Cluster Snapshot Block Device

Daniel Phillips

3rd September 2004

Abstract

For several years now, Linux has provided a block level snapshot facility, first as part of the logical volume manager, and more recently as part of the Device Mapper virtual block device subsystem. This work builds on the original concept and extends it to operate with shared storage devices accessed simultaneously by many cluster nodes over a storage network. In the process, some deficiencies of the existing design were corrected: when multiple snapshots are held simultaneously, multiple copying of snapshotted data is eliminated; memory footprint is reduced and made independent of volume size; the requirement for multiple separate snapshot store volumes is eliminated; and a single Device Mapper target serves for both snapshot and origin devices.

1 Background

1.1 Why Snapshots?

In a typical enterprise computing environment, a file system is relied on to store, organize, keep track of and make available the files generated by the users. The enterprise computing system may have hundreds or even thousands of users and the file system is required to support large numbers of users reading from and writing to the disk storage space managed by the file system. In today's business environment, the file system may be needed around the clock. In many systems, the file system must be continuously available and cannot be shutdown temporarily without incurring unacceptable business costs.

Full-system backup copies can be used to restore files in the event of system failure, accidental deletion, file corruption, unintended edits, etc. Regularly making backups is an essential safety measure in many systems. In order to make a backup of a file system using traditional copying techniques, it is necessary that the contents of the file system be stable. If the file system is live, then changes may be made to the contents, even as they are being copied. Thus, the file system must be "off-line" in order to make an effective back up. In a typical enterprise environment, this process may be performed at night or during off hours. The file system may be unavailable for several hours each night or once a week as the contents of the file system are read out and copied to tape. The end result is a physical copy of the original data. As enterprises grow ever larger, and the stored data accumulates, the volume of data that has to be copied increases and the time that it takes to make a backup of the file system using traditional techniques is no longer manageable. In some environments, it may take more than a day to make a backup copy. It may be unacceptable to take a filesystem offline for such a long time.

It is possible to achieve the effect of suspending and making a copy of an entire filesystem, but without interrupting the operation of the filesystem. A 'snapshot' is a virtually instant copy of a defined collection of data created at a particular instant in time. Backups and other functions can be performed in a leisurely way on a snapshot image without impacting system availability.

Beyond online backup, snapshots have various uses. For example, one might create a snapshot before making a trial install of some new system software. If the install is unsuccessful, the trial can be reverted by copying the snapshot back to the origin device.

1.2 Why Block-Level Snapshots?

Snapshots may be implemented at the block device level, where the disk contents are viewed as mere blocks of data, or at the filesystem level, where the filesystem contents are viewed as having a particular organization. It has been argued that the latter is the best choice, since without explicit knowledge of filesystem structure it is impossible to know which blocks are free and which are not. In the absence of that knowledge, a block level snapshot scheme will sometimes waste time preserving snapshot data that is just filesystem free space.

On the other hand, it is much simpler to implement snapshots at the block level than to incorporate them into the design of a possibly already-complex filesystem. Furthermore, a block level snapshot works for all filesystems, not just one particular filesystem. And finally, we already have a natural way of representing a snapshot to the system: as a virtual block device.

1.3 Multiple Snapshots

Multiple snapshots of the same filesystem may be maintained simultaneously, each created at a different point in time. In the past, this has been implemented in a straightforward way by giving each snapshot its own fixed-size storage area. This approach, though simple, has serious drawbacks. Each time a block of data is changed on the filesystem, it must be written to the storage for each snapshot to which it belongs. The cost of these multiple writes increases directly with the number of snapshots held. In practice, performance degrades noticeably with more than one or two simultaneous snapshots. Depending on the lifetime of a snapshot and the number of changes made to the original file system, each snapshot store may need to be as large as the original disk. In the absence of some clever scheme to allocate storage on demand, these multiple snapshot stores may be largely empty. In any event, it is probable that multiple redundant copies of data will be stored.

It is clear that to avoid the above problems, data must be shared between multiple snapshots whenever possible, both to reduce copying time and to save space. But this sharing requirement has profound implications, versus the simple existing design:

- We need structures to keep track of which data is shared by which snapshots
- We need a space allocation scheme for the snapshot store
- Both of the above need to be persistent and durable

I elected to use a filesystem-style bitmap allocation scheme to manage the allocation of snapshot store data and metadata, a btree to keep track of snapshot data membership, and a journal to make those structures durable. The code to manage a snapshot store thus ends up looking like a simple filesystem, a far cry from the existing implementation of snapshots, and even further from the simplicity of a physical disk. But this is the price that must be paid to solve the pressing problems of the existing implementation.

1.4 Cluster Snapshots

In many environments, storage systems are clustered, with multiple server-class systems accessing the same storage device. The clients of the storage systems need to communicate amongst themselves in order to use the storage system in a coordinated way to avoid interleaving their changes in a nonsensical way. All members of a cluster must receive a consistent view of possibly multiple snapshots, even while other members of the cluster may be simultaneously writing to the same snapshots.

Network communication is required between cluster members for three distinct purposes:

1. Discovering which physical data blocks belong to a given snapshot
2. Synchronizing access to block level data shared between snapshot and origin
3. Synchronizing access to snapshot metadata

A traditional approach would rely on a distributed lock manager for synchronization, augmented with a message passing protocol to implement the data exchange required for (1) above. I elected instead to adopt a pure message-passing approach that combines synchronization and data exchange into a single set of messages; after all, a distributed lock managers implements its locks, in the end, with messages. This eliminates an entire layer of network synchronization messages, and results in a compact, efficient protocol that is easy to verify.

1.5 Durability and Performance Goals

As general purpose block devices, cluster snapshots need to operate just like real block devices. In particular, performance should not be noticeably degraded and data written to a snapshot virtual device should not be at any greater risk than data written to a real disk. The latter is a challenging requirement in view of the fact that a real disk is not required to maintain complex data structures to do its work. The cluster snapshot on the other hand, must maintain complex data structures durably, in other words, without being subject to corruption in the event of sudden interruption such as a power failure. We would not expect a real disk to become corrupted in that case, and so have the same high expectations of a virtual disk. An additional challenge is to achieve this durability without degrading performance.

The main tactic brought to bear on this problem is batching together of data structure updates. A localized group of writes to virtual volume data should result in a batch of localized updates to the volume metadata, which can be transferred to disk as a single transaction. A journal is used to make the transaction atomic and durable. The batching makes it efficient.

1.6 Why a Client-Server Architecture?

Question: How does one distribute a database across multiple cluster nodes? Answer: With much blood, sweat and tears. This I did not wish to attempt. Instead I decided that the snapshot store database would be centrally managed by a server process running on one of the cluster nodes, and that synchronization and information distribution would be by way of network messages, taking the form of queries and replies. Since the query protocol is very efficient, the client server architecture does not impose a large network load and is expected to scale readily to several hundred clients.

For very large clusters, the client-server architecture will no doubt eventually become a bottleneck. Fortunately there remains plenty of time to solve that problem, and there are no examples yet of filesystems of the type that could benefit from a snapshot block device, running on such large clusters.

1.7 Terminology

Here we review some terminology introduced by the original LVM snapshot implementation:

Origin volume
One of two block devices underlying a virtual snapshot device. This volume is mapped one-to-one to a snapshot origin virtual device. The virtual device could be removed and the underlying origin volume accessed directly, at the risk of losing the integrity of any snapshots sharing data with the origin.

Snapshot store
One of two block devices underlying a virtual snapshot device. This volume contains data chunks that were copied from the origin in order to preserve the integrity of snapshot data, or were written directly to the snapshot store via a snapshot virtual device. It also contains all metadata required to keep track of which snapshot store chunks belong to which snapshots, among other things.

Copy-out
The act of preserving a chunk of origin data by copying it to snapshot store.

Chunk
The granularity of snapshot copy-outs, a user-definable binary multiple of 4K block size.

Exception
A chunk of data in the snapshot store, belonging to one or more snapshots

1.8 History

To the best of my knowledge, the concept of a copy-on-write block device level snapshot first arose in discussions between Heinz Mauelshagen and Michael Marxmeier, in 1999 at the Cologne Linux Kongress conference. Heinz created a functional implementation as part of his Linux LVM project shortly after. In 2002, Joe Thornber refined these ideas and re-implemented the snapshot target as a Device Mapper virtual device. I began design work for a cluster snapshot in September, 2003. Once a design was available, a prototype kernel client for Linux 2.4 was coded in five days by Patrick Caulfield in December, 2003. Observing that a simple network echo is a good simulation of a cluster snapshot server for an important class of access patterns, Patrick was able to obtain early benchmark numbers at that time. Encouraged by the excellent performance, I began implementation work on a user space snapshot server early in the new year, bringing it to a testable state by June.

In July, the prototype kernel client was developed into a functional client for the snapshot origin, and benchmarks of a functional virtual origin device holding a snapshot were obtained. These confirmed the predictions set out in the design document, nearly a year earlier. By Monday August 16th, snapshot block device support had been added to the kernel client, completing the basic block device functionality. Ben Marzinski ported the 2.4 kernel client to Linux 2.6 later that week, in three days. This brings us to today, with a pre-alpha cluster snapshot prototype ready to be released on the "release early, release often" plan. Several more months of hard work is required to complete the fault tolerance and cluster infrastructure integration, which will bring this cluster snapshot block device to a usable state.

2 How a Block Device Snapshot Works

Block level snapshots in Linux are implemented using a copy-on-write strategy. When a snapshot is first created, it is exactly the same as the origin volume. Afterwards, any attempted write to the origin volume would also be a write to the snapshot, which would destroy the integrity of the snapshot, and isn't allowed. Before the write is allowed to proceed, the data that would be overwritten is copied to a snapshot store. Snapshot metadata is also recorded on the snapshot store, to keep track of the parts of the origin that have been preserved in the snapshot store. This metadata is consulted on each write access to determine whether or not origin data has already been preserved in the snapshot store, and thus is no longer shared with the origin.

A snapshot is presented to the system as a virtual block device. Any read or write to this device is intercepted and the snapshot store metadata is consulted to determine whether the target of the IO lies on the origin (i.e., hasn't been written to yet by any filesystem mounted on the origin device) or is in the snapshot store. By preserving overwritten data, then remapping each access to the virtual snapshot device to either the origin or snapshot store as appropriate, a view of a snapshot can be reconstructed on demand. A snapshot view can be accessed through its virtual snapshot device simultaneously with the origin volume, and these two virtual devices will behave as if they were completely independent. The virtual snapshot device takes care of the tricky synchronization required to make this work.

Writing to a snapshot through its virtual device is allowed. Data to be overwritten in the snapshot store does not need to be preserved, because it is not shared. However, if a write to the snapshot device would overwrite origin data, space is allocated for the write in the snapshot store instead and the snapshot store metadata is updated to reflect that.

Multiple snapshots can be held against an origin volume simultaneously. Any number of these simultaneous snapshots and the origin can be accessed as writable block devices, also simultaneously.

3 Cluster Synchronization

What must be done to turn a single-node snapshot device into a cluster snapshot device? Surprisingly little. A relatively thin layer of network synchronization has to be added, and some fail-over hooks. Block data itself is written to or read from the shared block device directly by the cluster snapshot client in the same way as if the block device were local. The main thing that is new for a cluster snapshot client is that it must decide for each IO request if global synchronization is required, and query the snapshot server over the network if it is.

The essential questions that a snapshot client must ask are: is a given logical data chunk shared by any snapshot, or is it unique? Is it stored on the origin volume, or in the snapshot store, and if so, at what physical address? For a write, either to the origin or a snapshot, by the time the server replies to a query about a given logical chunk the chunk is always unique, because the server will make it so if necessary. To make a chunk unique, the server allocates a new chunk in snapshot store, copies the shared data there, and updates the snapshot metadata. Having ensured uniqueness, the server replies to the query, the client associates the reply with a waiting write request, and finally the device-level write proceeds. In the case of a snapshot store write, the reply includes the physical address of the snapshot store chunk, which is not needed in the origin case because the logical and physical addresses are the same.

With this simple scheme, we see that it is never possible for a client node to overwrite origin data that is shared by a snapshot, or a shared chunk in the snapshot store.

Reading from the origin is even simpler: no synchronization at all is required on the part of the snapshot device. This is because there are no metadata updates, and any possible races would also be races on a physical disk. It is the job of a higher level application (i.e., filesystem or database) to prevent such races, not the snapshot device.

Reading from a snapshot, in contrast, requires the most intricate synchronization of all. The problem is, if a snapshot read references an origin chunk, there could be a simultaneous origin write in progress to the same chunk. If nothing is done to prevent it, the snapshot data could be overwritten while it is in the process of being read. The snapshot device is supposed to act like a physical disk, but it is very unlike a physical disk for data to spontaneously change while being read. There is no help from cluster filesystem or application layers either, because they do not know about the possible interaction of origin and snapshot and cannot be expected to.

The solution to this race is to provide some locking between origin writes and snapshot reads. When a snapshot read references an origin chunk, the server locks the chunk internally before replying. The snapshot client then initiates the actual read and, on completion, sends a message to the server to release the read lock. Snapshot reads from the origin thus use a three message protocol, as opposed to origin or snapshot writes, which use two. This also means that there are two possible forms of reply to a snapshot read query, depending on whether the target chunk resides on the origin or in the snapshot store. The latter flavor must include the address of a snapshot store chunk, but does not require any final release message.

3.1 Client-side Query Cache

An interesting fact about the shared vs unique state of any given origin chunk is that it is a latch: once a chunk becomes unique, it will stay unique until a new snapshot is created, at which point all chunks on the origin are shared with the new snapshot. This means that an origin client can cache the one-bit result of any write query, since it can be relied on to stay unchanged, except when a new snapshot is created. The server broadcasts a message to all origin clients in that case, so that they may clear their caches. Any uncached or zero bit is either shared or 'don't know', the distinction being immaterial.

Snapshot clients may also cache query results, the cached value being a snapshot store address rather than a bit. This cache never needs to be cleared explicitly, although it might be partially or fully cleared in response to memory pressure.

4 The ACID Test

The snapshot store metadata is a simple database, and as such, should satisfy the ACID test: atomicity, consistency, isolation and durability. We consider each of these requirements briefly here, to see how the new design stacks up.

Atomicity

A write query that changes the state of the on disk metadata is always handled entirely within a single journal transaction, which commits the change to disk atomically.

Consistency

Just as with a physical disk, it is not the job of the snapshot device to make any guarantee about the internal consistency of data on the device, only that what was written is what will be read back, for any and all simultaneously accessed snapshots and the origin device. In other words, its only responsibility here is to keep the metadata straight.

Isolation

The prototype implementation achieves transaction isolation by the simple expedient of a single-threaded server that handles incoming queries one at a time. This will eventually show up as a bottleneck and more elaborate synchronization will be needed.

Durability

Thanks to the journal, the entire state of the metadata server (with on exception, see below) is always completely recorded on disk at the time any write is acknowledged. Thus, if the metadata server should fail a new one can be started, read the metadata root and continue as if nothing had happened.

The one exception to this is that locking state of snapshot read requests against origin writes is kept only in memory on the server. While it is enough to simply require all outstanding reads on clients to complete before a newly started metadata server can resume processing requests, there could be cases where this would cause an unnecessary delay of several seconds on server restart where there is a heavy backlog of IO. Since it is easy, clients will be asked to upload any outstanding locked snapshot reads to the new metadata server before the server resumes processing requests. This should only take a few tens of milliseconds. The total latency of starting a new metadata server then should be measured in tens of milliseconds (though detecting that a server has failed could easily take much longer).

5 Server Implementation Details

5.1 Exception BTree Format

Exceptions for all snapshots are stored in a single btree indexed by logical chunk address. For each chunk, a list of exceptions is stored. Each exception consists of a snapshot address and a bitmap specifying which snapshots share that exception.

The btree is meant to be operated on directly by the snapshot server, as opposed to being translated into some more efficient cache format. To support alignment-restricted architectures, all fields in btree blocks are aligned according to their size. [1]

An attempt has been made to keep the btree compact by designing the node formats carefully, without going to extremes such as using a serial compressed encoding which is unpacked into a memory structure in order to be accessed. In other words, difficult tradeoffs have been made here between compactness, simplicity and efficiency.

[1] This posturing may prove unnecessary if the compiler's ability to generate alignment-independent code for alignment-restricted architectures proves reliable.

Leaf nodes

Leaf block format is optimized for rapid lookup and efficient insertion. At the bottom of each leaf is a header and a directory map that grows up towards a table of exceptions, which grows down. Each entry in the directory map gives the logical chunk address relative to a base address stored in the header, and has a pointer to one of the exceptions in the table at the top of the block. The entries are stored in sorted order according to logical chunk address and the pointers increase monotonically.

Using relative addresses allows the map entries to be more compact. In the current prototype map entries consist of two 32 bit numbers, however two 16 bit numbers might work just as well and save more space, although a 16 bit relative block number might be so small as to cause a noticeable increase in the number of leaf blocks if exceptions.are distributed sparsely. With 32 bit map numbers, a single exception requires 24 bytes; with 16 bit map numbers that would fall to 20 bytes, a 16% savings. The final determination of which is best should probably be determined experimentally.

The difference between each two pointers in the map gives the number of exceptions for the chunk. The last entry in the map is a sentinel and points at the top of the block (this could be designed out to save a few bytes). Each entry in the exception table has the 64 bit sector address of an exception in the snapshot store and a bitmap to indicate which snapshots share the exception.

The basic operations to locate and determine sharing of exceptions are efficient. A binary search is used to locate the target chunk address in the map, if it is present. This yields a list of exceptions on which efficient bitwise operations can be performed to determine sharing. From the point of view of the origin, a logical chunk is shared unless all active snapshots have exceptions for that chunk. From the point of view of a snapshot, a logical chunk is shared if it has no exception (i.e., is shared with the origin) or it has the same snapshot store address as another snapshot.

A slight drawback of this leaf format is that insertion requires memory moves in order to maintain the entries in sorted order, and the memory moves get longer as the leaf block fills up. For relatively small leaf blocks, i.e. 4K, it is probably not a problem. This will be determined experimentally. Other, equivalently efficient leaf formats are certainly possible, though perhaps they will not be as simple.

A more serious drawback of this leaf format is that as the number of snapshots increases, update overhead of the btree increases more or less linearly, albeit with a gentle slope. Nonetheless, it might prove desirable to adopt a variant leaf format at some point capable of encoding runs of adjacent exceptions efficiently.

Index nodes

An index node contains a table of entries each of which consists of a 64 bit logical chunk address key and a 64 bit sector address of a lower level index node or, at the lowest index level, a leaf. The entries are in sorted order by logical chunk address. Two successive keys bound the range of entries contained by the lower level node.

To locate the leaf block in which exceptions, if any, are stored for a given logical address, we descend recursively from the root, doing a binary search on the address key in each block and descending recursively into the node referenced by the sector address lying between the two keys that bound the target key.

We search all the way to a leaf node even if we are examining a region of the address space that is completely empty. For write requests this is not inefficient because we will immediately add an exception to the leaf node we found if one is not present. For read requests it's a little more work than necessary but we probably do not care since this only affects snapshot reads, and only by a small amount (origin reads do not involve the server).

5.2 Journal

Any altered metadata block, i.e, btree leaf and index nodes, allocation bitmaps, etc, are written to a journal before being written to their final destinations. This guarantees that the metadata can be restored reliably to the state of the most recently committed exception or other metadata change.

The size and location of the journal are determined at the time the snapshot store is created and cannot be changed.

Each journal transaction consists of an arbitrary number of data blocks followed by a journal tag block. The tag block carries a magic number allowing it to be identified as such for the purpose of journal replay, and a sequence number used to locate the starting point for journal replay. Any data block written to the journal that happens to have the same number at the same location must be escaped by writing a zero to that location in a copy of the data. The tag block carries a list of snapshot store sector addresses which are the final destinations of the data blocks. The low bit of the address carries a bit flag indicating that the data block was escaped and the magic number needs to be restored before the data block is finally written. The tag block carries other miscellaneous information such as partial usage status of a chunk recently allocated for metadata.

5.3 Allocation Bitmaps

Free space in the snapshot store is managed via bitmaps with a resolution of one bit per chunk. Each bitmap is one 4K block in size and maps 2**15 chunks. The bitmap blocks are indexed via a radix tree rooted in the header. Each radix tree node contains 512 8-byte sector addresses. As a slight simplification this tree is always 3 levels deep, giving 2^27 * 2^15 = 4 trillion chunks, or 16 petabytes volume size limit with a minimal 4K chunk size. It is always fully populated, i.e., the tree is created at the time the snapshot store is created and changed only if the snapshot store is expanded. The second lowest level of the bitmap index tree is loaded into memory when the volume is activated, this will be about 512 KB per terabyte of snapshot store.

Bitmaps are cached in buffers and accessed via getblk. A pointer is kept to the most recently accessed bitmap, i.e., it is not released until a different bitmap is accessed, which eliminates the majority of getblk lookups assuming reasonably good locality of allocation. Likewise, a pointer is kept to the most recently accessed index block. Since nearly all accesses to bitmaps are associated with changing the bitmap, the bitmaps are kept near the journal rather than being distributed throughout the snapshot store. This is purely a matter of allocation policy since the actual locations of bitmaps are determined by the radix tree.

Since metadata is allocated in blocks but allocation granularity is chunks, some chunks allocated to metadata may be only partially full. To avoid leakage of this unallocated space on unexpected restart, any partial allocations are recorded in the journal tag block. As a side effect, this means that a few metadata blocks can be allocated before a bitmap needs to be modified, saving some journal bandwidth.

5.4 Allocation Policy

The prototype implementation uses a simple, wraparound allocation scheme for both snapshot store data and metadata. If snapshots are held over extended periods, the snapshot store will become fragmented, reducing the efficiency of IO transfers. While this may not be a serious problem for backup applications, it certainly will be a problem for a user that wishes to use a snapshot in place of the origin device. To cater to this type of usage, a more sophisticated allocator will be needed. To facilitate efficient data transfer, exception store chunks that are logically close together should be stored physically close together. Because of chunk sharing, this can not be done perfectly in general; rather, an improved average case should be sought. At a higher level, many filesystems implicitly rely on details of layout linearity, for performance reasons. Much effort is required by the allocator to avoid violating such assumptions too severely. This is work for the future.

5.5 Locking

Synchronization via locking is only required between snapshot reads and origin writes. This locking takes place entirely within the server so no cluster lock manager is involved. (In fact the server is a lock manager for the limited case of snapshot reads.) The locks are simple, hashed locks. The cost of this locking will be one hash lookup per snapshot read or origin write of a shared chunk, plus the unlock messages. This locking is only required when snapshot and origin virtual devices are active at the same time; e.g., the server does not have to take any locks to service origin write requests if no snapshot device is active, even if snapshots are being held.

5.6 Snapshot Deletion

Because it packs together information for multiple snapshots in each leaf node, the exception btree is optimized for lookup and exception insertion as it should be. However, snapshot deletion is not as simple an operation as it would be if each snapshot had its own tree. (But if each snapshot had its own tree then exception creation time would increase with the number of snapshots, much more space would be used for multiple snapshots and keeping track of exception sharing would be less efficient.) In general, deleting a snapshot requires examining the entire btree and modifying each leaf block that contains an exception for the snapshot. This could amount to quite a lot of IO traffic and take a significant amount of time. The snapshot server will therefore simply log the status of the snapshot as "in process of deleting" and indicate completion immediately to the requesting client. The actual deletion will proceed in the background. When the deletion is finished, which could require tens of seconds for a large volume, the snapshot is logged as available for reuse.

A possible optimization is to defer deletions until several snapshots can be deleted in one pass, which will require less time than deleting each individually. How much less depends on how common it is for exceptions of several snapshots being deleted to lie in the same btree node. Another possible optimization is to include in each index node a bitmap indicating which snapshots have exceptions in the subtree descending from that node so that entire subtrees can be skipped during the traversal if they do not need to be modified.

A more aggressive and considerably more difficult optimization would involve introducing the concept of snapshot set generations and tagging each leaf block with a the snapshot generation as of the most recent alteration. Then a snapshot could be deleted by creating a new generation that does not include the deleted snapshot. A leaf block tagged with an earlier generation would be seen as "stale" and would be modified when next encountered, to remap it to the current generation, removing exceptions belonging to deleted snapshots in the process. The complexity of this approach makes it unattractive, however if snapshot deletion performance turns out to be a problem it might turn out to be worth the effort.

5.7 Expanding the Snapshot Store

The only tricky part of expanding the snapshot store is increasing the size of the allocation bitmap table. These are held in a radix tree to facilitate this. New index and bitmap blocks are added on the right. Is it best to keep the bitmap blocks near the journal, or to disperse them throughout the snapshot store, nearby the snapshot store areas that they map? Currently, all bitmap blocks are located near the base of the snapshot store, which was the simplest thing to implement. But if the snapshot store is expanded, where should the new bitmap blocks come from? Should we try to leave some free space at the bottom of the snapshot store? Should we relocate some data blocks to make space? These are open questions.

Once the bitmap allocation table is expanded, the new size of the snapshot store is recorded in the superblock, and that is that.

5.8 User Space Server

Since the cluster snapshot client connects to the snapshot server solely by means of a network socket, it was natural to implement the server in user space rather than kernel. Since the server is rather complex, this likely saved some time, and certainly makes the code accessible to a wider community of programmers than would be the case for kernel code. On the other hand, fully asynchronous IO is considerably more difficult to achieve in user space than in kernel code, and certain operations could well be less efficient. It is expected that the server will be ported to kernel space at some point, if only to learn the truth about relative performance. This might also be desirable when it comes time to implement a single-node variant.

User Space Buffer Layer

Like many problems in computer science, the snapshot server's handling of its disk-based btree amounts to a caching problem. As such, the classic Unix "getblk" provides an elegant model. It lends itself well to incremental updating of the btree, bitmaps and other file-backed data items. It supports asynchronous writing and multi-threaded writing well. And importantly, it will greatly simplify the task of porting the server to kernel space. An analog of Linux's buffer layer was therefore implemented in user space, complete with buffer heads, object referencing, and a buffer hash. This is recommended reading for those interested in how Linux's buffer IO model works.

Memory Inversion Deadlock

Implementing portions of block devices (or filesystems for that matter) in user space introduces the possibility of memory inversion deadlock. This occurs when a user space program exists somewhere in the virtual memory writeout path, which will be the case for any block device or filesystem. When the VM system tries to flush some dirty data to disk, the user space program may be invoked and may knowingly or unknowingly attempt to allocate some memory. If the memory available for non-critical processes such as user space programs is nearly exhausted, the memory allocation request may block, which also blocks the memory writeout attempt, which can eventually deadlock all user space programs due to memory exhaustion. This problem has been observed in practice. The standard solution is to run any network servers on dedicated servers so that the server is not in any VM writeout path. This solution is not entirely satisfactory because it requires an extra server, which may end up underutilized.

Such inversion can be prevented by running the user space process in "PF_MEMALLOC" mode, in which a special process flag is set giving the process privileged access to a pool of system memory which will automatically be tapped when memory runs low. In this mode, a process must be sure that its memory usage is strictly bounded so that the system memory pool is guaranteed never to be exhausted. Enforcing such a bound is possible with the help of the mlockall system call, preallocation of sufficient process stack space, and careful analysis of the user space program in question. While some details remain to be worked out, it is anticipated that a workable solution to the memory inversion deadlock problem will be found, so that this problem in itself will not become a compelling reason to port the snapshot server to kernel space.

5.9 Messaging

The snapshot server uses an asynchronous stream messaging model over a socket transport. A simple message formatting model was devised in which each message header has just two fields: code and length. Since the messages themselves tend to be small and numerous, the size of the message header has a significant effect on network load. The philosophy is, every field that can be moved out of the header into the message body should be. This way, messages do not incur size penalties for fields they do not require.

Generally, two socket read operations are performed for each incoming message, the first to learn the size and a second to read the message body. The size of the message header could be further reduced, since the size of many messages can be inferred from the message code, however doing so would require three reads instead of two for variable length messages. For the time being, the current eight byte message headers are deemed satisfactory.

Outgoing messages are typically sent directly from a process stack in a single socket write operation. Syntactic macro sugar is provided that allows fixed-length messages to be expressed in a single-line functional style, using symbolic fields as opposed to positional.

Message contents are binary, which requires careful attention to considerations of field alignment, size and byte order. All numeric fields are converted to network byte order on sending and native order on receiving. Message format is defined using gcc-specific packed structure attributes, which in theory provides predictable field size and alignment, and alignment independence across processor architectures. We shall see how well this works in practice, in the face of possible compiler bugs in this relatively untested area.

A fully asynchronous message model is used in which the sender never waits for a reply. Instead, the sender implements an incoming message handler capable of handling any message that might be received, including message replies. Any required state transformations are implemented within the incoming message handler, which runs as its own task. Thus, synchronous messages requiring replies, such as the client's initial "identify" message, must be handled with interprocess communication while the common and performance-critical case of asynchronous messaging is the default.

6 Client Implementation Details

Compared to the server, the snapshot client is simple. Implemented as a device mapper target, it is responsible for deferring IO requests as appropriate, querying the server for information it does not have in cache, receiving replies, and submitting requests to the correct device. It must also worry about releasing snapshot read locks, and a handle a few system interface details, such as setting up a network socket connection to the snapshot server, reconnecting if the connection is broken, and responding to the occasional message to clear cache because a new snapshot was created. Essentially, the client implements some simple message handling and a cache, the latter being non-essential.

On initialization, the client is passed a snapshot number (-1 = origin) and the local device names of two shared storage devices: the origin and the snapshot store. The client starts two kernel daemons, one being a blocking thread to handle server replies and the other, a nonblocking worker thread to take care of various minutiae. At this point the client initialization completes and the virtual device appears in the /dev/mapper directory. It can receive IO requests, but not complete them yet, because it does not have a socket connection to the server. A user space utility supplies this later via an ioctl on the virtual device.

One might argue that the socket connection should be created before the virtual device is created, however the virtual device needs to be able to operate for short periods with the socket disconnected anyway, since the socket connection might break and need to be reconnected.

Finally, before it can complete any IO requests, the client identifies itself to the snapshot server, stating which snapshot it is a client for. The server informs the client of the chunk size, then the client initializes its cache and proceeds to send queries, receive replies and retire IO requests.

The client-side query cache is implemented lazily as an inode address space. This has the advantage that it is managed automatically by the VM system: when there is no memory pressure, the cache will grow without bound, but when there is, the VM system can easily shrink it.

7 Failure Tolerance

7.1 Server Restart

Though challenging to implement efficiently, the ACID properties of consistency and durability immediately yield an important benefit: fail-over of the snapshot server is trivial. This is because all the server state information connected with completed IO requests is always available on disk. State information for uncompleted IO requests is always maintained in the memory of clients, and none needs to be uploaded to the new server. Restarting the snapshot server is the same as starting it initially, except that snapshot clients (but not origin clients) must upload their pending snapshot read locks before the server may allow any origin writes to proceed.[2]Otherwise, the new server need only load the snapshot store superblock and replay the journal. Finally, there is no requirement to clear client cache. So server restart is very fast, on the order of tens of milliseconds. Detecting server failure is likely to take much longer. There will be some slight performance degradation due to the loss of server metadata cache and the need for snapshot clients to retry pending reads, but it is unlikely to be noticeable.

7.2 Client failure

The server notices that a client has failed when its socket connection breaks. In response, the server releases any snapshot read locks held by that client, and that is that.

7.3 Failure Detection

Cluster heartbeating in itself is insufficient to detect some common failure conditions. For example, a node might continue responding to heartbeat messages, even though its cluster snapshot device has stopped servicing IO requests for one reason or another, e.g., a memory failure. If that client holds crucial read locks, the entire cluster could lock up. So at least in the case of snapshot clients, the server must detect timeouts and ping the client for liveness. If the client fails the test, the server will break its socket connection and release its locks. It is possible that the client may have simply be slow, not dead, in which case it will reconnect to the cluster and carry on.

Similarly, clients need to detect server problems that heartbeating may miss. If a snapshot or origin client detects a timeout, it should report it to the cluster infrastructure so that further remedial action may be taken. It may be that the cluster resource manager or human administrator simply assigned the server to an under-powered or overloaded node, in which case the correct action is to restart the server on a more appropriate node. In any event, this is not a problem that heartbeating can detect.

[2]A future optimization to eliminate copy-on-write for snapshot writes will require similar treatment in-progress snapshot writes.

8 Utilities

Several utilities programs are provided to support the use of the cluster block device:

mksnapstore - Initialize a physical volume as a snapshot store

csnap-connect - Connect a virtual snapshot device to a snapshot server

csnap-create - Create a new snapshot

csnap-delete - Delete a snapshot

The latter two functions have traditionally been performed via an ioctl on the Device Mapper control device. However it makes more sense for a cluster snapshot implementation to request a new snapshot directly from the snapshot server so that the device itself is not required to act as a mere relay.

9 Interface to Cluster Infrastructure

The cluster snapshot block device was designed to be operated either manually or under the control of a cluster manager system. Its needs are modest: it requires a socket connection to a snapshot server, and it needs a way of reporting errors when things go wrong. For example, if its server connection breaks, it needs to be able to request a new one. These are the only things it requires beyond the existing Device Mapper infrastructure.

A user space program connects a socket to the virtual device after Device Mapper has instantiated it via an ioctl on the virtual device, passing the FD number of a socket it has opened. A simple utility is provided so that this may be done from the shell, for the purpose of running the snapshot block device without the aid of any cluster infrastructure.

A simple user space monitoring utility is planned, to receive reconnection requests from the snapshot client. These are delivered over a second, local socket connection. So the snapshot client has two socket connections: one to the server over the network, and a local connection to a monitor daemon. If both connections break, the snapshot target will give up in disgust and return failure for all further IO requests.

The snapshot server must rely on the cluster infrastructure to satisfy the requirement that all snapshot clients reconnect and upload their read locks on server restart. A suitable interface to support this requires further research.

10 Performance Characteristics

10.1 Assumptions

Performance estimates below are based on using the smallest chunk size, 4K. Each new exception uses 20 bytes (exception store address, sharing bitmap and directory entry) so each btree leaf node holds a maximum of about 200 exceptions. Due to splitting, leaf nodes are normally not full. In fact worst case fullness of 50% is expected for the early implementations, so leaf nodes will hold about 100 exceptions each. The performance estimates here assume asynchronous IO, which for user space is not yet a practical possibility in Linux, therefore a kernel implementation is assumed. The initial implementation however is in user space; without asynchronous IO the user space implementation will not perform as well as a kernel implementation. It is expected that both implementations will be developed and maintained; that the user implementation will be available first; that a kernel implementation will supercede it in performance; and that the user space implementation will eventually pull even with the kernel implementation by taking advantage of newly available asynchronous IO and user space locking facilities.

10.2 Effect of chunk size

Larger chunk size will help performance for sequential and hurt for random write loads. The total size of metadata reduces linearly with the chunk size, saving space, IO bandwidth and seeking. On the other hand, larger chunks increase internal fragmentation of the snapshot store, especially for sparse, random access loads, and the overhead of metadata updating is supposed to be small in relation to data transfers. Therefore it is hoped that the performance and metadata size cost of small chunk sizes will be outweighed reduced internal fragmentation, saving space in the snapshot store. This remains to be tested in practice.

10.3 Effect of metadata block size

Larger metadata blocks will improve performance somewhat on largely serial write loads due do requiring a fewer number of larger IOs, especially if the snapshot metadata is fragmented. However, for the time being Linux does not support IO buffers larger than physical page size, so it is expected that metadata block size will not increase until this issue is addressed, at least for a kernel implementation of the snapshot metadata server. For compatibility with the expected kernel metadata server, the user space implementation will use 4K blocks.

It is thought that communication overhead and server load will not be significant performance factors, due to these being highly optimized. Contention on large clusters with parallel loads should not be a significant factor either, since a single server should be able to handle the traffic of many nodes of similar power to itself. The exception to this is copy-out overhead which could easily saturate a server's bus; a simple solution is available: farm out the copy-out traffic to lightly-loaded nodes as necessary.

10.4 Effect of Holding Multiple Snapshots

The more snapshots that are held, the more btree leaf nodes will be required to hold them. Journalling the extra btree leaves to disk consumes IO bandwidth, causes more seeking and generates cache pressure. Reading in the extra btree nodes increases latency. However, because exceptions for all snapshots are stored adjacent in the btree, the overhead is not as large as if a separate map had to be updated on disk for each snapshot. Importantly, the process of determining whether a given chunk is shared never requires more than a single leaf node to be examined.

Sharing bitmaps are used within leaf nodes to avoid having to enter any given snapshot store address more than once into the node, and also performs the function of specifying which snapshot uses a given snapshot store address. The worst case arises when a given logical chunk is written at least once after every snapshot. Then the leaf node entries for that chunk have a bitmap and a snapshot store address for every snapshot. Since leaf nodes are expected to be 50% full in the initial implementation, we can end up with one exception stored in each leaf node. Then the number of btree nodes that have to be journalled is equal to the number of chunks written. The journalled node has to be written twice, once to the journal and once to its true destination. So the worst case is a factor of 3 degradation in write performance due to btree updating alone.

To ameliorate such degradation it would be wise to use a larger chunk size when large numbers of snapshots are expected.

The worst case degradation above can be tempered somewhat by improving the btree update algorithm to use a b+tree algorithm, which guarantees 2/3rds leaf fullness, enough to hold two exceptions instead of one. Larger metadata blocks will help reduce seeking overhead, when they become practical. Eventually though, the best strategy is to introduce variant leaf node formats that optimize for the many-snapshots case by representing ranges of snapshot store chunks compactly, especially where the snapshot store chunks are allocated sequentially, which is something we want to achieve anyway.

In brief, the metadata update component of origin and snapshot write performance will degrade linearly with the number of snapshots held, but with a much shallower slope than if snapshot store data were not shared and metadata were not grouped together by logical address. In the latter case, copy-out overhead would increase directly with number of snapshots. Exception table update overhead would increase rapidly as well, though the exact rate is harder to characterize because it depends on the chunk sharing patterns.

With the maximum number of snapshots held (64) the new design should perform better than the old one by a factor of thirty or more. Furthermore, some fairly straightforward improvements to the btree leaf format can make the slope much shallower, to the point where the overhead of holding 64 snapshots may be hard to notice.

With a single snapshot held, the new design not perform quite as well as the existing device-mapper design, but only because the existing design does not provide durable recording of snapshot store updates. In any case, the overhead of the durable snapshot recording is expected to be only about 2% worst-case overhead vs raw writing, far less than the 200% worst-case overhead of copy-outs when a single snapshot is held, and shrinks roughly linearly with the chunk size (extra seeking in the metadata region makes this relationship slightly more complex). So by using a 256K chunk size, metadata update can most likely be held to a few percent of first-time write overhead even when the maximum number of snapshots are held.

10.5 Origin Read Performance

Origin reads are passed straight through to the underlying volume. Since the overhead of the device mapper handling is insignificant, origin read performance is essentially unchanged

10.6 Sequential Origin Write Performance

Origin write throughput is affected mainly by the frequency of chunk copy-outs and metadata update overhead. Copy-outs require reading and writing, requiring a minimum of 200% additional bandwidth vs raw write and additional seeking as well, especially for the single-spindle case where the origin volume and snapshot store will be far apart. Throughput is improved at the expense of latency by batching the copy-out reads and copy-out writes, which happens naturally with asynchronous IO. There will thus be fewer long seeks between the origin and snapshot store.

Worst case origin write performance is obtained when the snapshot store is created with the smallest possible chunk size (4K) and the load requires a copy-out for every chunk write. Such a load is easy to generate, for example by setting a snapshot and then immediately unpacking an archive into the volume. Required IO bandwidth will triple, seeking between the origin and snapshot store will increase, and metadata updating will increase. Writing in this case should be largely linear and batching amortizes the seeking overhead, so the dominant effect is expected to be the increased IO bandwidth. For this load we should expect to see a 3 times slowdown versus raw volume access. Fragmentation of the snapshot store could make this considerably worse, perhaps by another factor of three.

Since such a load is easy to generate it is worrisome. It is possible that in the long run, general performance for a snapshot volume could become better than for the origin, see below.

Fragmentation of the snapshot store will introduce additional seeking and rotational latency penalties. Reducing such fragmentation by clever snapshot store allocation policy will yield significant performance gains, however such allocation policy improvements require considerable time to develop. A highly fragmented snapshot store could aggravate worst case write performance by an additional factor of a few hundred percent.

10.7 Random Origin Write Performance

A load that consists of 100% single-sector writes distributed randomly over the entire volume immediately after setting a snapshot will cause copy-out bandwidth to be much more than 200% of raw write bandwidth, and will also cause a great deal of additional seeking. Metadata overhead will also increase significantly since typically only a single chunk on each leaf node will be updated each time the node is journalled to disk; rotational latency will increase significantly during metadata access. Performance under this random load will typically be dominated by seeking rather than bandwidth. Analysis is complex, however I will speculate now that the performance of the snapshotted volume could degrade by a factor of 3 to 4 versus the raw volume due to additional seeking and rotational latency for copy-outs and metadata updating.

Fragmentation of the snapshot store can and should be addressed over time. For origin writes, nothing that can be done about the copy-out overhead. Snapshot writes on the other hand do not incur copy-out overhead. They do incur seeking and rotational penalties due to fragmentation in the snapshot store, but so do origin writes. Furthermore snapshot reads also suffer from fragmentation penalties whereas origin reads do not. Very good snapshot store layout optimization could reduce both the penalty for snapshot reading and writing, in which case general performance on a snapshot volume could be better than on a snapshotted origin volume. Whether this can be realized in practice remains to be seen.

10.8 Snapshot Read Performance

Unlike origin reads, snapshot read throughput is affected by snapshot store fragmentation. Snapshot read latency is increased by the requirement of locking against origin writes. Read-ahead results in a kind of lock-ahead, so under loads where read-ahead is effective, increased snapshot read latency will not hurt read throughput. The predominant visible effect is expected to be read fragmentation. With large chunk sizes, e.g., 256K and up, moderate fragmentation should cause only slight degradation in snapshot read performance. However, without special attention to snapshot store allocation policy, fragmentation can be expected to be fairly severe, so snapshot read performance is not expected to be stellar in early implementations. Fortunately, since the main purpose of reading from a snapshot is to back it up or restore a few files, some read performance degradation is acceptable and is unlikely to be noticed.

In the long run it is desirable to improve snapshot read performance by controlling snapshot store fragmentation as much as possible, in order to take advantage of the inherently superior performance of snapshot writing versus origin writing.

10.9 Snapshot Write Performance

Snapshot writes to not require copy-outs; if an origin chunk or shared snapshot store chunk needs to be written, the logical chunk is first remapped to a new chunk in the snapshot store. With some tweaking of the message protocol, writing to the chunk could proceed as soon as the new allocation is known, in parallel with the logging of the new exception. So snapshot writes are inherently quite efficient.

Snapshot write overhead comes from metadata update overhead and snapshot store fragmentation. The former is supposed to be small, on the order of a few percent. The latter could be very large, and probably will be in initial implementation, perhaps on the order of a factor of 10. Larger chunk sizes will reduced this seeking overhead, roughly linearly with the chunk size. Careful layout optimization could conceivably reduce this to a few percent, even with small chunks. We shall see.

10.10 Network Performance

The amount of message data needed for each chunk is small, especially since the message format is designed from the outset to handle ranges of chunks and multiple ranges in each message. Except for snapshot reads, each message sequence is only two messages long (note: approximately. Server responses do not correspond exactly requests; e.g., any unshared chunks can be acknowledged immediately). Message traffic is expected to be less than 1% of disk array traffic. Assuming that the general purpose network interconnect and storage array interconnect have similar bandwidth, this is where the expectation that this architecture will scale linearly to about 100 clients comes from.

10.11 Overall Performance

It is expected that typical usage of a snapshotted origin volume will show only slight reduction of performance versus the raw origin volume, due to reading being more common than writing. Rewriting chunks is optimized by the client's bitmap cache, which is compact and probably capable of caching all the hot spots of a volume, even for large volumes. So rewriting should show now visible degradation. The performance of fresh writes to snapshotted chunks will degrade significantly, due to copy-out bandwidth, and to snapshot store fragmentation, that latter being subject to optimization while the former is unavoidable. In general, more frequent snapshots cause more fresh writes, with the frequency of fresh writes peaking just after the snapshot and declining over time, till the next snapshot.

So: what will be the balance of fresh writes vs reads and rewrites? How frequently will we see will we see the balance shift for a short time in the direction of the worst case? How bad is the worst case? How likely is it that the user will notice the shifts in write performance? These all await measurement under live loads. However at this point I will speculate that even a relatively early implementation will show average performance degradation versus a raw volume of less than ten percent, and that, at worst, performance degradation will be limited to a factor of four or so just after a snapshot. For many users, and particularly enterprise users, the benefits of snapshotting will outweigh the performance loss: it is easy to buy bandwidth, not as easy to buy live backup capability. For others, the unavoidable performance degradation of origin writing will make snapshotting unattractive enough to discourage its use. Eventually we may be able to satisfy this group as well, by improving snapshot store allocation policy to the point where the origin can be made optional and all IO take place in the snapshot store.

The pessimism in this section should be tempered by observing that in many respects, performance is expected to be good:

- Large number of snapshots can be held without affecting performance much
- Snapshot store utilization is good
- Network traffic is minimal
- Rewrites are highly optimized

In other words, if you need snapshots then this implementation is likely to deliver good performance versus alternatives. Plus there is a clear path forward to achieving near-optimal performance, by working towards a system where the snapshot store can be used effectively alone, with no origin volume.

11 Further Work

11.1 Parallelizing the Architecture

Normally the first question I am asked about this clustered snapshot design is why isn't it symmetric? The answer: because a) it doesn't have to be in order to perform well on today's typical clusters and b) distributing a tree structure across independent caches is a complex, error-prone process, and introduces overhead of its own. At some point, however, the single node server architecture will become a bottleneck, so I discuss parallelizing strategies here.

The easiest thing we can do, and with the strongest immediate effect is to have the server distribute the copy-out work to underused nodes. This will take significant IO bandwidth load off the server's bus at the expense of a little messaging latency. By doing this, a single server can likely scale to handle a hundred or so busy nodes of similar power to itself: the real bottleneck will probably be the storage array. A user who can afford to upgrade the storage array to handle even larger numbers of clients can likely afford to upgrade the snapshot server as well.

At some point, perhaps two or three hundred clients, the snapshot server becomes a bottleneck again. Further scaling is easily achieved by dividing up the work between a number of snapshot servers, by logical address range. Each snapshot server maintains a separate btree in a distinct range of logical addresses and operates its own journal. Care must be taken that allocation bitmaps are divided up cleanly; this is not hard (e.g., even if a logical address range boundary lies in the middle of a bitmap block, the boundary bitmap can be replicated between two nodes, with logic to prevent allocation outside the boundary - needed anyway for error checking). Shared metadata such as the current snapshot list, superblock, etc., is updated using a RW locking strategy (i.e., using a DLM). Assuming that workload is distributed relatively evenly across the logical address range, this simple parallelization strategy will serve up to a thousand clients or so, and the disk will once again be the bottleneck.

If we want to scale to far larger numbers of clients we probably have to bite the bullet and distribute the btrees and allocation bitmaps. However I do not think this problem is imminent; there is plenty of time to think about it.

11.2 Adaptation to Single Node Client

It is expected that this cluster snapshot design will eventually be adapted for single-node use, replacing the current snapshot target, with the following benefits:

- Copy-out overhead will no longer increase linearly with number of snapshots held
- Excessive memory footprint for large volumes eliminated
- Client cache memory shrinks in response to memory pressure
- Only one snapshot store volume to manage instead of one per snapshot
- Sharing exceptions between snapshots saves disk space
- All snapshots allocate disk memory from common pool
- Just one device mapper target to know about instead of two

When it comes time to take this step, a decision must be taken whether to (lazily) keep the message-based interface and client-server factoring, or to create a variant where the server logic is integrated directly into the client.

12 Conclusion

Block device snapshots have a number of interesting applications, including the crucial one of enabling online backup for enterprise Linux users. With the imminent addition of cluster filesystems to the Linux kernel, cluster snapshot capability is required. This presents a number of interesting synchronization and structural problems, for which original solutions have been devised and presented in this paper.

As well as extending snapshot capability to clusters, some problems with the existing, single node snapshot design were solved, yielding significant improvements in IO performance, disk space consumption and memory usage. A prototype implementation exhibiting good performance characteristics has been created. Further work remains to be done to transform the prototype into a fully reliable, fault-tolerant facility ready to be deployed in a production environment.