An $O(1)$ Method for Storage Snapshots

Brian L. Stuart

ABSTRACT

The capability of taking snapshots is approaching ubiquity as a feature of file systems and data storage arrays. Here, we present an approach to structuring and managing snapshots in a storage space that provides for rapid creation and roll-back. This approach has been realized in the form of a Plan 9 kernel device that can be interposed between any pair of storage service and application. The kernel device has, in turn, been used to support an experimental file system. In this paper, we discuss the basic snapshot model, its implementation, and its application to a file system. Finally, we consider extensions to the model for supporting the deletion of snapshots.

1. Introduction

The ability of a file system or storage array to take snapshots has become de rigueur in the modern data storage world. As we discuss in Section 2., many of the recently developed file systems include snapshot capabilities as integral parts of their designs. Similarly, many Storage Area Network (SAN) products also include snapshot features as part of the block storage services they provide.

For the purposes of this work, we define a snapshot by the following characteristics:

1. Snapshots are immutable. We consider only the case of a snapshot which is preserved in time. In particular, clones (views of a data store which allow further modification) are outside the scope of this work.

2. Snapshots preserve the state of a data store at a particular moment in time. To view the state of a specific file as it existed at a specific point in history, the file as it exists in any snapshot taken between the specified point in time and the time of the next modification of the file will give that view. Collectively, a series of snapshots show a sampled view of the evolution of a data store over time.

3. Snapshots do not interfere with future use of the data store. Although the contents of a file are preserved by the act of taking a snapshot, the snapshot does not freeze the file for all time. Further modifications of the file can be made in the data store itself independent of the snapshot of its earlier state.

4. Snapshots can be accessed through the same mechanisms as the data store itself. This characteristic of snapshots distinguishes them from more traditional backups in which the data store is represented in a format that is in some ways different from the original data store preventing the normal file access tools and function calls from being able to act on the backup. Although the internal details of data representation may or may not be different in snapshots, we expect the implementation to provide the same interface as that of the primary data store. For data stores accessible via a network protocol such as 9P or NFS, we expect that the protocol and the implementation provide a way for a client to specify that it wishes to view the set of snapshots (or perhaps a particular snapshot).

There are two fundamental techniques whereby snapshots are usually implemented. The first of these techniques is block copying. As with a backup, a snapshot is taken by copying the blocks of the data store to a separate medium where the snapshots are stored. Normally, as an optimization, only the blocks (or files) that have changed since the last snapshot are copied. Of course, when using such an optimization, some form of indexing distinct from that used in the active store must indicate the location of files and blocks that are carried over from one snapshot to the next. This approach is particularly useful when the nature of the storage medium for snapshots is substantially different from that of the active data store. For example, the snapshots might reside on spinning disks where the active data store resides on solid-state flash memory devices.

The second major technique for implementing snapshots is copy-on-write (COW). Using this technique, a snapshot is taken by marking all currently writable blocks as copy-on-write. Later, when an attempt is made to write to such a block, a copy is made and the write is carried out on the copy. As with the block copying technique, indexing in the data store must identify which copy of a block is being referenced in a particular snapshot.

As detailed in Section 3., the present approach to snapshots is based on a COW mechanism where the physical location of a block determines its COW status. Specifically, this approach implements write once, read many (WORM)-like behavior such that blocks older than the most recent snapshot are frozen in time. In this way, we can implicitly and rapidly mark a large number of blocks as needing to be copied on the next write. In Sections 4.–7., we examine various considerations regarding the realization and application of this technique. The WORM-like characteristic of this approach isn't, however, immediately amenable to deleting snapshots. Section 8. discusses some potential methods of supporting snapshot deletion within the present snapshot technique.

2. Related Work

As mentioned earlier, snapshots in a file system or a storage space have become quite common. In this section, we take a look at several examples that are relevant to the present work. The particular examples chosen here are intended to be representative, rather than exhaustive.

2.1. Fossil/Venti and KenFS

Thompson's Plan 9 file server [5, 13] includes the ability to dump the state of the file system to WORM storage on demand or on an automated schedule. Later Plan 9 work includes the venti [6, 7] storage server which provides WORM-like behavior. It is most often used by fossil [8] for dump snapshots. The details of how our file system [11] presents its snapshots is modeled on that of the original Plan 9 file system and subsequently by fossil using venti. In both of these cases, snapshots are stored on what is conceptually (and in the earliest incarnations of the Plan 9 file system, literally) WORM media. The effect of "carving snapshots in stone" is a characteristic shared by the snapshots in the present work.

To avoid potential confusion, we point out that fossil supports two types of snapshots: ephemeral and archival. The mechanism for ephemeral snapshots is similar to that discussed in the present work, but the resulting snapshots are not persistent and their space is reclaimed in time. Fossil's archival snapshots are implemented by writing all active blocks to venti, providing the same persistence as the techique discussed here. Thus for purposes of comparison, one should read all references to snapshots in fossil as referring to the archival operation of block-wise copies to venti.

Both of the traditional Plan 9 file systems maintain caches where writes take place. When a snapshot is taken, the blocks in the cache that are to be part of the snapshot are scheduled for writing to a separate snapshot storage area. At the highest level of abstraction, this is very much like the active region distinct from the snapshots we discuss in Section 3.. There is, however, a fundamental difference of design. In the former cases, the active area is in a fixed location on potentially different media from the snapshots. In the case of the present system, the active region and the snapshots share common media, and the active area moves across the media like a sliding window. Although we continue to

investigate the relative merits of the two architectures, one benefit the present work does give is very rapid snapshot and roll-back operations.

2.2. Ext3cow

The Linux operating system has also provided a platform on which a number of storage research projects have been based. Among these is ext3cow [4] which implements a versioning file system based on similar copy-on-write techniques to those we use here. Because all file system updates in ext3cow induce copies, it effectively creates a snapshot on each write with one second resolution. Although the present work can be used for fine-grained snapshots, we have focused on applications where snapshot granularity on the order of one day is appropriate, and intra-day versions are unnecessary.

Ext3cow also illustrates a different approach to making snapshots available in the name space. Where our file system that uses the snapshots described here makes them available as individual hierarchical trees, ext3cow allows each component of a path name to be modified with version information. This difference is directly related to the granularity and intended application of the snapshots in the two different systems.

2.3. BTRFS

BTRFS [10] is one of the more recent major file systems developed for Linux. It is based on data extents arranged as B-trees with copy-on-write semantics [9] and uses reference counts to determine when extents can be reused. Although it uses COW for all updates (unless the nocow option is set), it does not keep all versions as ext3cow does. Instead, BTRFS uses a snapshot mechanism that creates a copy of the B-tree root, thus increasing the reference count on all nodes in the tree. BTRFS avoids walking the entire tree updating reference counts by deferring the reference count updates beyond a single generation of descendants in the tree. The effect of copying the root in this way is the creation of a clone, allowing updates to both "copies" of the tree.

The core technique of creating a snapshot by making a copy of a single metadata structure is similar to the techniques we report here. However, the details of our approach differ from those of BTRFS in a number of respects. For simplicity, we have chosen to use a large fixed-size block as the unit of COW rather than extents. Furthermore, we only perform a COW on those blocks that have not changed since the last snapshot following much the same policy as BTRFS does with the nocow option set.

2.4. ZFS

The SUN file system, ZFS, also provides a snapshotting facility [12]. At a superficial level, the relationship between snapshot support in ZFS and the approach described here is much like a combination the relationships described in this section. By that, we refer to the design of snapshots in ZFS as part of a file system and that clones can be created and snapshots deleted. Like several of the systems discussed in this section and the work reported here, ZFS uses a copy on write approach to implementing snapshots and clones. However, unlike the traditional Plan 9 file systems and the file system enhancements described here, ZFS doesn't make the set of snapshots collectively available in a unified name space.

2.5. Coraid VSX

For storage managed at the block level, the Coraid VSX product [1] includes a number of snapshotting features. Indeed, it provides a more rich set of snapshotting features than the system described in this report. In particular, it directly implements snapshot removal and the creation of clones (writable snapshots).

The basic technique of treating all pre-snapshot blocks as copy-on-write and using the new copy as the active block is modeled directly on the VSX approach. However, because the VSX implementation directly supports removal and clones and because it is integrated with the other functions of the VSX,

the process for creating a snapshot is relatively complex. As a result, its metadata updating as part of the snapshotting process is a $O(n)$ operation.

The divergent objectives between the two efforts are also apparent in the architectural approaches taken. Whereas the VSX implementation of snapshotting is aimed at providing a subset of the features of an integrated product, the present work is aimed at providing a snapshotting component that can be used in a number of contexts. It is this distinction that led to the implementation of this system as a kernel device, rather than integrating snapshot functionality into one or more applications.

2.6. NetApp WAFL

The Write Anywhere File Layout (WAFL) from NetApp also provides snapshotting based on copy on write. In a 2008 blog [3], the original developer of WAFL, Dave Hitz, stated "My current view is that WAFL *contains* a filesystem, multiple filesystems actually, but that's different from *being* a filesystem." This conclusion is based on the observation that WAFL has a top half which handles traditional file system functions, and a bottom half which handles disk and data management. The snapshot facility is described as being part of the bottom half. That certainly sounds similar to the architecture described here where the file system runs in a snapshotted space provided by devsnap. However, if we look deeper, we find that they are quite different. In [2], WAFL snapshots are described as being implemented by copying the root i-node. Furthermore, we find that the representations of a block's status within the file system and its status within the storage pool are directly coupled. Finally, all of the block management is based on physical block numbers, lacking the logical to physical block translation used in the system described here.

3. Snapshot Design

In the course of investigating how best to provide snapshotting in an experimental file system to be reported in a Coraid technical report [11], we have developed a technique and a mechanism for snapshotting the underlying storage space. The technique applies a COW approach to the raw blocks in a data store, residing in a layer underneath any file system. Thus, snapshots for a file system or a block storage structure fall out almost for free. In this paper, we report the details of the snapshotting design and implementation.

The core mechanism for this approach to snapshotting has its genesis in a question on an operating systems qualifying examination. The problem asked how one might approach supporting a UNIX-like file system directly on WORM media. One approach to this problem is the use of a block forwarding table. When a new block is allocated from the free list and written, no action outside of the normal operations is taken. However, when an existing block is modified, a new block is allocated from the free list and the modified block is written there. In conjunction with that, the original block's entry in the forwarding table is changed to point to the newly allocated block. We note that this technique implements the COW semantics. When reading a block, we consult the forwarding table and follow the chain of forwarding pointers until reaching one that is not forwarded. It is in that block where the current contents can be found.

We adapt this idea for implementing snapshots by recasting the forwarding table as a logical block number to physical block number translation table and use it to implement the COW semantics. The translation table here, denoted $\Pi : l \mapsto p$, differs from the WORM block forwarding table in two key respects. First, free blocks are not allocated from physical space, but from a logical space of blocks. This means that on the first allocation of a block an entry must be added to Π whereas in the forwarding table newly allocated blocks are left identified as unforwarded. Second, we assume that we are working with re-writable media. Thus prior to freezing by a snapshot, blocks and entries in the translation table can be modified multiple times. However, once frozen by a snapshot, blocks acquire COW status.

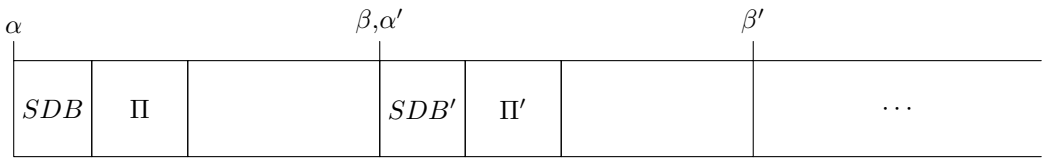

Figure 1: Version 1 Snapshot Structure

3.1. Snapshot Structure: Version 1

With this structure in place, we can see that a snapshot can be described uniquely with a translation table and a small amount of additional descriptive information. We place this additional information in a Snapshot Descriptor Block (SDB) which sits just before the snapshot's translation table. A snapshot is created, then, by writing an appropriately initialized SDB and copying the currently active Π table. The new SDB and table become the active ones and new blocks are allocated after the table. The effect is that snapshots are laid out sequentially in the storage media with the last one always being the active data store. This layout is illustrated in Figure 1.

The SDB stores several items. The value n gives the size of Π in blocks, α is the block number of the SDB, and β is the first free block in an active partition (as well as the SDB block number of a subsequent snapshot). Independent of β, the SDB also contains a pointer to the next SDB which is nil in the currently active data store. It also contains a textual string naming the snapshot.

The translation table, Π, is a simple array of block pointers, indexed by logical block number. Unallocated logical blocks are identified by a nil block pointer. Only the Π table in the currently active region can be written, because of the immutability of snapshots. Entries that point to physical blocks located to the left of the currently active SDB refer to blocks that are part of snapshots, which implicitly possess the COW property. Those that point to physical blocks to the right of the currently active Π table refer to blocks that are mutable, having been allocated or copied since the most recent snapshot was taken.

3.2. Snapshot Operation: Version 1

We now turn to the operation of the snapshot system. This section focuses on the four primary operations of reading blocks of data, writing blocks of data, creating a new snapshot, and reverting to an earlier snapshot. Throughout this discussion, the notation Π_l refers to the lth entry of the translation table Π. In other words, it is the physical block number where logical block l is stored in the relevant snapshot. Similarly, the notation Σ_p refers to the contents of physical block p in the storage media. If physical block p is part of a Π table, then the ith entry in that block of the table is identified by $(\Sigma_p)_i$. Finally, $\Sigma_{[n,m)}$ refers to the contents of blocks n through (but not including) m. The discussion is mostly structured in terms of single blocks, but support for partial blocks and multiple blocks is added in the obvious way.

With the structure described above, it is clear that we do not need much of the block meta-data that would normally be present. For example, from the perspective of snapshot handling, we know that a block is free if and only if it's physical block number is greater than or equal to β. Similarly, we know that a block has the COW status if and only if its physical block number is less than α. These observations make the following algorithms substantially simpler than they would otherwise be.

3.2.1. Block Reads

Reads are the simplest operations we perform in the snapshot system. Conceptually, we simply look up the block we want in the relevant translation table and then perform the read from the physical block identified there. More formally, we can describe the operation as in Algorithm 1.

ALGORITHM 1: Block Read

Input: Logical block number l to read
Output: Physical block contents
$p = \Pi_l$, where Π is the translation table for the snapshot
if $p = \Lambda$ **then** logical block l never allocated
    return 0 Block
else
    return Σ_p
end

Note that because contiguous logical blocks may be mapped to non-contiguous physical blocks, requests that span logical block boundaries must be broken down and processed as multiple operations, each fitting within one logical block.

3.2.2. Block Writes

As one might guess, writes are a bit more complicated than reads. This is because they require allocating blocks and copying when necessary. The algorithm for a write works as in Algorithm 2.

ALGORITHM 2: Block Write

Input: Logical block number l to write and data D
$p = \Pi_l$, Π must be the translation table for the currently active store, because snapshots are immutable
if $p = \Lambda$ **then** writing a block never yet written
    $\Pi_l \leftarrow \beta$
    Store D into block β
    $\beta \leftarrow \beta + 1$
end
else if $p \geq \alpha$ **then** writing a block already copied since last snapshot
    Store D into block p
else need a COW
    $\Pi_l \leftarrow \beta$
    $\Sigma_\beta \leftarrow \Sigma_p$
    Store D into block β
    $\beta \leftarrow \beta + 1$
end

Although we have accounted for all values of p, note that not all values of p are possible. If $\alpha \leq p < \alpha + n + 1$, then we have a logical block mapped to a physical block where an SDB or a Π block is stored. This is not possible in a correct implementation of these algorithms.

3.2.3. Taking a Snapshot

As illustrated in Algorthm 3 the act of taking a snapshot is relatively simple. We need only close the currently active region as a snapshot and open a new active region by making a copy of the current SDB and Π table. One effect of this is resetting α to the old value of β. This, in turn, implicitly marks all allocated blocks as COW. Thus we mark them all in a single step, rather than having to go through a table of blocks, marking each one.

Note that these operations are all $O(1)$ with respect to the number of dirty, unique, or allocated blocks. Thus the time to take a snapshot does not depend on the number of dirty or unique blocks in any earlier snapshot or the currently active store.

ALGORITHM 3: Taking a Snapshot

$\Sigma_{[\beta,\beta+n+1)} \leftarrow \Sigma_{[\alpha,\alpha+n+1)}$, Copy the SDB and Π table
Update the name of the old SDB and set its next pointer to β
Update the new SDB as: $\alpha' \leftarrow \beta$ and $\beta' \leftarrow \alpha' + n + 1$

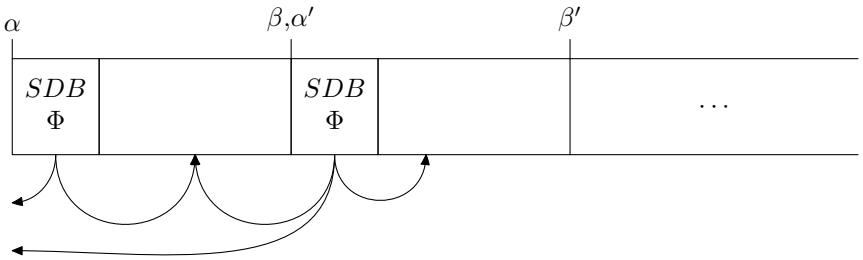

Figure 2: Version 2 Snapshot Structure

3.2.4. Reverting

The effect of reverting to Snapshot x is the removal of all changes made since x was taken and the resetting of x as the active region. This also results in x and all subsequent snapshots being removed. Implementing reverting is simply a matter of relabeling the SDB of x as the active region and as no longer pointing to any subsequent snapshots.

3.3. Snapshot Structure: Version 2

The initial prototype of this snapshot mechanism followed the structure and operation outlined in the previous subsections. However, it was noted that there are certain inefficiencies in that approach. Although the number of blocks copied in a snapshot operation (and, by extension, the number of blocks of overhead consumed by a snapshot) are constant with respect to the number of modified blocks, it is proportional to the overall size of the storage space. Furthermore, the subset of Π that changes from one snapshot to the next is relatively small. Therefore, a large subset of Π is redundant from snapshot to snapshot.

This deficiency is addressed in a second version of the basic design. In particular, a second level of indirection is added, distinguishing between logical and physical block numbers of the blocks that make up the translation table Π. The same COW strategy we use for data block also applies to the blocks making up Π. For the purposes of notation, we refer to this second-level translation table as Φ. With a suitable choice of block size, we can ensure that Φ is fully contained within the previously unused portion of the SDB. This, in turn, means that the snapshot procedure involves writing a single block, and that the storage overhead is a single block per snapshot. This revised structure is illustrated in Figure 2. The arrows leaving the SDBs in the figure are pointers in Φ identifying the physical blocks that make up Π. Those pointing to the left are entries of Φ that are inherited from the previous snapshot. Those pointing to the right are entries for blocks of Π that are either newly allocated or that are modified since the last snapshot was taken.

3.4. Snapshot Operation: Version 2

As expected, the details of the specific operations are a little more involved with this revised structure. However, as we see in this section, they are straightforward extensions of the previous algorithms. In the following discussion, B denotes the number of block numbers that are stored in a block.

3.4.1. Block Reads

The only thing different for reads in the new structure is the need for a two-step block number translation. This is illustrated in Algorithm 4

ALGORITHM 4: Block Read

Input: Logical block number l to read
Output: Physical block contents
$i = \lfloor \frac{l}{B} \rfloor$
$j = l \bmod B$
$m = \Phi_i$
if $m = \Lambda$ **then** unallocated block of Π
    return 0 Block
else
    $p = (\Sigma_m)_j$
    if $p = \Lambda$ **then** logical block l never allocated
        return 0 Block
    else
        return Σ_p
    end
end

3.4.2. Block Writes

Similarly, block writes require a two-step translation. Because not all blocks of Π are necessarily allocated, we might have to allocate a block for the translation table as well as for the data. We also note that a write operation might result in two COW operations. If the most recent version of the data block itself is in a frozen snapshot, then it is copied. Also if this is the first block mapped in a particular Π block since the last snapshot was taken, then the Π block must also be copied. Algorithm 5 shows the details of this.

3.4.3. Taking a Snapshot

Although the revised snapshot algorithm as shown in Algorithm 6 looks nearly identical to the first version, it involves copying only a single block as opposed to $n + 1$ blocks.

3.4.4. Reverting

Reverting to a previous snapshot is not changed by these revisions in the structure of the translation table.

4. Snapshot Kernel Device

This design for storage snapshots has been realized in the form of a Plan 9 kernel device. The structure and design of Plan 9 makes the inclusion of this type of mechanism particularly straightforward. Because all storage space servers present the spaces as simple data files and all clients expect to work through data files, we can construct a simple kernel device that both uses and presents the same data file interface. Such a device can then be interposed between any client and storage space, providing snapshotting for any storage application. Furthermore, this approach makes it easy for the snapshotting device to present all snapshots as data files too, allowing the clients to interact with snapshots in the same way that they interact with the active store. In the remainder of this section, we discuss the kernel device we

ALGORITHM 5: Block Write

Input: Logical block number l to write and data D
$i = \lfloor \frac{l}{B} \rfloor$
$j = l \bmod B$
$m = \Phi_i$
if $m = \Lambda$ **then** unallocated block of Π
    $\Phi_i \leftarrow \beta$
    $(\Sigma_\beta)_j \leftarrow \beta + 1$
    Store D into block $\beta + 1$
    $\beta \leftarrow \beta + 2$
end
else
    $p = (\Sigma_m)_j$
    if $p \geq \alpha$ **then** writing a block already copied since last snapshot
        Store D into block p
    else p is either never written or needs a COW
        if $m < \alpha$ **then** need to COW the relevant block of Π
            $\Phi_i \leftarrow \beta$
            $\Sigma_\beta \leftarrow \Sigma_m$
            $k = \beta$
            $\beta \leftarrow \beta + 1$
        end
        else
            $k = m$
        if $p \neq \Lambda$ **then** needs a COW
            $\Sigma_\beta \leftarrow \Sigma_p$
        $(\Sigma_k)_j \leftarrow \beta$
        Store D into block β
        $\beta \leftarrow \beta + 1$
    end
end

have implemented for this. Because the current code implements Version 2 of the design, the following discussion is based on that version.

The source file for the driver is devsnap.c and it uses the device letter, $\mathbb{P}$. (Plan 9 supports the full Unicode character set for device identifiers.) The current prototype version is less than 700 lines of code. When the system first comes up, devsnap presents a single file in its name space. The file, `ctl`, serves the conventional role of providing a control and status interface for the snap device. There are five commands that can be written to the control file:

bind The command, `bind`, is used to attach an existing snapshotted store to the snap device. It takes a single argument which is the data file representing the storage space where snapshots are stored. E.g.

```
bind '#ℙ' /n/snap
echo bind '#S/sdE1/data' > /n/snap/ctl
```

Devsnap scans the data file and creates a data file for each snapshot (including the active store) it finds there.

ream The `ream` command is used to initially format a storage space where snapshots are to be managed.

ALGORITHM 6: Taking a Snapshot

$\Sigma_\beta \leftarrow \Sigma_\alpha$, Copy the SDB and Φ table
Update the name of the old SDB and set its next pointer to β
Update the new SDB as: $\alpha' \leftarrow \beta$ and $\beta' \leftarrow \alpha' + 1$

It takes two arguments, a name given to the active store and the path name of the data file where the snapshots are to be managed. E.g.

```
echo ream fs '#S/sdE1/data' > /n/snap/ctl
```

Reaming is implemented by simply writing the first SDB and an empty Φ table at the beginning of the storage space.

snap Taking a snapshot is triggered by the `snap` command which takes two arguments. The first argument is the name of the active store to be snapshotted, and the second argument is the name to be given to the newly created snapshot. E.g.

```
echo snap fs fs.20131119 > /n/snap/ctl
```

In response to this command, devsnap applies Algorithm 6 given in Section 3.4.3..

unbind The `unbind` command takes a single argument which is the path name for the storage space data file previously connected with a `ream` or `bind` command. E.g.

```
echo unbind '#S/sdE1/data' > /n/snap/ctl
```

The effect of an `unbind` is the closing of the connection to the underlying device and the removal of the associated files from devsnap's name space.

revert As with taking a snapshot, reverting takes two arguments, the name of the active region and the name of the snapshot to which we should revert. E.g.

```
echo revert fs fs.20131119 > /n/snap/ctl
```

The effect of this command is to roll back the store to the same state it was in at the time that Snapshot `fs.20131119` was taken.

Reads of the `ctl` file return a list of the snapshots and active stores currently known to devsnap. E.g.

```
0 fs #S/sdE1/data 1 10720 10849 10863
1 fs.20131119 #S/sdE1/data 3 10683 0 10683
2 fs.201311191 #S/sdE1/data 3 10720 10683 10849
```

The items on each line are: the internal slot number, the name of the snapshot, the path name to the underlying data store, the internal flags, the number of blocks in use, the value of α, and the value of β.

For each snapshot, devsnap provides a data file in its name space. E.g.

```
--rw-rw-r-- ℙ ...             0 ... ctl
--rw-rw-r-- ℙ ... 1000204886016 ... fs
```

```
--r--r--r-- ℙ ...   11201937408 ... fs.20131119
--r--r--r-- ℙ ...   11240734720 ... fs.201311191
```

Note that the snapshots are given read-only permissions, and the active store is given read-write permissions. Also the sizes differ among the data files. The size of the active store is the size of the underlying data file. The size of a snapshot is the number of bytes used by that snapshot.

In devsnap, we have chosen to use a block size of 1MB and to specify logical and physical block numbers as 64-bit integers. It should be noted that devsnap's block size is only used as the unit of allocation and COW and in no way constrains the block sizes used either by clients or by the underlying storage. We do, however, assume the ability to write 512-byte sectors to the underlying storage. This happens when updating a value in the Π table. Instead of writing the whole 1MB block, we write only the 512-byte sector containing the change. I/O as a result of client read/write requests is carried out in the units requested by the client. Thus the behavior continues to be the same as it would be without the addition of devsnap.

The parameters we have chosen for devsnap mean that each block in the Π table contains 128K entries. Since each block is 1MB, a block of the Π table describes 128GB of logical space. Because very little of the SDB is used for descriptive information, almost all of it is available for the Φ table. Thus a two-level table with 1MB blocks can manage about 16PB. To support a larger storage space, we can use 4MB blocks which allow for up to 1EB in total space. Similarly, a three level table can be used which would support up to 2ZB with 1MB blocks. As well as the currently active SDB and Φ table, we keep a one block cache for the Π table for each snapshot, allocated when we first access a snapshot. Therefore, as long as jumps between 128GB regions of the logical space are relatively rare, the snap device introduces very little overhead.

5. Boot Modifications

To assist in booting from a storage space managed by devsnap, we have augmented /sys/src/9/boot/local.c. If the bootargs string or the string entered at the "root is from" prompt begins with "#ℙ" indicating that the root is to be taken from devsnap, the second token, if any, indicates the data file to bind to devsnap. For example, if we enter the string:

```
local!#ℙ/fs #S/sdE1/data
```

at the "root is from" prompt, the boot code will issue the message

```
bind #S/sdE1/data
```

to devsnap before running the file system code. This would result in the system running with its root taken from a snapshotted file system stored on a device handled by devsd.

6. File System Support

Because each snapshot is provided in its own data file, it would be relatively easy to add snapshot support to any file system by pointing the file system in a read-only mode at the appropriate snapshot data file. However, this would become quite cumbersome as the number of snapshots grew, particularly with respect to any attempts to export the set of snapshots collectively via 9P or NFS. Therefore, when adding support for snapshotting to an experimental file system, we chose instead to give the file system knowledge of the set of snapshot files and let it present them in a useful way.

Our support for snapshots takes advantage of support for the creation of multiple roots similar to that supported by ZFS. (Alternatively, these can be thought of as multiple independent file systems sharing the same storage space.) Access to them is specified by the attach name in 9P and by the mount path name in NFS.

For snapshot use, we create a root called /dump which provides semantics compatible with existing Plan 9 history tools. Contained within this root directory are a set of directories, each named by the year for which at least one snapshot was taken. Inside the year directories are directories named by the date on which the snapshot was taken. Each entry is of the form $mmdds$ where mm is the month, dd is the day of the month, and s is a sequence number which is nil on the first snapshot of the day and the numbers 1, 2, 3,... for subsequent snapshots. What makes these entries special is the presence of a meta-datum named snap. The value of the snap meta-datum is a string giving the name of the data file for that snapshot.

Continuing with the example we have been showing, we could access the snapshots as follows:

```
term% 9fs dump
term% lc /n/dump
2014
term% ls -l /n/dump/2014
d-rwxrwxr-x M 42 bootes bootes 0 Feb 12  2014 /n/dump/2014/0212
d-rwxrwxr-x M 42 bootes bootes 0 Feb 12  2014 /n/dump/2014/02121
d-rwxrwxr-x M 42 bootes bootes 0 Feb 13  2014 /n/dump/2014/0213
d-rwxrwxr-x M 42 bootes bootes 0 Feb 14  2014 /n/dump/2014/0214
...
term% history papers/acm/snapshot.tex
Aug 28 14:38:50 EDT 2014 papers/acm/snapshot.tex 44208 [stuart]
Aug 27 21:57:22 EDT 2014 /n/dump/2014/0827/.../snapshot.tex 46048 [stuart]
Aug 26 22:02:12 EDT 2014 /n/dump/2014/0826/.../snapshot.tex 45365 [stuart]
Aug 25 17:54:02 EDT 2014 /n/dump/2014/0825/.../snapshot.tex 45258 [stuart]
Aug 24 21:09:27 EDT 2014 /n/dump/2014/0824/.../snapshot.tex 43448 [stuart]
Aug 23 00:25:37 EDT 2014 /n/dump/2014/0823/.../snapshot.tex 42991 [stuart]
Aug 22 23:03:47 EDT 2014 /n/dump/2014/0822/.../snapshot.tex 43035 [stuart]
Aug 21 16:52:40 EDT 2014 /n/dump/2014/0821/.../snapshot.tex 38902 [stuart]
term% ls -lp /n/dump/2014/0821/usr/stuart/papers/acm/snapshot.tex
--rw-rw-r-- M 42 stuart stuart 38902 Aug 21 16:52 snapshot.tex
term% ls -lp /n/dump/2014/0827/usr/stuart/papers/acm/snapshot.tex
--rw-rw-r-- M 42 stuart stuart 46048 Aug 27 21:57 snapshot.tex
```

where the ellipsis is used to shorten the path names for formatting purposes. When walking a directory tree, we walk as normal until we reach a node with a snap value. At that point, we open the snapshot data file and continue the walk starting from its default root. The net effect is much like a mounted file system.

Because the details of the directory traversal are very different between 9P and NFS, so too are the details of our snapshot "mounting." In the case of 9P, we keep track of which tree we're in by attaching an open file descriptor to the internal Fid structure. The file descriptor is reference counted and on a destroyfid call where we are dropping the last reference to it, we close the file. In the case of NFS, we keep track of where we are by extending the file handle passed back to the client for files in a snapshot tree. In particular, we append a unique identifier of the file in the /dump tree whose meta-data includes the relevant value for snap. In both cases, accesses to snapshots are read-only and uncached.

To support creating snapshots, a new command has been added to the console interface of the file system. Writing the command snap to the console causes it to command devsnap to create a snapshot. The name is based on the dump conventions discussed earlier. Specifically, if the active store name is x, then the dump will be named x.yyyymmdds. The snapshots shown in the examples of this report were all created by a periodic scheduled triggering of this mechanism.

Similarly the revert command has been added to support rolling the file system back to an earlier

snapshot. The command takes a single argument which identifies the snapshot to which we are rolling back. For convenience, it can be specified either in the form it appears in the snapshot name, yyyymmdds or in the partial path name seen in the dump file system, yyyy/mmdds.

7. Applications in Block and Object Storage

The snapshotting provided by this technique is not restricted to applications in file systems. Because it operates directly on an array of blocks and presents the same array of blocks, it can just as easily be used with a storage space used to provide block or object storage. However, as with 9P and NFS exports of the file system, we must establish some mechanism by which a client can gain access to the snapshot. Most SAN designs identify exported block and object storage spaces by logical unit numbers (LUNs). Such a system can export a snapshot by assigning a LUN to the snapshot but with read-only permissions.

We should point out one interesting effect of snapshotting LUNs in this way. Just as the implementation of devsnap as a kernel device creates a very general snapshotting mechanism within the system, applying it to LUNs creates a very general snapshotting mechanism for clients. Whether the client is running Linux, Solaris, vmware, or Windows, it will have access to snapshots of its native file systems as stored on a block storage appliance.

8. The Snapshot Deletion Problem

From the foregoing discussion in this report, it is clear that this approach to snapshotting is fundamentally based on a WORM-like model of storage. In particular, once a region has been closed by taking a snapshot and by starting a new active region, the space in the first region is never reused. Another way of viewing this is that we never write to any physical block whose block number is less than α in the active system. Although this characteristic of our approach substantially simplifies and streamlines many of the implementational details, it does impede any desire to remove snapshots. In this section, we consider some ways of providing snapshot removal.

8.1. Redacting Snapshots

We identify the two primary motivations for removing snapshots as data obsolescence and space recovery. In the case of data obsolescence, the intent is usually the removal of references to sensitive data. Merely removing the mapping to old snapshots is actually quite easy. If we copy the succeeding SDB and Φ table on top of the ones we want to remove, we lose any mappings to the blocks unique to the snapshot we are removing while maintaining references to other blocks. More formally, we copy $\Sigma_{\alpha_n} \leftarrow \Sigma_{\alpha_{n+1}}$. Although the references to those blocks are removed, the data blocks themselves remain intact and are therefore subject to forensic recovery. To make the data less susceptible to recovery, we would need to overwrite the blocks that are uniquely allocated to the snapshot being removed. This would, in turn, require the reference counting technique discussed in the next subsection, and could be piggy-backed on the space compaction discussed there.

8.2. Space Compaction

If our purpose in removing snapshots is recovering space not used in recent snapshots, then there are a couple of approaches we can use to reclaim those blocks no longer referenced after overwriting an SDB and Φ table. The first approach we consider is compaction, sometimes referred to as defragmentation. The basic idea is a simple one. We shift logical blocks that are in use down into the physical blocks that are freed by the snapshot removal. Compaction has the benefits of not requiring any additional mapping of block positions and of maintaining (and even increasing) locality of reference. The disadvantage of compaction, however, is the large amount of block copying that it incurs. Consequently, we lose the $O(1)$ snapshot behavior mentioned earlier.

The primary technique for knowing which blocks are freed in a snapshot removal is to maintain a reference count of each physical block. Each reference count is incremented on block allocation and snapshot creation, and it is decremented on COW and snapshot removal. Blocks whose reference counts are decremented to zero are now free. Because the reference counts are global to the whole storage space, only a single copy is kept avoiding adding more overhead to the snapshot regions.

8.3. Unbounded Space Simulation

The second approach we consider for reclaiming space from removed snapshots is simulating an unlimited storage space. The basic idea is that when we remove a snapshot, we logically tack all of the blocks unique to it to the end of the storage space expanding it by that number of blocks. Then as β grows and new snapshots are taken, we reuse the blocks that had been part of earlier snapshots.

As with the compaction approach in the previous subsection, we need to maintain reference counts to know which blocks are freed. Simulating the unbounded logical storage space can be done relatively simply by maintaining another logical-to-physical translation table. This table is a little more involved than the ones we use as part of the snapshots. The reason is that the logical space is unbounded and thus potentially much larger than the physical space to which it maps. Consequently, a simple table indexed by logical block number would not be practical. An effective alternative would be a hash table keyed on the logical block number.

9. Conclusion

In this report, we have examined a novel technique for providing storage snapshots. The technique is both simple and general, and it places very little in the way of demands on the storage applications using it. Our prototype implementation is less than 700 lines of code and supports both file-structured and block-structured storage space. It can be used with individual raw disks, RAID configurations, and network attached block storage. This technique has proven to be quite powerful as evidenced by the simplicity of including support for it in an existing file system. The bulk of the necessary changes are a few hundred lines that make the set of snapshots available in a single hierarchical name space. Finally, as suggested by the example in Section 6., the text editing and formatting of this paper were carried out on a file system operating on the snapshot mechanism described here.

10. Acknowledgements

We would like to express our appreciation to Coraid for its support during this research. In particular, Brantley Coile established the research environment in which the work was conducted. Erik Quanstrom provided a valuable sounding board and helpful questions and suggestions, particularly in the transition from Version 1 to Version 2 of the design discussed here.

References

[1] Coraid Inc. Ethernet san storage virtualization etherdrive vsx3500, 2010.

[2] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[3] David Hitz. Is wafl a filesystem? https://web.archive.org/web/20140715102135/https://-communities.netapp.com/community/netapp-blogs/dave/blog/2008/12/08/is-wafl-a-filesystem, archived on July 15, 2014, 2008.

[4] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Trans. Storage*, 1(2):190–212, May 2005.

[5] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[6] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. Technical report, Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ, USA, May 2002.

[7] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[8] Sean Quinlan, Jim McKie, and Russ Cox. Fossil: an archival file server. World-Wide Web document, 2003.

[9] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Trans. Storage*, 3(4):2:1–2:27, February 2008.

[10] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3):9:1–9:32, August 2013.

[11] Brian L. Stuart. Toward unification of storage granularities. Coraid Technical Report, 2014.

[12] Sun Microsystems. ZFS On-Disk Specification. https://web.archive.org/web/20081230170058/-http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf, archived December 30, 2008, 2006.

[13] Ken Thompson. The Plan 9 file server. World-Wide Web document, Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ, USA, 2000.