

## Re: [RFC] JBD ordered mode rewrite

---

*Source:* <http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-03/msg04010.html>

---

- *From:* Jan Kara <[jack@xxxxxxx](mailto:jack@xxxxxxx)>
  - *Date:* Mon, 10 Mar 2008 20:54:40 +0100
- 

On Fri 07-03-08 16:52:10, Andreas Dilger wrote:

On Mar 06, 2008 18:42 +0100, Jan Kara wrote:

Below is my rewrite of ordered mode in JBD. Now we don't have a list of data buffers that need syncing on transaction commit but a list of inodes that need writeout during commit. This brings all sorts of advantages such as possibility to get rid of journal heads and buffer heads for data buffers in ordered mode, better ordering of writes on transaction commit, simplification of some JBD code, no more anonymous pages when truncate of data being committed happens. The patch has survived some light testing but it still has some potential of eating your data so beware :) I've run dbench to see whether we didn't decrease performance by different handling of truncate and the throughput I'm getting on my machine is the same (OK, is lower by 0.5%) if I disable the code in truncate waiting for commit to finish... Also the throughput of dbench is about 2% better with my patch than with current JBD.  
Any comments or testing most welcome.

Looks like a very good patch – thanks for your effort in moving this beyond the "hand-waving" stage that it's been in for the past few years.

I'm looking at what implications this has for delayed allocation in ext4, because the vast majority of file data will be unmapped in that case and a journal commit in ordered mode will no longer cause the data to be flushed to disk.

I think is OK, because the pdflushd will now be totally in charge of flushing the dirty pages to disk, instead of this previously being done by ordered mode in the journal. I know there have been some bugs in this area in the past, but I guess it isn't much different than running in writeback mode. That said, I don't know how many users run in writeback mode unless they are running a database, and the database is doing a lot of explicit fsync of file data so there may still be bugs lurking...

Yes, they basically get guarantees as in writeback mode. As I wrote to Mingming, I'm not sure how that is handled with current ordered-mode

## Re: [RFC] JBD ordered mode rewrite

handling because you cannot afford to do block allocation from the commit code anyway. We could possibly do the allocation if we were sure that we have enough credits in the transaction (but that currently isn't the case since we really count the number of buffers dirtied on the account of transaction handle and after the handle is dropped, we give back unused credits to the transaction).

Some care is still needed here because with ext4 delayed allocation the common case will be unmapped buffers, while the ext3 implementation will only have this in very rare cases (unmapped mmap writes), but it looks like the right mechanism is already in place for both.

I'll just put a bunch of comments inline, not necessarily problems, just walking through the code to ensure I understand what is going on...

```
@@ -1675,7 +1675,8 @@ static int __block_write_full_page(struct inode
*inode, struct page *page,
*/
clear_buffer_dirty(bh);
set_buffer_uptodate(bh);
- } else if (!buffer_mapped(bh) && buffer_dirty(bh)) {
+ } else if (!buffer_mapped(bh) && buffer_dirty(bh)
+ && !wbc->skip_unmapped) {
```

(comment) This is skipping writeout of unmapped pages during journal commit – good, otherwise we would deadlock trying to add block mappings...

```
@@ -183,6 +184,8 @@ void ext3_delete_inode (struct inode * inode)
{
handle_t *handle;

+ if (ext3_should_order_data(inode))
+ ext3_begin_ordered_truncate(inode, 0);
```

(comment) This is flushing out pages allocated in the previous transaction to keep consistent semantics in case of a crash loses the delete... There is some possibility for optimization here – comments below at journal\_begin\_ordered\_truncate().

```
@@ -1487,46 +1462,49 @@ static int ext3_ordered_writepage(struct page
*page,
+ if (!wbc->skip_unmapped) {
+ handle = ext3_journal_start(inode,
+ ext3_writepage_trans_blocks(inode));
+ if (IS_ERR(handle)) {
```

Re: [RFC] JBD ordered mode rewrite

```
+ ret = PTR_ERR(handle);
+ goto out_fail;
+ }
}
```

I guess the common case here for delalloc is that if someone cares to fsync the file then that inode would be added to the journal list and the pages would be mapped here in the process context and flushed with the journal commit. This is ideal because it avoids syncing out all of the dirty pages for inodes that were NOT fsync'd and fixes the "one process doing fsync kills performance for streaming writes" problem in ordered mode that was recently discussed on the list.

We in fact win twice because fsync\_page\_range() will potentially only map and flush the range of pages that the application cares about and does not necessarily have to write out all pages (though that will still happen in ordered mode if the pages had previously been mapped).

```
+ else if (!PageMappedToDisk(page))
+ goto out_fail;
```

```
(style) } else if (...) {
goto out_fail;
}
```

```
+ /* This can go as soon as someone cares about that ;) */
if (!page_has_buffers(page)) {
create_empty_buffers(page, inode->i_sb->s_blocksize,
(1 << BH_Dirty)|(1 << BH_Uptodate));
}
```

Is there any reason to keep this in the non-data-journal case? Adding buffer heads to every page is a non-trivial amount of RAM usage.

No. But I think that it's better to do nobh ordered mode in a separate patch. This one is complicated enough...

```
+static int journal_submit_data_buffers(journal_t *journal,
+ transaction_t *commit_transaction)
{
+ /*
+ * We are in a committing transaction. Therefore no new inode
+ * can be added to our inode list. We use JI_COMMIT_RUNNING
+ * flag to protect inode we currently operate on from being
+ * released while we write out pages.
```

Re: [RFC] JBD ordered mode rewrite

```
+ */
```

Should we J\_ASSERT() this is true? Probably better to put this comment before the function instead of inside it.

Comment moved. How would you like to assert for inode being freed? Maybe we could check `jinode->i_transaction == commit_transaction`. That should be good enough.

```
+ spin_lock(&journal->j_list_lock);
+ list_for_each_entry(jinode, &commit_transaction->t_inode_list, i_list) {
+ mapping = jinode->i_vfs_inode->i_mapping;
+ if (!mapping_cap_writeback_dirty(mapping))
+ continue;
+ wbc.nr_to_write = mapping->npages * 2;
+ jinode->i_flags |= JI_COMMIT_RUNNING;
+ spin_unlock(&journal->j_list_lock);
+ err = do_writepages(jinode->i_vfs_inode->i_mapping, &wbc);
+ if (!ret)
+ ret = err;
+ spin_lock(&journal->j_list_lock);
+ jinode->i_flags &= ~JI_COMMIT_RUNNING;
+ wake_up_bit(&jinode->i_flags, __JI_COMMIT_RUNNING);
+ }
+ spin_unlock(&journal->j_list_lock);
```

Hmm, this is one area I'm a bit worried about. In the old code the number of buffers to sync for a transaction are bounded because they can only be added to the transaction while it is open. With this new inode-list code there could be a process busily adding new dirty + mapped pages to the inode(s) in the running transaction, while we are blocked here writing out the pages in `kjournald` trying to commit the previous transaction.

Correct me if I'm wrong but as far as I look into the write-out code, we do just one-sweep scan of mapping when writing out dirty pages. So we could possibly write out more than we need but at most the whole file. Oh, ok, someone could be extending the file as well but we could fix that by setting `.range_end` to the current `i_size` - that's actually a nice optimization anyway so I've done that.

At some point we will eventually no longer be able to add new pages to the running transaction (because it is full or was closed due to timeout) and we won't be able to start a new transaction because the committing transaction has not yet committed. At that point we would be able to finish the commit of the previous transaction, but not until we had reached the point of blocking all operations in the filesystem waiting for the new transaction to open. This would essentially lead to livelock of the system, with

Re: [RFC] JBD ordered mode rewrite

stop–start–stop cycles for the transactions and IO.

What would be needed is some kind of mechanism to separate the dirty pages on the inode into "dirty from current transaction" and "dirty from the previous transaction". That seems non–trivial, however. I guess we could also force any process doing writing to flush out all of the dirty and mapped pages on the inode if it detects the inode is in two transactions.

This would at least parallelize the IO submission and also throttle the addition of new pages until the dirty ones on the committing transaction had been flushed but may essentially mean sync IO performance for streaming writes on large files without delalloc (which would not add new pages that are mapped normally).

```
@@ -655,7 +565,14 @@ start_journal_io:
so we incur less scheduling load.
*/
```

```
– jbd_debug(3, "JBD: commit phase 4\n");
+ jbd_debug(3, "JBD: commit phase 3\n");
```

Rather than numbering these phases, which has little meaning and often means they just get renumbered like here, it is probably more useful to give them descriptive names like "JBD: commit wait for data buffers", etc.

Yes, but I guess that's for a separate patch. I'll add it to my todo :).

```
+/*
+ * File inode in the inode list of the handle's transaction
+ */
+int journal_file_inode(handle_t *handle, struct jbd_inode *jinode)
+{
+ transaction_t *transaction = handle->h_transaction;
+ journal_t *journal = transaction->t_journal;
+
+ if (is_handle_aborted(handle))
+ return 0;
```

Should we be returning an error back to the caller at this point?

Yes, of course.

```
+ jbd_debug(4, "Adding inode %lu, tid:%d\n", jinode->i_vfs_inode->i_ino,
+ transaction->t_tid);
+
+ spin_lock(&journal->j_list_lock);
```

## Re: [RFC] JBD ordered mode rewrite

```
+
+ if (jinode->i_transaction == transaction ||
+     jinode->i_next_transaction == transaction)
+ goto done;
```

Is it possible/safe to do the above check outside of the `j_list_lock` as an optimization? We will be calling this function for every page that is dirtied and it will likely be quite highly contended. It used to be that we HAD to get this lock because we were almost certainly adding the new buffers to the transaction list, but in this updated code the common case is that the inode will already be on the list...

Probably we could do that – there are two places where we remove inode from a transaction list (and these are the only ones interesting for races). One is the commit code and the other one is `journal_release_jbd_inode()`. We are safe from the second one because we obviously hold a reference to the inode. The first one is more subtle but if `jinode->i_transaction == transaction`, it is filed on the running transaction's list from which we have handle and so commit code cannot touch it. If `jinode->i_next_transaction == transaction`, then commit code can change inode under us but it will file the inode to the running transaction's list anyway so there's nothing to lose.

Changed the code and added a big comment...

```
+/*
+ * This function must be called when inode is journaled in ordered mode
+ * before truncation happens. It starts writeout of truncated part in
+ * case it is in the committing transaction so that we stand to ordered
+ * mode consistency guarantees.
+ */
+int journal_begin_ordered_truncate(struct jbd_inode *inode, loff_t
new_size)
+{
+ journal_t *journal;
+ transaction_t *commit_trans;
+ int ret = 0;
+
+ if (!inode->i_transaction && !inode->i_next_transaction)
+ goto out;
+ journal = inode->i_transaction->t_journal;
+ spin_lock(&journal->j_state_lock);
+ commit_trans = journal->j_committing_transaction;
+ spin_unlock(&journal->j_state_lock);
+ if (inode->i_transaction == commit_trans) {
+ ret = __filemap_fdatawrite_range(inode->i_vfs_inode->i_mapping,
+ new_size, LLONG_MAX, WB_SYNC_ALL);
+ if (ret)
+ journal_abort(journal, ret);
+ }
```

Re: [RFC] JBD ordered mode rewrite

Is there any way here to entirely avoid writing blocks which were just allocated during the current transaction (e.g. temp files during compile or whatever)? One possibility is to store the transaction number when the inode was created (or first dirtied?) in `ext3_inode_info` and if that is equal to the committing transaction then we don't need to write anything at all?

Note that we write out blocks only if we truncate inode that is being currently committed. I.e., it has been created in the previous transaction and is truncated in the current one. Or did you mean something else?

Honza

--

Jan Kara <jack@xxxxxxx>  
SUSE Labs, CR

--

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to [majordomo@xxxxxxxxxxxxxxxx](mailto:majordomo@xxxxxxxxxxxxxxxx)  
More majordomo info at <http://vger.kernel.org/majordomo-info.html>  
Please read the FAQ at <http://www.tux.org/lkml/>