Meta-data Write-back Cache

6th March 2008

Started: 2007.12.18

Nikita Danilov `Nikita.Danilov@Sun.COM`

Contents

1 Introduction

Write-back cache is a client and server mechanism for caching meta-data operations in a manner similar to already existing cache for data operations (*i.e.*, a client data cache). Perceived advantages of write-back cache are:

⊢ more efficient network usage, due to the batched transfer of meta-data operations;

⊢ higher degree of concurrency on the client;

⊢ an ability, given suitable locking mechanism (*e.g.*, sub-tree locking), for client to operate without communicating to the server (including disconnected mode);

⊢ lower latency, particularly on wide area networks;

⊢ more efficient operations execution on the server.

Refer to the architecture page listed in the *References* section for the additional information.

2 Definitions

block a chunk of file data (data block) or representation of meta-data state updates (meta-data block). In the pre-existing implementation blocks are identical with pages.

co-dependency A situation in which an MD operation modifies multiple separate pieces of client state that are otherwise not related. These dependent pieces of state have to be reintegrated atomically (in the data-base ACID sense). For example:

⊢ link and unlink introduce a co-dependency between the directory where the entry is added or removed, and the target object whose nlink count is updated.

⊢ cross-directory rename makes the parent directories dependent.

- ⊢ unlinking the last name of a file introduces a co-dependency between the file inode and the inodes of its stripe objects that are to be destroyed.

Note: co-dependencies are bidirectional. While it is possible to have directed dependencies (*e.g.*, new inode has to be created before its name is inserted into directory, but not necessary visa versa), our locking and transactional models make these finer distinctions useless in practice.

coordinated reintegration A special case of reintegration that occurs when the client cache contains dependent state pertaining to multiple servers. In this case the servers have to act in concert to guarantee consistency. Coordinated reintegration is originated by the client, that sends dependent batches to the servers in parallel. Global file system consistency is maintained through use of *epochs*, which are described in another set of design specification, see [3], and Epoch HLD/DLD (forthcoming).

epoch (definition copied from Epochs Architecture page) a collection of state updates over multiple nodes of a cluster, that is guaranteed when applied to transfer a file system from one consistent state to another. Ultimate sources of epochs are:

- ⊢ a WBC client: for a single WBC client epoch boundaries can be as fine grained as a single operation;
- ⊢ a single meta-data server: when MDS executes operations on behalf of non-WBC clients, in packs state updates into epochs defined by the transactions of the underlying local file system;
- ⊢ a cluster of meta-data servers: when non-WBC client issues meta-data operation against the cluster of meta-data servers, this operation is automatically included into an epoch defined by the Cuts algorithm running on the cluster;

file system object An entity that has a persistent state, stored on a certain server, and possibly a cached state stored on other nodes; that can be updated individually (in the sense of transactions of the local file system), and locked individually (in the sense of DLM): a regular file, a stripe, a non-striped directory, a stripe of a striped directory.

md batch A group of MD state updates performed by a client such that:

- ⊢ the batch as a whole transforms the file system from one consistent state to another,
- ⊢ no other client depends on seeing the file system in any state where some, but not all of the MD operations in the batch are in effect.

md operation A meta-data operation performed on a client (create, unlink, rename, etc.).

reintegration The process of applying an MD batch on a server. Reintegration executes all the MD operations in the batch and changes the file system from one consistent state to another.

state update An effect of an MD operation on a particular file system object. Single operation can update state of multiple objects (*e.g.*, link updates state of a directory, where name is added to, and of an object to which new link is added). Cached state update is also referred to as a *cache element*.

wbc write-back cache.

3 Requirements

The following list of requirements is taken from the WBC architecture page.

scalability client should be able to execute 32K creations of 1–64KB files per second. Files maybe created in different directories with file counts per directory to range from 1K to 100K.

correctness reintegration changes the file system from one globally consistent state to another. Recovery leads to well-defined and consistent state.

transactionality reintegration assures that the disk image of the file system is consistent. This implies that reintegration is either done completely within a single transaction, or the batch contains enough information to cut reintegration into smaller pieces, each preserving consistency.

concurrency when a client surrenders a meta-data lock it only flushes enough of its cache to guarantee correctness (i.e., flushing the whole meta-data cache is not necessary).

resource leasing resources are leased to a client efficiently and correctly. Grants prevent spurious ENOSPC errors, and fid sequences prevent failure to create an object during reintegration.

3.1 Note

To keep the design manageable, this document tries to introduce as few external dependencies as possible. Specifically, while WBC performance critically depends on the sub-tree locking, no details of the latter are discussed here. Also, details of epochs implementation are, being the subject of another design document (forthcoming), left out,

4 Functional specification

4.1 General

When a client is invoked to execute meta-data or data operation, it checks whether WBC is applicable (see *Conditions* below). If it is, the client modifies local VFS and VM objects to reflect effects of the operation, and adds an operation record (see *Caching* sub-section below) to the cache. If WBC is not applicable, the client sends RPC to the server(s) involved (this might incur cache write-out, see *Write-back* below).

4.2 Caching

The meta-data cache is organized as a collection of per-object logs. Here an object is a file or a directory on which meta-data operation is performed, and a log is a sequence of operation records, in the order of their execution history (this is a well-defined order, as meta-data operations on a single object are serialized by the VFS locks). An operation record `R0` can be co-dependent with another operation record `R1`, (where usually, but not necessary, `R0` and `R1` belong to the logs for the different objects), meaning that `R1` has to be reintegrated together with `R0` (see *Batching* below). In terms of epochs, it can be stated that operations are dependent when they are part of the same epoch. Individual records and their co-dependencies are tracked so that client is able to write-out only a portion of its cache, while preserving file system integrity: if partial write-out is not required, then consistency can be trivially guaranteed by always reintegrating the whole cache. Partial cache write-out is necessary to:

- ⊢ achieve low latency of the DLM lock cancellation, and
- ⊢ have an ability to cache more meta-data operations than can be reintegrated in a single RPC. This is important, for example, for the meta-data proxy server.

4.3 Conditions

WBC might not be applicable for a given operation due to a number of reasons:

- ⊢ WBC is disabled by the administrator (*e.g.*, because exhaustive audit is required);
- ⊢ an amount of the cached state reached certain tunable threshold;
- ⊢ a synchronous operation is requested by the user (sync, fsync, O_SYNC);

- ⊢ an operation cannot be executed locally (*e.g.*, client lacks necessary locks on some of the objects involved in the operation) without communication with the server, or client doesn't have enough information to complete operation locally;
- ⊢ client runs out of a leased resource (fids, grants, locks, *etc.*). This is an important special case of the previous situation.

4.4 Write-back

Cache write-back occurs when

- ⊢ the client cannot proceed without contacting the server (see *Conditions* above), and policy wants to piggy-back cache write-back to the RPC being sent, or
- ⊢ the cache contains operations older than some configurable age, or
- ⊢ a lock protecting a part of the cache is invalidated either due to the arrival of the blocking AST from the server, or due to the lock LRU policy, or
- ⊢ write-back is explicitly requested by the user (*e.g.*, umount, or preparations for going into disconnected mode), or
- ⊢ write-back is implicitly requested by the environment (*e.g.*, in the form of memory pressure callback from VM subsystem).

Cache write-back starts with the construction of a batch (see *Batching* below). The batch consists of per-server sub-batches, each targeted at the specific server (mdt or ost). Each per-server sub-batch is a description of data and meta-data updates that are to be effected on the target server. The batch is converted into the ptlrpc format and transmitted to the servers as a set of RPCs.

Before transmission is started, client waits until the *maximal RPC in flight* limitation is satisfied for all servers involved in the write-out. With the current recovery design, the maximal number of RPCs in flight is necessary equal to 1.

Transmission of the sub-batch, followed by its execution on the server, and possibly followed by the recovery is referred to as a *reintegration*. Either all or none of the per-server sub-batches have to be reintegrated successfully. This is achieved through the epoch mechanism.

4.5 Batching

Construction of a batch starts with a certain "root set" of cache elements. In the case of age-based cache write-out, root set contains certain amount of oldest elements in the cache; in the case of blocking AST root set consists of objects for which conflicting lock is requested, *etc.* The batch then is iteratively expanded until it is closed w.r.t. the following properties:

- ⊢ local sequentiality: together with any operation, the batch also contains all preceding, not yet integrated, operations on the same object;
- ⊢ co-dependency: together with any operation, the batch also contains all operations (on this or other objects) co-dependent with a given operation. For a definition of co-dependency, see *Definitions* section.

If resulting closure is too small to form an efficient RPC, more elements (selected according to certain policy) are added to the batch and construction of the closure repeats.

4.6 Resource leasing

4.6.1 Grants

A client indicates that it uses

4.6.2 Fid sequences

4.6.3 Locking, sub-tree locks

WBC client caches meta-data locks, granted by servers. Compare this with:

- ⊢ non-WBC client, that never caches PW meta-data locks, and instead relies on server to take locks on its behalf, and
- ⊢ caching extent locks for data.

4.6.4 Local lookups

An immediate and important consequence of lock caching is that WBC client can also safely and efficiently cache directory blocks, containing (name, fid) pairs. As Lustre readdir indexes directory blocks by a hash or name, such caching entails an ability to do local lookups without going to the server: to check for the presence or absence of a directory entry with a name “foo” client has to check whether its local cache contains data block for HASH(“foo”), and if it does, whether this block contains entry with the desired name. If block is missing it has to be fetched from the server.

Cached directory blocks has to be kept consistent with local updates (e.g., when file is created locally, a record has to be made in the corresponding data block).

4.6.5 Sub-tree locks

Sub-tree lock (STL) is a special type of meta-data lock that delegates to a client an ownership of the whole sub-tree, rooted at the object for which STL was granted. See STL design documentation for more details.

4.7 Security

4.7.1 Partial write-out

The question of how much cached state has to be written out to cancel a meta-data lock is a subtle one. Consider a case, where client holds a `PW` lock (not necessary a sub-tree lock) on directory `/DIR`, and a `PW` lock on directory `/DIR/d0/d`. In seems plausible to assume that cancellation of the lock on `/DIR` invalidates only cached state associated with `/DIR` and shouldn't cause write-out of cached updates on `/DIR/d0/d`, but consider the following scenario

```
$ rm /DIR/d0/d/secret # destroy sensitive information
$ chmod go+rx /DIR    # allow everyone to access sub-tree
```

If cancellation of `/DIR` lock at this point writes out only updates to `/DIR` (including change of the permission bits), no immediate problem arise, as nobody can access `d` yet without first canceling lock on it, and this cancellation will cause write-out of cached updates, including removal of directory entry for `secret` file. Potential problem arises if client crashes after writing out state updates for `/DIR`, but with cached updates for `d` still in the cache: this creates security hole. It can be noted that

⊢ such situation is already possible in non-transactional file systems with asynchronous meta-data updates, like ext2, where meta-data updates reach the storage out of order, and crash can occur at any moment;

⊢ also such situation is possible with UFS+softupdates—default FreeBSD file system configuration;

⊢ also such situation is possible with the file systems that allow multiple concurrent transactions (*e.g.*, NTFS, reiser4);

⊢ generally speaking, the problem is due to the out-of-order reintegration that breaks sequentiality which appears to break causality (*i.e.*, some later event happened, while some earlier one didn't). One possible solution is to introduce some weaker notion of sequentiality. *E.g.*, to postulate that all operations executed by a single *thread* or process have to be reintegrated in their causal order, without enforcing total ordering of all operations. This can be easily achieved by adding co-dependencies between state updates of all operations issues by a given thread or process. The drawbacks of this are

- it is insufficient in some cases, *e.g.*, when threads or processes synchronize their execution through the non-file-system means (*e.g.*, by using interprocess communication)—in such cases users might expect that operations executed by independent processes will always be executed in order;
- this solution doesn't address the case of meta-data proxy server, where unbounded amount of state updates can be (and has to be) cached;

⊢ [Nathan's proposal] it seems to be enough to enforce full write-out only when access restrictions are weakened. This observation allows to avoid full write-outs most of the time, but it doesn't address meta-data proxy server case;

⊢ a notion of *strong consistency* can be introduced. Let's say that a file system is strongly consistent when in addition to being consistent (see [3] for a definition of a file system consistency), it contains effects of all operations executed by every client up to certain moment in the local time of this client. *Strong epoch* is then defined as a sequence of epochs transferring file system from one strongly consistent state to another. For example, any strong epoch containing state updates for `chmod a+r /DIR` also contains updates for `rm /DIR/d0/d/secret`. Changing batching algorithm to always produce batches for strong epochs closes the security hole in question;

⊢ allow partial write-out, but keep track on the server of where points of strong consistency (in addition to the points of usual consistency, defined by the epoch boundaries) are. If the client fails before communicating whole strong epoch to the server, the latter rolls back all epochs that are part of the incomplete strong epoch. That might cause roll back of epochs from other clients (that depends on epochs being rolled back), and eviction of clients that obtained locks on the objects in rolled back epochs (that process is done iteratively).

Reasonable solution for this is to specify consistency guarantees through a mount option (procfs tunables, ioctl() invocation, *etc.*), and to support two modes:

⊢ total write-out, where invalidation of any meta-data lock causes whole cached state to be written out, and

⊢ careful: compose a batch according to normal co-dependency rules, but once a *suspicious* record is added to the batch, also add to it all the records preceding in the global node time, or at least all preceding operations in the sub-tree rooted at the object affected by the suspicious record. Here suspicious record is one that changes name-space visibility: relaxation of permissions on a file or a directory, cross-directory rename, creation of a hard-link, *etc*.

⊢ relaxed: ignore the issue completely, construct batches based solely on co-dependencies. This is compatible with ext2, UFS+softupdates, NTFS.

Let's look at the symmetrical case:

```
$ chmod go-rx /DIR
$ touch /DIR/d0/d/secret
```

By the similar reasoning it is obvious that SETATTR (resulting from chmod(2)) has to be reintegrated not later than ADD_NAME("secret"), otherwise the secret file might become visible as a result of a client crash. Solution to this problem (in addition to total write-out, and ignoring the issue) is nicely dual too: in the case of permission relaxation all the *earlier* records in the sub-tree are written out together with the record relaxing permissions. In the case of permissions tightening, a record tightening permissions if written out together with any *later* record in the sub-tree.

4.7.2 Capabilities

They are by definition not cacheable (for a long time). How to generate them locally?

4.8 Quota

5 Use cases

The following list of use cases is taken from the WBC architecture page.

5.1 Sub-tree-operations

A client creates a new sub-directory and populates it with a large number of small files (and sub-directories, recursively).

⊢ when new directory is created, server grants the client a sub-tree lock for it (subject to some server policy);

⊢ creation of a new object F within the directory:

- client checks VFS permissions
- client selects an mds where F is to reside (using usual placement policy);
- client checks that it is able to execute operation locally that it:
  - has an unallocated fid in fid sequences;
  - has enough inode grants from (maybe a part of the previous check);
  - has enough inode quota;
  - can extent existing sub-tree lock to the F;
- once all checks above passed, client creates an inode and attaches it to the dentry, created by VFS;
- client maintaining per-mount-point operation counter (*epoch number*, in other terms) that is used to tag update-records;
- client creates update-records:
  - a NAME_ADD record. It is appended to the per-directory operation log;
  - a FILE_CREATE record. It is appended to the per-inode operation log (necessary empty at that point);

  There is a co-dependency between the NAME_ADD record to the FILE_CREATE record (implemented as a pointer), guaranteeing that creation of the file (*i.e.*, of inode on the mds) is reintegrated not later than the insertion of the name referring to this file.

⊢ if one of the checks mentioned above fails due to the shortage of the leased resources, or addition of a new state update record pushes cache size to the limit (either global or per-object), synchronous cache write-out takes place, following the description given in the *Write-back* sub-section:

- a root set is selected according to certain policy (*e.g.*, selecting oldest elements in the cache seems one possible reasonable policy);
- closure of the root set is built. Data structures, described below in the *Logic Specification* section make this step efficient.

5.2 Sub-tree-conflict

A client C0 enters directory /DIR and populates it with a large number of files (and sub-directories, recursively). Another client C1 obtains a conflicting lock on /DIR.

⊢ when new directory is created, server grants the client a sub-tree lock for it (subject to some server policy);

⊢ C0 then proceeds to create new files and sub-directories as described in the previous use case;

⊢ after some time C0 ends up with the following cached state:

- a number of ADD_NAME state update records attached to the top-level directory;
- a number of inodes, representing newly created files. State-update logs for these inodes contain at least FILE_CREATE records, but may also create other records, *e.g.*, for SETATTR, ADD_NLINK, *etc*. There are co-dependency pointers between records in the object logs and records in the logs of their parent directories;
- a number of osc-level data structures (struct lov_oinfo, or struct osc_object in the new client code) representing stripe sub-objects of newly created files. There is a per-object update log, containing at least OBJECT_CREATE operation (see note above), possibly followed by SETATTR and TRUNCATE records.
- a number of osc-level data structures (struct osc_async_page, or struct osc_page in the new client code) representing blocks with the cached file data, attached to the osc objects. An implicit co-dependency pointer between a cached data block and corresponding OBJECT_CREATE record is assumed;
- a set of DLM locks protecting cached file data. These locks were implicitly granted by the server together with fid sequence that it allocated to this client at the connect time;
- capabilities;
- cached grants:
  - disk space grants from OST, consumed by the dirty data blocks;
  - disk space grants from MDT, consumed by the directory entries;
  - inode grants from OST, implicitly given out together with the fid sequences;
  - inode grants from MDT, implicitly given out together with the fid sequences;
  - OST quota
  - MDT quota;
- all update records are tagged with an epoch number;

⊢ all objects with non-empty logs are kept on some list hanging off mount-point.

- ⊢ `C1` does `GETATTR(/DIR)` (*e.g.*, as a part of “`ls -l /`”). Supposing that `C1` is a non-WBC client, server acquires `LCK_PR` lock on `/DIR` on C1's behalf, sending blocking AST to `C0`.
- ⊢ client can handle blocking AST for an STL in multiple ways. Let's for simplicity assume that the client decides to completely cancel STL lock on `/DIR`, which implies write-out of all cached state protected by this lock.
- ⊢ client forms a batch (as a collection of sub-batches, one for each server involved):
  - batch has to contain at least the modifications to objects that were protected by the lock being cancelled. Client maintains lock<->object mapping to find such objects efficiently;
  - starting from such root set, batch is built as described in the *Batching* section. Construction of the closure might require addition of update records from logs other objects not protected by the lock being cancelled. In such a situation, client is free to include only a part of object's update log in the batch.

5.3 Undo

Client creates new sub-directory, populates it with some number of files, and then removes them all.

- ⊢ file creation proceeds as described in the Sub-tree-operations use case.
- ⊢ unlink() results in two state update records: DELETE_NAME record in the parent directory log, and FILE_DESTROY record in the file log (if nlink drops to 0 in the result of unlink). Before adding record to the log, log is scanned to check whether new record can be coalesced with existing one. *E.g.*, DELETE_NAME(“foo”) record coalesces with the last preceding ADD_NAME(“foo”) in the log, resulting in SETATTR(MTIME) record. Similarly, pair of records

  ```
  RENAME(“foo”, “bar”), ..., DELETE_NAME(“bar”)
  ```

  coalesces into DELETE_NAME(“foo”), and pair of

  ```
  SETATTR(ATIME), ...  SETATTR(ATIME|CTIME)
  ```

  records coalesces into SETATTR(ATIME|CTIME). Coalescing can cause more coalescing further down the log;

⊢ when a record R is removed from the log due to coalescing, following is done:

  – records dependent on R, become dependent on the record preceding R in the log. If R is the first record in the log, these co-dependencies are removed;

⊢ when record R is moved in the log due to coalescing, co-dependencies are also updated. At this point co-dependency relation graph might change significantly. Such modifications of co-dependency graph don't change epoch boundaries, because all graph nodes involved are already part of the same epoch.

⊢ if no reintegration happened during creation phase, then removal of all files would result in the empty logs, except for the SETATTR(MTIME | ATIME | CTIME) record for the top-level directory;

⊢ if reintegration happened during creation phase, then all records for file creation, not integrated when removal phase started are undone. Records for removals of files whose creation is already reintegrated are sent to the server(s).

5.4 Data-consistency

Client executes data and meta-data operations on existing files, when conflicting lock on some data is requested by other client.

Data cache for stripes is grafted to the meta-data name-space tree: for the purpose of a subtree, as used in Partial write-out subsection, stripe sub-objects are immediate descendants of the corresponding file object, and cached data blocks are immediate descendants of the corresponding stripe objects. Such an arrangement guarantees that normal batch building rules (including ones described in Security sections) are sufficient to maintain the same level of consistency as for meta-data.

5.5 Unlink

Client removes a number of (not hard-linked) files.

After all removal operations are completed locally, meta-data cache contains

⊢ DELETE_NAME records in the parent directory log;

⊢ FILE_DESTROY records in the file logs;

⊢ STRIPE_DESTROY records in the stripe logs;

FILE_DESTROY records are not sent to the MDS, because the server knows when last link to the file is removed and will destroy object automatically. Reintegration of DELETE_NAME's to the MDS'es and STRIPE_DESTROY's to the OST's is an example of coordinated reintegration. Client sends sub-batches, tagged with epoch numbers, to the MD and OS servers in parallel, and keeps RPC's in memory until batch is committed on every participating server.

5.6 Recovery

Client performs a number of MD operations. (A) Sends batch to the server. (B) Server executes batch. (C) Client gets reply. (D) Server commits batch. (E) Client gets commit notification. (F). Server crashes at either A, B, C, D, E, or F.

See Epoch design documents for detailed description of recovery use cases.

5.7 Co-dependency, rename, CMD-rename

Client performs MD operation, involving more than one object (link, unlink, etc.). Lock protecting of the objects involved is revoked.

Specifically rename, which is a special case of the same use case in which dependency is manifestly bi-directional: both parent directories depend on each other.

Even more specifically, a situation where client renames file across directories, located on different MD servers. Lock, protecting one of these directories, is revoked.

Batching logic guarantees that file system consistency is preserved in this case. See Sub-tree-conflict use case for a description.

6 Logic specification

6.1 Data structures:

⊢ state update record: represents state update for a file system object. Fields:

- an epoch number;
- a pointer to the object this record is updating state for;
- an amount of space record consumes in the RPC (both message and bulk);
- a linkage into per-object log;
- a linkage to the dependent operations;

  - optionally (for debugging) a list of records that are dependent on this one;
  - a linkage into per-mount-point sequential log of all operations, ordered by time. This is necessary to keep track of strong epochs;
  - a pointer to operation vector. This includes operations to
    - check whether this record commutes with a given one;
    - check whether this record can be coalesced with a given one, and if so, what is the resulting record;
    - add record to the RPC
    - destructor: called when record is destroyed by generic caching code;
  - record specific state, *e.g.*, attributes for SETATTR, file name for NAME_ADD, *etc*. Some state is generic, for example, any record mutating object state has an mtime value as one of its attributes;

⊢ record log: an ordered list of records, attached to particular file system object (directory or file). Can be implemented as a simple double-linked list, protected by a spin-lock which is used to protect fields of all records on the log.

⊢ batch: a sequence of records selected for reintegration. Batch has per-server sub-batches, allocated lazily as records are added to the batch.

6.2 Caching

Meta-data caching is implemented as a new layer `wbcc` in the client md-stack, replacing or augmenting mdc. Changes to llite and other modules are necessary to get rid of the assumptions pre-existing code makes about synchronous RPC processing.

6.3 Batching

Batch construction starts from a certain root set. Batching is by its very nature a single-threaded process: it doesn't make sense to build two batches concurrently (unless these concurrent processes can discover each other and cooperate to build a single consistent batch). Algorithms below assume (and implementation should check) that only one batch is constructed at a time.

```
batch_build(root_set) {
        list batch; /* batch of records being constructed */
        batch = root_set;
        do {
                more1 = batch_build_sequential_closure(batch);
                more2 = batch_build_dependency_closure(batch);
                more3 = batch_build_subtree_up(batch);
                more4 = batch_build_subtree_down(batch);
        } while (more1 || more2);
}
int batch_build_sequential_closure(batch) {
        foreach(record in batch) {
                lock(record->log);
                /* move all records preceding @record in the log
                   into @batch, and remove them from the log. */
                list_splice(record->log_linkage.prev, batch);
                unlock(record->log);
        }
        return nr_moved > 0;
}
int batch_build_dependency_closure(batch) {
        foreach(record in batch) {
                lock(record->log);
                foreach(deprec in record->dependency_list_linkage) {
                        LASSERT(record->log != deprec->log);
                        /* to lock both logs in a dead-lock safe manner
                           standard try-and-repeat lock sequence is
                           used. */
                        try_repeat_lock(deprec->log);
                        if (!(deprec in batch))
                                list_add(deprec, batch);
                        unlock(deprec->log);
                }
                unlock(record->log);
        }
        return nr_added > 0;
}
int batch_build_subtree_up(batch) {
        foreach(record in batch) {
                /* this implies a loop over _all_ parents for a
                   hard-linked file */
                for (uprec = record->parent; uprec != NULL;
                     uprec = uprec->parent) {
                        if (rec_tightens_permissions(uprec) &&
                            record->time >= uprec->time))
                                list_add(uprec, batch);
                }
        }
        return nr_added > 0;
}
```

```
int batch_build_subtree_down(batch) {
        foreach(record in batch) {
                if (rec_relaxes_permissions(record)) {
                        foreach (downrec in subtree at record) {
                                if (record->time >= downprec->time)
                                        list_add(downrec, batch);
                        }
                }
        }
        return nr_added > 0;
}
```

Note: pseudo-code above only shows the logic of the execution and outline of locking model. Actual implementation will be optimized in the obvious ways (*e.g.*, to not take and release the same lock over and over again).

6.4 Reintegration

6.4.1 client side

Once batch is built, an RPC is constructed from it, and sent to the server.

```
wbcc_rpc_build(subbatch) {
        struct ptlrpc_request *req;
        int nr_pages;
        size_t mssg_size = 0;
        size_t bulk_size = 0;
        cfs_page_t **pages;
        struct lu_epoch_header *head;
        foreach(record in subbatch) {
                /* check for possible merging here... */
                mssg_size += record->mssg_size;
                bulk_size += record->bulk_size;
        }
        nr_pages = bulk_size >> CFS_PAGE_SHIFT;
        pages = alloc(nr_pages, sizeof pages[0]);
        for (i = 0; i < nr_pages; ++i)
                pages[i] = alloc_page();
        req = ptlrpc_prep_req_pool(..., MDS_REINT, ..., mssg_size);
        desc = ptlrpc_prep_bulk_imp(req, nr_pages, BULK_GET_SOURCE,
                                    MDS_BULK_PORTAL);
        head = lustre_msg_buf(req->rq_reqmsg, ...);
        /* fill in head: epoch number, other servers containing parts of
           this epoch, etc. */
```

```
                /* a buffer for inline parts of records (if any) */
                buf = lustre_msg_buf(req->rq_reqmsg, ...);
                foreach(record in subbatch) {
                        record->ops.put_in_rpc(record, buf, address);
                        /* update buf (pointing into message) and address
                           (pointing into page), kmap()/kunmap(), ... */
                }
                for (i = 0; i < nr_pages; ++i)
                        ptlrpc_prep_bulk_page(desc, pages[i], ???, CFS_PAGE_SIZE)
                wbcc_announce_cached(mssg_size + bulk_size);
                req->rq_interpret_reply = wbcc_interpret_reint;
                ptlrpcd_add_req(req);
        }
```

Locks cannot be released until reply is received from the server (modulo NRS reordering). Details of recovery are encapsulated inside of PTLRPC layer and described in Epochs design documentation

6.4.2 server side

⊢ when batch is received, store information from its header (including epoch number and a list of other servers containing parts of this epoch) into special epoch llog hanging off LAST_RCVD file entry;

⊢ then start bulk transfers and apply batch records one by one as blocks arrive. This can be done my multiple threads concurrently, as updates in the batch are not conflicting;

⊢ server assumes (and checks) that locks, on the objects involved into reintegration records are held by the client, and doesn't attempt to acquire locks by itself.

⊢ whenever record is applied, write undo record into llog as part of the same transaction;

⊢ on recovery after a server failure, scan LAST_RCVD file, and all epoch llogs, and for every not globally committed epoch contact every server mentioned in the epoch header, to initiate epoch recovery process (see Epochs design documentation for details);

7 State management

7.1 State invariants

⊢ cached meta-data update is protected by a DLM lock;

- ⊢ batch is a union of full epochs;

7.2 Scalability & performance

WBC is supposed to improve scalability for the most work-loads. Batched reintegration reduces number of messages communicated over network, further reducing interrupt and context-switch activity on the servers and clients. Meta-data caching allows operations to be executed locally, reducing system call latency.

Possible scenarios where WBC can degrade performance are

- ⊢ high contention for meta-data. In this case extra RPCs are necessary to cancel locks cached on the client;
- ⊢ frequent forced meta-data synchronization, *e.g.*, fsync-intensive workload. In this case overhead of maintaining cache can be important. Also, sending small meta-data updates through bulks is sub-optimal, but this can be optimized out;

Scalability on a single client is achieved by using per-object operation logs, protected by separate locks.

7.3 Recovery changes

Recovery is changed significantly, see Epochs design documents for details.

7.4 Locking changes

Locking is changed significantly, see Sub-tree Locks design documents for details.

7.5 Disk format changes

Disk format is changed (undo logs, epochs tracking, *etc.*), see Epochs design documentation for details.

7.6 Wire format changes

7.7 Protocol changes

New rpc type for bulk meta-data updates is introduced.

7.8 API changes

Caching API is introduces. See Logical Specification.

7.9 RPCs order changes

RPC ordering requirements are changed, see Epochs design documents for details.

8 Alternatives

Assorted list

- [Eric Barton] instead of having separate data-structure for operation log, keep cached meta-data updates in bulk RPC blocks from the very beginning. Advantage: avoids extra copy during RPC construction. Disadvantage: makes space management more difficult;
- [CMD2 WBC] keep cached meta-data operations as in a global log, and send them as a sequence of RPCs. Advantage: simplicity, server can check validity of meta-data updates. Disadvantage: less efficient.

9 Focus for inspections

- partial write-out;
- error handling during reintegration;

10 References

1. Architecture page: Write Back Cache `http://arch.lustre.org/index.php?title=Write_Back_Cache`
2. bugzilla 14170: METADATA WBC basic functionality `https://bugzilla.lustre.org/show_bug.cgi?id=14170`
3. Architecture page: Epochs `http://arch.lustre.org/index.php?title=Epochs`