# An Efficient Snapshot Technique for Ext3 File System in Linux 2.6

**Seungjun Shim*, Woojoong Lee and Chanik Park**
Department of CSE/GSIT*
Pohang University of Science and Technology, Kyungbuk, Republic of Korea
{zephyr*,wjlee,cipark}@postech.ac.kr

**Abstract**

Snapshot is to create an instant image of a file system. Creating a snapshot image with little processing and space overheads is critical to provide the continuous data service during backup.

In Linux, there are two types of snapshot techniques available : volume-based approach like Logical Volume Manager(LVM) and file system-based approach like SnapFS. The volume-based LVM provides efficient space management capability, but requires some space to be exclusively reserved to store snapshot images and results in high snapshot processing overhead. The file system-based SnapFS performs better than LVM due to its less snapshot processing overhead. Moreover, the SnapFS does not have to reserve some space exclusively for snapshot images. Therefore, the file system-based SnapFS is considered a better solution in the desktop environment where large-scale space management capability is not that criticial. However, SnapFS is available only in Linux kernel 2.2.

In this paper, we develop a file system-based snapshot for the ext3 file system in Linux kernel 2.6. The main concept of the file system-based snapshot mainly come from old-version SnapFS. Experimental evaluation shows that our implementation is quite efficient and working correctly.

## 1 Introduction

In recent days, as mass storage devices are being broadly used on desktop PCs, large-scale data backup techniques become more and more important. However, in spite of its importance, off-the-shelf data backup methods available on Linux such as cpio and rsync[1] are not sufficient to satisfy some requirements of large-scale backup. Most of all, while these methods are running, all I/Os must be blocked to guarantee data integrity. Thus, they are not applicable to large-scale data backup.

The snapshot is an advanced technique that can make an image of file system at a point of time. The image, called a snapshot image, is useful as a large-scale backup. It provides high backup performance and allows concurrent data services to users during backup.

There are two types of snapshot methods. One is a file system level approach which depends on a specific file system, and the other is a volume level on the the device driver layer. Each of them has some pros and cons. Because the file system level approach depends on an underlying file system, it has much lower portability. However, it is more effective to construct a snapshot image by using some information provided by the file system. In the volume level approach, there is another serious problem with its processing. It needs exclusively reserved free space for snapshot images. Moreover, the size of the reserved space is determined by monitoring the access pattern of its target volume on run-time.

In Linux systems, the SnapFS[2] is the most well-known file system which was implemented using the snapshot technique on the ext2 file system in Linux kernel 2.2. However, it is not available on the current Linux kernel. The LVM/LVM2[3], one of the latter approaches, is an implementation of device virtualization layer on a block device driver in Linux kernel. Although it provides a large-scale backup mechanism and is available on the current kernel, it has the problems described above.

In this paper, a new version of SnapFS is developed for the ext3 file system[4] on Linux kernel 2.6 in order to realize an effective large-scale backup. The experimental evaluation shows that our implementation is quite efficient in snapshot performance when compared with the old version SnapFS in Linux kernel 2.2 and with LVM2.

The rest of this paper is organized as follows. Section 2 surveys background works. Section 3 describes the design of the SnapF and interesting im-

plementation aspects. In section 4, we evaluate the SnapFS performance. Finally, we conclude in Section 6 and suggest future directions.

# 2 Related Works

As mentioned above, the off-the-shelf snapshot methods are difficult to apply in the current Linux environment, because they heavily depend on a specific file system or requires an exclusively reserved disk volume for snapshot images. In this section, we describe two approaches for the snapshot: file system-level and volume-level approaches.

## 2.1 File System-level Snapshot

As a file system level snapshot approach, there are some file systems such as WAFL[5], VxFS[6], FFS[7] and ext3cow[8]. The Write Anywhere File Layout (WAFL) file system is based on Log-structured File System (LFS)[9]. It maintains all data with its metadata on a specific tree structure and provides the snapshot through managing this structure.

Veritas File System (VxFS) is a file system that was developed by Veritas Software as the first commercial journaling file system. VxFS also supports the snapshot based on block-level Copy-on-Write (COW). VxFS has a problem that free space called the snapshot file system must be reserved and it is only available while mounted.

Fast File System (FFS) is a popular file system in BSD. It also supports the snapshot based on COW and maintains snapshot images with a special file, called the snapshot file. The problem with FFS is that it is heavily dependent on the BSD environment.

Ext3cow was developed in Linux kernel 2.4. It extends the ext3 file system to support the snapshot through improving the in-memory and disk metadata structure. It not only shows efficient performance compared with ext3, but also supports the snapshot with low overhead. However, to use ext3cow, mkfs.ext3, a file system format utility, have to be patched. That is, although ext3cow extends the ext3 file system, the ext3cow is an entirely different file system. Thus, the snapshot in the ext3cow file system also has very low portability.

## 2.2 Volume-level Snapshot

In this section, some of volume-level snapshot approaches are described. The volume-level snapshot is usually performed by a volume manager such as LVM/LVM2[3] or PSDD[10]. These methods have a common problem in that they must have reserved disk volume dedicated for snapshot images. Logical Voume Manager (LVM)/LVM2 is a volume manager located between a file system and a device driver. The snapshot in LVM is achieved by block-level COW and is performed on a unit of Logical Volume (LV) in the Volume Group (VG). The main problem of LVM/LVM2 is that it needs a reserved disk volume called Snapshot Volume (SV). If the size of snapshot image is larger than SV, LVM/LVM2 will invalidate a snapshot image. Therefore SV must be at least equivalent in size to its target LV.

PSDD is a device driver for the persistent snapshot and supports data backup without any dependence on file systems. The snapshot under the PSDD is performed on a block device and the blocks that are modified during the backup, are copied to the reserved snapshot disk and the mapping information between target disk and the snapshot disk is maintained to guarantee the data integrity. The PSDD also requires an exclusivly reserved disk volume called the snapshot disk for snapshot images and may lose all snapshot images when a disk failure occurs.

# 3 A File System-based Snapshot Solution in Linux 2.6

SnapFS is an open source light-weight file system residing on top of the ext2 file system in Linux kernel 2.2. It supports a snapshot operation by the block-level copy-on-write (COW) technique and the entire disk spaces defined by a file system will be used to store snapshot images as well as file information. SnapFS maintains file's metadata related to snapshot while the underlying file system (that is, ext2 file system) manages file's data as well as snapshot image and should support the block-level copy-on-write technique.

## 3.1 SnapFS Architecture

In this section, we describe details of the SnapFS architecture.

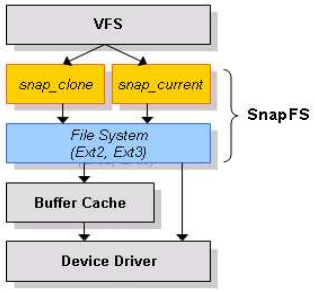


**FIGURE 1:** *SnapFS Architecture*

Figiure 1 shows the SnapFS architecture.

SnapFS consists of two sub components: *snap_current* and *snap_clone*. The *snap_current* is an interface for supporting block-level COW to the underlying file system and the *snap_clone* handles a read-only file system for snapshot images only.

SnapFS operates by the operation-filtering between VFS and the underlying file system. When VFS invokes functions of the underlying file system, SnapFS captures it and performs some operations which are necessary to block-level COW or read snapshot images.

Especially, because *snap_clone* is explicitly separated from currently used file system, called *snap_current*, there is no I/Os block to guarantee data integrity even if snapshot images are accessed.

## 3.2 Snapshot in the SnapFS

In this section, we describe details of the snapshot operation in SnapFS. As mentioned above, the SnapFS support the snapshot base on block-level COW, which is achieved through managing address space information. The address space information defines memory mapping of blocks in the file, and it is managed by *struct address_space* in the Linux kernel. Because the block-level COW is performed by using this structure rather than blocks on disk volume directly, the SnapFS supports the snapshot more efficiently.

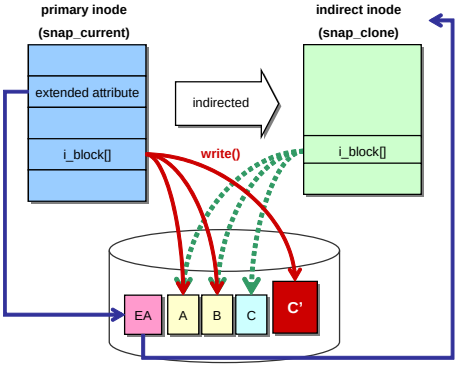


**FIGURE 2:** *Block COW operation of SnapFS*

Figure 2 describes this operation. When the snapshot command is issued, SnapFS allocates a new inode, called the *indirect inode* and copies the address space information of the *primary inode* to the *indirect inode*. By copying the address space, the *primary inode* shares all its blocks with the *indirect inode* without any actual block copies. If any block modification occurs, SnapFS only has to allocate new blocks to the primary inode and apply these blocks to its address space.

SnapFS also uses the extended attributes of the primary inode for managing the relationship between the primary and indirect inodes. The relationship which is stored into extended attributes with a table form is used to access indirect inodes, called snapshot images, in *snap_clone*.

## 3.3 Porting Issues

As mentioned above, because the old version of SnapFS was developed in Linux kernel 2.2, there are some issues that should be considered. We summarize these issues below:

- separation of inode and file operation
- creation of super block on underlying file system rather than VFS layer
- use of 32bit device type
- access to the address space structure through the file structure

### 3.3.1 The separation of inode and file operation

Most of all, the big change is a separation of inode and file operation structure. In Linux kernel 2.2, there is "has a" relationship between inode operation and file operation structure even though conceptually there is not. However, in 2.6, file operation structure is explicitly separated from inode operation structure.

The inode operation is managed by *struct inode_operations* inside the Linux kernel. It defines operations that are related to a file, such as open or close and also defines what operations are available with opened files. The file operations are also managed by a data structure, called *struct file_operations*. It consists of operations that are normally available to any file on the file system. Therefore, *struct file_operations* contains create, remove and rename operations and *struct inode_operations* contains open, close, read and write operations .

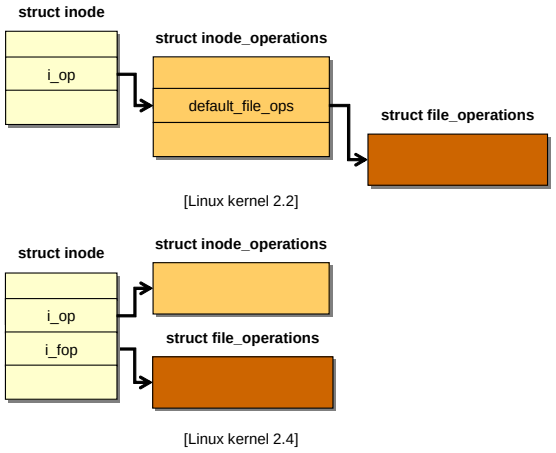


**FIGURE 3:** *inode and file operation structures in Linux kernel 2.2 and 2.4.6*

In Linux kernel 2.2, because the inode operation contained file operation, there is unnecessary dependency between the two structures. Thus, sometimes a file operation may have an unrelated operation. Since Linux kernel 2.4 these two structures are explicitly separated, as in figure 3, so many functions are modified.

### 3.3.2 The super block structure

In Linux kernel 2.6, a super block of the file system is allocated on the underlying file system rather than the VFS layer. Due to the change of the policy about allocating super blocks, we developed features to allocate a super block structure and set its values for the *snap_current* and *snap_clone* file system.

Linux kernel 2.2 and also 2.4 allocates the super block to the VFS layer; however, in 2.6, the super block is allocated on the underlying file system. This causes the following two modifications.

First, it means that *snap_current* and *snap_clone* can not use the super block that is made from VFS and contents of the super block structure may be quite different from lower versions of Linux. For this reason, we developed a new way to allocate a new super block in *snap_current* and *snap_clone*. In addition, because the super block structure of the *snap_clone* has almost the same contents as that of *snap_current*, we modified it to reuse the super block of *snap_current*, except for some values such as the flag field which has to be set read-only.

Second, SnapFS must handle the block device information shared between *snap_current* and *snap_clone*. The block device information defines the block device to which the super block belongs. It is not necessary to take care of the block device information in the old version of SnapFS because VFS allocates super block structure and SnapFS just uses it. However, as SnapFS has to make super block structure itself, it should be handled in *snap_current* and *snap_clone*.

### 3.3.3 Change of the device type

Linux kernel 2.6 uses *dev_t* rather than *kdev_t* for describing a block device type.

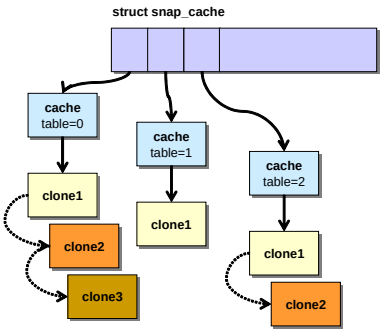


**FIGURE 4:** *snap_cache structure*

Figure 4 shows the *snap_cache* structure maintained inside of SnapFS. The *snap_cache* structure consists of the information about *snap_current*, *snap_clone* and snapshot images. This structure is the most important information and is heavily used by SnapFS. For example, currently used *snap_current*, how many snapshot images each *snap_current* has, which snapshot image *snap_clone* is related to and other important information are managed by *snap_cache*. SnapFS identifies each information with a device type which is stored in the block device structure.

Since Linux kernel 2.5, device type is described by 32bit *dev_t* type rather than *kdev_t* type, 16bit. Therefore, we should modify many parts of the SnapFS to support this modified device type.

### 3.3.4 The Address space structure

In Linux 2.6, the address space structure is accessed through not only *struct inode*, but also *struct file*. The *struct file* is created when the file is opened and defines how a process interacts with a opened file and the *struct inode* defines all information needed by the file system to handle a file. Conceptually, *struct inode* is include in *struct file*.

As mentioned above, SnapFS supports the snapshot with the block-level COW. In SnapFS, a block-level COW operation is performed by managing the address space structure rather than handling blocks on the disk volume directly. In lower versions of Linux such 2.2 or 2.4, the address space is managed through *struct inode* in *struct file* rather than *struct*

*file* directly. For this reason, the old SnapFS maintains only the address space in *struct inode*. However, Linux kernel 2.6 accesses the address space through *struct inode* in *struct file* as well as *struct file* itself.

Therefore, the SnapFS should manage address space structures of both *struct inode* and *struct file*, so we implemented that the address space of *struct inode* and of *struct file* have the same information when they are modified for a COW operation.

# 4 Performance Evaluation

We evaluated our implementation of SnapFS in Linux kernel 2.6 compared with the old version SnapFS and LVM2. We ran all the benchmarks on a 1.4GHz Pentium 3 machine with 1GB of RAM. All experiments were located on Segate 120GB 7200RPM SCSI hard drives. The machine was running either the new SnapFS and the 2.6.10 kernel or the old SnapFS and the 2.4.20 kernel.

We use the Bonnie benchmark program which was modified to evaluate performance of sequential read on the *snap_clone* mounted read-only mode. Considering effects of buffer cache on Linux, all experiments only ran on 500MB, 700MB, 1GB and 2GB file sizes. All experiment results are averages of twenty tests, and we measured performance of the block read and write operations among many Bonnie benchmark results.

## 4.1 Filtering Overhead

Firstly, we evaluated filtering overhead of SnapFS in Linux kernel 2.6. Because SnapFS is operated by the operation filtering between VFS and the underlying file system such as ext3, it is important to show the filtering overhead of SnapFS.

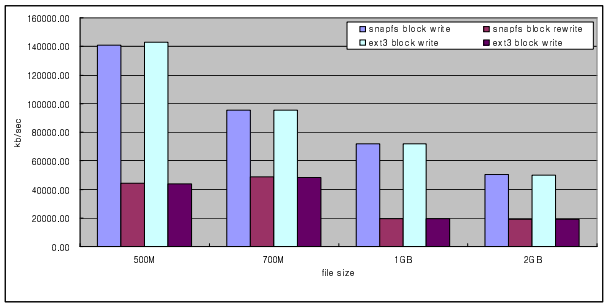


**FIGURE 5:** *Filtering overhead to write operation*

Figure 5 shows the result of filtering overhead to write operation of the SnapFS with normal write performance of ext3 file system. For each file size, the SnapFS performance averaged about 3% lower than ext3.

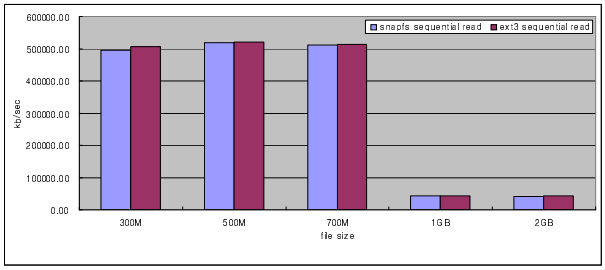


**FIGURE 6:** *Filtering overhead to read operation*

Figure 6 shows the result of filtering overhead to read operation. This result also shows that read filtering overhead is very low, averaging below three percent. Therefore, SnapFS has almost no effect on normal operations of the underlying file system.

## 4.2 The performance of snapshot image read

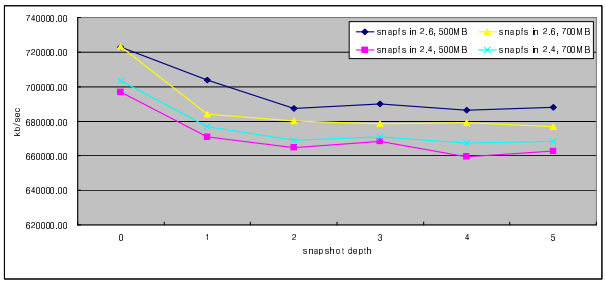


**FIGURE 7:** *Performance of the snapshot image read (500MB and 700MB file size)*

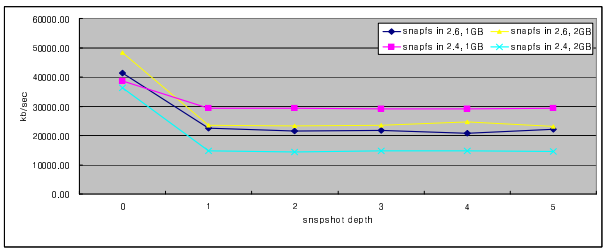


**FIGURE 8:** *Performance of the snapshot image read (1GB and 2GB file size)*

Figure 7 and 8 show performance of the snapshot image read with the snapshot depth from 1 to 5 compared with the old version of SnapFS. We use two separate graphs because the results of the 500MB and 700MB file size have largely different values from those of 1GB and 2GB file size. Read performance in SnapFS, especially *snap_clone*, decreases an average of 37MB/sec for below 1GB file size and 22MB/sec for above 1GB file size, regardless of the snapshot depth from 1 to 5.

The *snap_clone* of the old version shows that read performance between 1GB and 2GB file size is quite different; however the *snap_clone* in the newly developed SnapFS shows more stabilized results. The *snap_clone* in Linux kernel 2.6 also shows much better read performance than the old version in 2GB file size where it has no effects from the buffer cache. Additionally, the result about 1GB file size shows that SnapFS in Linux 2.6 has much lower performance than the lower version because the SnapFS was developed and optimized in the Linux kernel 2.2 environment and effects of buffer cache in Linux kernel 2.2 and 2.6 are quite different.

### 4.3 The efficiency of COW operation

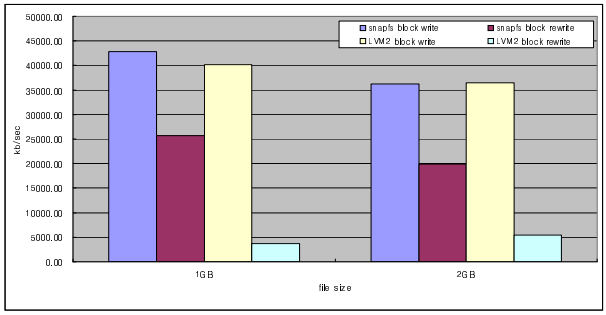


**FIGURE 9:** *The efficiency of COW operation (1GB, 2GB)*

Figure 9 is the result of write operation performance in *snap_current* after the snapshot is issued. This result shows how efficient block-level COW in SnapFS operates compared to LVM2.

Whenever a write operation occurs to *snap_current*, the SnapFS performs COW to blocks of files, and LVM2 does the same. In this case, block-level COW in the SnapFS averages 3.5% faster than LVM2. Bonnie's rewrite test performs the following series of operations: read each 16kb chunk of the file, dirty it, and then rewrite it. That is, the rewrite result describes how efficient COW operates against localized I/O in this experiment. Especially, block-level COW operation of the SnapFS is about five times faster than LVM2 against localized I/O.

## 5 Conclusion

Creating a snapshot image with little overhead is important for continuous data service as well as data backup. In this paper, we have developed an efficient file system-based snapshot and evaluated its effectiveness in ext3 file system in Linux kernel 2.6. Our solution has low snapshot processing overhead, resulting in 10.26% better read throughput for snapshot images compared with SnapFS. However, it is observed that the read throughput drops significantly when snapshot images are read. Our future work is to optimize read performance for snapshot images by prefetching the related metadata.

## Acknowledgement

The authors would like to thank the Ministry of Education of Korea for its support towards the Electrical and Computer Engineering Division at POSTECH through the BK21 program. This research has also been supported in part by HY-SDR IT Research Center, in part by the grant number R01-2003-000-10739-0 from the basic research program of the Korea Science and Engineering Foundation, in part by the grant number F01-2005-000-10241-0 from international cooperative research program of the Korea Science and Engineering Foundation, in part by the regional technology innovation program of the Korea Institute of Industrial Technology Evaluation and Planning, and in part by the wearable computer platform from ETRI.

## References

[1] A. Tridgell and P. MacKerras, 1996, *The rsync algorithm*, Technical Report TR-CS-96-05

[2] *SnapFS*, http://sourcefrog.net/projects/snapfs/

[3] D. Teigland, H. Mauelshagen, 2001, "Volume Managers in Linux", USENIX Technical Conference

[4] M. K. Johnson, 2001, *RedHat's New Journaling File System: ext3*, RedHat Inc.

[5] D. Hitz, J. Lau, and M. Malcolm, 1994, *File System Design for an NFS File Server Applicance.*, USENIX Technical Conference

[6] Toby Creek, 2003, *VERITAS VxFS, Technical Report TR-3281*

[7] M. K. McKusick, et al., 1994, *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley

[8] Z. Peterson and R. Burns, 2005, *Ext3cow: A Time-Shifting File System for Regulatory Compliance*, ACM Transactions on Storage.

[9] M. Rosenblum and J. K. Ousterhout, 1991, *The Design and Implementation of a Log-Structured File System*, Symposium on Operating Systems Principles

[10] S. B. Siddha, 2001, *A Persistent Snapshot Device Driver for Linux*, Linux Showcase