XFS Self Describing Metadata

Introduction

The largest scalability problem facing XFS is not one of algorithmic scalability, but of verification of the filesystem structure. Scalabilty of the structures and indexes on disk and the algorithms for iterating them are adequate for supporting PB scale filesystems with billions of inodes, however it is this very scalability that causes the verification problem.

Almost all metadata on XFS is dynamically allocated. The only fixed location metadata is the allocation group headers (SB, AGF, AGFL and AGI), while all other metadata structures need to be discovered by walking the filesystem structure in different ways. While this is already done by userspace tools for validating and repairing the structure, there are limits to what they can verify, and this in turn limits the supportable size of an XFS filesystem.

For example, it is entirely possible to manually use xfs_db and a bit of scripting to analyse the structure of a 100TB filesystem when trying to determine the root cause of a corruption problem, but it is still mainly a manual task of verifying that things like single bit errors or misplaced writes weren't the ultimate cause of a corruption event. It may take a few hours to a few days to perform such forensic analysis, so for at this scale root cause analysis is entirely possible.

However, if we scale the filesystem up to 1PB, we now have 10x as much metadata to analyse and so that analysis blows out towards weeks/months of forensic work. Most of the analysis work is slow and tedious, so as the amount of analysis goes up, the more likely that the cause will be lost in the noise. Hence the primary concern for supporting PB scale filesystems is minimising the time and effort required for basic forensic analysis of the filesystem structure.

Self Describing Metadata

One of the problems with the current metadata format is that apart from the magic number in the metadata block, we have no other way of identifying what it is supposed to be. We can't even identify if it is the right place. Put simply, you can't look at a single metadata block in isolation and say "yes, it is supposed to be there and the contents are valid".

Hence most of the time spent on forensic analysis is spent doing basic verification of metadata values, looking for values that are in range (and hence not detected by automated verification checks) but are not correct. Finding and understanding how things like cross linked block lists (e.g. sibling pointers in a btree end up with loops in them) are the key to understanding what went wrong, but it is impossible to tell what order the blocks were linked into each other or written to disk after the fact.

Hence we need to record more information into the metadata to allow us to quickly determine if the metadata is intact and can be ignored for the purpose of analysis. We can't protect against every possible type of error, but we can ensure that common types of errors are easily detectable. Hence the concept of self describing metadata.

The first, fundamental requirement of self describing metadata is that the metadata object contains some form of unique identifier in a well known location. This allows us to identify the expected contents of the block and hence parse and verify the metadata object. IF we can't independently identify the type of metadata in the object, then the metadata doesn't describe itself very well at all!

Luckily, almost all XFS metadata has magic numbers embedded already - only the AGFL, remote symlinks and remote attribute blocks do not contain identifying magic numbers. Hence we can change the on-disk format of all these objects to add more identifying information and detect this simply by changing the magic numbers in the metadata objects. That is, if it has the current magic number, the metadata isn't self identifying. If it contains a new magic number, it is self identifying and we can do much more expansive automated verification of the metadata object at runtime, during forensic analysis or repair.

As a primary concern, self describing metadata needs some form of overall integrity checking. We cannot trust the metadata if we cannot verify that it has not been changed as a result of external influences. Hence we need some form of integrity check, and this is done by adding CRC32c validation to the metadata block. If we can verify the block contains the metadata it was intended to contain, a large amount of the manual verification work can be skipped.

CRC32c was selected as metadata cannot be more than 64k in length in XFS and hence a 32 bit CRC is more than sufficient to detect multi-bit errors in metadata blocks. CRC32c is also now hardware accelerated on common CPUs so it is fast. So while CRC32c is not the strongest of possible integrity checks that could be used, it is more than sufficient for our needs and has relatively little overhead. Adding support for larger integrity fields and/or algorithms does really provide any extra value over CRC32c, but it does add a lot of complexity and so there is no provision for changing the integrity checking mechanism.

Self describing metadata needs to contain enough information so that the metadata block can be verified as being in the correct place without needing to look at any other metadata. This means it needs to contain location information. Just adding a block number to the metadata is not sufficient to protect against mis-directed writes - a write might be misdirected to the wrong LUN and so be written to the "correct block" of the wrong filesystem. Hence location information must contain a filesystem identifier as well as a block number.

Another key information point in forensic analysis is knowing who the metadata block belongs to. We already know the type, the location, that it is valid and/or corrupted, and how long ago that it was last modified. Knowing the owner of the block is important as it allows us to find other related metadata to determine the scope of the corruption. For example, if we have a extent btree object, we don't know what inode it belongs to and hence have to walk the entire filesystem to find the owner of the block. Worse, the corruption could mean that no owner can be found (i.e. it's an orphan block), and so without an owner field in the metadata we have

no idea of the scope of the corruption. If we have an owner field in the metadata object, we can immediately do top down validation to determine the scope of the problem.

Different types of metadata have different owner identifiers. For example, directory, attribute and extent tree blocks are all owned by an inode, while freespace btree blocks are owned by an allocation group. Hence the size and contents of the owner field are determined by the type of metadata object we are looking at. The owner information can also identify misplaced writes (e.g. freespace btree block written to the wrong AG).

Self describing metadata also needs to contain some indication of when it was written to the filesystem. One of the key information points when doing forensic analysis is how recently the block was modified. Correlation of set of corrupted metadata blocks based on modification times is important as it can indicate whether the corruptions are related, whether there's been multiple corruption events that lead to the eventual failure, and even whether there are corruptions present that the run-time verification is not detecting.

For example, we can determine whether a metadata object is supposed to be free space or still allocated if it is still referenced by its owner by looking at when the free space btree block that contains the block was last written compared to when the metadata object itself was last written. If the free space block is more recent than the object and the object's owner, then there is a very good chance that the block should have been removed from the owner.

To provide this "written timestamp", each metadata block gets the Log Sequence Number (LSN) of the most recent transaction it was modified on written into it. This number will always increase over the life of the filesystem, and the only thing that resets it is running xfs_repair on the filesystem. Further, by use of the LSN we can tell if the corrupted metadata all belonged to the same log checkpoint and hence have some idea of how much modification occurred between the first and last instance of corrupt metadata on disk and, further, how much modification occurred between the corruption being written and when it was detected.

Runtime Validation

Validation of self-describing metadata takes place at runtime in two places:

- immediately after a successful read from disk
- immediately prior to write IO submission

The verification is completely stateless - it is done independently of the modification process, and seeks only to check that the metadata is what it says it is and that the metadata fields are within bounds and internally consistent. As such, we cannot catch all types of corruption that can occur within a block as there may be certain limitations that operational state enforces of the metadata, or there may be corruption of interblock relationships (e.g. corrupted sibling pointer lists). Hence we still need stateful checking in the main code body, but in general most of the per-field validation is handled by the verifiers.

For read verification, the caller needs to specify the expected type of metadata that it should see, and the IO completion process verifies that the metadata object matches what was expected. If the verification process fails, then it marks the object being read as EFSCORRUPTED. The caller needs to catch this error (same as for IO errors), and if it needs to take special action due to a verification error it can do so by catching the EFSCORRUPTED error value. If we need more discrimination of error type at higher levels, we can define new error numbers for different errors as necessary.

The first step in read verification is checking the magic number and determining whether CRC validating is necessary. If it is, the CRC32c is calculated and compared against the value stored in the object itself. Once this is validated, further checks are made against the location information, followed by extensive object specific metadata validation. If any of these checks fail, then the buffer is considered corrupt and the EFSCORRUPTED error is set appropriately.

Write verification is the opposite of the read verification - first the object is extensively verified and if it is OK we then update the LSN from the last modification made to the object, After this, we calculate the CRC and insert it into the object. Once this is done the write IO is allowed to continue. If any error occurs during this process, the buffer is again marked with a EFSCORRUPTED error for the higher layers to catch.

Structures

A typical on-disk structure needs to contain the following information:

```
struct xfs_ondisk_hdr {
        __be32  magic;              /* magic number */
        __be32  crc;                /* CRC, not logged */
        uuid_t  uuid;               /* filesystem identifier */
        __be64  owner;              /* parent object */
        __be64  blkno;              /* location on disk */
        __be64  lsn;                /* last modification in log, not logged */
};
```

Depending on the metadata, this information may be part of a header structure separate to the metadata contents, or may be distributed through an existing structure. The latter occurs with metadata that already contains some of this information, such as the superblock and AG headers.

Other metadata may have different formats for the information, but the same level of information is generally provided. For example:

- short btree blocks have a 32 bit owner (ag number) and a 32 bit block number for location. The two of these combined provide the same information as @owner and @blkno in eh above structure, but using 8 bytes less space on disk.
- directory/attribute node blocks have a 16 bit magic number, and the header that contains the magic number has other information in it as well. hence the additional metadata headers change the overall format of the metadata.

A typical buffer read verifier is structured as follows:

```
#define XFS_FOO_CRC_OFF             offsetof(struct xfs_ondisk_hdr, crc)

static void
xfs_foo_read_verify(
        struct xfs_buf      *bp)
{
    struct xfs_mount *mp = bp->b_mount;

        if ((xfs_sb_version_hascrc(&mp->m_sb) &&
            !xfs_verify_cksum(bp->b_addr, BBTOB(bp->b_length),
                                    XFS_FOO_CRC_OFF)) ||
            !xfs_foo_verify(bp)) {
                XFS_CORRUPTION_ERROR(__func__, XFS_ERRLEVEL_LOW, mp, bp->b_addr);
                xfs_buf_ioerror(bp, EFSCORRUPTED);
        }
}
```

The code ensures that the CRC is only checked if the filesystem has CRCs enabled by checking the superblock of the feature bit, and then if the CRC verifies OK (or is not needed) it verifies the actual contents of the block.

The verifier function will take a couple of different forms, depending on whether the magic number can be used to determine the format of the block. In the case it can't, the code is structured as follows:

```
static bool
xfs_foo_verify(
        struct xfs_buf              *bp)
{
        struct xfs_mount    *mp = bp->b_mount;
        struct xfs_ondisk_hdr       *hdr = bp->b_addr;

        if (hdr->magic != cpu_to_be32(XFS_FOO_MAGIC))
                return false;

        if (!xfs_sb_version_hascrc(&mp->m_sb)) {
                if (!uuid_equal(&hdr->uuid, &mp->m_sb.sb_uuid))
                        return false;
                if (bp->b_bn != be64_to_cpu(hdr->blkno))
                        return false;
                if (hdr->owner == 0)
                        return false;
        }

        /* object specific verification checks here */

        return true;
}
```

If there are different magic numbers for the different formats, the verifier will look like:

```
static bool
xfs_foo_verify(
        struct xfs_buf              *bp)
{
        struct xfs_mount    *mp = bp->b_mount;
        struct xfs_ondisk_hdr       *hdr = bp->b_addr;

        if (hdr->magic == cpu_to_be32(XFS_FOO_CRC_MAGIC)) {
                if (!uuid_equal(&hdr->uuid, &mp->m_sb.sb_uuid))
                        return false;
                if (bp->b_bn != be64_to_cpu(hdr->blkno))
                        return false;
                if (hdr->owner == 0)
                        return false;
        } else if (hdr->magic != cpu_to_be32(XFS_FOO_MAGIC))
                return false;

        /* object specific verification checks here */

        return true;
}
```

Write verifiers are very similar to the read verifiers, they just do things in the opposite order to the read verifiers. A typical write

verifier:

```
static void
xfs_foo_write_verify(
        struct xfs_buf      *bp)
{
        struct xfs_mount    *mp = bp->b_mount;
        struct xfs_buf_log_item     *bip = bp->b_fspriv;

        if (!xfs_foo_verify(bp)) {
                XFS_CORRUPTION_ERROR(__func__, XFS_ERRLEVEL_LOW, mp, bp->b_addr);
                xfs_buf_ioerror(bp, EFSCORRUPTED);
                return;
        }

        if (!xfs_sb_version_hascrc(&mp->m_sb))
                return;


        if (bip) {
                struct xfs_ondisk_hdr       *hdr = bp->b_addr;
                hdr->lsn = cpu_to_be64(bip->bli_item.li_lsn);
        }
        xfs_update_cksum(bp->b_addr, BBTOB(bp->b_length), XFS_FOO_CRC_OFF);
}
```

This will verify the internal structure of the metadata before we go any further, detecting corruptions that have occurred as the metadata has been modified in memory. If the metadata verifies OK, and CRCs are enabled, we then update the LSN field (when it was last modified) and calculate the CRC on the metadata. Once this is done, we can issue the IO.

Inodes and Dquots

Inodes and dquots are special snowflakes. They have per-object CRC and self-identifiers, but they are packed so that there are multiple objects per buffer. Hence we do not use per-buffer verifiers to do the work of per-object verification and CRC calculations. The per-buffer verifiers simply perform basic identification of the buffer - that they contain inodes or dquots, and that there are magic numbers in all the expected spots. All further CRC and verification checks are done when each inode is read from or written back to the buffer.

The structure of the verifiers and the identifiers checks is very similar to the buffer code described above. The only difference is where they are called. For example, inode read verification is done in xfs_inode_from_disk() when the inode is first read out of the buffer and the struct xfs_inode is instantiated. The inode is already extensively verified during writeback in xfs_iflush_int, so the only addition here is to add the LSN and CRC to the inode as it is copied back into the buffer.

XXX: inode unlinked list modification doesn't recalculate the inode CRC! None of the unlinked list modifications check or update CRCs, neither during unlink nor log recovery. So, it's gone unnoticed until now. This won't matter immediately - repair will probably complain about it - but it needs to be fixed.