Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Zumastor Linux Storage Server

Daniel Phillips
Google, Inc.
phillips@google.com

Abstract

Zumastor provides Linux with network storage functionality suited to a medium scale enterprise storage role: live volume backup, remote volume replication, user accessible volume snapshots and integration with Kerberized network filesystems. This paper examines the design and functionality of the major system components involved. Particular attention is paid to the subjects of optimizing server performance using NVRAM, and reducing the amount of network bandwidth required for remote application using various forms of compression. Future optimization strategies are discussed. Benchmark results are presented for currently implemented optimizations.

1 Introduction

Linux has done quite well at the edges of corporate networks—web servers, firewalls, print servers and other services that rely on well standardized protocols—but has made scant progress towards the center, where shared file servers define the workflow of a modern organization. The barriers are largely technical in that Linux storage capabilities, notably live filesystem snapshot and backup, have historically fallen short of the specialized proprietary offerings to which users have become accustomed.

Zumastor provides Linux with network storage functionality suited to medium scale enterprise storage roles: live volume backup, remote volume replication, user accessible volume snapshots and integration with Kerberized network filesystems. Zumastor takes the form of a set of packages that can be added to any Linux server and will happily coexist with other roles that the server may serve. There are two major components: the ddsnap virtual block device, which provides base snapshot and replication functionality, and the Zumastor volume monitor, which presents the administrator with a simple command line interface to manage volume snapshotting and replication. We first examine the low level components in order to gain an understanding of the design approach used, its capabilities and limitations.

2 The ddsnap Virtual Block Device

The ddsnap virtual block device provides multiple read/write volume snapshots. As opposed to the incumbent lvm snapshot facility, ddsnap does not suffer from degraded write performance as number of snapshots increases and does not require a separate underlying physical volume for each snapshot. A ddsnap virtual device requires two underlying volumes: an origin volume and a snapshot store. The origin, as with the lvm snapshot, is a normal volume except that write access is virtualized in order to protect snapshotted data. The snapshot store contains data copied from the origin in in the process of protecting snapshotted data. Metadata in the snapshot store forms a btree to map logical snapshot addresses to data chunks in the snapshot store.

The snapshot store also contains bitmap blocks that control allocation of space in the snapshot store, a list of currently held snapshots, a superblock to provide configuration information, and a journal for atomic and durable updates to the metadata. The ddsnap snapshot store resembles a simple filesystem, but with only a single directory, the btree, indexed by the logical address of a snapshot. Each leaf of the btree contains a list of logical chunk entries; each logical chunk entry in the btree contains a list of one or more physical chunk addresses; and for each physical chunk address, a bitmap indicates which snapshots currently share that physical chunk. Share bits are of a fixed, 64 bit size, hence the ddsnap limitation of 64 simultaneous snapshots. This limitation may be removed in the future with a redesign of the btree leaf format.

2.1 Client-server design

DDsnap was conceived as a cluster snapshot—*DD* stands for *distributed data*—with a client-server architecture. Each ddsnap client (a device mapper device) provides access to either the underlying, physical origin volume or to some snapshot of the origin volume. To implement the copy-before-write snapshot strategy, a ddsnap origin client communicates with a userspace server over a socket, requesting permission from this snapshot server before writing any block so that the server may move any data shared with a snapshot to the snapshot store. Similarly, a ddsnap snapshot client requests permission from the snapshot server before writing any block so that space may be allocated in the snapshot store for the write. A snapshot client also requests permission before reading snapshot data, to learn the physical location of the data and to serialize the read against origin writes, preventing the data being read from moving to a new location during the read. On the other hand, an origin client need not consult the server before reading, yielding performance nearly identical to the underlying physical volume for origin reads.

Since all synchronization with the snapshot server is carried out via messages, the server need not reside on the same node as an origin or snapshot client, although with the current single-node storage application it always does. Some more efficient means of synchronization than messages over a socket could be adopted for a single node configuration, however messaging overhead has not proved particularly bothersome, with a message inter-arrival time measured in microseconds due to asynchronous streaming. Some functionality required for clustering, such as failing over the server, uploading client read locks in the process, is not required for single-node use, but imposes no overhead by its presence.

Creating or deleting a snapshot is triggered by sending a message to the snapshot server, as are a number of other operations such as obtaining snapshot store usage statistics and obtaining a list of changed blocks for replication. The *ddsnap* command utility provides a commandline syntax for this. There is one snapshot server for each snapshot store, so to specify which snapshot server the command is for, the user gives the name of the respective server control socket. A small extra complexity imposed by the cluster design is the need for a ddsnap *agent*, whose purpose on a cluster is to act as a node's central cluster management communication point, but which serves no useful purpose on a single node. It is likely that the functionality of agent and snapshot server will be combined in future, somewhat simplifying the setup of a ddsnap snapshot server. In any event, the possibility of scaling up the Zumastor design using clustering must be viewed as attractive.

2.2 Read/Write Snapshots

Like lvm snapshots, ddsnap snapshots are read/write. Sometimes the question is raised: why? Isn't it unnatural to write to a snapshot? The answer is, writable snapshots come nearly for free, and they do have their uses. For example, virtualization software such as Bochs, QEMU, UML or Xen might wish to base multiple VM images on the same hard disk image.[1] The copy-on-write property of a read/write snapshot gives each VM a private copy in its own snapshot of data that it has written. In the context of zumastor, a root volume could be served over NFS to a number of diskless workstations, so each workstation is able to modify modify part of its own copy while continuing to share the unmodified part.

2.3 Snapshotted Volume IO Performance

Like the incumbent lvm snapshot, origin read performance is nearly identical to native read performance, because origin reads are simply passed through to the underlying volume.

As with the incumbent lvm snapshot, ddsnap uses a copy-before-write scheme where snapshotted data must be copied from the origin to the snapshot store the first time the origin chunk is written to after a new snapshot. This can degrade write performance markedly under some loads. Keeping this degradation to a tolerable level has motivated considerable design effort, and work will continue in this area. With the help of several optimization techniques discussed below, a satisfactory subjective experience is attained for the current application: serving network storage.

Compared to the incumbent lvm snapshot, the ddsnap snapshot design requires more writes to update the metadata, typically five writes per newly allocated physical chunk:

[1] `http://en.wikipedia.org/wiki/Copy-on-write`

1. Write allocation bitmap block to journal.
2. Write modified btree leaf to journal.
3. Write journal commit block.
4. Write allocation bitmap block to store.
5. Write modified btree leaf to store.

This update scheme is far from optimal and is likely to be redesigned at some point, but for now a cruder approach is adopted: add some nonvolatile memory (NVRAM) to the server.

The presence of a relatively small amount of nonvolatile RAM can accelerate write performance in a number of ways. One way we use NVRAM in Zumastor is for snapshot metadata. By placing snapshot metadata in NVRAM we reduce the cost of writing to snapshotted volume locations significantly, particularly since ddsnap in its current incarnation is not very careful about minimizing metadata writes. Unfortunately, this also limits the maximum size of the btree, and hence the amount of snapshot data that can be stored. This limit lies roughly in the range of 150 gigabytes of 4K snapshot chunks per gigabyte of NVRAM. NVRAM is fairly costly, so accommodating a large snapshot store be expensive. Luckily, much can be done to improve the compactness of the btree, a subject for another paper.

Using NVRAM, snapshot performance is no worse than the incumbent lvm snapshot, however the size of the btree and hence the amount of data that can be stored in the snapshot store is limited by the amount of NVRAM available. Future work will relax this limitation.

2.3.1 Filesystem Journal in NVRAM

For filesystems that support separate journals, the journal may be placed in NVRAM. If the filesystem is further configured to journal data writes as well as metadata, a write transaction will be signalled complete as soon as it has been entered into the journal, long before being flushed to underlying storage. At least until the journal fills up, this entirely masks the effect of slower writes to the underlying volume. The practical effect of this has not yet been measured.

2.3.2 Effect of Chunk Size and Number of Snapshots on Write Performance

Untar time on the native (Ext3) filesystem is about 14 seconds. Figure 1 shows that untar time on the virtual block device with no snapshots held is about 20 seconds, or slower by a factor of 1.43. This represents the overhead of synchronizing with the snapshot server, and should be quite tractable to optimization. Snapshotted untar time ranges from about 3.5 times to nearly 10 times slower than native untar time.

Figure 1 also shows the effect of number of currently held snapshots on write performance and of varying the snapshot chunk size. At each step of the test, a tar archive of the kernel source is unpacked to a new directory and a snapshot is taken. We see that (except for the very first snapshot) the untar time is scarcely affected by the number of snapshots. For the smallest chunk size, 4K, we see that untar time does rise very slightly with the number of snapshots, which we may attribute to increased seek time within the btree metadata. As chunk size increases, so does performance. With a snapshot store chunk size of 128KB, the untar runs nearly three times faster.

2.3.3 Effect of NVRAM on Write Performance

Figure 2 shows the effect of placing the snapshot data in NVRAM. Write performance is dramatically improved, and as before, number of snapshots has little or no effect. Interestingly, the largest chunk size tested, 128KB, is no longer the fastest; we see best performance with 64K chunk size. The reason for this remains to be investigated, however this is good news because a smaller chunk size improves snapshot store utilization. Write performance has improved to about 2 to 5 times slower than native write performance, depending on chunk size.

2.3.4 NVRAM Journal compared to NFS Write Log

NVRAM is sometimes used to implement a NFS write log, where each incoming NFS write is copied to the write log and immediately acknowledged, before being written to the underlying filesystem. Compared to the strategy of putting the filesystem journal in NVRAM, performance should be almost the same: in either case,

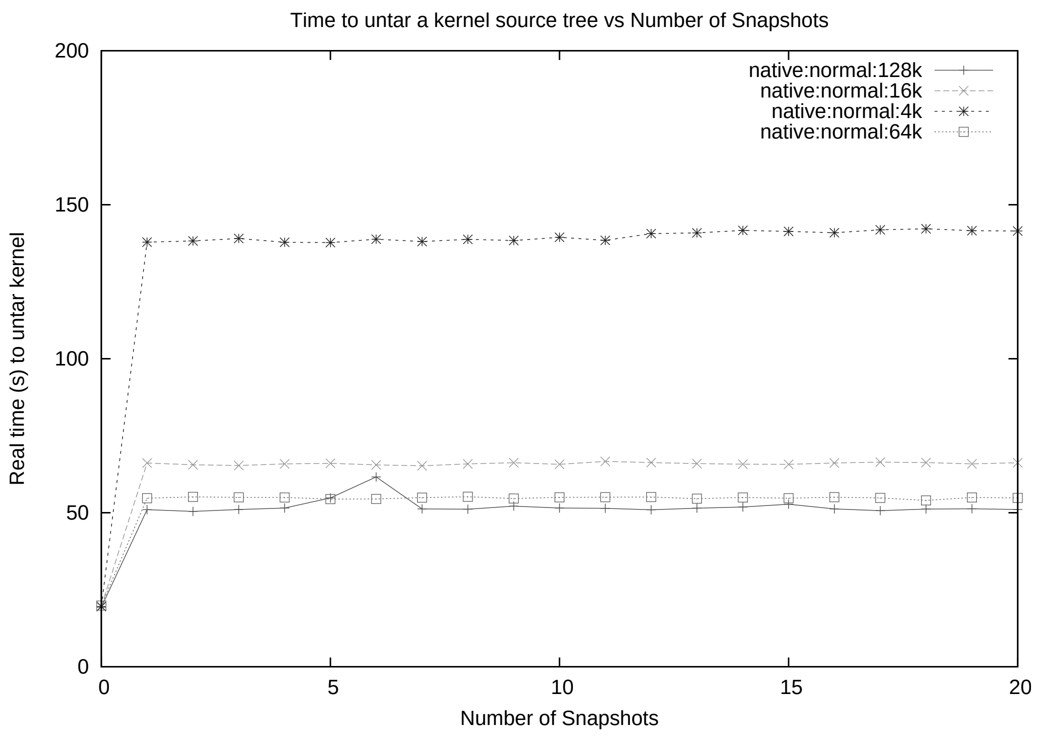

Figure 1: Write performance without NVRAM

a write is acknowledged immediately after being written to NVRAM. There may be a small difference in the overhead of executing a filesystem operation as opposed to a potentially simpler transaction log operation, however the filesystem code involved is highly optimized and the difference is likely to be small. On the other hand, the transaction log requires an additional data copy into the log, likely negating any execution path efficiency advantage. It is clear which strategy requires less implementation effort.

2.3.5 Ongoing Optimization Efforts

A number of opportunities for further volume snapshot write optimization remain to be investigated. For example, it has been theorized that writing to a snapshot instead of the origin can improve write performance a great deal by eliminating the need to copy before writing. If read performance from a snapshot can be maintained, then perhaps it would be a better idea to serve a master volume from a snapshot than an origin volume.

3 Volume Replication

Volume replication creates a periodically updated copy of a master volume at some remote location. Replication differs from mirroring in two ways: 1) changes to the remote volume are batched into volume deltas so that multiple changes to the same location are collapsed into a single change and 2) a write operation on the master is not required to wait for write completion on the remote volume. Batching the changes also allows more effective compression of volume deltas, and because the changes are sorted by logical address, applying a delta to a remote volume requires less disk seeking than applying each write to a mirror member in write completion order. Replication is thus suited to situations where the master and remote volume are not on the same local network, which would exhibit intolerable remote write latency if mirrored. High latency links also tend to be relatively slow, so there is much to be gained by good compression of volume deltas.

Zumastor implements remote replication via a two step process: 1) Compute difference list; 2) Generate delta. To generate the difference list for a given pair of snapshots, the ddsnap server scans through the btree to find all snapshot chunks that belong to one snapshot and not the other, which indicates that the data for the corresponding chunks was written at different times and is most probably different. To generate the delta, a ddsnap utility runs through the difference list reading data from one or both of the snapshots which is incorporated into

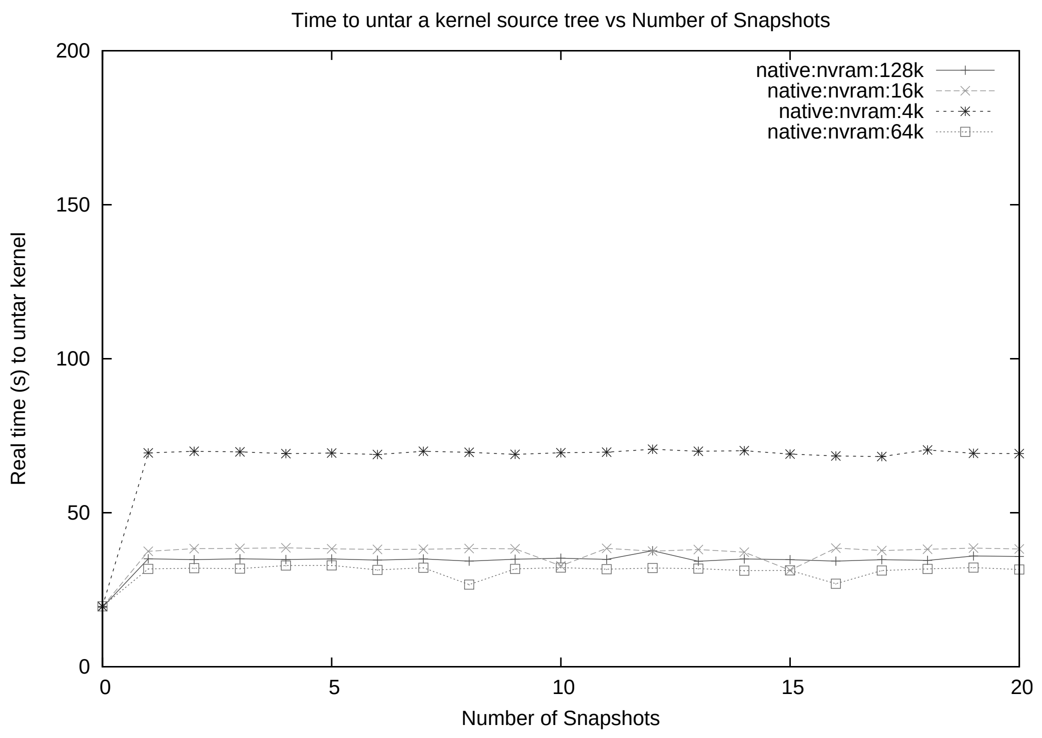

Figure 2: Write performance with NVRAM

the output delta file. To allow for streaming replication, each volume delta is composed of a number of extents, each corresponding to some number of contiguous logical chunks.

A variety of compression options are available for delta generation. A "raw" delta incorporates only literal data from the destination snapshot. An "xdelta" delta computes the binary difference between source and destination snapshot. Raw delta is faster to generate and requires less disk IO, is faster to apply to the target volume, and is more robust in the sense that the code is very simple. Computing an xdelta delta requires more CPU and disk bandwidth but should result in a smaller delta that is faster to transmit.

A volume delta may be generated either as a file or as a TCP stream. Volume replication can be carried out manually using delta files:

1. Generate a snapshot delta as a file
2. Transmit the delta or physically transport it to the downstream host
3. Apply the delta to the origin volume of the downstream host

It is comforting to be able to place a snapshot delta and to know that the replication algorithm is easy enough to carry out by hand, which might be important in some special situation. For example, even if network connectivity is lost, volume replication can still be carried out by physically transporting storage media containing a delta file.

Zumastor uses ddsnap's streaming replication facility, where change lists and delta files are never actually stored, but streamed from the source to target host and applied to the target volume as a stream. This saves a potentially large amount of disk space that would be otherwise be required to store a delta file on both the source and target host.

From time to time it is necessary to replicate an entire volume, for example when initializing a replication target. This ability is provided via a "full volume delta" that generates a raw, compressed delta as if every logical chunk had appeared in the difference list. Incidentally, only the method of delta generation is affected by this option, not the delta file format.

To apply a volume delta, ddsnap overwrites each chunk of the target volume with the new data encoded in the delta in the case of a raw delta, or reads the source snapshot and applies the binary difference to it in the case of xdelta. Clearly, it is required that the source snapshot exist on the downstream host and be identical to the source snapshot on the upstream host.

To create the initial conditions for replication:

1. Ensure that upstream and downstream origin volumes are identical, for example by copying one to the other
2. Snapshot the upstream and downstream volumes

The need to copy an entire volume over the network in the first step can be avoided in some common cases. When a filesystem is first created, it is easy to zero both upstream and downstream volumes, which sets them to an identical state. The filesystem is then created after step 2. above, so that relatively few changed blocks are transmitted in the first replication cycle. In the case where the downstream volume is known to be similar, but not identical to the upstream volume (possibly as a result of an earlier hardware or software failure) then the remote volume differencing utility rdiff may be used to transmit a minimal set of changes downstream.

Now, for each replication cycle:

1. Set a new snapshot on the upstream volume.
2. Generate the delta from old to new upstream snapshot.
3. Transmit the delta downstream.
4. Set a new snapshot on the downstream volume (downstream origin and new snapshot are now identical to the old upstream snapshot).
5. Apply the delta to the downstream origin (downstream origin is now identical to the new upstream snapshot).

For the streaming case, step 4 is done earlier so that the transmit and apply may iterate:

1. Set a new snapshot on upstream and downstream volumes.
2. Generate the delta from old to new upstream snapshot.
3. Transmit the next extent of the delta downstream.
4. Apply the delta extent to the downstream origin.
5. Repeat at 3 until done.

For streaming replication, a server is started via ddsnap on the target host to receive the snapshot delta and apply it to the downstream origin.

Fortunately for most users, all these steps are handled transparently by the Zumastor volume manager, described below.

Multi level replication from a master volume to a chain of downstream volumes is handled by the same algorithm. We require only that two snapshots of the master volume be available on the upstream volume and that the older of the two also be present on the downstream volume. A volume may be replicated to multiple targets at any level in the chain. In general, volume replication topology is a tree, with the master volume at the root and an arbitrary number of target volumes at interior and leaf nodes. Only the master is writable; all the target volumes are read-only.

3.1 Delta Compression

Compression of delta extents is available as an option, either using zlib (gzip) or in the case of xdelta, an additional Huffman encoding stage. A compressed xdelta difference should normally be more compact than gzipped literal data, however, one can construct cases where the reverse is true. A further (extravagant) "best" compression option computes both the gzip and xdelta compression for a given extent and use the smaller for the output delta. Which combination of delta generation options is best depends largely on the amount of network bandwidth available.

Figure 3 illustrates the effect of various compression options on delta size. For this test, the following steps are performed:

1. Set snapshot 0.
2. Untar a kernel tree.
3. Set snapshot 1.

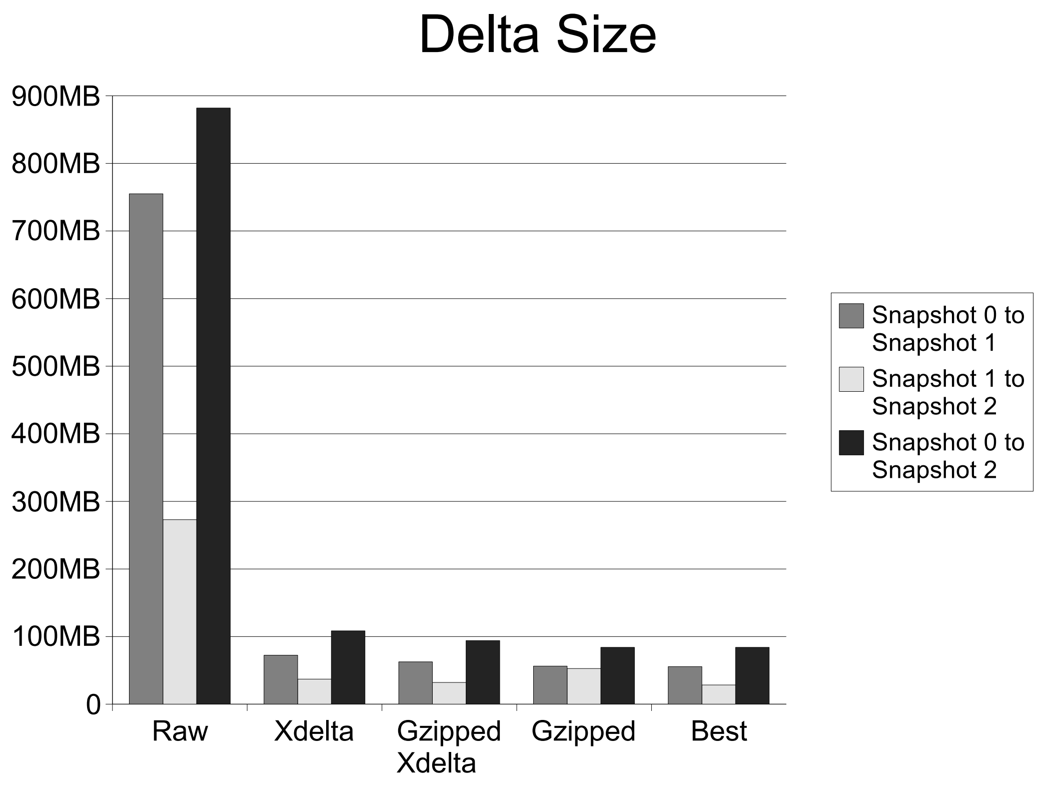

Figure 3: Delta Compression Effectiveness

4. Apply a (large) patch yielding the next major kernel release.

5. Set snapshot 2.

Three delta files are generated, the two incremental deltas from snapshot 0 to snapshot 1 and from snapshot 1 to snapshot 2, and the cumulative delta from snapshot 0 to snapshot 2. We observe a very large improvement in delta size, ranging up to a factor of 10 as compared to the uncompressed delta.

XDelta performs considerably better on the snapshot 1 to snapshot 2 delta, which is not surprising because this delta captures the effect of changing many files as opposed to adding new files to the filesystem, so it is only on this delta that there are many opportunities to take advantage of similarity between the two snapshots.

The "best" method gives significantly better compression on the snapshot 1 to snapshot 2 delta, which indicates that some delta extents compress better with gzip than they do with xdelta. (As pointed out by the author of xdelta, this may be due to suboptimal use of compression options available within xdelta.) Figure 4 re-expresses the delta sizes of Figure 3 as compression ratios, ranging from 5 to 13 for the various loads.

Delta compression directly affects the time required to transmit a delta over a network. This is particularly important when replicating large volumes over relatively low bandwidth network links, as is typically the case. The faster a delta can be transmitted, the fresher the remote copy will be. We can talk about the "churn rate" of a volume, that is, the rate at which it changes. This could easily be in the neighborhood of 10% a day, which for a 100 gigabyte disk would be 10 gigabytes. Transmitting a delta of that size over a 10 megabit link would require 10000 seconds, or about three hours. An 800 gigabyte volume with 10% churn a day would require more than a day to transmit the delta, so the following delta will incorporate even more than 10% churn, and take even longer. In other words, once replication falls behind, it rapidly falls further and further behind, until eventually nearly all the volume is being replicated on each cycle, which for our example will take a rather inconvenient number of days.

In summary, good delta compression not only improves the freshness of replicated data, it delays the point at

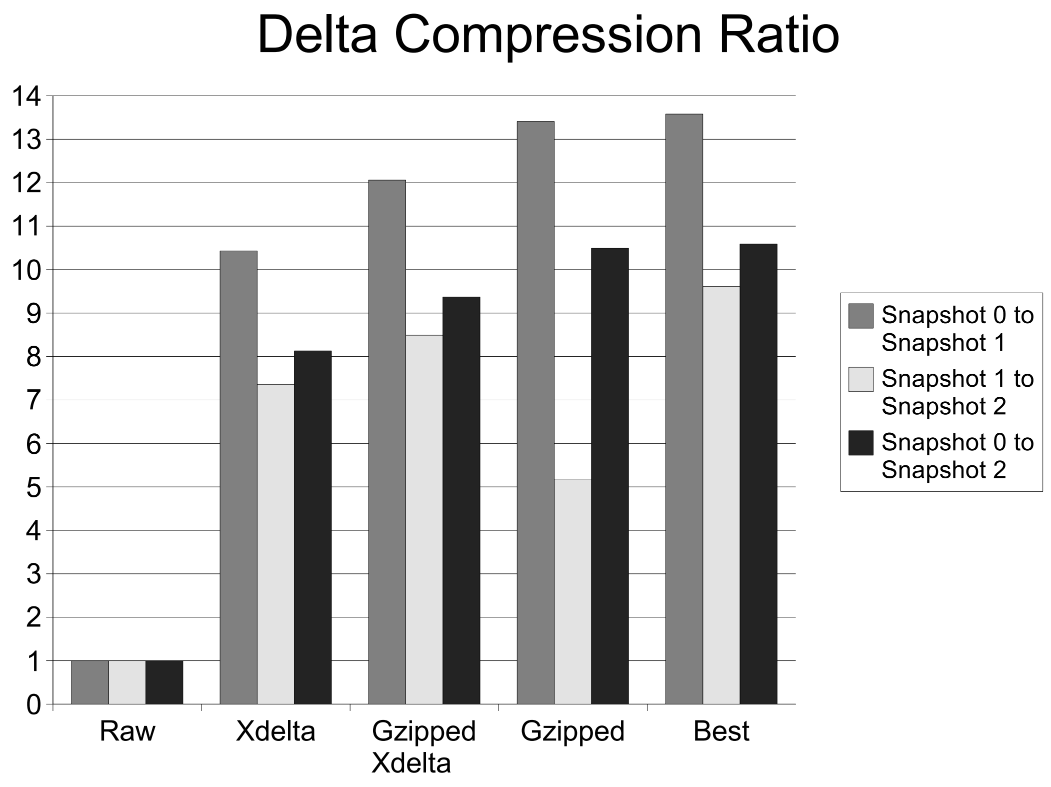

Figure 4: Delta Compression Effectiveness

which replication lag begins to feed on itself and enables timely replication of larger, busier volumes over lower speed links.

3.2 NFS Snapshot Rollover

Exporting a replicated volume via NFS sounds easy except that we expect the filesystem to change state "spontaneously" each time a new volume delta arrives, without requiring current NFS clients to close their TCP connections. To create the effect of jumping the filesystem from state to state as if somebody had been editing the filesystem locally, we need to unmount the old snapshot and mount the new snapshot so that future NFS accesses will be to the new snapshot. The problem is, the Linux server will cache some elements of the client connection state such as the file handle of the filesystem root, which pins the filesystem and prevents it from being unmounted.

Zumastor solves this problem by introducing a nfsd suspend/resume operation. This flushes all cached client state which forces the use count of the exported filesystem snapshot to one, the mount point. This allows the old snapshot to be unmounted and the new snapshot to be mounted in its place, before resuming. Interestingly, the patch to accomplish this is only a few lines, because most of the functionality to accomplish it already existed.

3.3 Incremental Backup using Delta Files

Ddsnap delta files are not just useful for replication, they can also be used for incremental backup. From time to time, a "full volume" delta file can be written to tape, followed periodically by a number of incremental deltas. This should achieve very rapid backup and economical use of tape media, particularly if deltas are generated with more aggressive compression options. To restore, a full (compressed) volume must be retrieved from tape, along with some number of delta files, which are applied sequentially to arrive at a volume state as some particular point in time. Restoring a single file would be a very slow process, however it is also expected to be a rare event. It is more important that backup be fast, so that it is done often.

4 Zumastor volume monitor

The Zumastor volume monitor takes care of most of the chores of setting up virtual block devices, including making socket connections between the ddsnap user space and kernel components, creating the virtual devices with dmsetup, organizing mount points and mounting volumes. It maintains a simple database implemented as a directory tree that stores the configuration and operating status of each volume on a particular host, and provides the administrator with a simple set of commands for adding and removing volumes from the database, and defining their operational configuration. Finally, it takes care of scheduling snapshots and initiating replication cycles.

The Zumastor volume database is organized by volumes, where each volume is completely independent from the others, not even sharing daemons. Each Zumastor volume is either a master or a target. If a master, it has a replication schedule. If a target, then it has an upstream source. In either case, it may have any number of replication targets. Each replication target has exactly one source, which prevents cycles and also allows automatic checking that the correct source is replicating to the correct target.

The replication topology for each volume is completely independent. A given host may offer write/write access to volumes that are replicated to other hosts and read-only access to volumes replicated to it from other hosts. So for example, two servers at widely separated geographic locations might each replicate a volume to the other, which not provides a means of sharing data, but also provides a significant degree of redundancy, particularly if each server backs up both its own read/write volume and the replicated read-only volume to tape.

The replication topology for each volume is a tree, where only the master (root of the tree) behaves differently from the other nodes. The master generates snapshots either periodically or on command. Whenever one of its target hosts is ready to receive a new snapshot delta, the master creates a new snapshot and replicates it to the target, ensuring that the downstream host receives as fresh as possible a view of the master volume. On all other hosts, snapshot deltas are received from an upsteam source and simply passed along down the chain.

Zumastor replication is integrated with NFS in the sense that Zumastor knows how to suspend NFS while it remounts a replicated volume to the latest snapshot, effecting the snapshot rollover described above.

5 The Future

In the future, Zumastor will continue to gain new functionality and improve upon existing functionality. It would be nice to have a graphical front end to the database, and a web interface. It would be nice to see the state of a whole collection of Zumastor servers together in one place, including the state of any replication cycles in progress. It would be natural to integrate more volume management features into Zumastor, such as volume resizing. Zumastor ought to be able to mirror itself to a local machine and fail over NFS service transparently. Zumastor should offer its own incremental backup using delta files. Another creative use of delta files would be to offer access to “nearline” snapshots, where a series of archived reverse deltas are applied to go back further in time than is practical with purely online snapshots.

There is still plenty of room for performance optimization. There is a lot more that can be done with NVRAM, and things can be done to improve the performance without NVRAM, perhaps making some of Zumastor’s replication and backup capabilities practical for use on normal workstations and the cheapest of the cheap servers.

All in all, there remains plenty of work to do and plenty of motivation for doing it.