SMR Layout Optimisation for XFS (v0.2, March 2015)

Contents

1 Overview **1**

2 Concepts **1**

3 Journal modifications **1**

4 Data zones **2**

4.1 Zone Cleaner 2

4.2 Reverse mapping btrees 2

5 Mkfs **3**

6 Repair **3**

7 Quantification of Random Write Zone Capacity **3**

8 Kernel implementation **3**

9 Problem cases **3**

9.1 Concurrent writes 4

9.2 Crash recovery 4

9.3 Preallocation Issues 4

9.4 Allocation Alignemnt 5

9.5 RAID on SMR. 5

1 Overview

This document describes a relatively simple way of modifying XFS using existing on disk structures to be able to use host-managed SMR drives.

This assumes a userspace ZBC implementation such as libzbc will do all the heavy lifting work of laying out the structure of the filesystem, and that it will perform things like zone write pointer checking/resetting before the filesystem is mounted.

2 Concepts

SMR is architected to have a set of sequentially written zones which don't allow out of order writes, nor do they allow overwrites of data already written in the zone. Zones are typically in the order of 256MB, though may actually be of variable size as physical geometry of the drives differ from inner to outer edges.

SMR drives also typically have an outer section that is CMR technology - it allows random writes and overwrites to any area within those zones. Drive managed SMR devices use this region for internal metadata journalling for block remapping tables and as a staging area for data writes before being written out in sequential fashion ito zones after block remapping has been performed.

Recent research has shown that 6TB seagate drives have a 20-25GB CMR zone, which is more than enough for our purposes. Information from other vendors indicate that some drives will have much more CMR, hence if we design for the known sizes in the Seagate drives we will be fine for other drives just coming onto the market right now.

For host managed/aware drives, we are going to assume that we can use this area directly for filesystem metadata - for our own mapping tables and things like the journal, inodes, directories and free space tracking. We are also going to assume that we can find these regions easily in the ZBC information, and that they are going to be contiguous rather than spread all over the drive.

XFS already has a data-only device call the "real time" device, whose free space information is tracked externally in bitmaps attached to inodes that exist in the "data" device. All filesystem metadata exists in the "data" device, except maybe the journal which can also be in an external device.

A key constraint we need to work within here is that RAID on SMR drives is a long way off. The main use case is for bulk storage of data in the back end of distributed object stores (i.e. cat pictures on the intertubes) and hence a filesystem per drive is the typical configuration we'll be chasing here. Similarly, partitioning of SMR drives makes no sense for host aware drives, so we are going to constrain the architecture to a single drive for now.

3 Journal modifications

Because the XFS journal is a sequentially written circular log, we can actually use SMR zones for it - it does not need to be in the metadata region. This requires a small amount of additional complexity - we can't wrap the log as we currnetly do, we'll need to split the log across two zones so that we can push the tail into the same zone as the head, then reset the now unused zone and then when the log wraps it can simply start again form the beginning of the erased zone.

Like a normal spinning disk, we'll want to place the log in a pair of zones near the middle of the drive so that we minimise the worst case seek cost of a log write to half of a full disk seek. There may be advantage to putting it right next to the metadata zone, but typically metadata writes are not correlated with log writes.

Hence the only real functionality we need to add to the log is the tail pushing modificaitons to move the tail into the same zone as the head, as well as being able to trigger and block on zone write pointer reset operations.

The log doesn't actually need to track the zone write pointer, though log recovery will need to limit the recovery head to the current write pointer of the lead zone. Modifications here are limited to the function that finds the head of the log, and can actually be used to speed up the search algorithm.

However, given the size of the CMR zones, we can host the journal in an unmodified manner inside the CMR zone and not have to worry about zone awareness. This is by far the simplest solution to the problem.

4 Data zones

What we need is a mechanism for tracking the location of zones (i.e. start LBA), free space/write pointers within each zone, and some way of keeping track of that information across mounts. If we assign a real time bitmap/summary inode pair to each zone, we have a method of tracking free space in the zone. We can use the existing bitmap allocator with a small tweak (sequentially ascending, packed extent allocation only) to ensure that newly written blocks are allocated in a sane manner.

We're going to need userspace to be able to see the contents of these inodes; read only access wil be needed to analyse the contents of the zone, so we're going to need a special directory to expose this information. It would be useful to have a ".zones" directory hanging off the root directory that contains all the zone allocation inodes so userspace can simply open them.

THis biggest issue that has come to light here is the number of zones in a device. Zones are typically 256MB in size, and so we are looking at 4,000 zones/TB. For a 10TB drive, that's 40,000 zones we have to keep track of. And if the devices keep getting larger at the expected rate, we're going to have to deal with zone counts in the hundreds of thousands. Hence a single flat directory containing all these inodes is not going to scale, nor will we be able to keep them all in memory at once.

As a result, we are going to need to group the zones for locality and efficiency purposes, likely as "zone groups" of, say, up to 1TB in size. Luckily, by keeping the zone information in inodes the information can be demand paged and so we don't need to pin thousands of inodes and bitmaps in memory. Zone groups also have other benefits...

While it seems like tracking free space is trivial for the purposes of allocation (and it is!), the complexity comes when we start to delete or overwrite data. Suddenly zones no longer contain contiguous ranges of valid data; they have "freed" extents in the middle of them that contian stale data. We can't use that "stale space" until the entire zone is made up of "stale" extents. Hence we need a Cleaner.

4.1 Zone Cleaner

The purpose of the cleaner is to find zones that are mostly stale space and consolidate the remaining referenced data into a new, contigious zone, enabling us to then "clean" the stale zone and make it available for writing new data again.

The real complexity here is finding the owner of the data that needs to be move, but we are in the process of solving that with the reverse mapping btree and parent pointer functionality. This gives us the mechanism by which we can quickly re-organise files that have extents in zones that need cleaning.

The key word here is "reorganise". We have a tool that already reorganises file layout: xfs_fsr. The "Cleaner" is a finely targetted policy for xfs_fsr - instead of trying to minimise fixpel fragments, it finds zones that need cleaning by reading their summary info from the /.zones/ directory and analysing the free bitmap state if there is a high enough percentage of stale blocks. From there we can use the reverse mapping to find the inodes that own the extents those zones. And from there, we can run the existing defrag code to rewrite the data in the file, thereby marking all the old blocks stale. This will make almost stale zones entirely stale, and hence then be able to be reset.

Hence we don't actually need any major new data moving functionality in the kernel to enable this, except maybe an event channel for the kernel to tell xfs_fsr it needs to do some cleaning work.

If we arrange zones into zoen groups, we also have a method for keeping new allocations out of regions we are re-organising. That is, we need to be able to mark zone groups as "read only" so the kernel will not attempt to allocate from them while the cleaner is running and re-organising the data within the zones in a zone group. This ZG also allows the cleaner to maintain some level of locality to the data that it is re-arranging.

4.2 Reverse mapping btrees

One of the complexities is that the current reverse map btree is a per allocation group construct. This means that, as per the current design and implementation, it will not work with the inode based bitmap allocator. This, however, is not actually a major problem thanks to the generic btree library that XFS uses.

That is, the generic btree library in XFS is used to implement the block mapping btree held in the data fork of the inode. Hence we can use the same btree implementation as the per-AG rmap btree, but simply add a couple of functions, set a couple of flags and host it in the inode data fork of a third per-zone inode to track the zone's owner information.

5 Mkfs

Mkfs is going to have to integrate with the userspace zbc libraries to query the layout of zones from the underlying disk and then do some magic to lay out al the necessary metadata correctly. I don't see there being any significant challenge to doing this, but we will need a stable libzbc API to work with and it will need ot be packaged by distros.

If mkfs cannot find ensough random write space for the amount of metadata we need to track all the space in the sequential write zones and a decent amount of internal fielsystem metadata (inodes, etc) then it will need to fail. Drive vendors are going to need to provide sufficient space in these regions for us to be able to make use of it, otherwise we'll simply not be able to do what we need to do.

mkfs will need to initialise all the zone allocation inodes, reset all the zone write pointers, create the /.zones directory, place the log in an appropriate place and initialise the metadata device as well.

6 Repair

Because we've limited the metadata to a section of the drive that can be overwritten, we don't have to make significant changes to xfs_repair. It will need to be taught about the multiple zone allocation bitmaps for it's space reference checking, but otherwise all the infrastructure we need ifor using bitmaps for verifying used space should already be there.

THere be dragons waiting for us if we don't have random write zones for metadata. If that happens, we cannot repair metadata in place and we will have to redesign xfs_repair from the ground up to support such functionality. That's jus tnot going to happen, so we'll need drives with a significant amount of random write space for all our metadata. . . . . .

7 Quantification of Random Write Zone Capacity

A basic guideline is that for 4k blocks and zones of 256MB, we'll need 8kB of bitmap space and two inodes, so call it 10kB per 256MB zone. That's 40MB per TB for free space bitmaps. We'll want to suport at least 1 million inodes per TB, so that's another 512MB per TB, plus another 256MB per TB for directory structures. There's other bits and pieces of metadata as well (attribute space, internal freespace btrees, reverse map btrees, etc.

So, at minimum we will probably need at least 2GB of random write space per TB of SMR zone data space. Plus a couple of GB for the journal if we want the easy option. For those drive vendors out there that are listening and want good performance, replace the CMR region with a SSD. . . .

8 Kernel implementation

The allocator will need to learn about multiple allocation zones based on bitmaps. They aren't really allocation groups, but the initialisation and iteration of them is going to be similar to allocation groups. To get use going we can do some simple mapping between inode AG and data AZ mapping so that we keep some form of locality to related data (e.g. grouping of data by parent directory).

We can do simple things first - simply rotoring allocation across zones will get us moving very quickly, and then we can refine it once we have more than just a proof of concept prototype.

Optimising data allocation for SMR is going to be tricky, and I hope to be able to leave that to drive vendor engineers. . . .

Ideally, we won't need a zbc interface in the kernel, except to erase zones. I'd like to see an interface that doesn't even require that. For example, we issue a discard (TRIM) on an entire zone and that erases it and resets the write pointer. This way we need no new infrastructure at the filesystem layer to implement SMR awareness. In effect, the kernel isn't even aware that it's an SMR drive underneath it.

9 Problem cases

There are a few elephants in the room.

9.1 Concurrent writes

What happens when an application does concurrent writes into a file (either by threads or AIO), and allocation happens in the opposite order to the IO being dispatched. i.e., with a zone write pointer at block X, this happens:

```
Task A                  Task B
write N                 write N + 1
allocate X
                        allocate X + 1
submit_bio              submit_bio
<blocks in Io stack>    IO to block X+1 dispatched.
```

And so even though we allocated the IO in incoming order, the dispatch order was different.

I don't see how the filesystem can prevent this from occurring, except to completely serialise IO to zone. i.e. while we have a block allocation and no write completion, no other allocations to that zone can take place. If that's the case, this is going to cause massive fragmentation and/or severe IO latency problems for any application that has this sort of IO engine.

There is a block layer solution to this in the works - the block layer will track the write pointer in each zone and if it gets writes out of order it will requeue the IO at the tail of the queue, hence allowing the IO that has been delayed to be issued before the out of order write.

9.2 Crash recovery

Write pointer location is undefined after power failure. It could be at an old location, the current location or anywhere in between. The only guarantee that we have is that if we flushed the cache (i.e. fsync'd a file) then they will at least be in a position at or past the location of the fsync.

Hence before a filesystem runs journal recovery, all it's zone allocation write pointers need to be set to what the drive thinks they are, and all of the zone allocation beyond the write pointer need to be cleared. We could do this during log recovery in kernel, but that means we need full ZBC awareness in log recovery to iterate and query all the zones.

Hence it's not clear if we want to do this in userspace as that has it's own problems e.g. we'd need to have xfs.fsck detect that it's a smr filesystem and perform that recovery, or write a mount.xfs helper that does it prior to mounting the filesystem. Either way, we need to synchronise the on-disk filesystem state to the internal disk zone state before doing anything else.

This needs more thought, because I have a nagging suspiscion that we need to do this write pointer resynchronisation **after log recovery** has completed so we can determine if we've got to now go and free extents that the filesystem has allocated and are referenced by some inode out there. This, again, will require reverse mapping lookups to solve.

9.3 Preallocation Issues

Because we can only do sequential writes, we can only allocate space that exactly matches the write being performed. That means we **cannot preallocate extents**. The reason for this is that preallocation will physically separate the data write location from the zone write pointer. e.g. if we use preallocation to allocate space we are about to do random writes into to prevent fragmentation. We cannot do this on ZBC drives, we have to allocate specifically for the IO we are going to perform.

As a result, we lose almost all the existing mechanisms we use for preventing fragmentation. Speculative EOF preallocation with delayed allocation cannot be used, fallocate cannot be used to preallocate physical extents, and extent size hints cannot be used because they do "allocate around" writes.

We're trying to do better without much investment in time and resources here, so the compromise is that we are going to have to rely on xfs_fsr to clean up fragmentation after the fact. Luckily, the other functions we need from xfs_fsr (zone cleaning) also act to defragment free space so we don't have to care about trading contiguous filesystem for free space fragmentation and that downward spiral.

I suspect the best we will be able to do with fallocate based preallocation is to mark the region as delayed allocation.

9.4 Allocation Alignemnt

With zone based write pointers, we lose all capability of write alignment to the underlying storage - our only choice to write is the current set of write pointers we have access to. There are several methods we could use to work around this problem (e.g. put a slab-like allocator on top of the zones) but that requires completely redesigning the allocators for SMR. Again, this may be a step too far....

9.5 RAID on SMR....

How does RAID work with SMR, and exactly what does that look like to the filesystem?

How does libzbc work with RAID given it is implemented through the scsi ioctl interface?

How does RAID repair parity errors in place? Or does the RAID layer now need a remapping layer so the LBA or rewritten stripes remain the same? Indeed, how do we handle partial stripe writes which will require multiple parity block writes?

What does the geometry look like (stripe unit, width) and what does the write pointer look like? How does RAID track all the necessary write pointers and keep them in sync? What about RAID1 with it's dirty region logging to minimise resync time and overhead?