DRAFT: Ext4 with Parity and Checksums

Brian Kroth
bpkroth@cs.wisc.edu

Suli Yang
suli@cs.wisc.edu

Abstract

NOTE: Due to a number of reasons we have abandoned the pursuit of handling this within the filesystem layer. Instead, we've decided to adapt Linux's MD software RAID implementation to include checksums and to reuse the techniques previously developed in [2] in order to speed up recovery of the RAID to a consistent state following a crash.

The reasons for this are as follows:

1. The implementation of this method is far less intrusive. In the fs layer we would have had to fight with the entire fs layout, page cache, behavior, block allocation logic, etc.
2. Our journalling protocol didn't really make use of the fact that we had journalled full checksum/parity blocks upon crash recovery. Rather, all it did was use their locations to figure out where it should recalculate the checksums/parity. This is effectively the same scheme as that presented in [2].
3. Without being aware of it from top to bottom the filesystem can't really do anything interesting to recovery from the fact that checksums don't match anyways. RAID on the other hand can use this data to rebuild a stripe similar to ZFS.

1 Introduction

This is the intro.

Disks fail.

Sometimes silently.

We want to detect it.

We'd even like to repair it.

? Why at the filesystem level ?

End-End ...

Here's a citation for ext4 [3].

We designed a system that introduces checksumming and parity into ext4/jbd2 at a stripe level.

2 Background

This is the background section.

3 Design Decisions

3.1 Problem Overview

We wish to add checksuming and parity features to ext4 such that we can tolerate a single block error. Checksums are necessary to determine when and which block has gone bad so that it can potentially be recovered from parity.

The summary of the major problem to overcome is that without data journalling mode (ie: ordered or writeback data journalling where only metadata is journalled), checksums and the datablocks they protect must be written separately to their fixed disk locations and due to the nature of the disk these updates are not atomic. Thus, in a model where we store a single checksum/parity block for an extent or more generally a group of blocks, no matter the order they're written, we can reach a state (eg: after a partial write crash or error) where the data and its checksum do not match up, but we cannot be bad aware of this after a crash. We solve this by storing two checksums on disk, the current and previous, and maintaining the invariant that each data block will at all times match one of these two checksums, else we must invoke a recovery from parity.

In general we will not address data journalling mode as it's somewhat a simplification of ordered journalling mode. That is, if everything is logged to the journal, we don't have any consistency problems.

First, let us address some basic design comparisons and questions.

3.2 Checksum Block Association

Given that we have two checksum/parity block versions to store, where should we store them?

Associating them with an extent is an incomplete solution since it does not handle all metadata. Additionally, it's an inefficient solution due to the fact that extent sizes can vary radically. Also, there is an inconsistent recovery guarantee in the extent model.

Thus, we arrive at the alternative of associating them based on a stripe size of N blocks, where N is specified by the user as the level of recovery guarantee that they wish to have at `mkfs` time. In this way we can ignore what the content in the "datablocks" actually is to a certain degree since the checksum/parity block pair is simply a protection over an individual stripe. It should be noted that since we pair checksum and parity blocks, the maximum value of N is restricted based upon the number of checksums we can store in a filesystem block. See sections 3.5 and 3.4 for further discussions of this topic.

3.3 Checksum Block Layout

In either association case, we need to consider where to store the checksum/parity block pairs.

For backwards compatibility with the current ext4 on disk structure we could consider storing them in a separate fixed location, much like the addition of a journal file was backwards compatible with ext2. However, we probably also wish to read and compare the checksum upon every read of a datablock, else how will we know when something has gone wrong and that we need to issue recovery. This last desire implies that we should store them close to the data to avoid extra long seeks during uncached read operations. Moreover, this behavior cooperates better with the internal caching and read-ahead behavior of the disk.

Thus, we originally came up with the following layout, in which blocks were organized as a stripe group with leading and trailing checksum/parity block pairs:

FIXME: Turn this into a figure.

```
+-----+-----+---+---+---+-----+-----+ | P_new
| C_new | D_1 | ...  | D_N | P_old | C_old |
+-----+-----+---+---+---+-----+-----+
```

In this model the beginning and end checksum/parity block pairs would alternate between which matched the in progress (new) and current (old) data. We now refer to this model as the LFS style layout since it bears some resemblance to LFS's checkpoint regions.

A challenge here is determining which block pair to write during a rewrite of data within the stripe. Upon first write we have no information, so at best we must guess and potentially have to read the next block, at worst we have to read both blocks and compare some generation number. Now, let's suppose we cache this data after that point.

In a 512GB drive (not especially large by today's standards), we have 131072000 4K fs blocks. Supposing a 16 block stripe size we have about 8192000 checksum/parity new/old pairs to be concerned with. We need at least 2 bits of information for each of these: one to denote if we've read it, another to denote which is the live pair. This results in about 2MB of cache for the entire filesystem. For a 16TB fs we'd need 64MB, which begins to look like an excess amount. **FIXME: This isn't nearly as bad as I originally thought. Perhaps a comparison with ZFS?**

Upon writes, we have two options in this model for the order in which we write datablocks and checksum blocks.

1. We could hook into the usual jbd2 journalling code and write all affected checksum blocks for the datablocks of a given transaction before the data blocks.
2. We could write the checksum blocks along with their corresponding datablocks, one stripe at a time.

The first involves a single pass over the disk in one direction with a number of short seeks followed by a wait and a long seek.

The second involves a series of short seeks and waits. In the LFS style layout outlined above it would also potentially involve backwards seeks in each of the short seek groups. One potential way of solving that particular issue is to change the stripe layout to store an extra checksum/parity block so that writes can always happen in a unidirectional fashion yet partial writes won't obliterate our old values. However, this uses extra space and still involves seeks for non full stripe writes. Further, if the disk reorders I/O operations we cannot guarantee correctness.

FIXME: Turn this into a figure.

```
+-----+-----+-----+-----+---+---+---+---+-----+-----+
| P_new | C_new | P_old | C_old | D_1
| D_2 | ...  | D_N | P_new | C_new |
+-----+-----+-----+-----+---+---+---+---+-----+-----+
```

In either case, it appears that we would spend the majority of our time waiting for the disk head to seek and settle rather than writing.

Thus we arrive at an alternative that journals new checksum/parity blocks in a special way before writing the datablocks associated with a transaction. In this model, for writes we do as follows:

1. As before, we hook into the usual jbd2 journalling code to collect and write all affected checksum blocks for the datablocks of a given transaction before the data blocks. However, we write them to a

fixed location (ie: journal) before the final destination and due so in such a way that we are confident of their validity but have not yet marked them as replayable.

2. We write the datablocks for the transaction.

3. At this point the datablock checksums should match the new checksum blocks that we journalled, so we commit the journal of the checksum/parity blocks (ie: mark it as replayable).

4. We proceed with the rest of the journalling operations at this point and at some point (potentially later) checkpoint everything to disk. If we don't crash at this point the correct checksums and data are at least still in page cache, so whether or not they've made it to their final location doesn't matter. In principle this step is really separate, but we like to include it just to understand how the whole process fits together.

Note that in data journalling mode all of these steps can go straight to the journal. Further note, that we have not addressed whether the metadata journal and checksum journal are separate. Conceptually we consider them to be so, however for implementation simplicity, we reuse and modify the existing jbd2 system.

During a read operation we do as follows:

1. Read the data.

2. Read the checksum.

3. Compare the two, and if necessary begin recovery procedures for that stripe. Recovery is effectively an XOR of the remain blocks in the stripe, provided they all check out as still valid. If that fails, we issue an error and remount-ro.

There are a number of advantages and a few disadvantages in this model.

1. Our write of the new checksums is sequential and contiguous. Thus, we've introduced only a single extra seek and wait rather than O(number of checksum blocks in the transaction). The number of seeks for datablocks is unchanged from a non-checksumming filesystem. However, the checksum journal is also potentially further away, which means a potentially long seek. The hope is that since ext4 buffers data a bit more aggressively that previous ext filesystems, and because it batches many datablock writes into a single transaction, then the number of checksum blocks affected is large enough to make better use of disk bandwidth compared to the seek times.

2. During normal running operations we no longer need to worry about which checksum is the current one. It's understood based on the location and journalling protocol semantics. Further, this potentially integrates better into the existing journalling code.

3. During crash recovery, based on the journal, we are aware of which stripes were potentially in the process of being written so we can deal with them immediately rather than waiting for them to be read again at some later time (possibly much later) as would have been the case in the LFS style layout. In that case we would not be able to tell the difference between a corrupted block due to bitrot and a partial rot and would typically have to fail the filesystem.

3.4 Pairing Checksum and Parity Blocks

The pairing of checksum and parity blocks allows us to deal with recovery intelligently and efficiently. **FIXME: discuss the complications of allowing checksums for N blocks and parity of M blocks.**

3.5 Checksum Block Design

Our current design of the checksum block includes a brief header that stores the block number of the checksum a checksum of the whole block including perhaps some of the metadata followed by a list of checksums whose index in the list denotes the datablock in the stripe it represents (including parity blocks).

Each of those entries is composed of a valid bit (if the block is unused we can avoid the overhead of calculating it's parity or checksums), an inode bit and 32-bit inode number for datablocks that are associated with an inode, a 6-bit checksum type index (eg: crc32, md5, sha512, etc.), and the checksum calculation up to 472-bits, rounding the entire entry out to 512-bits. We reserve this space ahead of time for flexibility and because being ultra space efficient really doesn't matter since the number of checksums we're able to store in a single fs block is typically much larger than the parity stripe size, which is determined at `mkfs` time by the user. Since the parity stripe size is a metric of recovery guarantee, the user is likely to make it much less than even 50. For comparison, if we assume a 4K fs block, and each entry is 512 bits, we can store 63 entries plus our header. For further comparison, consider that typical RAID5 arrays max out their number of spindles at 16. **FIXME: A better probabilistic analysis could be made here to motivate one stripe size versus the other and to make sure we're comparing apples to apples.**

We store the inode number in the checksum blocks as a potential aid for `fsck` operations to recover extra data, however we do not depend upon it.

An example checksum block: **FIXME: Turn this into a figure.**

```
+---------+-------------+ | Block
number | Block checksum | +---------+-----+--------+
| V | I | Inode | Type | Checksum |
+--+--+-----+----+--------+ | V | I | Inode | Type
| Checksum | +--+--+-----+----+--------+ | ...  |
+----------------------+
```

We included the block number in the checksum block in order to detect misdirected writes and a checksum over the block to detect errors in that block. Unfortunately, we still cannot detect phantom writes to the checksum block, though we can detect phantom writes to the datablocks it represents.

Alternatively, rather than storing a checksum for the checksum block in the block itself we could store them in a fixed location that would allow us to detect phantom and misdirected writes of checksum blocks. Updates could be done in a manner similar to ZFS. **FIXME: what exactly did we mean by this?** However, as discussed in the section 3.3, this would involve extra seeks and not make the best use of internal disk cache. We rejected the idea of storing the checksum of the checksum block in a nearby checksum block since that would introduce cascading I/O operations and a significant performance penalty.

Also, we could potentially store a hamming code like OCFS2 does for metadata [1], instead of just a checksum. This would allow us some us some basic per block recovery possibilities, up to a single bit error, so that we could possibly get closer to a state where we can use our parity block to recover a more damaged block (eg: one that couldn't be repaired from a hamming code). Another spin on this design is that we store checksums for each sector instead of an fs block. However, we can do no better in terms of recovery (from our parity block) in that case anyways, so it doesn't appear to matter, even though it's more likely that within the disk we fail at the granularity of a sector. In both of these cases, as the IRON filesystems paper [4] pointed out, sector and bit errors are likely to exhibit spatial locality, so having only a single bit flip in multiple sectors or blocks is unlikely, so the extra work and storage cost is of little value.

3.6 Discussion

One major problem with this is that it's a departure from the ext4 fs layout. **FIXME: Is there an excuse, argument, etc. for/against this?**

Another is that we can not recover from stripes that were partially written and crashed in the middle, but we can detect it, which is in itself an improvement over current ext4 semantics, and we can do so at remount time without a full `fsck`, which is an improvement over our original LFS style design. We gain this benefit by journalling checksum updates. Without this, in ordered journalling mode the journal doesn't yet contain the inodes that were being updated until the data has been fully written. Thus, we don't realize that's there's inconsistent data until we next read the blocks during normal operations and find that we have inconsistencies that we cannot recover from so we otherwise wouldn't have been able to just distrust and recalculate those checksums at remount time as we do. **FIXME: Did we say this already?**

FIXME: Says this in a less snarky way. We mostly ignored writeback mode because people who use that don't care about their data anyways, so we should prevent them from mounting it in that mode with checksums enabled anyways.

In this design, we believe that we are still able to make use of the performance advantages extents offer such as smaller metadata requirements and contiguous writes and reads. This is because when reads are issued they follow through the VFS layer to the filesystem **(FIXME: we should reference the code paths here. I just don't recall what they are at the moment.)** where we translate the logical block number to a sector which is then returned from page cache or issued as a read to the disk. In the latter case we still need to calculate and compare checksums so a larger read can still be issued without being wasted. However, since the page cache is indexed by both physical block number and file offset, and our checksums sit within these extents we must exclude the checksum/parity blocks within this structure from being returned to the user, which imposes an extra header to separate these structures in memory. We feel this is a reasonable sacrifice.

FIXME: @Remzi: Thoughts? Writes on the other hand mostly just involves some modulo arithmetic and a few changes to the block allocator to take them into consideration when finding contiguous regions for new extents. Another possibility is to change the semantics of how the fs blocks are indexed, which would change the layout of the bitmap. We have not yet fully explored either of these ideas. **TODO: Do that.**

4 Design

In this section we describe our final design by discussing the process of writing, reading, and recovery including more system call code path specifics.

4.1 Write Operation

First, let's consider a write operation.

1. When a system call is issuing a write, it must calculate the block number within a given extent subject to the in stripe parity/checksum blocks, then it grabs a handle from the current transaction of the in memory journal structure (by start_journal()), writes in page caches, dirties the affected page, files the affected inode to the current transaction's inode list, files the affected metadata block to the transaction's metadata list, then stops the journal.

2. At some point later (due to memory pressure, a commit timer, fsync call, etc.), the journal freezes (ie: stops accepting any new updates) and later commits.

3. During the journal committing process:

   (a) From the inode list in the transaction, we identify all the dependent data blocks.

   (b) From the data block numbers and the corresponding metadata block numbers in the metadata list we determine the checksum/parity block number using some modulo arithmetic and its new content including the inode number for the checksum block entry.

   Note, that here we are currently glossing over the specifics of how to update the checksum and parity blocks efficiently. We discuss a couple of different strategies we can potentially use in section 4.1.1.

   Rather than naively keeping the checksum and metadata transactions separate, an optimization here is to write the checksum/parity blocks so that they take into account the new metadata in the transaction right away as well. This has the advantage that in the most common case we are avoiding an extra set of I/Os to journal checksums for the metadata. As discussed in section 4.3, even if we crash between commiting the checksum and metadata we will be able to detect it and appropriately recalulate the necessary checksum/parity blocks.

   (c) Journal the checksum and parity blocks together. Mark the journal as ready but not checkpointable. See section 4.1.2 for a discussion on how we mark the journal as ready, but not checkpointable.

   (d) Write data block in place.

   (e) Journal the metadata block. Note that as in ext4, the last two steps can happen in parallel.

   (f) Mark both the metadata journal and checksum journal as checkpointable (replayable) atomically. Note that we must be able to write the commit block for our metadata and checksums atomically, else the checksums and metadata may not appear to be replayable at the same time, which will result in non-matching data upon a journal recovery. To do this we simply use a single commit block for both checksum and metadata transactions.

4. Journal checkpointing: write the checksum/parity and metadata blocks in place. If the storage system allows, this too could happen in parallel.

4.1.1 Efficient Parity and Checksum Calculations

During a write process for we previously ignored how to efficiently update the corresponding parity and checksum blocks.

For both the checksum and parity blocks, independently we can either update the block's data in the page cache or defer until write out time.

For heavily updated pages, it's probably better to avoid the overhead of checksum/parity recalculation upon each write call. However, if we do not do this immediately, then in the naive implementation, upon flush to disk we may need to recalculate parity from the other blocks in the stripe, which can involve more I/O to read them into memory and can be equally if not more expensive.

In a hybrid approach, at a write system call, we could take the difference of the page cache version and the user version and recalculate the parity block in the page cache immediately. This is a relatively inexpensive operation. [1] Both pages are marked as dirty. Checksums however, are not updated until the pages are flushed to disk. This imposes the restriction that upon a read system call, pages read from the cache cannot be compared with their checksums if they are marked as dirty, else they will not match. This has the potential of missing some memory errors, but other solutions such as ECC memory can be used to account for that issue. An alternative is that a mount time option affects whether or not we recalculate and recheck checksums for blocks upon every write and read respectively or only upon flush to and read from disk.

In another approach we could cache the original on disk version of each block until write out time. Given enough

[1] In our napkin math estimations we could do over 1000 of these operations in the time it would take for a single seek to occur.

cache memory this would allow us to avoid any calculations or extra I/O until it was time to flush to disk.

4.1.2 Validating Non-Checkpointable Checksum Journal Transactions

FIXME: I think this can already be done with existing ext4 journalling techniques, else the idea I have in mind is:

In order to mark checksum journal transactions as ready and valid, but not checkpointable we propose the following scheme:

1. Add a checksum in the journal header that is a checksum over all the blocks and the journal header in the transaction.
2. Since it contains a length field and we can check the checksum we know later on whether or not those blocks represent a full journal record (minus the commit block).
3. Later when we write the commit block it signifies that the journal is replayable.

This means one less I/O since we won't have to write the commit block twice or update anything else.

4.2 Read Operation

Next, let's consider a read operation.

1. The system call checks the page cache, if contents don't already exist in page cache, it calls into ext4_readpage() to start reading from disk.
2. Submit read request to disk and register an I/O completion function.
3. (a) When the read completes, it calls into the I/O completion function we setup in the last step which calculates its checksum and reads its corresponding checksum block, which could already be in the page cache. **FIXME: Verify this.** Since jbd2 doesn't mark pages as clean until the corresponding block has been checkpointed, then the block is either in the page cache or in its fixed disk location, but we don't need to check the journal.
   (b) In the case of reading the checksum block from disk we register a different I/O completion function that verifies the consistency of the checksum block itself. If the checksum block is not consistent there's not much we can currently do, so we issue an error and remount the filesystem read-only. Else, we continue to the next step.
4. If checksum for the datablock agrees, the read successfully completes, and we return the data. Else, we start recovery processes. If the recovery process completes successfully, we retry.

4.3 Recovery Operations

Next, let's consider recovery operations. There are actually two forms of recovery in our design: checksum mismatch and filesystem crash.

4.3.1 Checksum Mismatch Recovery

First, let's deal with the case of checksum mistmatches. Recall that this situation arrises during step 3a of the read process, further that we have already verified the validity of the checksum block based on its checksum.

1. Log an error so that the user knows that we detected a corrupt block.
2. Read in all the blocks, including the parity block, in the stripe group, and check their checksums. If more than one block fails to match, we cannot recover from a single parity block, so return an error and remount the filesystem read-only. Note, that for file data we could argue as others have [4] that remounting the filesystem read-only is perhaps an overly heavy handed approach, however we like the uniform treatment of this case. Further, it is consistent with current ext3/4 approaches.
3. To recover a single block failure we XOR all the good blocks, including the parity block, to get the content of the bad block.
4. Calculate the checksum of the recovered bad block. Check if it matches the stored checksum.
   (a) If it matches, rewrite the recovered block to disk. Recovery has succeeded and we can retry the previous read operation and log another message that we recovered the corrupt block we detected.
   (b) If it doesn't match, something really bad happened like a memory or I/O system, or programmer error. We should remount read-only.

4.3.2 Filesystem Crash Recovery

Next, we address the issue of recovery after a filesystem crash.

1. Check the journal:
   (a) If both the checksum journal and metadata journal are checkpointable, then checkpoint both journals.

   (b) If the checksum journal is marked ready (ie: valid) but not checkpointable, it indicates a partial write of either file data or the metadata journal.

   Based on the information we have about the checksum blocks, we can find the corresponding stripe groups.

   For each group, we calculate the checksum of each block in the group.

   If the datablocks do not as a group match either entirely match the original checksum block or entirely match the new one, then we have a partial write of the stripe that we cannot recover from since the data on disk will not as a whole match either the new or old parity blocks paired with those checksums.

   Thus, our only recourse is the recalculate the checksum/parity blocks based upon the data we currently have on disk. This is not different from the current ext3 or ext4 semantics except that we can notify the user of that situation. Additionally, since we stored inode numbers in the checksum blocks we can inform the user of the inodes involved, though a more intensive disk scan would be required to identify the file name(s) associated with that inode.

   Note that this partial write could happen within a block (eg: 3 out of 8 sectors new, the rest old data). However, the results are the same.

   To recalulate the checksum/parity block pairs, for each group, we must recalculate the checksum of each block within the group.

   Write the checksum block (w/ the appropriate checksum entries) and the new calculated parity block.

5 Implementation

This is the implementation section.

6 Evaluation

This is the evaluation section.

Here's a possible set of things to measure:

- Recovery correctness. **FIXME: How?**
- Parity/Checksum caclulation schemes (eg: at write() or at flush to disk).
- Stripe size performance affects.
- Checksum function performance affects.

7 Related Work

7.1 RAID

Our work is in some sense similar to RAID since we calculate parity in terms of stripes and are more or less data contents agnostic.

On the other hand, RAID is incapable of detecting per block bit rot, misdirected writes, or phantom writes. Furthermore, RAID is only able to detect data corruption or partial writes during a read (either in operation or scrubbing). Finally, typical RAID repair policies are to throw the entire disk with the faulty sector and rebuild the entire array.

In our model, we can avoid this and perform a single parity stripe repair operation. Also, since the checksums are journalled first, we can detect partial write failures (eg: due to a crash) at remount time as opposed to some lengthy period later. This is in fact the biggest argument for journalling the checksum/parity block pairs since without that our only recourse for a partial write crash would be a full `fsck`, which is at least as expensive as a RAID rebuild.

Also, our system can work with a RAID to make use of multiple disk spindles and bandwidth. Ideally, the checksum/parity stripe size in our journalling and buffering layers can be tuned to match the RAID level so that we can perform matching full stripe writes.

@Remzi: Since this is getting to be very similar to RAID and is becoming more of a general journaling system as opposed to just ext4 specific, it would be good to discuss the arguments for doing this in the fs/journal layer as opposed to the storage layer.

8 Conclusions

This is the conclusions section.

Acknowledgments

References

[1] J. Becker. Block-based error detection and correction for ocfs2. `http://oss.oracle.com/osswiki/OCFS2/DesignDocs/BlockErrorDetection`.

[2] T. Denehy, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Journal-guided resynchronization for software RAID. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies-Volume 4*, page 7. USENIX Association, 2005.

[3] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.

[4] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. Gunawi, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 206–220. ACM, 2005.