Metadata Stat-ahead DLD

Lai Siyao <lsy@clusterfs.com>

2007.03.26

1 Introduction

This document describes metadata stat-ahead, which is a part of metadata improvements. The client will perform metadata stat-ahead when it detects readdir and sequential stat of dir entries therein.

2 Functional specification

2.1 struct md_statahead_info

This structure contains all stat-ahead related informations:

```
struct md_statahead_info {
        struct inode         *sai_inode;     /* inode */
        int                   sai_max;       /* max ahead of lookup, dynamic */
        atomic_t              sai_ahead;     /* count ahead of lookup*/
        atomic_t              sai_count;     /* total count */
        atomic_t              sai_hit;       /* hit count */
        atomic_t              sai_miss;      /* miss count */
        unsigned              sai_ls_all:1;  /* flag indicates `ls -al` is executed */
        struct ptlrpc_thread  sai_thread;    /* stat-ahead thread */
        void                 *sai_cb;        /* callback for async enqueue */
};
```

This structure is part of ll_inode_info, and is allocated for dir inode when client start to do stat-ahead:

```
struct ll_inode_info {
        ...
        pid_t                     lli_opendir_pid;  /* opendir pid */
        struct md_statahead_info *lli_sai;          /* md_statahead_info */
};
```

2.2 struct mdc_enqueue_args

Similar to struct osc_enqueue_args, to support MDC asynchronous enqueue lock, this structure is introduced:

```
struct mdc_enqueue_args {
        struct obd_export        *ma_exp;
        struct obd_enqueue_info  *ma_ei;
        struct lookup_intent     *ma_it;
        struct lustre_handle      ma_lockh;
        struct mdc_op_data       *ma_data;
        struct it_cb_data        *ma_icbd;
};
```

2.3 int ll_statahead_enter(struct inode parent, struct dentry *dentry)

It will start stat-ahead thread if needed, else it will check whether stat-ahead has fetched results back, if so, return 0.

2.4 int is_first_dirent(struct inode *dir, struct dentry *dentry)

Check whether dentry is the first dirent under dir. Its return value might be:

- LS_NONE_FIRST_DE: not the first dirent.
- LS_FIRST_DE: the first dirent, and whose name doesn't start with ".", as means this is not a hidden dirent.
- LS_PARENT_DE: dirent is "..", which implies user issues 'ls -al'.

Note: actually "." is always the first dirent of a directory, but because applications tend to do stat("."), we won't start stat-ahead for ".".

2.5 int ll_statahead_exit(struct dentry *dentry, int result)

This will be called in ll_lookup_it()/ll_revalidate_it(), it will update sai_hit/miss/ahead statistics.

if result == 1, which means hit, otherwise miss.

2.6 int ll_statahead_thread(void *arg)

stat-ahead thread, read dir entries in a loop, and stat them by turns.

2.7 int ll_statahead_one(struct dentry *parent, struct md_statahead_info *sai, struct ext2_dirent *de)

stat-ahead for one dir entry.

2.8 int ll_statahead_interpret(struct obd_export *exp, struct ptlrpc_request *req, struct lookup_intent *it, struct it_cb_data *icbd, int rc)

callback function for MDC asynchronous enqueue. It will finish lookup(prepare inode and dentry for stat-ahead dirent).

2.9 int mdc_statahead(struct obd_export *exp, struct mdc_op_data *op_data, struct lookup_intent *it, ldlm_blocking_callback cb_blocking, struct it_cb_data *icbd, struct md_statahead_info *sai)

Similar to mdc_enqueue(), but enqueue lock asynchronously. The enqueue request will be sent by ptlrpcd.

2.10 int mdc_statahead_interpret(struct ptlrpc_request *req, void *unused, int rc)

This is the callback function for request sent by ptlrpcd: req->rq_interpret_reply. After handling lock issues it will call ll_statahead_interpret() to finish lookup.

3 Use cases

3.1 user issues ‘ls -l‘

1. the process of ‘ls -l‘ open(“.”), and in ll_file_open() ll_inode_info.lli_opendir_pid is set to current->pid.
2. it then calls getdents(), which finally calls ll_readdir().
3. stat(“file1”) is called, which calls ll_lookup_it(), it will call ll_statahead_enter().
4. firstly it finds ll_inode_info.lli_sai is NULL, as means stat-ahead thread is not started yet. It then check the pid, and finds current pid is the same as the one open dir lately, besides, it is also the first dir entry of this dir. So it starts the stat-ahead thread ’ll_sa’, and then returns.
5. ll_lookup_it() continues with lookup for “file1” as before.

6. meanwhile 'll_sa' thread starts to stat from the second dir entry.
7. 'll_sa' thread keeps stating the remaining dir entries, until it is md_statahead_info.sai_max(dynamic, 3 from the starting, the maximum value is 50) ahead of the 'ls -l' process.
8. the next stat("file2") is called by 'ls -l' process, it calls ll_lookup_it() and then ll_statahead_enter().
9. firstly it finds an 'll_sa' thread is running, and if stated count is 0, as means 'll_sa' thread hasn't got any result yet, it will queue itself in the waitqueue of 'll_sa' thread.
10. once 'll_sa' thread gets some reply, it will wake up all processes on its waitqueue.
11. the stat("file2") is woken up, once the "file2" dentry->d_inode != NULL, it returns 1, otherwise 0. (This is to avoid double-stat, see below Logic specfication)
12. if ll_statahead_enter() for "file2" return 1, it means stat-ahead thread has fill with inode informations in dentry, and it will return from ll_lookup_it() immediately. Otherwise it will continue ll_lookup_it() as before.
13. stat("file2") will continue with ll_getattr_it().
14. step 8 is repeated until all dir enties have been stated.
15. 'll_sa' thread will exit when it hits the end of dir.
16. dir is closed, the lli_opendir_pid is cleared.

3.2 acceptance test

3.2.1 performance examination

1. create 1 million empty files under a lustre directory.
2. clear client cached locks.
3. disable stat-ahead, and run 'ls -l', record the time used.
4. clear client cached locks.
5. enable stat-ahead, and run 'ls -l', compare the time with the last result, it should be at least 50% less.
6. create 1, 5, 10, 100, 1000, 10000, 100000 empty files, and redo step 1-4, the time used should be 0% ~ 50% less.

3.2.2 sanity test

1. ‘ls -al‘ will do stat-ahead successfully.
2. run several ‘ls -l‘ simultaneously, the performance shouldn’t become worse.
3. run dbench, the performance shouldn’t become worse.

4 Logic specification

4.1 opendir pid registration

In ll_file_open(), if current inode is a dir, current pid is saved in ll_inode_info.lli_opendir_pid. And in contrary, lli_opendir_pid is cleared in ll_file_release().

4.2 ll_statahead_enter

```
int ll_statahead_enter(struct inode *parent, struct dentry *dentry)
{
        if (lli->lli_sai) {
                /* avoid double-stat */
                wait_event(sai->sai_thread.t_waitq,
                                atomic_read(&sai->sai_ahead) > 0 ||
                                sai->sai_thread.t_flags & SVC_STOPPED);
                return (dentry->d_inode != NULL);
        }

        /* pid check */
        if (lli->lli_opendir_pid != current->pid)
                return -EBADF;

        /* first dirent? */
        rc = is_first_dirent(parent, dentry);
        if (!rc)
                return -EBADF;
        if (rc == LS_PARENT_DE)
                sai->sai_ls_all = 1;

        /* start stat-ahead thread */
        kernel_thread(ll_statahead_thread, dentry->d_parent, 0);

        return 0;
}
```

In ll_lookup_it(), if IT_GETATTR, calls ll_statahead_enter(). Once this function returns 0, it will return current dentry back, otherwise it will lookup as before.

In ll_revalidate_it(), if IT_GETATTR, calls ll_statahead_enter(), but don't check the return value, because ll_revalidate_it() validates a dentry anyway.

4.3 ll_statahead_exit

```
void ll_statahead_exit(struct dentry *dentry, int result)
{
        spin_lock(&lli->lli_lock);
        if (lli->lli_sai) {
                if (result == 1) {
                        atomic_inc(&sai->sai_hit);
                        if (sai->sai_max < MD_STATAHEAD_MAX)
                                sai->sai_max = min(2 * sai->sai_max, MD_STATAHEAD_MAX);
                } else {
                        atomic_inc(&sai->sai_miss);
                        if (sai->sai_max > MD_STATAHEAD_MIN)
                                sai->sai_max = max(sai->sai_max / 2, MD_STATAHEAD_MIN);
                }
                if (atomic_read(&sai->sai_ahead) > 0)
                        atomic_dec(&sai->sai_ahead);
                wake_up(&sai->sai_thread->t_ctl_waitq);
        }
        spin_unlock(&lli->lli_lock);
}
```

This function will be called in ll_lookup_it() and ll_revalidate_it(), for ll_lookup_it() ***result*** is the return value of ll_statahead_enter(), and for ll_revalidate_it() is mdc_intent_lock(), 1 means it's found in local cache.

BTW, the sai_max is adjusted here, it's MD_STATAHEAD_MIN by default, and will increase till MD_STATAHEAD_MAX.

Also the sai_ahead is decreased here to push stat-ahead move forward, but note the lookup/revalidate function might be called several times for one dentry, so this value might be decreased more times than expected, however we will try to avoid it below zero.

4.4 ll_statahead_thread

```
int ll_statahead_thread(void *arg)
{
        struct dentry *parent = dget(arg);
        struct inode *inode = parent->d_inode;

        cfs_daemonize("ll_sa");

        while (1) {
                l_wait_event(thread->t_ctl_waitq,
```

```
                                        ll_sa_check_stop() || !ll_sa_full(),
                                        &lwi);

                        if (ll_sa_check_stop())
                                break;

                        for (; index < npages && !ll_sa_full(); index++, offset = 0) {
                                page = ll_get_dir_page(inode, index);
                                for (; (char *)de <= limit; de = ext2_next_entry(de)) {
                                        if (!de->inode)
                                                continue;
                                        if (de->d_name[0] == ".” && !sai->sai_ls_all)
                                                continue;
                                        rc = ll_statahead_one(parent, sai, de);
                                        if (rc)
                                                break;
                                 }
                                offset = de - kaddr;
                                ext2_put_page(page);
                                if (rc)
                                        goto out;
                                else if ((char *)de <= limit)
                                        break;   // ll_sa_full(), goto outer loop
                        }
                }
        out:
                return 0;
        }
```

If ***sai_ls_all*** is not set, we don't do stat-ahead for hidden dirent(the name of which start with ".").

4.5 ll_statahead_one

```
int ll_statahead_one(struct dentry *parent, struct md_statahead_info *sai, struct ext2_dirent *de
{
        struct inode *dir = parent->d_inode;
        intent_init(&it, IT_GETATTR);
        it.it_op_release = ll_intent_release;

        dirent2qstr(&name, de);
        down(&dir->i_sem);
        dentry = d_lookup(parent, &name);
        if (!dentry) {
                dentry = d_alloc(parent, &name);
                rc = do_sa_lookup(dir, dentry, &it);
                up(&dir->i_sem);
                return rc;
```

```
        }
        up(&dir->i_sem);

        rc = do_sa_revalidate(dentry, &it);
        return rc;
}
```

Note dir->i_sem is down before do_sa_lookup(), and since stat-ahead is done asynchronously, do_sa_lookup() will return soon, it won't hold this semaphore for long. However in ll_statahead_interpret() this semaphore will be used to protect the part of code which prepares inode and dentry after reply.

4.6 do_sa_lookup

```
int do_sa_lookup(struct dentry *parent, struct dentry *dentry, struct lookup_intent *it)
{
        rc = ll_prepare_mdc_op_data(&op_data, parent, NULL, dentry->name,
                                                    dentry->name_len, 0);
        if (rc)
                return rc;

        rc = mdc_statahead(exp, &op_data, it, ll_mdc_blocking_ast, sai);
        return rc;
}
```

4.7 do_sa_revalidate

```
int do_sa_revalidate(struct dentry *dentry, struct lookup_intent *it)
{
        if (dentry->d_inode == NULL)
                return 0;
        if (d_mountpoint(dentry))
                return 0;

         /* don't stat-ahead once lock cached already */
         rc = mdc_revalidate_lock(exp, it, &fid);
         if (rc == 1)
                 return 0;

         /* just do lookup by fid */
         rc = ll_prepare_mdc_op_data(&op_data, parent, NULL, dentry->name,
                                                    dentry->name_len, 0);
        if (rc)
                return rc;

        rc = mdc_statahead(exp, &op_data, it, ll_mdc_blocking_ast, sai);
```

```
        return rc;
}
```

4.8 ll_statahead_interpret

```
int ll_statahead_interpret(struct obd_export *exp, struct ptlrpc_request *req,
                                    struct it_cb_data *icbd, int rc)
{
        struct inode *parent = icbd->icbd_parent;
        struct dentry *dentry = *icbd->icbd_childp;
        if (rc)
                goto out;

        down(&parent->i_sem);
        rc = lookup_it_finish(req, DLM_REPLY_REC_OFF, it, icbd);
        up(&parent->i_sem);
        if (rc != 0)
                goto out;

        ll_lookup_finish_locks(it, dentry);
        atomic_inc(&sai->sai_ahead);
out:
       ptlrpc_req_finished(req);
       iput(parent);
       dput(dentry);
       ll_intent_release(it);
       OBD_FREE_PTR(it);
       OBD_FREE_PTR(icbd);
       /* wake up all processes which are waiting to avoid double-stat */
       wake_up(&sai->sai_thread.t_ctl_waitq);
       return rc;
}
```

Intent is released right after preparing inode and dentry for stat-ahead, thus request will be freed here.

4.9 mdc_statahead

```
int mdc_statahead(struct obd_export *exp, struct mdc_op_data *op_data,
                            struct lookup_intent *it, ldlm_blocking_callback cb_blocking,
                            md_statahead_cb_t cb_statahead, struct md_statahead_info *sai)
{
        req = mdc_intent_getattr_pack(exp, it, op_data);

        mdc_enter_request(&obddev->u.cli);
        rc = ldlm_cli_enqueue(exp, &req, res_id, lock_type, &policy, lock_mode,
```

```
                                        &flags, cb_blocking, ldlm_completion_ast, NULL,
                                        cb_data, NULL, 0, NULL, lockh, 1);
        aa = (struct mdc_enqueue_args *)&req->rq_async_args;
        aa->ma_exp = exp;
        aa->ma_sai = md_sai_get(sai);
        req->rq_interpret_reply = mdc_statahead_interpret;
        ptlrpcd_add_req(req);
        return 0;
}
```

Though stat-ahead requests are sent asynchronously, it should be limited by max_rpcs_in_flight, so before doing enqueue, mdc_enter_request() is called, and mdc_exit_request() will be called in mdc_statahead_interpret().

4.10 mdc_statahead_interpret

```
int mdc_statahead_interpret(struct ptlrpc_request *req, void *unused, int rc)
{
        ma = (struct mdc_enqueue_args *)&req->rq_async_args;
        mdc_exit_request(&obddev->u.cli);

        rc = ldlm_cli_enqueue_fini(exp, req, ei->ei_type, 1, ei->ei_mode, ei->ei_flags,
                                                NULL, 0, NULL, &ma->ma_lockh, rc);
        rc = mdc_finish_enqueue(exp, it, lock_mode, lockh, req, rc);
        if (rc)
                goto out;
        rc = mdc_finish_intent_lock(exp, ma->ma_it, ma->ma_data, &ma->ma_lockh, req);
out:
        sai->sai_cb(exp, req, rc);
        return 0;
}
```

Two functions are listed here: mdc_finish_enqueue() and mdc_finish_intent_lock(), actually they are not new code, but the part to finish enqueue and lock, which is part of current code of mdc_enqueue() and mdc_intent_lock().

5 State management

5.1 stat-ahead lookup intent

Stat-ahead lookup intent will be released right after ll_statahead_interpret(), as is different from ll_lookup_it() and ll_revalidate_it(), for the latter two cases, lookup/revalidate might be called several times for one dentry.

6 Lock

ll_inode_info.lli_sai will be protected by ll_inode_info.lli_lock, and the stat-ahead thread and each on wire stat-ahead RPC need hold refcount of md_statahead_info.sai_refc. The last one who put this refcount will free ll_inode_info.lli_sai.

7 Compatibility

Metadata-statahead needs be ported to both 1.6 and CMD. Because most code is in llite layer, there are only two issues to handle:

1. Add m_statahead function to md_ops, and add lmv_statahead().
2. CMD uses lu_dirent instead of ext2_dirent, so all the code related with dirent needs modification to support it.

8 Alternatives

N/A

9 Focus for inspections

- Can we do stat-ahead for lustre root?