Shared snapshots

Mikulas Patocka
Red Hat Czech, s.r.o.
Purkynova 99
612 45, Brno
Czech Republic
mpatocka@redhat.com

1 Abstract

Shared snapshots enable the administrator to take many snapshots of the same logical volume. With existing non-shared snapshots, multiple snapshots are possible, but they are implemented very inefficiently — a write to the logical volume copies the data to every snapshot separately, consuming disk space and i/o bandwidth. New shared snapshots implement this efficiently, a write to the logical volume copies data to just one place, possibly sharing this piece of data among large number snapshots.

2 Introduction

Snapshot is a fixed image of a logical volume taken at a specific time. When the user writes to the volume, the snapshot is unchanged.

The most common use for snapshots is online backup. Without snapshots, the administrator has to unmount the volume when performing a backup. Snapshots enable taking backup of a live volume — the administrator takes a snapshot, copies the snapshot to the backup store and meanwhile applications can write to the volume.

Snapshots are implemented as logical volumes that hold the state of the origin before modifications. When the user writes to the origin volume, the previous data is copied to the snapshot and then the write to the origin is allowed. Snapshot maintains its own metadata that specifies what piece of data in the origin was placed to what location in the snapshot. When the user reads the origin volume, he sees the data that he wrote there. When the user reads the snapshot volume, the kernel driver looks up the location in the metadata — if the location is not there, the origin was not yet written at this place, and the read request is redirected to the origin. If the location is in the snapshot metadata, the origin volume was written and the old copy of data before the modification is read from the snapshots. With this mechanism, the snapshot present a static image of the volume that doesn't change with writes to the origin.

Snapshots work in the units of "chunks". A chunk is a configurable number of sectors (for implementation efficiency, it must be a power of two). With write to the origin, the whole chunk (or several chunks) is copied to the snapshot. Snapshots with smaller chunk sizes are more space efficient, snapshots with larger chunk sizes are faster because copying larger pieces of data causes less disk head seeks. The chunk size is configurable because different chunk sizes are optimal for different workloads. Generally, if the workload consist of many small writes to random parts of the origin, then small chunk size is preferred. If the workload consists of large writes or small writes near each other, big chunk size gives better performance.

When the snapshot overflows — there is not any free chunk — the snapshot is invalidated and all the data in the snapshot is lost. This is OK for backup purposes (the user has just to re-run the backup, but he loses no real data). If the snapshot is used for other purposes, the user must monitor the snapshot and extend it so that it doesn't overflow.

The current snapshot implementation supports writing to the snapshots. It is typically not used for backup purposes, but it can be used if the user wants to mount the snapshot volume — journaled filesystems need to write to the volume to replay the journal and get a consistent image. Another use for writable snapshots is a sparse volume — if the user creates a zero volume (a volume containing zeros and consuming no disk space) and takes a big snapshot of that volume, he gets a sparse volume — a volume that appears to be big but consumes a smaller amount of disk storage. Obviously, the user may write only as much data to the sparse volume as its physical size. If he writes more, the snapshot overflows and is invalidated.

2.1 Current snapshot shortcomings

The Linux kernel currently supports *unshared snapshots*[1]. It allows multiple snapshots of the same origin logical volume, but the snapshots are independent. Thus, if the user writes to the origin, the data is copied to each snapshot. Write performance is degraded linearly with the number of snapshots. In practice, the user rarely ever creates more than one of two snapshots, if he created more, performance would drop to unusable state.

Another problem of the current snapshot implementation is the metadata format. The metadata is stored on disk as a linear list of pairs (*origin chunk*, *snapshot chunk*). This metadata format cannot be efficiently searched. Thus, the implementation loads all the metadata in memory, into a hash table, and searches it there. On 64-bit machines it consumes 32 bytes per chunk. In Linux kernel 2.6.25, compression was introduced, up to 256 consecutive entries can be compressed into one entry. It helps to keep memory consumption down for the most cases, but memory consumption becomes nondeterministic — if the entries are not consecutive, it still consumes 32 bytes per chunk.

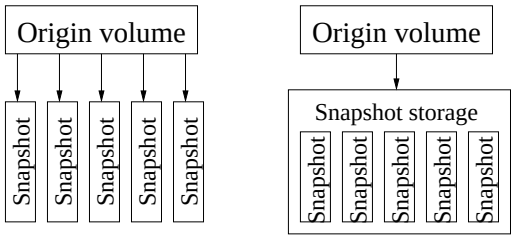

Figure 1: Unshared and shared snapshots

2.2 Shared snapshots

I designed and implemented a new snapshot store called *shared snapshots*. Shared snapshots overcome the disadvantages of current snapshot implementation and enable new possibilities.

With shared snapshots, there can be multiple snapshots of the same logical volume and they are all stored within one volume with one metadata. As a consequence, there is no write inefficiency with multiple snapshots, a write to the origin copies the chunk to just one place in the snapshot store and marks it in the metadata to be shared by multiple snapshots. The implementation supports up to 2^{32} snapshots.

My implementation allows two levels of snapshots — i.e. snapshots-of-snapshots are allowed, but snapshots-of-snapshots-of-snapshots are not.

My implementation stores metadata on disk in B+tree [2] and caches only a small (configurable) part in memory, thus it is extendible to arbitrarily large devices.

Shared snapshot enable new uses: One possible use is to take snapshots at timed intervals, for example every hour, to record system activity. If something goes wrong with the system, the user can find out when did it happen and he can restore the system or a part of it from the last good snapshot. Another possible use for shared snapshots is sharing common data among many virtual machines — the origin contains the initial image and virtual machines take the snapshots, so that common data is stored only once.

3 Shared snapshot format

Shared snapshot store consists of chunks. The chunks may be used for data (those are copies of chunks from the origin volume) or metadata.

3.1 B+tree

Metadata forms a B+tree [2]. B+tree represents the translation of chunk numbers on the origin volume to chunk numbers on the snapshot store. B+tree is keyed by (*chunk number, range of snapshot IDs*); chunk number is the most significant (see figure 2). If we need to look up a certain chunk in a certain snapshot, it searches the tree for the appropriate key.

This range of snapshots IDs allows the snapshot store to share data between multiple snapshots. If the entry has range for example $(1 - 3)$, the appropriate chunk represents data for snapshots 1, 2 and 3.

The snapshot IDs grow constantly, they are never reused. Thus, the snapshot store is always exhausted after creating 2^{32} snapshots (even if the snapshots are deleted meanwhile), however I don't think it imposes a practical limit — if the user created a snapshot every second, the store would last for 136 years.

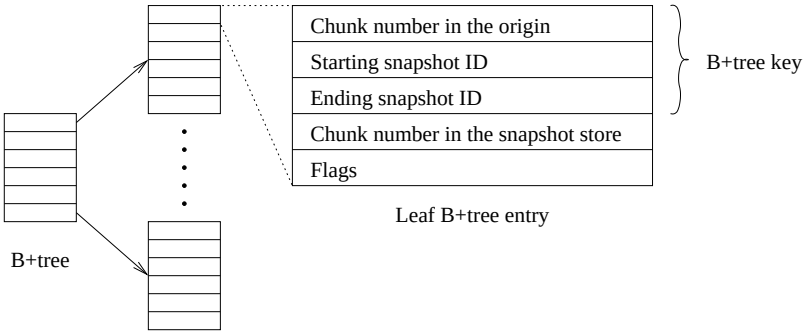

Figure 2: B+tree

3.1.1 Writing to the origin

If the user writes to the origin, we need to check if all the snapshots hold an entry for the given chunk. If there is one or more snapshots that don't hold the chunk, the chunk must be reallocated to these snapshots and only after that, the write to the origin may be allowed.

We scan in-memory map of used snapshots IDs (stored as rb-tree containing ranges), simultaneously we scan B+tree on disk, and we look for ranges of IDs that are in in-memory map and not in the B+tree. We must make new entries for these ranges, add them to the B+tree and copy appropriate data chunks from the origin device to the snapshot store. When the data copy finishes, metadata are committed (see the next section for the description of commit), and finally, user's write to the origin device is allowed. Writing to the origin is shown in figure 3.

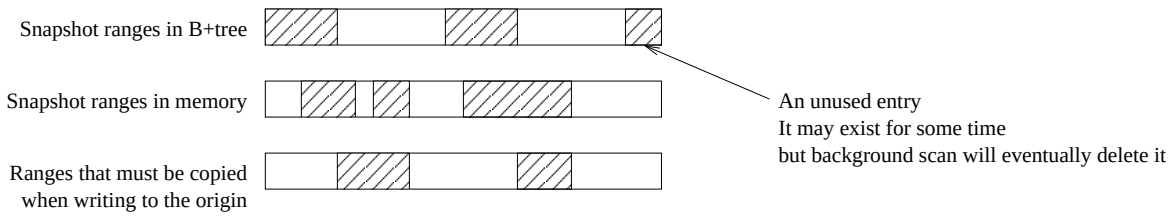

Figure 3: Writing to the origin: compare the ranges stored in B+tree and in memory

3.1.2 Writing to the snapshot

If the user writes to the snapshot to the given chunk, there are several possibilities:

- The snapshot doesn't have the appropriate chunk in the B+tree. In this case, we create B+tree entry for it, copy the chunk from the origin to the snapshot store, commit metadata and then redirect the write to the snapshot store.
- The snapshot has the chunk in the B+tree and that chunk isn't shared with any other snapshots. In this case, we redirect the write directly to the snapshot store and we don't change snapshot store metadata at all.
- The snapshot has the chunk in the B+tree and it is shared with another snapshot. In this case we must copy the data from the snapshot store to another place in the snapshot store to break sharing. In worst case, there will be at most two copies, for example, if there is an entry with range $(1-7)$ and the user writes to the snapshot 4, we must break this entry to three entries with ranges: $(1-3)$, $(4-4)$, $(5-7)$ and copy the original data to two new places.

  Note that entries $(1-3)$ and $(5-7)$ contain the same data, but they point to different chunks. Data for these entries cannot be shared. The sharing isn't perfect. If I wanted to share data for these two entries, I would have to introduce reference counts for data chunks and then it could be allowed for two B+tree entries to point to the same data chunk. I decided against this solution because reference counts would introduce additional complexity (and performance degradation) and this imperfect sharing only happens when the user writes both to the origin and to the snapshots. I assume that writing extensively to both origin and snapshots is uncommon scenario and it is not worth to try to optimize it while slowing down other common cases.

3.1.3 An optimization for large amount of snapshots

There may be a large number of snapshots (like in the example when the user takes timed snapshots) and the implementation should not be dependent on the number of snapshots. I aim for $O(\log n)$ efficiency w.r.t. the number of snapshots.

The algorithm for writing to the origin actually scans the ranges in the B+tree and valid snapshot ranges in the memory and fills in the missing ranges. If the user doesn't write to the snapshots, it results in a copy of at most one data chunk — assume that the user last wrote to the given chunk in the origin after snapshot k and before snapshot $k+1$ was created. Then, all snapshots with ID $\leq k$ are reallocated and all snapshots with ID $> k$ are not reallocated. Thus we create a single ranged entry from $k+1$ to the last snapshot ID.

Nonetheless, even if we copy just one chunk, the algorithm, as described would scan all the entries in the B+tree for the given chunk. That would give $O(n \log n)$ complexity w.r.t. the number of snapshots (scan at most n entries and each entry scan walks a B+tree that is $O(\log n)$ operations). To avoid this tree scanning, I implemented a special flag `DM_MULTISNAP_BT_PREVIOUS_COVERED`. The flag may be set for every B+tree leaf entry and it means that the chunk is reallocated for all snapshots with lower IDs than this entry.

So, the origin reallocation routine takes the last entry in the B+tree for the given chunk, checks its `DM_MULTISNAP_BT_PREVIOUS_COVERED` flag, if it is set, it knows that it doesn't have to check the previous entries. If the flag is not set, it must scan all the entries. An example is shown in figure 4.

`DM_MULTISNAP_BT_PREVIOUS_COVERED` is set when making reallocations because of a write to the origin, it is not set when making reallocations because of a write to the snapshot. The result is that shared snapshots have complexity $O(\log n)$ if the user writes to the origin and not to the snapshots (because `DM_MULTISNAP_BT_PREVIOUS_COVERED` eliminates scanning in every write to the origin) or if the user writes to the snapshots and not to the origin (if there is no write to the origin, there is no scanning of all entries). If the user writes to both, the implementation is $O(n)$. It should be noted that if the user writes mostly to the origin and a little bit to the snapshots (for example to replay journal), only

those few written chunks show this performance anomaly and complexity is still $O(\log n)$ for most of the chunks.

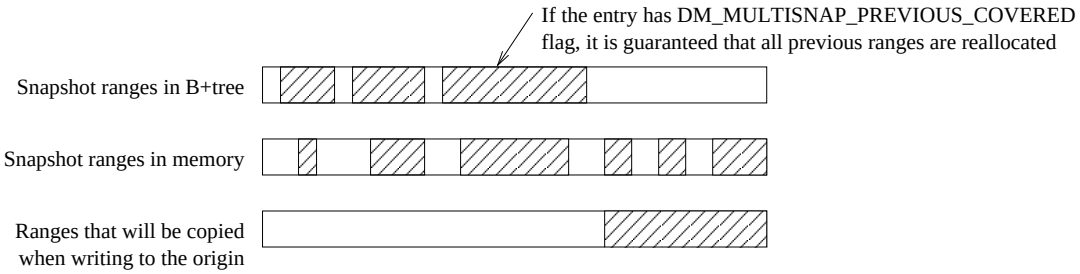

Figure 4: Optimized writing to the origin

3.1.4 Deleting the snapshot

When snapshot is deleted, the whole B+tree must be walked to clean up entries that point to this snapshot. This may be slow, so it is done in background. If the user deletes a snapshot, a snapshot is immediately marked as deleted in the list of snapshots and a background task is spawned that scans the whole B+tree and deletes entries that are not used by any valid snapshot. And example of such entry is shown in figure 3.

This background scan is transparent to the user — except that when he deletes some snapshot, it takes some time until he sees increased free space in the snapshot store.

3.2 Log-structured layout

Shared snapshots use log-structured layout to maintain its consistency. New data are always written to unallocated parts of the device. When the commit block is written, the device atomically moves from one consistent state to another consistent state.

Log-structured layout is efficient for writing because most of the writes are done sequentially and there is little disk head seeking. On the other hand, log structured layout is inefficient for reading and so it is not used in filesystems much — a typical filesystem stores files that are adjacent in the directory tree near each other on disk, so that it minimizes disk seeks on operations like copying all the files in a directory. Log structured layout can't maintain this proximity, it writes any data and metadata to the place where the log head is.

I assume that the snapshot store is stressed mostly with write requests, so I decided to use the log structured layout.

The snapshot store contains one superblock and a number of commit blocks, commit blocks are preallocated at fixed places with constant stride (figure 5). Commit block contains pointers to the root B+tree node, root bitmap directory and other data, thus writing the commit blocks transfers the store from one consistent state to another. Technically, it would be sufficient to have just one commit block — but to minimize seek times, there are many commit blocks and the commit block that is near the chunk most recently written is used. Each commit block contains a commit sequence (a 64-bit number that is constantly increased), when loading the snapshot store, the commit block with the highest sequence number is considered valid.

Superblock contains pointer to chunk from where to start searching for the most recent commit block. If we didn't use the pointer in the superblock, we would have to scan all the commit blocks for the one with the highest sequence number when loading the snapshot store. That would be slow. If we updated the pointer in the superblock always with each commit, the commit would be slow (disk head would have to seek to the superblock). Thus, I decided for a compromise and update the pointer in the superblock only sometimes.

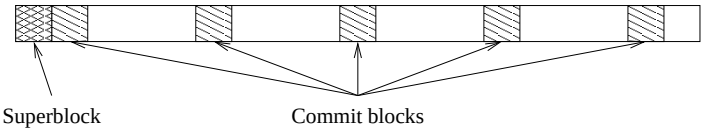

Figure 5: Superblock and commit blocks on the device

Some log-structured filesystems place commit blocks at variable locations (they are placed at the end of data and metadata to be committed) and checksums are used [3] to make sure that all the data and metadata were written correctly. If the checksum matches, the commit is considered completed, if it doesn't match, previous commit state is used. This solution avoids disk head seeks but I decided against it because:

- If we use a simple checksum function (such as CRC32 or a XOR), the function can be trivially fooled into a collision. The result of the collision is that an incomplete commit has matching checksum, it is considered valid and data corruption happens. This can be caused either intentionally (a user storing specific data pattern on the disk) or unintentionally (due to simplicity of these functions, one can't rule out the possibility of a collision under a normal workload).
- If we use a cryptographic hash function to checksum all committed data and metadata, it can't supposedly generate collisions and cause false commits, but these functions are slow to calculate.

3.3 Allocation bitmaps

Free space is managed with bitmaps. We may write only to unallocated parts of the device, so bitmaps must not be at static places. Multi-level bitmap directory points to bitmaps (figure 6). If the device is so small or the chunk size is so big that one bitmap suffices to cover the whole device, the bitmap directory has a depth of zero and there is in fact no bitmap directory.

The bitmap contains simple bits for allocated and free chunks, the bitmap directory contains 8-byte chunk numbers for a lower level of the directory. There are no other metadata in the bitmaps or the directory, so that both bitmap and directory chunks have power-of-two entries and they can be accessed without expensive division.

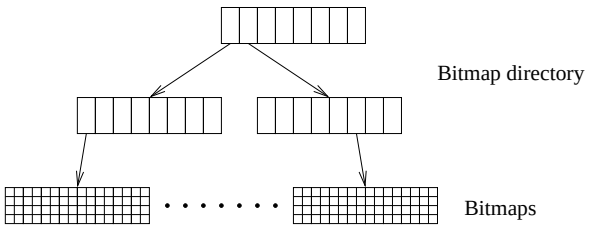

Figure 6: Bitmap directory and bitmaps

The following allocation algorithm is used: A pointer to the last chunk allocated is maintained. We try to allocate the chunk immediately after the pointer (eventually wrapping around to the beginning of the device), if this chunk is not free, we search for one free byte (8 continuous chunks) from the last allocated place to the end of the bitmap. If we find it, we allocate here (possibly stepping back up to 7 chunks to the beginning of the free extent), if we don't find a free byte, we search for one free bit from the last allocated place to the end of the bitmap. If we can't find even a single bit, we continue with the next bitmap, from its beginning.

This is very similar to the algorithm used in ext2/3 filesystem — it also allocates only forward, searches for a free byte first, then for a free bit. I assume that if the algorithm in ext2/3 worked well, so that it hasn't been revised for more than a decade, my implementation will work well too.

3.4 Temporary remaps

Using strict log-structured layout would mean that with each commit we have to write path down from the root to the leaves for B+tree and the bitmap directory. That would write unnecessarily too much data on small commits. To avoid this problem, I use temporary remaps. A commit block has few (27) remap entries that contain pairs (*old chunk number, new chunk number*). If we make a change to leaf nodes in the B+tree or bitmap, we write the leaf node to the new chunk and store pair (*old chunk number, new chunk number*) to the remap table. The remap table is then written with the commit to the commit block.

When the remap table is near overflow, we write paths from the root to the leaves for each entry in the remap table and then we clear the remap table.

3.5 Freelists

Freeing the chunks could not be managed with traditional log-structured design. Freeing a block would make a copy of the bitmap with the appropriate bit cleared, but making that copy causes another freed block (the previous bitmap location), and so on. So, freeing has the potential of causing an infinite loop. Another specialty with freed blocks is that they can't be reused until a commit is performed. If we reused a free block immediately after freeing it, system crash could happen and that free block would be needed in the previous state that is restored after the crash.

To deal with these specialties, I use another structure, a *freelist*. A freelist contains a list of extents that should be freed. If the freelist fills up, another freelist is allocated and the previous freelist is linked to it (so freelists form a single-linked chain). This chain of freelists is attached to the commit block. After the commit is performed, the bits in the freelist are cleared in the bitmaps. If the machine crashes after the commit, the freelist chain is walked on the next load and bits are cleared again.

In practice, the freelist almost always fits into one chunk.

There is a possible optimization (that I haven't yet implemented, but I likely will do) of stuffing a small freelist directly to the commit block to the temporary remap entries that are unused. Testing shows that the freelist would fit there most of the time.

3.6 Snapshots of snapshots

I initially designed the shared snapshot store to hold arbitrary number of simple snapshots, snapshot-of-snapshot functionality was not considered. When new use cases were introduced — such as using the shared snapshot store to hold many images of similar virtual machines — the requirement of snapshots-of-snapshots came up. So I extended the snapshot store format to allow two levels of snapshots.

Initially, there was simple 32-bit snapshots ID. Each snapshot got a new ID incremented by one, IDs of deleted snapshots were never reused. I extended the ID to 64 bits, the upper 32 bits are the snapshot ID, again, incremented by one for every new snapshot and never reused. The lower 32 bits are subsnapshot ID. For master snapshots (i.e. snapshots of the origin device), the subsnapshot ID is `0xFFFFFFFF`. For snapshots-of-snapshots, the upper 32 bits are the same as the master snapshot ID, the lower 32 bits increase from 0 upwards and are never reused (thus, the sequence runs out after creating $2^{32} - 1$ snapshots of a given snapshot, I don't consider this a practically reachable limit).

The reason for this kind of numbering is that it preserves all good qualities of the initial design and the code works almost the same way as if snapshots-of-snapshots didn't exist. It works in the following way:

- If the user creates a snapshot with ID `0x00000001FFFFFFFF` and writes to this snapshot, a B+tree entry is created for range `0x0000000100000000` – `0x00000001FFFFFFFF`. The entry spans the snapshot and all possible future subsnapshots of this snapshot.
- If the user creates a subsnapshot of this snapshot, it gets ID `0x0000000100000000`. The B+tree remains the same and now it represents both the master snapshot and the subsnapshot.

- Now if the user writes to the master snapshot again, an entry with range `0x0000000100000000` – `0x00000001FFFFFFFF` is broken to two entries, one for `0x0000000100000000` and the other for `0x0000000100000001` – `0x00000001FFFFFFFF`. The code is actually the same as when there are no subsnapshots and B+tree breaking is performed because of writes to the chunk that is used by multiple snapshots.

- If the user creates 10 new master snapshots, they get snapshots IDs `0x00000002FFFFFFFF` to `0x0000000BFFFFFFFF`. Now, write to the origin volume creates just one entry with range `0x0000000200000000` – `0x0000000BFFFFFFFF`. That entry spans all the new snapshots and their existing or future subsnapshots. This kind of numbering thus preserves $O(\log n)$ complexity with respect to the number of snapshots.

4 Performance evaluation

In this section I will show some performance measurements of shared and unshared snapshots. The measurements were made on a system with two quad-core 2.3GHz Opterons, on Seagate 300GB 15k SCSI disk.

I applied patches for delayed unplugging of request queue, clearing `SYNC` flag and for increasing size of copy buffer to 16MiB. Without these patches the copy rate would be much slower for both shared and unshared snapshots. It is expected that these patches will be soon applied to the mainstream Linux kernel.

I do measurements for sequential and random writes to the origin. 100MiB was written to the origin device, sequentially from the beginning (with `dd` command), and then randomly scattered in 4k chunks (with custom made program). Sequential write rate on the device without any snapshots is 79 MB/s and random write rate 2.7 MB/s.

First, I show scalability of shared snapshots with respect to the number of snapshots. Chunk 32KiB was used in these tests.

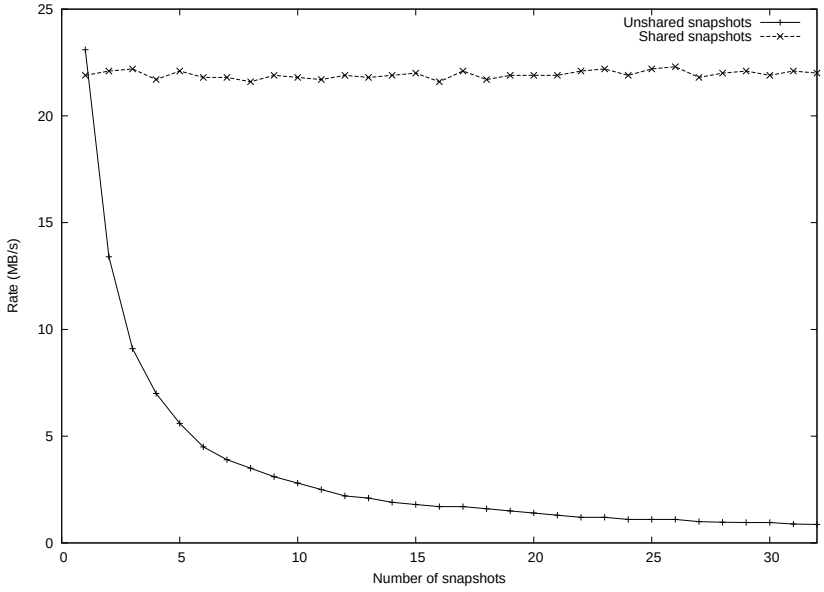

Figure 7: Sequential write rate depending on the number of snapshots

In figures 7 and 8 we can see how write rate depends on the number of snapshots. For 1 snapshot unshared snapshots are slightly more efficient (because they are simpler), for more snapshots, performance degrades with unshared snapshots, but stays the same with shared snapshots. In these figures we can see that performance of shared snapshots is independent on the number of snapshots.

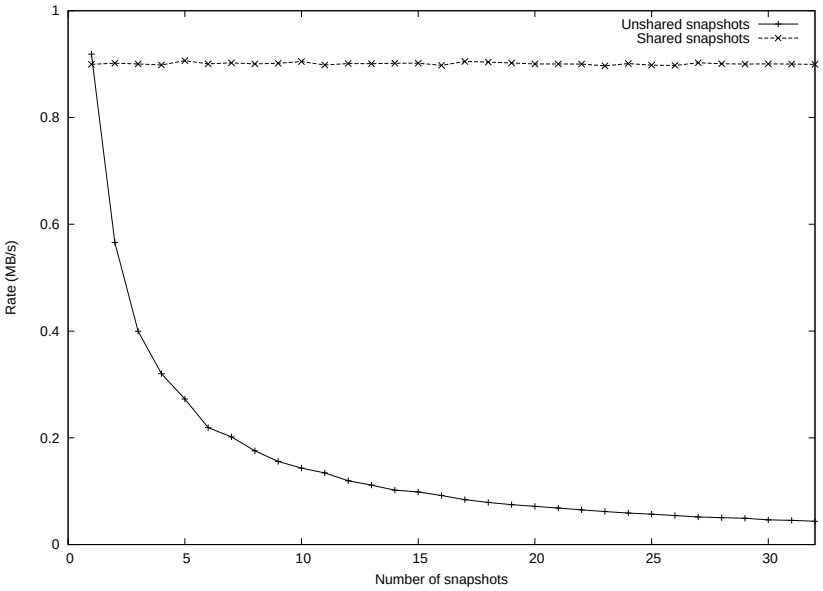

Figure 8: Random write rate depending on the number of snapshots

Next, to give some idea about chunk sizes, I tested performance of both unshared and shared snapshots with different chunk sizes. Only one snapshots was created in both cases.

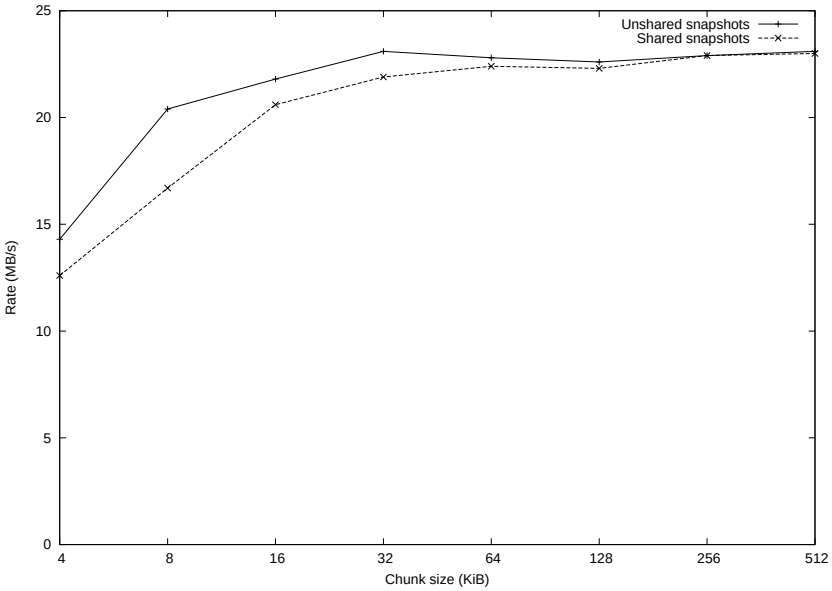

Figure 9: Sequential write rate depending on the chunk size

Figure 9 shows performance for sequential access, figure 10 shows performance for random access. As I expected, sequential performance improves with increasing chunk size (because data is copied in larger blocks and fewer metadata accesses are needed to copy the same amount of data). For shared snapshots, you see more performance decrease with decreased chunk size; chunk size 64KiB seems to be ideal becuase it is not too high and performance of shared snapshots is only 2% lower.

Note that one write to the chunk means three operations on the disk (read old chunk, write it to the snapshots, write new chunk). Thus, performance 23MB/s in figure 9 really saturates the disk with 69MB/s combined read/write transfer rate. The disk has 79MB/s transfer rate at that region, the remaining time is used for disk head seeks. Thus, with large chunk size, both shared and unshared snapshots

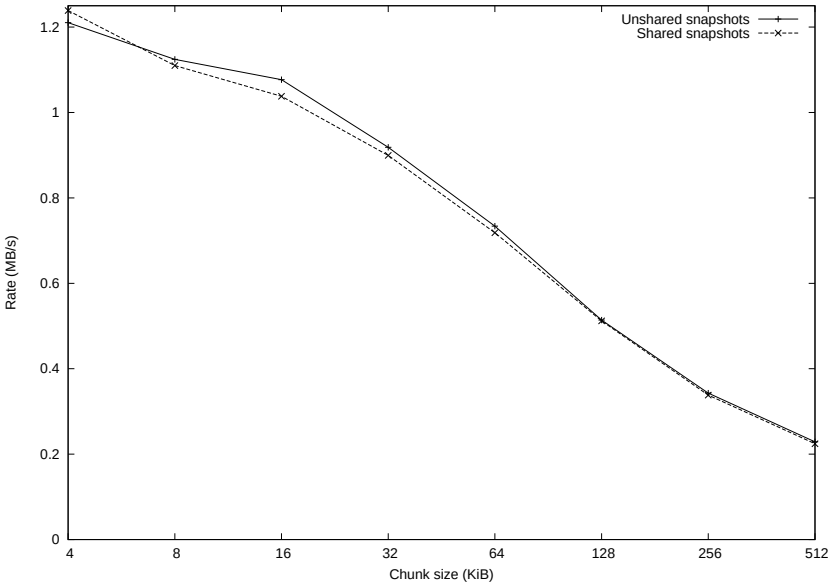

Figure 10: Random write rate depending on the chunk size

utilize 87% of the time for transfers and 13% for seeks.

Random access performance (figure 10) decreases with increasing chunk size because whole chunks are being copied, even on small write to the middle of the chunk.

5 Conclusion

In this paper I described the design of shared snapshots, a new feature that will be implemented in Linux LVM. The user can create arbitrarily high number of snapshots without associated performance degradation. Shared snapshots use B+tree to access data efficiently, log structured design to maintain consistency across crashes.

Shared snapshots enable new use cases: snapshots can be taken periodically and kept to record system activity. Shared snapshots can also be used to store images of virtual machines that have many common blocks.

Patches for the Linux kernel and userspace LVM tool can be downloaded from
`http://people.redhat.com/mpatocka/patches/kernel/new-snapshots/` and
`http://people.redhat.com/mpatocka/patches/userspace/new-snapshots/`.

References

[1] LVM snapshots — `http://tldp.org/HOWTO/LVM-HOWTO/snapshotintro.html`

[2] B+tree — `http://en.wikipedia.org/wiki/B%2Btree`

[3] M. Jambor, T. Hruby, J. Taus, K. Krchak, V. Holub: *Implementation of a Linux log-structured file system with a garbage collector*, ACM SIGOPS Operating Systems Review <2007>, <volume 41>, <24–32>