A Persistent Snapshot Device Driver for Linux

Suresh B Siddha
Indian Institute of Science, Bangalore, India
suresh.siddha@alumnus.csa.iisc.ernet.in
K Gopinath
Indian Institute of Science, Bangalore, India
gopi@csa.iisc.ernet.in

Abstract

Web servers and large enterprises demand online backup capability to protect data that must be available continuously and eliminate the down-time needed to perform conventional off-line backups. Online backup demand is fueled by growing data capacities that have lengthened backup window time frames and by the significant loss in productivity that occurs when servers must be taken offline.

Snapshots allow applications to take online backup. This paper discusses the *design and implementation* of a layered device driver in Linux for persistent snapshots. This paper also discusses the *design issues* involved in developing a snapshot device driver in a clustered environment.

1 Introduction

An application such as a file system or a database stores data on a device. To protect against loss of data, the device must be backed up regularly. But one cannot keep the applications offline during large backup intervals. Backup interval is typically large, because of slow devices like tape devices. Down time is a critical problem as it translates into lost business for sites like e-commerce sites.

1.1 Snapshots

Snapshots add a 'fourth dimension' - time - to the disk contents as a whole. Data can be viewed in its current state or as it was at selected instants in the past. While one access may apply to a particular static image of the data as of an hour ago, other users of the data can simultaneously read even older data sets, or read and write the current image of the data.

A snapshot efficiently creates logical copies of disk partitions, which simplifies online backup application development. Rather than backing-up live data, backup application access snapshot logical copies while the server remains online and fully functional. Snapshot copies of live data are logically equivalent to copying a real partition, but use significantly less disk space. Before taking a snapshot, the file system or raw partition has to be frozen. This freezing ensures that the data on the disk is in a consistent state. Once the file system or raw partition is frozen, a snapshot can be taken. A snapshot manages any changes to data via the snapshot save area. Before a physical block is modified, a snapshot invokes a *copy on write* (COW) technique, copying the contents of blocks that are to be modified into the snapshot save area.

After the block is copied, its physical location can be overwritten by the changed data. After setting up a snapshot, only the first write of a given block causes a COW operation ("a COW push"). Subsequent writes are allowed to go directly to the real disk. Since block copy I/O activities occur in real time and only as blocks are changed, snapshot I/O has a significantly smaller performance impact than alternative online backup approaches.

There are two different paths to access original data and snapshot data. Consider Figure 1 where the file `project.tex` is made of three blocks on disk: A, B and C. When a snapshot is taken, the version of `project.tex` that exists in the snapshot is identical to the one in the original file system. Assume that

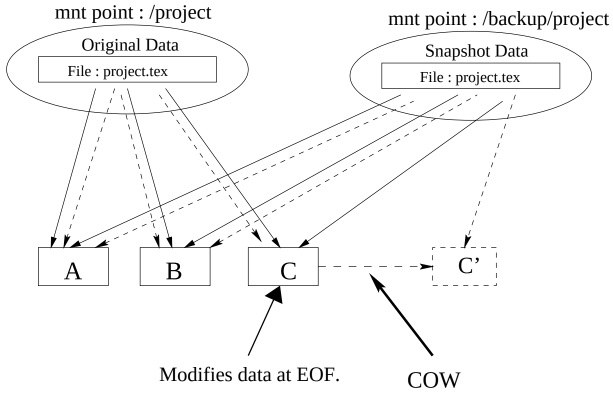

Figure 1: Snapshot COW Mechanism

an application modifies data at the end of the file, causing the contents of the block C to change. The snapshot facility uses the COW policy that copies the original block C to a location in snapshot save area, creating a block $C^{'}$. Dotted arrows in Figure 1 shows the resulting situation.

Snapshots happen at the block level. Our design uses a layered device driver that enables snapshots to be persistent across panics and power failures. As this is at a device driver level, the snapshot facility will be available to all file systems and databases that work on block devices directly. Journalling as in ext3fs is an orthogonal issue as it is at the file system level.

Before starting the snapshot, data needs to be in frozen state. This paper doesn't address the way data is frozen. This is left to the applications like file system or database.

This paper is structured as follows. First, we discuss previous work. Next, we discuss the *design and implementation* of a layered device driver in Linux for persistent snapshots for a single node. We then discuss the *design issues* involved in developing a snapshot device driver in a clustered environment. We then present our implementation for a single node system along with details of the difficulties faced given the device driver environment in Linux 2.2. We then give some performance indicators and end with conclusions and future work. Note that we often use the term disk as a synonym for a partition but it should be clear what is being meant.

2 Previous Work

In this section, some of the existing solutions that can be used for taking online backup will be discussed along with their advantages and disadvantages.

RAID 1 Solution: Solutions like *drbd* also comes under this category. In RAID 1, disks are grouped into mirrored pairs. Two copies of the same data are maintained on each of the disks in a mirrored pair. Every write of the data has to be applied to both the disks whereas a read request can be satisfied from any of them. During backup, one disk can be removed from the mirror and the data on that disk can be backed up. Once backup is over, the disk removed from the mirror has to be re-synchronized with the active disk, preferably in the background. But synchronization even in the background impacts I/O performance. During backup and synchronization, the disk that is removed from the mirror will not be available for servicing requests from normal applications, thus losing the redundancy advantage of RAID 1. To overcome this disadvantage, we have to use 3-way mirroring, which results in a large storage overhead.

Snapshots in File System: Some file systems like VxFS [1] [1] have a snapshot feature in their file system. Logically snapshots happen at file system level. The implementation is actually at the block I/O level. But each filesystem in the system has to provide the snapshot feature to take online backup of its disk

Clones in File system: Some file systems such as VxFS and WAFL [2] provide clones in their file system that enable online backup. When a snapshot is created, the entire file system is marked copy on write. Whenever any data is about to be changed, metadata (like inodes) that point to the data are copied and given new data to point at. A new superblock points to the old data and we now have file system structures that point to both old and new data. The old data and new data along with their file system metadata constitute two different file systems and therefore can be mounted at two different mount points. Backup can be taken from the filesystem that has the old data. This design, however, requires major changes in the file system.

Log-Structured Filesystem: Log-structured file systems (LFS) [2] use a sequential, append-only log as their only on-disk structure. Since writes are always at the tail of the log, they are all sequential and disk seeks can be eliminated. The data that has to be retrieved from the disk will always be located by traversing towards the left of tail of the log. When the snapshot is on, the data towards the left portion of the log tail position at checkpoint time corresponds to snapshot data. All new writes after the checkpoint will be written at the tail of the log and so all the new writes will be towards right of log tail position at checkpoint. But changes are required in file system to retrieve snapshot data from the log and garbage [3] collection should not be done when the snapshot is on.

LVM For Linux: The Logical Volume Manager (LVM) [3] adds an additional layer between the physical peripherals and the I/O interface in the kernel to get a logical view of disks. This allows the concatenation of several disks ("physical volumes") to form a storage pool ("volume group"). Recent releases include support for snapshot logical volumes where snapshots can be taken for any file system. However, the implementation in the 2.2 linux kernel does not support persistent snapshots. In addition to this, the current implementation is not clean: it is not a separate module but hacked into the Linux source code to get the required mapping between the logical devices and original devices, so that the actual request never reaches the *lvm* pseudo device.

3 Design Issues

This section discusses the various design issues involved in development of a persistent snapshot device driver and in developing such a device driver in a clustered environment.

3.1 Design of Persistent Snapshot Device Driver for a Single Node

The snapshot device driver is a layered device driver to enable online backup of data on the disk irrespective of the file systems existing on the disk. As described earlier, when a snapshot of a disk is taken, the blocks that are modified during backup interval are copied into the snapshot disk and a map is maintained between the block on the original disk and the snapshot disk. This map is the metadata of the snapshot.

3.1.1 Consistent Persistence

To enable the snapshot to survive across panics and power failures, we need to store the map and modified blocks in persistent storage. In addition, we have to keep the map and snapshot blocks consistent. i.e., whenever a COW push occurs, the update of the map and copy of the contents of the original block to snapshot disk has to be atomic. This atomicity can be achieved in two ways.

1. **Ordered Writes:** Here, the original block is first copied to the snapshot block and then the map block is written to the disk. Even if power fails in between, on the next write to the same block of the original disk, a COW push will be done as the map for that block has not yet been updated. The opposite order is problematic.

2. **Logging:** In this case, intentions are logged onto the log disk first and the original disk updates can be done in background. This is called as REDO logging [2]. In case of a crash, when the machine comes up, an user application scans through and replays the log (the data is taken from the log and written to the disk). Thus the snapshot data remains consistent.

Out of the above two methods, method 2 is more efficient. This is because the original disk writes can resume immediately after logging the transaction onto the log disk. And log disk writes are more efficient as they are written sequentially. Actual updates of map and the snapshot block can be done in background. Hence, the delay introduced for original writes will be smaller in the second method. In our implementation, we use data logging.

3.1.2 Log Format

In data logging, data is logged along with the metadata onto the log disk. The original disk block contents that will be copied on to the snapshot disk is logged onto the log disk along with a map that contains information about the mapping of the original and snapshot block numbers. In case of a power crash while writing to the log, we must be able to detect that writing to the log is not complete. So we have to introduce an identifier, Xid or transaction id, that grows linearly.

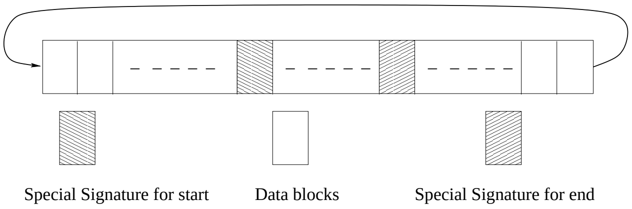

Figure 2: Initial Design of the Log

In logging, data is transient. We do not require the data of the log after the data is actually written to the data disk. To avoid problems like compaction, we use a circular log format. One design of the log is given in Figure 2. But there is a problem with the use of a special signature. Since we are writing data also, anything can be present in the data. So, there is a possibility of mistaking some data as the end signature. The next design is to have an initial signature, its offset in the log and the size of the log. We check for the signature just at that offset and if it is a valid signature, we assume that the data is completely written. However, there is still a possibility that there is some old data at that offset that exactly matches the signature. Our final design[4] reserves two entries in every block of the log disk for identification. Each block has its first few bytes as (transaction id, offset) where offset refers to the offset of the sector within the transaction. This eliminates the problem of data being mistaken as the signature since the transaction id is checked only in these places and transaction id is always unique. Hence, the structure of the transaction log has a transaction prologue, data followed by a transaction epilogue as shown in Figure 3. The space allocated to the transaction is freed only when the transaction is committed (after the data and map are written to the disk). When committing a given transaction, we change the offset value in the transaction's prologue to some special value.

3.1.3 Tree Structured Writable Snapshots

Generally snapshots are read-only. We can have multiple snapshots taken at different instances. The snapshots at different instances and Original view of the data form a chain like structure, with original view(also called as primary) at the head(root) of the chain and the oldest snapshot at the tail. Assume a chain of snapshots exist named C_n...,C_1,C_0 such that C_0 is the oldest and C_n the latest in the chain followed by the 'root' that we will call C_{n+1}. This

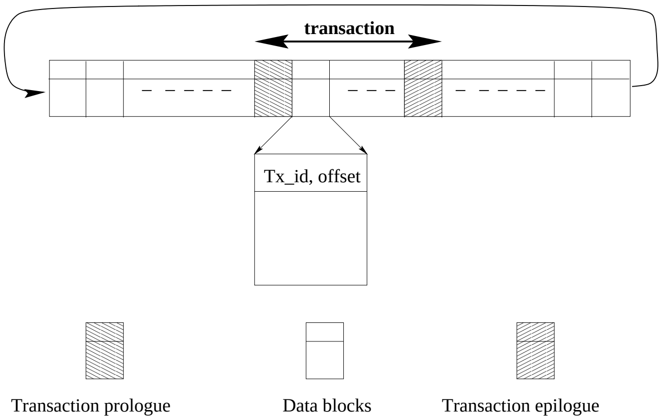

Figure 3: Final Design of the Log

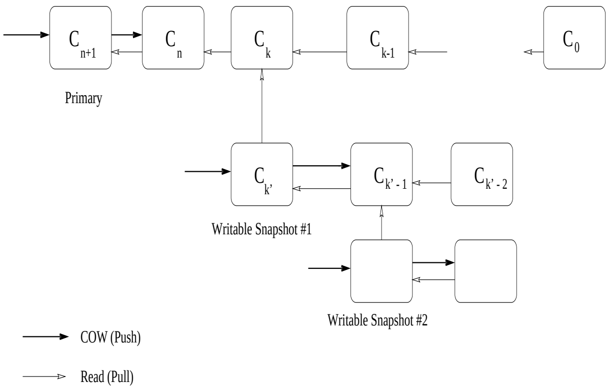

Figure 4: Snapshot Tree

is shown in Figure 4. All the COPY-ON-WRITE pushes resulting from writes to root, will go to the recent snapshot which is next to the root node in the chain. For snapshot read on some snapshot C_k proceeds as follows. First map of the snapshot C_k will be checked. If that has an entry for the block to be read, then the read request will be serviced from snapshot C_k. Otherwise the maps of the snapshots C_{k+1},C_{k+2},.... taken after the snapshot C_k, will be checked in order and if any one of them have an entry for the block to be serviced then that block will be read from the corresponding snapshots. If no snapshot has that mapping, then the read request will be serviced from the root.

Nothing prevents us from having a read-write snapshot. In this section, we present a design of tree structured writable snapshots, which we have implemented. The basic idea is to associate an additional snapshot to every snapshot that needs to be mounted writable. All actual writes are then made to this new snapshot and the snapshot chain is modified such that no downstream snapshot references this one.

The first time a snapshot C_k is to be mounted writable, we create a child node($C_{k'}$) rooted at the snapshot referenced C_k and mount this child snapshot $C_{k'}$ as writable. No (downstream) snapshot (C_j) with $j < k$, will depend on snapshot $C_{k'}$ because it doesn't occur in the path of root node(root node or primary will be the original view of data). With the tree structured snapshot instances, there will be additional dependencies. Snapshot removal cannot be permitted as long as writable children exist. Reading from the writable snapshot $C_{k'}$ works just as reading from any standard read-only snapshot does. Writes to the snapshot $C_{k'}$ will be directly written in $C_{k'}$. Now pushes are never required to be made downstream to snapshot C_{k-1}. We can have again read-only snapshots from read-write snapshots and from these read-only snapshots we can again have read-write snapshots. Thus we will have a tree like structure as shown in Figure 4.

3.2 A Clustered Persistent Snapshot Design

A cluster is a group of computers (usually referred to as nodes) connected in a way that lets them work as a single, highly available system. In clusters, resources are made highly available through redundancy: redundant servers, networking, disks and controllers. In a clustered environment, there will be shared disks across the cluster. Each node has direct access to the shared disk through a different interface. The number of nodes that can connect to a disk directly depends on the number of interfaces for that disk (typically 2-4). In our design of persistent snapshot device driver for a cluster, we assume the following components are available in the cluster environment:

- A global lock manager that is required for serializing access to metadata of the snapshots
- An external log manager for logging metadata changes of the snapshots
- Messaging components like global atomic broadcast to send messages from one node to any other node in cluster.

3.2.1 Metadata Management of Snapshots

Applications or kernel components running in a cluster environment, such as a database or a cluster file system can ensure that read and writes to the same block do not proceed at the same time from two nodes in the cluster. This is ensured by taking a cluster wide lock by the application or kernel component.

In a clustered environment, however, there can still be a conflict with respect to concurrent updates to the metadata of the snapshot device. The metadata of a snapshot is its map, which gives information about the mapping of original and snapshot blocks.

For example, suppose the application issues a `write (blkno1)` on node A, that takes time from *t0* to *t1*, and node B issues `write (blkno1)` or `write (blkno2)` that takes time from *t2* to *t3*. The constraint $t1 <= t2$, say, has to be satisfied if the write from B is to `blkno1`, otherwise there is none. This assumes that a correct application will ensure that writes on same block cannot come simultaneously from two different nodes in the cluster. Now consider how the snapshot device driver must behave. At the first write from node A, it does allocate a new block and push the old data there. The problem is about snapshot device driver (SDD) instance at node B that needs to know if a block has already been COW pushed.

The simplest (and lowest performance) solution is to have a single metadata area with some cluster wide lock on it. Node B, then, takes this lock, reads the translation table, sees that *blkno1* is already pushed, and allows the write to succeed. Or, it sees that *blkno2* is not yet pushed, and updates SDD metadata for block allocation and does a COW push. This solution is slow because each write from each node is totally serialized on a single lock. One of the alternatives is to divide the block allocation area, with each piece having a different lock so that there is more concurrency. This still serializes writes to a set of blocks that are controlled by the same lock.

3.2.2 Logging

All the changes to the snapshot metadata have to be logged to the log disk before writing anything to the disk. When a node X fails, some other node that shares the log disk with X can replay the log of X.

When all nodes in the group fail, and some nodes come up, the first node that comes up can check all the log disks that it shares with other nodes in that group and replay the logs that are dirty.

3.2.3 Membership updates in the Cluster

If A, B, and C are all registered members of a group and C unregisters, A and B will receive a membership update with only A and B. This message is identical to the message that would be generated if C failed. To distinguish these two events, we assume that the client uses the broadcast[4]messaging facility to announce its intention to leave the group. When a node leaves the group intentionally, this is not a big issue. When a node fails, a global atomic broadcast facility sends a membership update to every other node in the list. One of the remaining nodes (we can select the node with the smallest id) that shares the log disk with the failed node, replays the log of the failed node. Similarly, if a node joins the group, the new node has to be synchronized with the rest of the nodes in the group.

3.2.4 Performance

The update of metadata of snapshots is done by taking a clusterwide lock using a global lock manager. This global lock manager sends and receives messages on a network when transferring access rights of the locks between nodes. Each of the message involves some fixed overhead that is independent of the size of the message. If the data associated with a lock can 'piggy-back' on the message used to transfer an access right, there is a potential benefit in performance. Thus we can divide the map into sections of upto. say, 64K, with each section protected by a cluster lock.

To increase the performance of disk writes when the snapshot is on, we can cache snapshot maps and snapshot blocks. The following methods can be used for better performance:

Method A - Each node that does a COW push first takes a clustered wide lock and logs the transaction onto the log disk. Then it broadcasts the updated map information along with the original disk block to all the nodes. Each node updates its map and caches the original block. Whenever a node tries to read a snapshot block (for backup), it will check in its local cache for its map. If there is a mapping and the original block is in the cache, then that is used. If the original block is not in the cache, the node broadcasts a message requesting the snapshot block. Nodes that have this block in their cache will respond to this message. If no node responds to this message, it means that the block is already written to the snapshot disk and can be read from the disk. If there is no mapping for that block, then the node can take a cluster wide lock on the map section and read the block from the original disk. Similarly, if the map section is not in the cache, then the node can broadcast a request for it. If no node responds to that request, the map section can again be read from the snapshot disk.
Advantages If a block is already COW pushed, there is no need to take a cluster wide lock on the snapshot map section to check whether to do a COW push. Similarly, a snapshot-read need not take a cluster wide lock to read the snapshot block if it is already COW pushed. This method performs better when large number of nodes in the cluster do a large number of disk writes in a small, concentrated portion of the original disk.
Disadvantages We need one broadcast message for each COW and one broadcast message, if the map-section is not in cache, and one more broadcast message for snapshot block if it is not in the cache.

Augmentations to Method A - We can keep additional information in the map: the node that has COW pushed the block along with the snapshot block number where it is copied. For example, if some node A does a COW push of block x, x's map information contains A also and the snapshot block is in A's cache. Later, if some node B tries to read x when it is not in its cache, it gets the node number from the map and contacts that node (in this case, node A). If map is not in its cache, it broadcasts a message for it. Node B now requests node A for snapshot block x. If A has it in its cache, it can reply with that block. Otherwise, we can do the read of the snapshot block x in 2 ways.
1. Node A reads block x and sends it to node B
2. Node B reads block x and broadcasts that it has block x so that all other nodes can update the node information for the map-entry of snapshot block x
Advantages The broadcast messages in method A become unicast messages when map or snapshot block is not in cache. This method is useful if many nodes are write to the original disk, but the writes are not concentrated in one region.
Disadvantages The map size increases as we incorporate node information also in the maps. This

results in an increase in the number of locks that are used to serialize access to different sections of maps.

Method B - To do a write, a node x takes a cluster lock for the map section and logs its COW push onto the log disk. Whenever some other node y asks for a lock on the map section, node x transfers the log along with the lock to y. After transferring the log, x logs an entry in its log disk that the log has been transferred to y. If x fails, y need not replay the log as x's log has already been transferred to y's log disk.
Advantages There are no broadcast messages. This method is useful if only a few nodes in the cluster write to the original disk.
Disadvantages A node has to take a cluster wide lock on the snapshot map section to check whether to do a COW push, even if the block is already COW pushed. Similarly, a snapshot-read needs a cluster wide lock to read the snapshot block even if it is already COW pushed. When a cluster wide lock is transferred from one node to another node, the dirty log has to be transferred also.

4 Implementation Details

A persistent snapshot device driver has been implemented for Linux-2.2.5 kernel for a single node.

4.1 Block Drivers in Linux

Whenever the buffer cache cannot satisfy a read request or pending writes must be flushed to disk, the driver will be called to perform the data transfer. Since `struct file_operations` does not carry any entry point other than read and write, an additional structure, `blk_dev_struct` is used to deliver data transfer requests. The definition of the structure as found in the Linux 2.2 kernel [5] is as follows.

```
struct blk_dev_struct {
        request_fn_proc *request_fn;
        ...
        /*queue function*/
        queue_proc  *queue;
        /*request queue*/
        struct request *current_request;
        struct request plug;
        ...
};
```

When the kernel needs to spawn an I/O operation, it calls the `blk_dev[dev_major].request_fn`, where `device_major` is the major number of the device. `ll_rw_block()` is used to request a number of buffers from the block device or to write a number of buffers into the block device. `ll_rw_block()` queues the request structure into the corresponding device queue. The device queue can be mentioned by the queue function of the device. Since there are only a fixed number of request structures, if `ll_rw_block()` does not find any free request structures, it blocks till it gets a free request structure.

Plug: A *plug* request in `blk_dev_struct` is used to "plug" the device. A device is plugged to force the transfer to start only after we have put all the requests on the list. The request function, corresponding to the major number of the block device, is not called until the plug is removed. This allows clustering of adjacent blocks to speed up disk transfers.

4.2 The Pseudo Device Driver

4.2.1 Problems in writing a layered device driver in Linux

- *Request function of a Pseudo device*: `request_fn` for a device can be called from process context or interrupt context [5]. It is invoked from process context when the request function is not already running, i.e., when device queue is empty. At the end of the first request, if there are some pending requests in the queue, then it is the responsibility of interrupt handler to call `request_fn`. When writing a `request_fn` for a pseudo device, one have to take care that it will not block (as blocking routines cannot be called from interrupt context [6]). Hence a `request_fn` cannot call blocking routines like `ll_rw_block()`. This problem has been solved by making `request_fn` of our pseudo device use a loop that serves all the requests in the device queue one by one. Hence the `request_fn` will always be invoked from the process context and we can use blocking routines also.

- *Performance penalty*: For a layered device driver all the buffers will be first queued to pseudo device queue and, in `request_fn`, it will be put into the underlying device queue. Processing time for each buffer is now the delay of passing through two device queues.

- *Dynamic introduction of modules*: If we want to dynamically push a pseudo device on top of a real device, then the requests going to the real device have to be diverted to pass through the pseudo device. For example, for snapshots, the pseudo layer is required only when the snapshot is on for some extra work like COW push and logging. However, the introduction of dynamic layer is not easy in the current device driver interface framework in Linux. One solution (though not clean) for introduction of a dynamic layer is to interchange `request_fn` and queue function of the original device with the pseudo device.

  For example, if we are taking the snapshot of a SCSI device, before start of the snapshot (before introduction of the pseudo layer) SCSI major number will point to its own `request_fn` and queue function. Once the snapshot is on, SCSI major number will point to pseudo devices `request_fn` and queue function and the pseudo device major number will point to the SCSI device `request_fn` and queue function. We can thus introduce the pseudo device layer. But this is not a good solution. `ll_rw_block()` does different types of optimization's for different types of devices [5]. So, in this method `ll_rw_block()` code has to do all those optimization's for the original device major number if there is no pseudo layer and same optimization's has to be done for pseudo major number if there is a pseudo layer. Such an implementation will again become somewhat similar to implementation of LVM as explained in section 2 and thus will not be a simple, separate module and has to be (hacked!) integrated into the kernel.

  This problem has been solved by having a pseudo layer always on top of the original device driver layer. When ever snapshots are on, the required work such as COW push and logging will be effected by the driver. If there is no snapshot, the pseudo request function simply transfers the requests from the pseudo device to the original device.

4.2.2 Pseudocode of Driver

We have implemented the Persistent Snapshot Driver Driver (PSDD) in Linux 2.2.5-15 as a separate module [7]. As discussed earlier, the pseudo layer will always exist, independent of whether the snapshot is on or off. The following pseudo code explains the `request_fn` of the pseudo device that implements snapshots.

```
psdd_request() {
 while (1) {
  get the request from the pseudo device queue;
  if request is snapshot block read {
     if mapping exists for this block
        read the block from the snapshot disk;
     else
        read the block from the original disk;
     continue;
  }
  if request is original block read {
     read the block from the original disk;
     continue;
  }
  if request is a write request {
     if mapping exists for this block
        write the block to the disk;
     else {
        read the block from  disk;
        copy to the snapshot disk;
        update the map;
        write the block to the disk;
     }
   }
 }
}
```

4.2.3 Snapshot Disk Layout

The snapshot disk layout is shown in Figure 5. We use the first few blocks in the snapshot disk for logging. To get the full benefit of logging, this log should be on some other disk. The maps for all the existing snapshots use a set of blocks reserved at the end of the snapshot disk to copy the original blocks. Each map contains information about the parent in the tree structured snapshots.

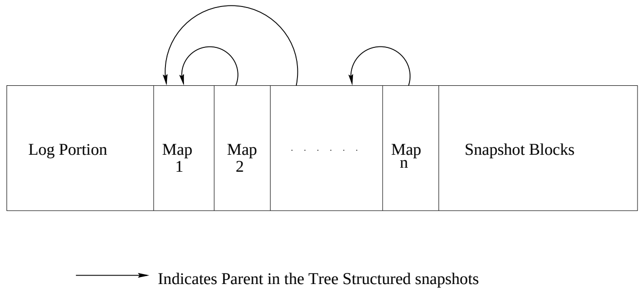

Figure 5: Snapshot Disk Layout

4.3 Changes to Linux Kernel 2.2.5

Even if the device driver framework is desired to be followed strictly, there are some places in `ll_rw_block()` which need modification. Otherwise, deadlocks may occur and performance may be impacted.

- Deadlock may occur as for each request structure in pseudo device queue `request_fn` of pseudo device will create (one or more) request structures that will be queued to the underlying device queue. If all the request structures are used up for the pseudo device queue, none of them can be processed as processing them in the underlying device requires some request structures. `ll_rw_block()` code has been modified so that the pseudo device queue can occupy only half of the maximum slots in the request structures array.
- Performance will be affected because the kernel will use plug to gather requests for the pseudo device but this will add to the delay in processing the requests. For a pseudo device, clustering of requests is not required, so the `ll_rw_block()` code has been modified so that plugging is avoided for the pseudo device.

5 Performance

To improve the performance of the original disk writes, the map block and snapshot blocks corresponding to a COW transaction are cached in buffer cache. These blocks can be cached till the log is completely full. In our case, we flush the cached snapshot disk blocks if the log disk is full or if the system free memory falls below certain limit. This limit can be a input parameter. `ll_rw_block()` code has been changed to calculate the time taken for disk writes. Field *b_dev_id* in *struct buffer_head* is used for this performance calculation. Table 1 shows the delays introduced for the original disk writes by the pseudo layer with and without snapshots and with ordered writes and with logging. Disk writes that result in COW pushes were selected in evaluating performance. Access patterns of the data in all the methods are almost similar. The delays for original disk writes due to snapshots with logging are much smaller compared to snapshots with ordered writes, since log writes are fast (as they are sequential writes). The delay time has been averaged over 30 block writes.

	Ordered Writes	**With Logging**
Without Snapshots	0.256 ms	0.263 ms
With Snapshots	0.121 s	0.068 s

Table 1: Snapshot Performance

6 Conclusions and Future Work

6.1 Conclusions

Data Backup - especially for disaster recovery or to return to an earlier uncorrupted view of the data - can be achieved through snapshots. The design of a persistent snapshot device driver and its implementation details in Linux have been presented. We also have discussed the design options in a clustered environment.

6.2 Future Work

- The device driver which we have implemented assumes that block size of original disk and snapshot disk is the same. The device driver can be extended for differences in block sizes of original disk and snapshot disk.
- Deletion and renaming of snapshots has to be implemented.

- In the current implementation, the first few blocks of the snapshot disk are used for logging. To get the full advantage of logging, a separate disk can be used for logging.
- Port the snapshot device driver to Linux 2.4 and release the snapshot device driver as open source to the Linux community for general use and future enhancements.
- The implementation of a clustered driver has to be undertaken. The algorithms needs to be formally verified and performance evaluated by simulation or other means.

Acknowledgments: We thank Dilip Ranade, Veritas Software Corp., Pune for suggesting that we look into this area and Radha Shelat, also of Veritas, for her help and interest. Financial support from Veritas Software, Pune is also gratefully acknowledged.

References

[1] *Snapshots in VxFS.* http://www.veritas.com.

[2] Uresh Vahalia. *Unix Internals.* Prentice Hall, 1996.

[3] *Logical Volume Manager for Linux.* http://linux.msede.com/lvm/.

[4] K Gopinath, Nitin Muppalaneni, N Suresh Kumar and Pankaj Risbood. *A 3-tier RAID Storage System with RAID1, RAID5 and compressed RAID5 for Linux.* 2000 USENIX Annual Technical Conference, June 21-23, 2000 - San Diego, California.

[5] *The Linux Kernel Sources version 2.2.5.*

[6] Alessandro Rubini. *Linux Device Drivers.* O'REILLY, Feb 1998.

[7] Beck et al. *Linux Kernel Internals.* Addison-Wesley, 1998.

[1] VxFS is a journalling file system from Veritas Software.

[2] Write Any-where File Layout(WAFL) is a file system from NetApp.

[3] In LFS garbage collection is done by a process, allowing the log to wrap around.

[4] Message sent to each and every node in a group of cluster, which has registered for certain service, like snapshot service.