes the details of the blockstore data structure. We implemented the blockstore as a loopback device created by generating a 1 GIG flat file from /dev/zero device and attaching it to /dev/loop0. As of now, it has not been tested on actual physical device.



1: Radix superblock-keeps a magic number that shows that it's a valid superblock.
2: Radix registry-keeps track of available free blocks within blockstore.
3: Universal radix root-has pointers to all individual VDI radix superblocks.
4: Radix root block for VDI 1.
5: Superblock for VDI 1. Has pointer that points to address of VDI radix root.
6: Other radix nodes (metadata blocks) & data blocks for VDI 1.
7: Radix root for VDI 2. The sequence begins again.

Figure 1: Blockstore Data structure

4 RADIX TREE ARCHITECTURE

At present we have three levels in our radix tree implementation. We use least significant 27 bits of the logical address to select a specific offset within a radix node. The 27 bits are separated into three sets of 9 bits each for each of the three levels to select a specific offset address within that node that contains the pointer to the next node in the next level. For example bits 26-18 designate an offset into the first level node that has pointer (an address) for the next second level node. The middle 9 bits designates offset into the second level node that points to the next, i.e., third level node & so on. In our code, we use the variable "key" to keep track of the logical address. The last level radix nodes contain addresses of physical blocks where data is to be stored.

The least significant bit of each of the entries within the radix node designates whether the next level node is read-write or read-only. If bit is 1, the node (or block) is writable. Else, it is only readable. There are 512 entries for each radix node each of them pointing to a next level radix node. Each entry of a radix node is of length 64 bits. The function `getid()`extracts the address part of the field whereas the function `iswritable()` checks whether the last bit is set so that the node or block is writable. The definitions of the two functions are given below:

```
# define
getid(x)(((x)>>1)&0x7fffffffffffffffLL)

#define writable(x) (((x)<<1)|1LL)
```

The data structure `target_type` gives the entry point to kernel for radix targets in blockstore. Figure 2 shows the details of the `target_type` data structure.

```
static struct target_type radix_target = {
       .name    = "radix",
       .version= {1, 0, 1},
       .module = THIS_MODULE,
       .ctr     = radix_ctr,
       .dtr     = radix_dtr,
       .map     = radix_map,
       .status = radix_status,
};
```

Figure 2: Kernel data structure that defines entry point for radix target operations.

In Figure 2, member `.name` specifies the string to use in the `table` for `dmsetup` to recognize a radix type target. The member `.ctr` specifies the radix constructor that will get executed to initialize a radix blockstore. It reads the `table`, checks the operation to perform, like "`copy`" or "`snap`" & executes the corresponding block of code. Similarly `.dtr` points to the destructor function. "`.map`" designates the mapping function that maps from logical address to physical block address. In our case, this is the `radix_map` which uses a synchronous kernel IO function to read a physical block from disk.

5 SNAPSHOT MECHANISM

The snapshot mechanism is discussed in two sections. Section 5.1 describes the initial snap shot creation operation & provides the pseudo code for it. Section 5.2 describes the copy on write mechanism that is employed when one attempts to write to a snap shot.

5.1 SNAPSHOT CREATION

The user specifies the source VDI device for which snapshot is to be created in the `table` that is read by `dmsetup`. There after, the algorithm that is used to create initial snapshot is described in Figure 3. The actual kernel code is provided in the appendix.

```
Create_Radix_Snapshot:
blockstore, device_id

Step 1: Read vdi superblock from source
device specified in the table.

Step 2: Find the radix root address from
the superblock pointer.

Step 3: Using the source radix root
address, create a clone of the source
radix root & allocate the clone in the
blockstore.

Step 4: Allocate a new radix superblock
for the snapshot device & make it point to
the cloned radix root.
```

Figure 3: Radix Snapshot creation Algorithm

5.2 COPY-ON-WRITE ON A SNAPSHOT DEVICE

To allow for read-write snapshots we use the copy on write mechanism. For this purpose we observe that if the operation is writing, the mapping function should allocate a new block on the physical disk (which is either a data block or a meta data block), copy the contents of the old block of the original device into the newly allocated block & return the address for the new block. Hence we modify our mapping function to accommodate for this operation. Our copy on write algorithm is described in Figure 4 & original kernel code is provided in the appendix.

In this algorithm, root denotes the address of a radix node at each level & key denotes the logical block number address. The variable "op" here stands for the operation that caused the lookup to be invoked, either READ or WRITE. The argument "`phy_blk_num`" actually is a pointer to the location containing the actual address of the newly created block in the disk.

It is clear from the algorithm that on each write operation, a new disk block is allocated on the physical device & all the node entries along the path from the root of the tree to the physical block become writable. This is explained in the Figure 5 which is actually taken from the Parallax paper[1].

```
Radix_lookup_copy_on_write:
device, blockstore,height, root,
key,phy_blk_num, op

Step 1: IF height=0 & op=WRITE
Step 2: Clone the original data block
Step 3: Make the new block writable
Step 4: Store the address of the
        newly allocated block in
        phy_blk_num & also return it.
Step 5: ELSE IF op=READ & height=0
Step 6: Return the physical address of
        data block in secondary device.
       END IF
Step 7: IF height>0 & root not writable
Step 8: Clone root & make all entries
        read_only
         END IF
Step 9: Calculate new height for child.
Step 10: Calculate offset from key.
Step 11: Call this function recursively
         with new height &
         root=node[offset] & save the
         child address returned in
         child_address.
Step 12: Assign node[offset]=child_address
Step 13: Save the radix node on disk &
         make its address writable
Step 14: Return root's address.
```

Figure 4: Lookup Copy on write algorithm

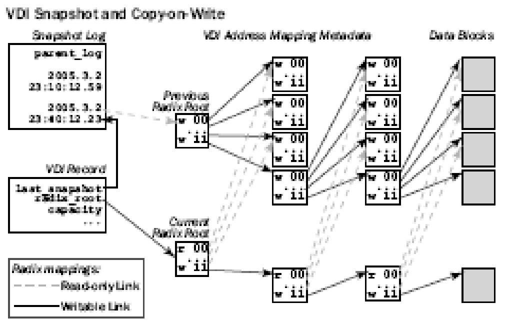


Figure 5: The snapshot mechanism (taken from the Parallax paper).

6 CONCLUSIONS AND FUTURE WORK

With our design, it is possible to create read-write snapshots of existing virtual disk images in the blockstore & also make recursive snapshots, that is, create snap shots of snap shots. We have tested this against pulling in loopback devices in blockstore & creating their snapshots. We have tested the writable logic by creating a junk file in VI editor & trying to save it on the snapshot device. We have also tested our system by creating snapshots of already existing snapshots & testing their writable nature. We have not done any benchmarking though on these writings or creation of snapshots. These are left as a future extension of this work.

Another major work that we were unable to accomplish is to make the writable logic asynchronous by using an asynchronous kernel call from the mapping function. To make snapshot creation fast & effective, we must also incorporate buffer cache mechanism for faster writes. Currently, when two or more processes try to write on the snapshot device at the same time one of them blocks. This can be prevented by using asynchronous calls along with buffer cache protection during updates. It would be really interesting to benchmark these optimizations against traditional LVM operations.

REFERENCES

[1] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing Storage for a Million Machines. In Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X), Santa Fe, NM, June 2005.
[2] Xen and the Art of Virtualization, by P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Proceedings of the Nineteenth ACM symposium on Operating systems principles, October 2003, 164--177.
[3] Efficient Disk management for Virtual Machines, Abhishek Gupta & Norman C. Hutchinson
[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. War_eld. Live migration of virtual machines. In Proc. USENIX Symposium on Networked Systems Design and Implementation, 2005
[5] LVM 2 user manual at http://sources.redhat.com/lvm2/
[6] The Linux Kernel Source Code from kernel.org
[7] LVM How to page from http://www.tldp.org/HOWTO/LVM-HOWTO/index.html
[8] Debugging operating systems with time-traveling virtual machines, by S. King, G. Dunlap, P. Chen, Proceedings of 2005 USENIX Annual Technical Conference, 2005.

Appendix: Source Codes

A. Radix Snapshot Creation

```
static int radix_ctr(struct dm_target *ti, unsigned int argc, char **argv)
{
   /* ... other codes  */

   if(strcmp("snap",argv[2]) == 0) { /*
                              “snap” is the keyword that specifies snap operation */
            int dev_id;
            vdi_superblock *target_vds = NULL;
            uint64_t snap_radix_root;

            if (sscanf(argv[3], "%d", &dev_id) != 1) {
                  ti->error = "dm-radix: Invalid device address";

                  error = EINVAL;
                  goto bad_bs;
            }
            printk ("%s: dev id = %d\n", __func__, dev_id);
            if ((target_vds = vdi_superblock_lookup(rc->dev, dev_id)) == NULL) {
                  ti->error = "dm-radix: Source device for snapshot not found in block
                                                                       store";
                  error = ENXIO;
                  goto bad_bs;
            }
            /* Ok, now we have confirmed that source device exists in blockstore */

            /* rc->vds also gives us the address of the vdi superblock */

            /* read vdi radix root address & clone the target radix root*/
            printk ("%s: target vdi radix root = %llu\n", __func__,
                                                  vdi_get_radix_root(target_vds));
            snap_radix_root = clone_radix_root (rc->dev, bs, vdi_get_radix_root
                                                                  (target_vds));
            printk ("radix root for snapshot = %llu\n", snap_radix_root);
            if (snap_radix_root == 0) {
                  error = ENXIO;
                  goto bad_bs;
            }
            /* create superblock, but do not create radix root */
            vds = create_vdi_superblock (rc->dev, bs,0x4565, dm_table_dev_minor(ti->table),
                                                                             0);
            vdi_set_radix_root (vds, snap_radix_root);

            r = write_vdi_superblock(rc->dev, bs, vds);
            if (!r) {
                  ti->error = "dm-radix: Failed to write virtual disk superblock to disk";
                  kfree (vds);
                  dm_put_device (ti, source->dev);
                  kfree (source);
                  error = EIO;
                  goto bad_dev;
            }
            else
                  rc->vds = vds;
            ti->split_io = 8;
      }

   /* other codes follow ...*/

   }
```

B. Copy on Write Mechanism

```
uint64_t radix_lookup_copy_on_write(struct dm_dev *dev, blockstore *bs, int height,
                        uint64_t root, uint64_t key, uint64_t *phy_blk_num, char rw)
{
      int      offset;
      uint64_t child;
      radix_tree_node node = NULL;
      struct page *page;

      if (height == 0)
      {
            if (rw == READ) {
                  *phy_blk_num = root;
            }
            else {
                  *phy_blk_num = writable (clone_radix_block(dev, bs, getid(root)));
            }

            return (*phy_blk_num);
      }

      /* the root block may be smaller to ensure all leaves are full */
      height = ((height - 1) / RADIX_TREE_MAP_SHIFT) * RADIX_TREE_MAP_SHIFT;
      offset = (key >> height) & RADIX_TREE_MAP_MASK;

      /* We should never get a root which is Zero. */
      if (root == ZERO) {
            *phy_blk_num = ZERO;
            return ZERO;
      } else {
            page = read_radix_block(dev, getid(root));
            if (page != NULL) {
                  node = (radix_tree_node)page_address(page);

                  if (!iswritable(root)) {
                        /* need to clone this node */
                        int i;
                        struct page *oldpage = page;
                        radix_tree_node oldnode = node;


                        page = alloc_page(GFP_KERNEL);
                        memset (page_address(page), 0, PAGE_SIZE);
                        node = (radix_tree_node)page_address(page);

                        for (i=0; i<RADIX_TREE_MAP_ENTRIES; i++) {
                              node[i] = oldnode[i] & ONEMASK;
                        }

                        block_page_put(oldpage);
                        root = ZERO;
                  }
            }
            else /* if page == NULL */ {
                  printk ("Node %llu could not be read from device \n", getid(root));
            }
      }

      if (node == NULL) {
            return ZERO;
      }
```

```
        child = radix_lookup_copy_on_write(dev, bs, height, node[offset], key,
                                                          phy_blk_num, rw);

        if (child == ZERO) {
              block_page_put (page);
              return ZERO;
        } else if (child == node[offset]) {
              /* no change, so we already owned the child */
              block_page_put (page);
              return root;
        }

        node[offset] = child;

        /* new/cloned blocks need to be saved */
        if (root == ZERO) {
              /* mark this as an owned block */
              lock_blockstore (bs);
              root = alloc_radix_block(dev, bs, page);
              unlock_blockstore (bs);
              if (root)
                    root = writable(root);
        } else {
              if (write_radix_block(dev, getid(root), page) < 1) {
                    block_page_put (page);
                    return ZERO;
              }
        }
        block_page_put(page);
        return root;
  }
```