```
**********************************************************
   80627 Tue Apr 23 15:35:05 2013
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
3743 zfs needs a refcount audit
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
int
dsl_dataset_hold_obj(dsl_pool_t *dp, uint64_t dsobj, void *tag,
    dsl_dataset_t **dsp)
{
        objset_t *mos = dp->dp_meta_objset;
        dmu_buf_t *dbuf;
        dsl_dataset_t *ds;
        int err;
        dmu_object_info_t doi;

        ASSERT(dsl_pool_config_held(dp));

        err = dmu_bonus_hold(mos, dsobj, tag, &dbuf);
        if (err != 0)
                return (err);

        /* Make sure dsobj has the correct object type. */
        dmu_object_info_from_db(dbuf, &doi);
        if (doi.doi_type != DMU_OT_DSL_DATASET) {
                dmu_buf_rele(dbuf, tag);
        if (doi.doi_type != DMU_OT_DSL_DATASET)
                return (SET_ERROR(EINVAL));
        }
#endif /* ! codereview */

        ds = dmu_buf_get_user(dbuf);
        if (ds == NULL) {
                dsl_dataset_t *winner = NULL;

                ds = kmem_zalloc(sizeof (dsl_dataset_t), KM_SLEEP);
                ds->ds_dbuf = dbuf;
                ds->ds_object = dsobj;
                ds->ds_phys = dbuf->db_data;

                mutex_init(&ds->ds_lock, NULL, MUTEX_DEFAULT, NULL);
                mutex_init(&ds->ds_opening_lock, NULL, MUTEX_DEFAULT, NULL);
                mutex_init(&ds->ds_sendstream_lock, NULL, MUTEX_DEFAULT, NULL);
                refcount_create(&ds->ds_longholds);

                bplist_create(&ds->ds_pending_deadlist);
                dsl_deadlist_open(&ds->ds_deadlist,
                    mos, ds->ds_phys->ds_deadlist_obj);

                list_create(&ds->ds_sendstreams, sizeof (dmu_sendarg_t),
                    offsetof(dmu_sendarg_t, dsa_link));

                if (err == 0) {
                        err = dsl_dir_hold_obj(dp,
                            ds->ds_phys->ds_dir_obj, NULL, ds, &ds->ds_dir);
                }
                if (err != 0) {
                        mutex_destroy(&ds->ds_lock);
                        mutex_destroy(&ds->ds_opening_lock);
                        refcount_destroy(&ds->ds_longholds);
                        bplist_destroy(&ds->ds_pending_deadlist);
                        dsl_deadlist_close(&ds->ds_deadlist);
```

```
                        kmem_free(ds, sizeof (dsl_dataset_t));
                        dmu_buf_rele(dbuf, tag);
                        return (err);
                }

                if (!dsl_dataset_is_snapshot(ds)) {
                        ds->ds_snapname[0] = '\0';
                        if (ds->ds_phys->ds_prev_snap_obj != 0) {
                                err = dsl_dataset_hold_obj(dp,
                                    ds->ds_phys->ds_prev_snap_obj,
                                    ds, &ds->ds_prev);
                        }
                } else {
                        if (zfs_flags & ZFS_DEBUG_SNAPNAMES)
                                err = dsl_dataset_get_snapname(ds);
                        if (err == 0 && ds->ds_phys->ds_userrefs_obj != 0) {
                                err = zap_count(
                                    ds->ds_dir->dd_pool->dp_meta_objset,
                                    ds->ds_phys->ds_userrefs_obj,
                                    &ds->ds_userrefs);
                        }
                }

                if (err == 0 && !dsl_dataset_is_snapshot(ds)) {
                        err = dsl_prop_get_int_ds(ds,
                            zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
                            &ds->ds_reserved);
                        if (err == 0) {
                                err = dsl_prop_get_int_ds(ds,
                                    zfs_prop_to_name(ZFS_PROP_REFQUOTA),
                                    &ds->ds_quota);
                        }
                } else {
                        ds->ds_reserved = ds->ds_quota = 0;
                }

                if (err != 0 || (winner = dmu_buf_set_user_ie(dbuf, ds,
                    &ds->ds_phys, dsl_dataset_evict)) != NULL) {
                        bplist_destroy(&ds->ds_pending_deadlist);
                        dsl_deadlist_close(&ds->ds_deadlist);
                        if (ds->ds_prev)
                                dsl_dataset_rele(ds->ds_prev, ds);
                        dsl_dir_rele(ds->ds_dir, ds);
                        mutex_destroy(&ds->ds_lock);
                        mutex_destroy(&ds->ds_opening_lock);
                        refcount_destroy(&ds->ds_longholds);
                        kmem_free(ds, sizeof (dsl_dataset_t));
                        if (err != 0) {
                                dmu_buf_rele(dbuf, tag);
                                return (err);
                        }
                        ds = winner;
                } else {
                        ds->ds_fsid_guid =
                            unique_insert(ds->ds_phys->ds_fsid_guid);
                }
        }
        ASSERT3P(ds->ds_dbuf, ==, dbuf);
        ASSERT3P(ds->ds_phys, ==, dbuf->db_data);
        ASSERT(ds->ds_phys->ds_prev_snap_obj != 0 ||
            spa_version(dp->dp_spa) < SPA_VERSION_ORIGIN ||
            dp->dp_origin_snap == NULL || ds == dp->dp_origin_snap);
        *dsp = ds;
        return (0);
}
```

```
int
dsl_dataset_hold(dsl_pool_t *dp, const char *name,
    void *tag, dsl_dataset_t **dsp)
{
        dsl_dir_t *dd;
        const char *snapname;
        uint64_t obj;
        int err = 0;

        err = dsl_dir_hold(dp, name, FTAG, &dd, &snapname);
        if (err != 0)
                return (err);

        ASSERT(dsl_pool_config_held(dp));
        obj = dd->dd_phys->dd_head_dataset_obj;
        if (obj != 0)
                err = dsl_dataset_hold_obj(dp, obj, tag, dsp);
        else
                err = SET_ERROR(ENOENT);

        /* we may be looking for a snapshot */
        if (err == 0 && snapname != NULL) {
                dsl_dataset_t *ds;

                if (*snapname++ != ’@’) {
                        dsl_dataset_rele(*dsp, tag);
                        dsl_dir_rele(dd, FTAG);
                        return (SET_ERROR(ENOENT));
                }

                dprintf("looking for snapshot ’%s’\n", snapname);
                err = dsl_dataset_snap_lookup(*dsp, snapname, &obj);
                if (err == 0)
                        err = dsl_dataset_hold_obj(dp, obj, tag, &ds);
                dsl_dataset_rele(*dsp, tag);

                if (err == 0) {
                        mutex_enter(&ds->ds_lock);
                        if (ds->ds_snapname[0] == 0)
                                (void) strlcpy(ds->ds_snapname, snapname,
                                    sizeof (ds->ds_snapname));
                        mutex_exit(&ds->ds_lock);
                        *dsp = ds;
                }
        }

        dsl_dir_rele(dd, FTAG);
        return (err);
}

int
dsl_dataset_own_obj(dsl_pool_t *dp, uint64_t dsobj,
    void *tag, dsl_dataset_t **dsp)
{
        int err = dsl_dataset_hold_obj(dp, dsobj, tag, dsp);
        if (err != 0)
                return (err);
        if (!dsl_dataset_tryown(*dsp, tag)) {
                dsl_dataset_rele(*dsp, tag);
                *dsp = NULL;
                return (SET_ERROR(EBUSY));
        }
        return (0);
}

int
```

```
dsl_dataset_own(dsl_pool_t *dp, const char *name,
    void *tag, dsl_dataset_t **dsp)
{
        int err = dsl_dataset_hold(dp, name, tag, dsp);
        if (err != 0)
                return (err);
        if (!dsl_dataset_tryown(*dsp, tag)) {
                dsl_dataset_rele(*dsp, tag);
                return (SET_ERROR(EBUSY));
        }
        return (0);
}

/*
 * See the comment above dsl_pool_hold() for details.  In summary, a long
 * hold is used to prevent destruction of a dataset while the pool hold
 * is dropped, allowing other concurrent operations (e.g. spa_sync()).
 *
 * The dataset and pool must be held when this function is called.  After it
 * is called, the pool hold may be released while the dataset is still held
 * and accessed.
 */
void
dsl_dataset_long_hold(dsl_dataset_t *ds, void *tag)
{
        ASSERT(dsl_pool_config_held(ds->ds_dir->dd_pool));
        (void) refcount_add(&ds->ds_longholds, tag);
}

void
dsl_dataset_long_rele(dsl_dataset_t *ds, void *tag)
{
        (void) refcount_remove(&ds->ds_longholds, tag);
}

/* Return B_TRUE if there are any long holds on this dataset. */
boolean_t
dsl_dataset_long_held(dsl_dataset_t *ds)
{
        return (!refcount_is_zero(&ds->ds_longholds));
}

void
dsl_dataset_name(dsl_dataset_t *ds, char *name)
{
        if (ds == NULL) {
                (void) strcpy(name, "mos");
        } else {
                dsl_dir_name(ds->ds_dir, name);
                VERIFY0(dsl_dataset_get_snapname(ds));
                if (ds->ds_snapname[0]) {
                        (void) strcat(name, "@");
                        /*
                         * We use a "recursive" mutex so that we
                         * can call dprintf_ds() with ds_lock held.
                         */
                        if (!MUTEX_HELD(&ds->ds_lock)) {
                                mutex_enter(&ds->ds_lock);
                                (void) strcat(name, ds->ds_snapname);
                                mutex_exit(&ds->ds_lock);
                        } else {
                                (void) strcat(name, ds->ds_snapname);
                        }
                }
        }
}
```

```
static int
dsl_dataset_namelen(dsl_dataset_t *ds)
{
        int result;

        if (ds == NULL) {
                result = 3;     /* "mos" */
        } else {
                result = dsl_dir_namelen(ds->ds_dir);
                VERIFY0(dsl_dataset_get_snapname(ds));
                if (ds->ds_snapname[0]) {
                        ++result;       /* adding one for the @-sign */
                        if (!MUTEX_HELD(&ds->ds_lock)) {
                                mutex_enter(&ds->ds_lock);
                                result += strlen(ds->ds_snapname);
                                mutex_exit(&ds->ds_lock);
                        } else {
                                result += strlen(ds->ds_snapname);
                        }
                }
        }

        return (result);
}

void
dsl_dataset_rele(dsl_dataset_t *ds, void *tag)
{
        dmu_buf_rele(ds->ds_dbuf, tag);
}

void
dsl_dataset_disown(dsl_dataset_t *ds, void *tag)
{
        ASSERT(ds->ds_owner == tag && ds->ds_dbuf != NULL);

        mutex_enter(&ds->ds_lock);
        ds->ds_owner = NULL;
        mutex_exit(&ds->ds_lock);
        dsl_dataset_long_rele(ds, tag);
        if (ds->ds_dbuf != NULL)
                dsl_dataset_rele(ds, tag);
        else
                dsl_dataset_evict(NULL, ds);
}

boolean_t
dsl_dataset_tryown(dsl_dataset_t *ds, void *tag)
{
        boolean_t gotit = FALSE;

        mutex_enter(&ds->ds_lock);
        if (ds->ds_owner == NULL && !DS_IS_INCONSISTENT(ds)) {
                ds->ds_owner = tag;
                dsl_dataset_long_hold(ds, tag);
                gotit = TRUE;
        }
        mutex_exit(&ds->ds_lock);
        return (gotit);
}

uint64_t
dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
    uint64_t flags, dmu_tx_t *tx)
{
```

```
        dsl_pool_t *dp = dd->dd_pool;
        dmu_buf_t *dbuf;
        dsl_dataset_phys_t *dsphys;
        uint64_t dsobj;
        objset_t *mos = dp->dp_meta_objset;

        if (origin == NULL)
                origin = dp->dp_origin_snap;

        ASSERT(origin == NULL || origin->ds_dir->dd_pool == dp);
        ASSERT(origin == NULL || origin->ds_phys->ds_num_children > 0);
        ASSERT(dmu_tx_is_syncing(tx));
        ASSERT(dd->dd_phys->dd_head_dataset_obj == 0);

        dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
            DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
        VERIFY0(dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
        dmu_buf_will_dirty(dbuf, tx);
        dsphys = dbuf->db_data;
        bzero(dsphys, sizeof (dsl_dataset_phys_t));
        dsphys->ds_dir_obj = dd->dd_object;
        dsphys->ds_flags = flags;
        dsphys->ds_fsid_guid = unique_create();
        (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
            sizeof (dsphys->ds_guid));
        dsphys->ds_snapnames_zapobj =
            zap_create_norm(mos, U8_TEXTPREP_TOUPPER, DMU_OT_DSL_DS_SNAP_MAP,
            DMU_OT_NONE, 0, tx);
        dsphys->ds_creation_time = gethrestime_sec();
        dsphys->ds_creation_txg = tx->tx_txg == TXG_INITIAL ? 1 : tx->tx_txg;

        if (origin == NULL) {
                dsphys->ds_deadlist_obj = dsl_deadlist_alloc(mos, tx);
        } else {
                dsl_dataset_t *ohds; /* head of the origin snapshot */

                dsphys->ds_prev_snap_obj = origin->ds_object;
                dsphys->ds_prev_snap_txg =
                    origin->ds_phys->ds_creation_txg;
                dsphys->ds_referenced_bytes =
                    origin->ds_phys->ds_referenced_bytes;
                dsphys->ds_compressed_bytes =
                    origin->ds_phys->ds_compressed_bytes;
                dsphys->ds_uncompressed_bytes =
                    origin->ds_phys->ds_uncompressed_bytes;
                dsphys->ds_bp = origin->ds_phys->ds_bp;
                dsphys->ds_flags |= origin->ds_phys->ds_flags;

                dmu_buf_will_dirty(origin->ds_dbuf, tx);
                origin->ds_phys->ds_num_children++;

                VERIFY0(dsl_dataset_hold_obj(dp,
                    origin->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &ohds));
                dsphys->ds_deadlist_obj = dsl_deadlist_clone(&ohds->ds_deadlist,
                    dsphys->ds_prev_snap_txg, dsphys->ds_prev_snap_obj, tx);
                dsl_dataset_rele(ohds, FTAG);

                if (spa_version(dp->dp_spa) >= SPA_VERSION_NEXT_CLONES) {
                        if (origin->ds_phys->ds_next_clones_obj == 0) {
                                origin->ds_phys->ds_next_clones_obj =
                                    zap_create(mos,
                                    DMU_OT_NEXT_CLONES, DMU_OT_NONE, 0, tx);
                        }
                        VERIFY0(zap_add_int(mos,
                            origin->ds_phys->ds_next_clones_obj, dsobj, tx));
                }
                dmu_buf_will_dirty(dd->dd_dbuf, tx);
                dd->dd_phys->dd_origin_obj = origin->ds_object;
                if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
                        if (origin->ds_dir->dd_phys->dd_clones == 0) {
                                dmu_buf_will_dirty(origin->ds_dir->dd_dbuf, tx);
                                origin->ds_dir->dd_phys->dd_clones =
                                    zap_create(mos,
                                    DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
                        }
                        VERIFY0(zap_add_int(mos,
                            origin->ds_dir->dd_phys->dd_clones, dsobj, tx));
                }
        }

        if (spa_version(dp->dp_spa) >= SPA_VERSION_UNIQUE_ACCURATE)
                dsphys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;

        dmu_buf_rele(dbuf, FTAG);

        dmu_buf_will_dirty(dd->dd_dbuf, tx);
        dd->dd_phys->dd_head_dataset_obj = dsobj;

        return (dsobj);
}

static void
dsl_dataset_zero_zil(dsl_dataset_t *ds, dmu_tx_t *tx)
{
        objset_t *os;

        VERIFY0(dmu_objset_from_ds(ds, &os));
        bzero(&os->os_zil_header, sizeof (os->os_zil_header));
        dsl_dataset_dirty(ds, tx);
}

uint64_t
dsl_dataset_create_sync(dsl_dir_t *pdd, const char *lastname,
    dsl_dataset_t *origin, uint64_t flags, cred_t *cr, dmu_tx_t *tx)
{
        dsl_pool_t *dp = pdd->dd_pool;
        uint64_t dsobj, ddobj;
        dsl_dir_t *dd;

        ASSERT(dmu_tx_is_syncing(tx));
        ASSERT(lastname[0] != ’@’);

        ddobj = dsl_dir_create_sync(dp, pdd, lastname, tx);
        VERIFY0(dsl_dir_hold_obj(dp, ddobj, lastname, FTAG, &dd));

        dsobj = dsl_dataset_create_sync_dd(dd, origin,
            flags & ~DS_CREATE_FLAG_NODIRTY, tx);

        dsl_deleg_set_create_perms(dd, tx, cr);

        dsl_dir_rele(dd, FTAG);

        /*
         * If we are creating a clone, make sure we zero out any stale
         * data from the origin snapshots zil header.
         */
        if (origin != NULL && !(flags & DS_CREATE_FLAG_NODIRTY)) {
                dsl_dataset_t *ds;

                VERIFY0(dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
                dsl_dataset_zero_zil(ds, tx);
```

```
                dsl_dataset_rele(ds, FTAG);
        }

        return (dsobj);
}

/*
 * The unique space in the head dataset can be calculated by subtracting
 * the space used in the most recent snapshot, that is still being used
 * in this file system, from the space currently in use.  To figure out
 * the space in the most recent snapshot still in use, we need to take
 * the total space used in the snapshot and subtract out the space that
 * has been freed up since the snapshot was taken.
 */
void
dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds)
{
        uint64_t mrs_used;
        uint64_t dlused, dlcomp, dluncomp;

        ASSERT(!dsl_dataset_is_snapshot(ds));

        if (ds->ds_phys->ds_prev_snap_obj != 0)
                mrs_used = ds->ds_prev->ds_phys->ds_referenced_bytes;
        else
                mrs_used = 0;

        dsl_deadlist_space(&ds->ds_deadlist, &dlused, &dlcomp, &dluncomp);

        ASSERT3U(dlused, <=, mrs_used);
        ds->ds_phys->ds_unique_bytes =
            ds->ds_phys->ds_referenced_bytes - (mrs_used - dlused);

        if (spa_version(ds->ds_dir->dd_pool->dp_spa) >=
            SPA_VERSION_UNIQUE_ACCURATE)
                ds->ds_phys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;
}

void
dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
    dmu_tx_t *tx)
{
        objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
        uint64_t count;
        int err;

        ASSERT(ds->ds_phys->ds_num_children >= 2);
        err = zap_remove_int(mos, ds->ds_phys->ds_next_clones_obj, obj, tx);
        /*
         * The err should not be ENOENT, but a bug in a previous version
         * of the code could cause upgrade_clones_cb() to not set
         * ds_next_snap_obj when it should, leading to a missing entry.
         * If we knew that the pool was created after
         * SPA_VERSION_NEXT_CLONES, we could assert that it isn’t
         * ENOENT.  However, at least we can check that we don’t have
         * too many entries in the next_clones_obj even after failing to
         * remove this one.
         */
        if (err != ENOENT)
                VERIFY0(err);
        ASSERT0(zap_count(mos, ds->ds_phys->ds_next_clones_obj,
            &count));
        ASSERT3U(count, <=, ds->ds_phys->ds_num_children - 2);
}
```

```
blkptr_t *
dsl_dataset_get_blkptr(dsl_dataset_t *ds)
{
        return (&ds->ds_phys->ds_bp);
}

void
dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx)
{
        ASSERT(dmu_tx_is_syncing(tx));
        /* If it’s the meta-objset, set dp_meta_rootbp */
        if (ds == NULL) {
                tx->tx_pool->dp_meta_rootbp = *bp;
        } else {
                dmu_buf_will_dirty(ds->ds_dbuf, tx);
                ds->ds_phys->ds_bp = *bp;
        }
}

spa_t *
dsl_dataset_get_spa(dsl_dataset_t *ds)
{
        return (ds->ds_dir->dd_pool->dp_spa);
}

void
dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx)
{
        dsl_pool_t *dp;

        if (ds == NULL) /* this is the meta-objset */
                return;

        ASSERT(ds->ds_objset != NULL);

        if (ds->ds_phys->ds_next_snap_obj != 0)
                panic("dirtying snapshot!");

        dp = ds->ds_dir->dd_pool;

        if (txg_list_add(&dp->dp_dirty_datasets, ds, tx->tx_txg)) {
                /* up the hold count until we can be written out */
                dmu_buf_add_ref(ds->ds_dbuf, ds);
        }
}

boolean_t
dsl_dataset_is_dirty(dsl_dataset_t *ds)
{
        for (int t = 0; t < TXG_SIZE; t++) {
                if (txg_list_member(&ds->ds_dir->dd_pool->dp_dirty_datasets,
                    ds, t))
                        return (B_TRUE);
        }
        return (B_FALSE);
}

static int
dsl_dataset_snapshot_reserve_space(dsl_dataset_t *ds, dmu_tx_t *tx)
{
        uint64_t asize;

        if (!dmu_tx_is_syncing(tx))
                return (0);

        /*
```

```
         * If there’s an fs-only reservation, any blocks that might become
         * owned by the snapshot dataset must be accommodated by space
         * outside of the reservation.
         */
        ASSERT(ds->ds_reserved == 0 || DS_UNIQUE_IS_ACCURATE(ds));
        asize = MIN(ds->ds_phys->ds_unique_bytes, ds->ds_reserved);
        if (asize > dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE))
                return (SET_ERROR(ENOSPC));

        /*
         * Propagate any reserved space for this snapshot to other
         * snapshot checks in this sync group.
         */
        if (asize > 0)
                dsl_dir_willuse_space(ds->ds_dir, asize, tx);

        return (0);
}

typedef struct dsl_dataset_snapshot_arg {
        nvlist_t *ddsa_snaps;
        nvlist_t *ddsa_props;
        nvlist_t *ddsa_errors;
} dsl_dataset_snapshot_arg_t;

int
dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
    dmu_tx_t *tx)
{
        int error;
        uint64_t value;

        ds->ds_trysnap_txg = tx->tx_txg;

        if (!dmu_tx_is_syncing(tx))
                return (0);

        /*
         * We don’t allow multiple snapshots of the same txg.  If there
         * is already one, try again.
         */
        if (ds->ds_phys->ds_prev_snap_txg >= tx->tx_txg)
                return (SET_ERROR(EAGAIN));

        /*
         * Check for conflicting snapshot name.
         */
        error = dsl_dataset_snap_lookup(ds, snapname, &value);
        if (error == 0)
                return (SET_ERROR(EEXIST));
        if (error != ENOENT)
                return (error);

        error = dsl_dataset_snapshot_reserve_space(ds, tx);
        if (error != 0)
                return (error);

        return (0);
}

static int
dsl_dataset_snapshot_check(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_snapshot_arg_t *ddsa = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        nvpair_t *pair;
        int rv = 0;

        for (pair = nvlist_next_nvpair(ddsa->ddsa_snaps, NULL);
            pair != NULL; pair = nvlist_next_nvpair(ddsa->ddsa_snaps, pair)) {
                int error = 0;
                dsl_dataset_t *ds;
                char *name, *atp;
                char dsname[MAXNAMELEN];

                name = nvpair_name(pair);
                if (strlen(name) >= MAXNAMELEN)
                        error = SET_ERROR(ENAMETOOLONG);
                if (error == 0) {
                        atp = strchr(name, '@');
                        if (atp == NULL)
                                error = SET_ERROR(EINVAL);
                        if (error == 0)
                                (void) strlcpy(dsname, name, atp - name + 1);
                }
                if (error == 0)
                        error = dsl_dataset_hold(dp, dsname, FTAG, &ds);
                if (error == 0) {
                        error = dsl_dataset_snapshot_check_impl(ds,
                            atp + 1, tx);
                        dsl_dataset_rele(ds, FTAG);
                }

                if (error != 0) {
                        if (ddsa->ddsa_errors != NULL) {
                                fnvlist_add_int32(ddsa->ddsa_errors,
                                    name, error);
                        }
                        rv = error;
                }
        }
        return (rv);
}

void
dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,
    dmu_tx_t *tx)
{
        static zil_header_t zero_zil;

        dsl_pool_t *dp = ds->ds_dir->dd_pool;
        dmu_buf_t *dbuf;
        dsl_dataset_phys_t *dsphys;
        uint64_t dsobj, crtxg;
        objset_t *mos = dp->dp_meta_objset;
        objset_t *os;

        ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

        /*
         * If we are on an old pool, the zil must not be active, in which
         * case it will be zeroed.  Usually zil_suspend() accomplishes this.
         */
        ASSERT(spa_version(dmu_tx_pool(tx)->dp_spa) >= SPA_VERSION_FAST_SNAP ||
            dmu_objset_from_ds(ds, &os) != 0 ||
            bcmp(&os->os_phys->os_zil_header, &zero_zil,
            sizeof (zero_zil)) == 0);


        /*
         * The origin's ds_creation_txg has to be < TXG_INITIAL
         */
```

```
        if (strcmp(snapname, ORIGIN_DIR_NAME) == 0)
                crtxg = 1;
        else
                crtxg = tx->tx_txg;

        dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
            DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
        VERIFY0(dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
        dmu_buf_will_dirty(dbuf, tx);
        dsphys = dbuf->db_data;
        bzero(dsphys, sizeof (dsl_dataset_phys_t));
        dsphys->ds_dir_obj = ds->ds_dir->dd_object;
        dsphys->ds_fsid_guid = unique_create();
        (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
            sizeof (dsphys->ds_guid));
        dsphys->ds_prev_snap_obj = ds->ds_phys->ds_prev_snap_obj;
        dsphys->ds_prev_snap_txg = ds->ds_phys->ds_prev_snap_txg;
        dsphys->ds_next_snap_obj = ds->ds_object;
        dsphys->ds_num_children = 1;
        dsphys->ds_creation_time = gethrestime_sec();
        dsphys->ds_creation_txg = crtxg;
        dsphys->ds_deadlist_obj = ds->ds_phys->ds_deadlist_obj;
        dsphys->ds_referenced_bytes = ds->ds_phys->ds_referenced_bytes;
        dsphys->ds_compressed_bytes = ds->ds_phys->ds_compressed_bytes;
        dsphys->ds_uncompressed_bytes = ds->ds_phys->ds_uncompressed_bytes;
        dsphys->ds_flags = ds->ds_phys->ds_flags;
        dsphys->ds_bp = ds->ds_phys->ds_bp;
        dmu_buf_rele(dbuf, FTAG);

        ASSERT3U(ds->ds_prev != 0, ==, ds->ds_phys->ds_prev_snap_obj != 0);
        if (ds->ds_prev) {
                uint64_t next_clones_obj =
                    ds->ds_prev->ds_phys->ds_next_clones_obj;
                ASSERT(ds->ds_prev->ds_phys->ds_next_snap_obj ==
                    ds->ds_object ||
                    ds->ds_prev->ds_phys->ds_num_children > 1);
                if (ds->ds_prev->ds_phys->ds_next_snap_obj == ds->ds_object) {
                        dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
                        ASSERT3U(ds->ds_phys->ds_prev_snap_txg, ==,
                            ds->ds_prev->ds_phys->ds_creation_txg);
                        ds->ds_prev->ds_phys->ds_next_snap_obj = dsobj;
                } else if (next_clones_obj != 0) {
                        dsl_dataset_remove_from_next_clones(ds->ds_prev,
                            dsphys->ds_next_snap_obj, tx);
                        VERIFY0(zap_add_int(mos,
                            next_clones_obj, dsobj, tx));
                }
        }

        /*
         * If we have a reference-reservation on this dataset, we will
         * need to increase the amount of refreservation being charged
         * since our unique space is going to zero.
         */
        if (ds->ds_reserved) {
                int64_t delta;
                ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
                delta = MIN(ds->ds_phys->ds_unique_bytes, ds->ds_reserved);
                dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV,
                    delta, 0, 0, tx);
        }

        dmu_buf_will_dirty(ds->ds_dbuf, tx);
        ds->ds_phys->ds_deadlist_obj = dsl_deadlist_clone(&ds->ds_deadlist,
            UINT64_MAX, ds->ds_phys->ds_prev_snap_obj, tx);
        dsl_deadlist_close(&ds->ds_deadlist);
        dsl_deadlist_open(&ds->ds_deadlist, mos, ds->ds_phys->ds_deadlist_obj);
        dsl_deadlist_add_key(&ds->ds_deadlist,
            ds->ds_phys->ds_prev_snap_txg, tx);

        ASSERT3U(ds->ds_phys->ds_prev_snap_txg, <, tx->tx_txg);
        ds->ds_phys->ds_prev_snap_obj = dsobj;
        ds->ds_phys->ds_prev_snap_txg = crtxg;
        ds->ds_phys->ds_unique_bytes = 0;
        if (spa_version(dp->dp_spa) >= SPA_VERSION_UNIQUE_ACCURATE)
                ds->ds_phys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;

        VERIFY0(zap_add(mos, ds->ds_phys->ds_snapnames_zapobj,
            snapname, 8, 1, &dsobj, tx));

        if (ds->ds_prev)
                dsl_dataset_rele(ds->ds_prev, ds);
        VERIFY0(dsl_dataset_hold_obj(dp,
            ds->ds_phys->ds_prev_snap_obj, ds, &ds->ds_prev));

        dsl_scan_ds_snapshotted(ds, tx);

        dsl_dir_snap_cmtime_update(ds->ds_dir);

        spa_history_log_internal_ds(ds->ds_prev, "snapshot", tx, "");
}

static void
dsl_dataset_snapshot_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_snapshot_arg_t *ddsa = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        nvpair_t *pair;

        for (pair = nvlist_next_nvpair(ddsa->ddsa_snaps, NULL);
            pair != NULL; pair = nvlist_next_nvpair(ddsa->ddsa_snaps, pair)) {
                dsl_dataset_t *ds;
                char *name, *atp;
                char dsname[MAXNAMELEN];

                name = nvpair_name(pair);
                atp = strchr(name, ’@’);
                (void) strlcpy(dsname, name, atp - name + 1);
                VERIFY0(dsl_dataset_hold(dp, dsname, FTAG, &ds));

                dsl_dataset_snapshot_sync_impl(ds, atp + 1, tx);
                if (ddsa->ddsa_props != NULL) {
                        dsl_props_set_sync_impl(ds->ds_prev,
                            ZPROP_SRC_LOCAL, ddsa->ddsa_props, tx);
                }
                dsl_dataset_rele(ds, FTAG);
        }
}

/*
 * The snapshots must all be in the same pool.
 * All-or-nothing: if there are any failures, nothing will be modified.
 */
int
dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors)
{
        dsl_dataset_snapshot_arg_t ddsa;
        nvpair_t *pair;
        boolean_t needsuspend;
        int error;
        spa_t *spa;
        char *firstname;
```

```
        nvlist_t *suspended = NULL;

        pair = nvlist_next_nvpair(snaps, NULL);
        if (pair == NULL)
                return (0);
        firstname = nvpair_name(pair);

        error = spa_open(firstname, &spa, FTAG);
        if (error != 0)
                return (error);
        needsuspend = (spa_version(spa) < SPA_VERSION_FAST_SNAP);
        spa_close(spa, FTAG);

        if (needsuspend) {
                suspended = fnvlist_alloc();
                for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
                    pair = nvlist_next_nvpair(snaps, pair)) {
                        char fsname[MAXNAMELEN];
                        char *snapname = nvpair_name(pair);
                        char *atp;
                        void *cookie;

                        atp = strchr(snapname, ’@’);
                        if (atp == NULL) {
                                error = SET_ERROR(EINVAL);
                                break;
                        }
                        (void) strlcpy(fsname, snapname, atp - snapname + 1);

                        error = zil_suspend(fsname, &cookie);
                        if (error != 0)
                                break;
                        fnvlist_add_uint64(suspended, fsname,
                            (uintptr_t)cookie);
                }
        }

        ddsa.ddsa_snaps = snaps;
        ddsa.ddsa_props = props;
        ddsa.ddsa_errors = errors;

        if (error == 0) {
                error = dsl_sync_task(firstname, dsl_dataset_snapshot_check,
                    dsl_dataset_snapshot_sync, &ddsa,
                    fnvlist_num_pairs(snaps) * 3);
        }

        if (suspended != NULL) {
                for (pair = nvlist_next_nvpair(suspended, NULL); pair != NULL;
                    pair = nvlist_next_nvpair(suspended, pair)) {
                        zil_resume((void *)(uintptr_t)
                            fnvpair_value_uint64(pair));
                }
                fnvlist_free(suspended);
        }

        return (error);
}

typedef struct dsl_dataset_snapshot_tmp_arg {
        const char *ddsta_fsname;
        const char *ddsta_snapname;
        minor_t ddsta_cleanup_minor;
        const char *ddsta_htag;
} dsl_dataset_snapshot_tmp_arg_t;
```

```
static int
dsl_dataset_snapshot_tmp_check(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_snapshot_tmp_arg_t *ddsta = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        int error;

        error = dsl_dataset_hold(dp, ddsta->ddsta_fsname, FTAG, &ds);
        if (error != 0)
                return (error);

        error = dsl_dataset_snapshot_check_impl(ds, ddsta->ddsta_snapname, tx);
        if (error != 0) {
                dsl_dataset_rele(ds, FTAG);
                return (error);
        }

        if (spa_version(dp->dp_spa) < SPA_VERSION_USERREFS) {
                dsl_dataset_rele(ds, FTAG);
                return (SET_ERROR(ENOTSUP));
        }
        error = dsl_dataset_user_hold_check_one(NULL, ddsta->ddsta_htag,
            B_TRUE, tx);
        if (error != 0) {
                dsl_dataset_rele(ds, FTAG);
                return (error);
        }

        dsl_dataset_rele(ds, FTAG);
        return (0);
}

static void
dsl_dataset_snapshot_tmp_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_snapshot_tmp_arg_t *ddsta = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;

        VERIFY0(dsl_dataset_hold(dp, ddsta->ddsta_fsname, FTAG, &ds));

        dsl_dataset_snapshot_sync_impl(ds, ddsta->ddsta_snapname, tx);
        dsl_dataset_user_hold_sync_one(ds->ds_prev, ddsta->ddsta_htag,
            ddsta->ddsta_cleanup_minor, gethrestime_sec(), tx);
        dsl_destroy_snapshot_sync_impl(ds->ds_prev, B_TRUE, tx);

        dsl_dataset_rele(ds, FTAG);
}

int
dsl_dataset_snapshot_tmp(const char *fsname, const char *snapname,
    minor_t cleanup_minor, const char *htag)
{
        dsl_dataset_snapshot_tmp_arg_t ddsta;
        int error;
        spa_t *spa;
        boolean_t needsuspend;
        void *cookie;

        ddsta.ddsta_fsname = fsname;
        ddsta.ddsta_snapname = snapname;
        ddsta.ddsta_cleanup_minor = cleanup_minor;
        ddsta.ddsta_htag = htag;

        error = spa_open(fsname, &spa, FTAG);
```

```
        if (error != 0)
                return (error);
        needsuspend = (spa_version(spa) < SPA_VERSION_FAST_SNAP);
        spa_close(spa, FTAG);

        if (needsuspend) {
                error = zil_suspend(fsname, &cookie);
                if (error != 0)
                        return (error);
        }

        error = dsl_sync_task(fsname, dsl_dataset_snapshot_tmp_check,
            dsl_dataset_snapshot_tmp_sync, &ddsta, 3);

        if (needsuspend)
                zil_resume(cookie);
        return (error);
}


void
dsl_dataset_sync(dsl_dataset_t *ds, zio_t *zio, dmu_tx_t *tx)
{
        ASSERT(dmu_tx_is_syncing(tx));
        ASSERT(ds->ds_objset != NULL);
        ASSERT(ds->ds_phys->ds_next_snap_obj == 0);

        /*
         * in case we had to change ds_fsid_guid when we opened it,
         * sync it out now.
         */
        dmu_buf_will_dirty(ds->ds_dbuf, tx);
        ds->ds_phys->ds_fsid_guid = ds->ds_fsid_guid;

        dmu_objset_sync(ds->ds_objset, zio, tx);
}

static void
get_clones_stat(dsl_dataset_t *ds, nvlist_t *nv)
{
        uint64_t count = 0;
        objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
        zap_cursor_t zc;
        zap_attribute_t za;
        nvlist_t *propval = fnvlist_alloc();
        nvlist_t *val = fnvlist_alloc();

        ASSERT(dsl_pool_config_held(ds->ds_dir->dd_pool));

        /*
         * There may be missing entries in ds_next_clones_obj
         * due to a bug in a previous version of the code.
         * Only trust it if it has the right number of entries.
         */
        if (ds->ds_phys->ds_next_clones_obj != 0) {
                ASSERT0(zap_count(mos, ds->ds_phys->ds_next_clones_obj,
                    &count));
        }
        if (count != ds->ds_phys->ds_num_children - 1)
                goto fail;
        for (zap_cursor_init(&zc, mos, ds->ds_phys->ds_next_clones_obj);
            zap_cursor_retrieve(&zc, &za) == 0;
            zap_cursor_advance(&zc)) {
                dsl_dataset_t *clone;
                char buf[ZFS_MAXNAMELEN];
                VERIFY0(dsl_dataset_hold_obj(ds->ds_dir->dd_pool,
                    za.za_first_integer, FTAG, &clone));
                dsl_dir_name(clone->ds_dir, buf);
                fnvlist_add_boolean(val, buf);
                dsl_dataset_rele(clone, FTAG);
        }
        zap_cursor_fini(&zc);
        fnvlist_add_nvlist(propval, ZPROP_VALUE, val);
        fnvlist_add_nvlist(nv, zfs_prop_to_name(ZFS_PROP_CLONES), propval);
fail:
        nvlist_free(val);
        nvlist_free(propval);
}

void
dsl_dataset_stats(dsl_dataset_t *ds, nvlist_t *nv)
{
        dsl_pool_t *dp = ds->ds_dir->dd_pool;
        uint64_t refd, avail, uobjs, aobjs, ratio;

        ASSERT(dsl_pool_config_held(dp));

        ratio = ds->ds_phys->ds_compressed_bytes == 0 ? 100 :
            (ds->ds_phys->ds_uncompressed_bytes * 100 /
            ds->ds_phys->ds_compressed_bytes);

        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRATIO, ratio);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_LOGICALREFERENCED,
            ds->ds_phys->ds_uncompressed_bytes);

        if (dsl_dataset_is_snapshot(ds)) {
                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_COMPRESSRATIO, ratio);
                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USED,
                    ds->ds_phys->ds_unique_bytes);
                get_clones_stat(ds, nv);
        } else {
                dsl_dir_stats(ds->ds_dir, nv);
        }

        dsl_dataset_space(ds, &refd, &avail, &uobjs, &aobjs);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_AVAILABLE, avail);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFERENCED, refd);

        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATION,
            ds->ds_phys->ds_creation_time);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATETXG,
            ds->ds_phys->ds_creation_txg);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFQUOTA,
            ds->ds_quota);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRESERVATION,
            ds->ds_reserved);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_GUID,
            ds->ds_phys->ds_guid);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_UNIQUE,
            ds->ds_phys->ds_unique_bytes);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_OBJSETID,
            ds->ds_object);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERREFS,
            ds->ds_userrefs);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_DEFER_DESTROY,
            DS_IS_DEFER_DESTROY(ds) ? 1 : 0);

        if (ds->ds_phys->ds_prev_snap_obj != 0) {
                uint64_t written, comp, uncomp;
                dsl_pool_t *dp = ds->ds_dir->dd_pool;
                dsl_dataset_t *prev;
```

```
                int err = dsl_dataset_hold_obj(dp,
                    ds->ds_phys->ds_prev_snap_obj, FTAG, &prev);
                if (err == 0) {
                        err = dsl_dataset_space_written(prev, ds, &written,
                            &comp, &uncomp);
                        dsl_dataset_rele(prev, FTAG);
                        if (err == 0) {
                                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_WRITTEN,
                                    written);
                        }
                }
        }
}

void
dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat)
{
        dsl_pool_t *dp = ds->ds_dir->dd_pool;
        ASSERT(dsl_pool_config_held(dp));

        stat->dds_creation_txg = ds->ds_phys->ds_creation_txg;
        stat->dds_inconsistent = ds->ds_phys->ds_flags & DS_FLAG_INCONSISTENT;
        stat->dds_guid = ds->ds_phys->ds_guid;
        stat->dds_origin[0] = ’\0’;
        if (dsl_dataset_is_snapshot(ds)) {
                stat->dds_is_snapshot = B_TRUE;
                stat->dds_num_clones = ds->ds_phys->ds_num_children - 1;
        } else {
                stat->dds_is_snapshot = B_FALSE;
                stat->dds_num_clones = 0;

                if (dsl_dir_is_clone(ds->ds_dir)) {
                        dsl_dataset_t *ods;

                        VERIFY0(dsl_dataset_hold_obj(dp,
                            ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &ods));
                        dsl_dataset_name(ods, stat->dds_origin);
                        dsl_dataset_rele(ods, FTAG);
                }
        }
}

uint64_t
dsl_dataset_fsid_guid(dsl_dataset_t *ds)
{
        return (ds->ds_fsid_guid);
}

void
dsl_dataset_space(dsl_dataset_t *ds,
    uint64_t *refdbytesp, uint64_t *availbytesp,
    uint64_t *usedobjsp, uint64_t *availobjsp)
{
        *refdbytesp = ds->ds_phys->ds_referenced_bytes;
        *availbytesp = dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE);
        if (ds->ds_reserved > ds->ds_phys->ds_unique_bytes)
                *availbytesp += ds->ds_reserved - ds->ds_phys->ds_unique_bytes;
        if (ds->ds_quota != 0) {
                /*
                 * Adjust available bytes according to refquota
                 */
                if (*refdbytesp < ds->ds_quota)
                        *availbytesp = MIN(*availbytesp,
                            ds->ds_quota - *refdbytesp);
                else
                        *availbytesp = 0;
        }
        *usedobjsp = ds->ds_phys->ds_bp.blk_fill;
        *availobjsp = DN_MAX_OBJECT - *usedobjsp;
}

boolean_t
dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds)
{
        dsl_pool_t *dp = ds->ds_dir->dd_pool;

        ASSERT(dsl_pool_config_held(dp));
        if (ds->ds_prev == NULL)
                return (B_FALSE);
        if (ds->ds_phys->ds_bp.blk_birth >
            ds->ds_prev->ds_phys->ds_creation_txg) {
                objset_t *os, *os_prev;
                /*
                 * It may be that only the ZIL differs, because it was
                 * reset in the head.  Don't count that as being
                 * modified.
                 */
                if (dmu_objset_from_ds(ds, &os) != 0)
                        return (B_TRUE);
                if (dmu_objset_from_ds(ds->ds_prev, &os_prev) != 0)
                        return (B_TRUE);
                return (bcmp(&os->os_phys->os_meta_dnode,
                    &os_prev->os_phys->os_meta_dnode,
                    sizeof (os->os_phys->os_meta_dnode)) != 0);
        }
        return (B_FALSE);
}

typedef struct dsl_dataset_rename_snapshot_arg {
        const char *ddrsa_fsname;
        const char *ddrsa_oldsnapname;
        const char *ddrsa_newsnapname;
        boolean_t ddrsa_recursive;
        dmu_tx_t *ddrsa_tx;
} dsl_dataset_rename_snapshot_arg_t;

/* ARGSUSED */
static int
dsl_dataset_rename_snapshot_check_impl(dsl_pool_t *dp,
    dsl_dataset_t *hds, void *arg)
{
        dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
        int error;
        uint64_t val;

        error = dsl_dataset_snap_lookup(hds, ddrsa->ddrsa_oldsnapname, &val);
        if (error != 0) {
                /* ignore nonexistent snapshots */
                return (error == ENOENT ? 0 : error);
        }

        /* new name should not exist */
        error = dsl_dataset_snap_lookup(hds, ddrsa->ddrsa_newsnapname, &val);
        if (error == 0)
                error = SET_ERROR(EEXIST);
        else if (error == ENOENT)
                error = 0;

        /* dataset name + 1 for the "@" + the new snapshot name must fit */
        if (dsl_dir_namelen(hds->ds_dir) + 1 +
            strlen(ddrsa->ddrsa_newsnapname) >= MAXNAMELEN)
                error = SET_ERROR(ENAMETOOLONG);
```

```

        return (error);
}

static int
dsl_dataset_rename_snapshot_check(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *hds;
        int error;

        error = dsl_dataset_hold(dp, ddrsa->ddrsa_fsname, FTAG, &hds);
        if (error != 0)
                return (error);

        if (ddrsa->ddrsa_recursive) {
                error = dmu_objset_find_dp(dp, hds->ds_dir->dd_object,
                    dsl_dataset_rename_snapshot_check_impl, ddrsa,
                    DS_FIND_CHILDREN);
        } else {
                error = dsl_dataset_rename_snapshot_check_impl(dp, hds, ddrsa);
        }
        dsl_dataset_rele(hds, FTAG);
        return (error);
}

static int
dsl_dataset_rename_snapshot_sync_impl(dsl_pool_t *dp,
    dsl_dataset_t *hds, void *arg)
{
        dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
        dsl_dataset_t *ds;
        uint64_t val;
        dmu_tx_t *tx = ddrsa->ddrsa_tx;
        int error;

        error = dsl_dataset_snap_lookup(hds, ddrsa->ddrsa_oldsnapname, &val);
        ASSERT(error == 0 || error == ENOENT);
        if (error == ENOENT) {
                /* ignore nonexistent snapshots */
                return (0);
        }

        VERIFY0(dsl_dataset_hold_obj(dp, val, FTAG, &ds));

        /* log before we change the name */
        spa_history_log_internal_ds(ds, "rename", tx,
            "-> @%s", ddrsa->ddrsa_newsnapname);

        VERIFY0(dsl_dataset_snap_remove(hds, ddrsa->ddrsa_oldsnapname, tx));
        mutex_enter(&ds->ds_lock);
        (void) strcpy(ds->ds_snapname, ddrsa->ddrsa_newsnapname);
        mutex_exit(&ds->ds_lock);
        VERIFY0(zap_add(dp->dp_meta_objset, hds->ds_phys->ds_snapnames_zapobj,
            ds->ds_snapname, 8, 1, &ds->ds_object, tx));

        dsl_dataset_rele(ds, FTAG);
        return (0);
}

static void
dsl_dataset_rename_snapshot_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *hds;

        VERIFY0(dsl_dataset_hold(dp, ddrsa->ddrsa_fsname, FTAG, &hds));
        ddrsa->ddrsa_tx = tx;
        if (ddrsa->ddrsa_recursive) {
                VERIFY0(dmu_objset_find_dp(dp, hds->ds_dir->dd_object,
                    dsl_dataset_rename_snapshot_sync_impl, ddrsa,
                    DS_FIND_CHILDREN));
        } else {
                VERIFY0(dsl_dataset_rename_snapshot_sync_impl(dp, hds, ddrsa));
        }
        dsl_dataset_rele(hds, FTAG);
}

int
dsl_dataset_rename_snapshot(const char *fsname,
    const char *oldsnapname, const char *newsnapname, boolean_t recursive)
{
        dsl_dataset_rename_snapshot_arg_t ddrsa;

        ddrsa.ddrsa_fsname = fsname;
        ddrsa.ddrsa_oldsnapname = oldsnapname;
        ddrsa.ddrsa_newsnapname = newsnapname;
        ddrsa.ddrsa_recursive = recursive;

        return (dsl_sync_task(fsname, dsl_dataset_rename_snapshot_check,
            dsl_dataset_rename_snapshot_sync, &ddrsa, 1));
}

static int
dsl_dataset_rollback_check(void *arg, dmu_tx_t *tx)
{
        const char *fsname = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        int64_t unused_refres_delta;
        int error;

        error = dsl_dataset_hold(dp, fsname, FTAG, &ds);
        if (error != 0)
                return (error);

        /* must not be a snapshot */
        if (dsl_dataset_is_snapshot(ds)) {
                dsl_dataset_rele(ds, FTAG);
                return (SET_ERROR(EINVAL));
        }

        /* must have a most recent snapshot */
        if (ds->ds_phys->ds_prev_snap_txg < TXG_INITIAL) {
                dsl_dataset_rele(ds, FTAG);
                return (SET_ERROR(EINVAL));
        }

        if (dsl_dataset_long_held(ds)) {
                dsl_dataset_rele(ds, FTAG);
                return (SET_ERROR(EBUSY));
        }

        /*
         * Check if the snap we are rolling back to uses more than
         * the refquota.
         */
        if (ds->ds_quota != 0 &&
            ds->ds_prev->ds_phys->ds_referenced_bytes > ds->ds_quota) {
                dsl_dataset_rele(ds, FTAG);
```

```
                return (SET_ERROR(EDQUOT));
        }

        /*
         * When we do the clone swap, we will temporarily use more space
         * due to the refreservation (the head will no longer have any
         * unique space, so the entire amount of the refreservation will need
         * to be free).  We will immediately destroy the clone, freeing
         * this space, but the freeing happens over many txg's.
         */
        unused_refres_delta = (int64_t)MIN(ds->ds_reserved,
            ds->ds_phys->ds_unique_bytes);

        if (unused_refres_delta > 0 &&
            unused_refres_delta >
            dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE)) {
                dsl_dataset_rele(ds, FTAG);
                return (SET_ERROR(ENOSPC));
        }

        dsl_dataset_rele(ds, FTAG);
        return (0);
}

static void
dsl_dataset_rollback_sync(void *arg, dmu_tx_t *tx)
{
        const char *fsname = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds, *clone;
        uint64_t cloneobj;

        VERIFY0(dsl_dataset_hold(dp, fsname, FTAG, &ds));

        cloneobj = dsl_dataset_create_sync(ds->ds_dir, "%rollback",
            ds->ds_prev, DS_CREATE_FLAG_NODIRTY, kcred, tx);

        VERIFY0(dsl_dataset_hold_obj(dp, cloneobj, FTAG, &clone));

        dsl_dataset_clone_swap_sync_impl(clone, ds, tx);
        dsl_dataset_zero_zil(ds, tx);

        dsl_destroy_head_sync_impl(clone, tx);

        dsl_dataset_rele(clone, FTAG);
        dsl_dataset_rele(ds, FTAG);
}

int
dsl_dataset_rollback(const char *fsname)
{
        return (dsl_sync_task(fsname, dsl_dataset_rollback_check,
            dsl_dataset_rollback_sync, (void *)fsname, 1));
}

struct promotenode {
        list_node_t link;
        dsl_dataset_t *ds;
};

typedef struct dsl_dataset_promote_arg {
        const char *ddpa_clonename;
        dsl_dataset_t *ddpa_clone;
        list_t shared_snaps, origin_snaps, clone_snaps;
        dsl_dataset_t *origin_origin; /* origin of the origin */
        uint64_t used, comp, uncomp, unique, cloneusedsnap, originusedsnap;
        char *err_ds;
} dsl_dataset_promote_arg_t;

static int snaplist_space(list_t *l, uint64_t mintxg, uint64_t *spacep);
static int promote_hold(dsl_dataset_promote_arg_t *ddpa, dsl_pool_t *dp,
    void *tag);
static void promote_rele(dsl_dataset_promote_arg_t *ddpa, void *tag);

static int
dsl_dataset_promote_check(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_promote_arg_t *ddpa = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *hds;
        struct promotenode *snap;
        dsl_dataset_t *origin_ds;
        int err;
        uint64_t unused;

        err = promote_hold(ddpa, dp, FTAG);
        if (err != 0)
                return (err);

        hds = ddpa->ddpa_clone;

        if (hds->ds_phys->ds_flags & DS_FLAG_NOPROMOTE) {
                promote_rele(ddpa, FTAG);
                return (SET_ERROR(EXDEV));
        }

        /*
         * Compute and check the amount of space to transfer.  Since this is
         * so expensive, don't do the preliminary check.
         */
        if (!dmu_tx_is_syncing(tx)) {
                promote_rele(ddpa, FTAG);
                return (0);
        }

        snap = list_head(&ddpa->shared_snaps);
        origin_ds = snap->ds;

        /* compute origin's new unique space */
        snap = list_tail(&ddpa->clone_snaps);
        ASSERT3U(snap->ds->ds_phys->ds_prev_snap_obj, ==, origin_ds->ds_object);
        dsl_deadlist_space_range(&snap->ds->ds_deadlist,
            origin_ds->ds_phys->ds_prev_snap_txg, UINT64_MAX,
            &ddpa->unique, &unused, &unused);

        /*
         * Walk the snapshots that we are moving
         *
         * Compute space to transfer.  Consider the incremental changes
         * to used by each snapshot:
         * (my used) = (prev's used) + (blocks born) - (blocks killed)
         * So each snapshot gave birth to:
         * (blocks born) = (my used) - (prev's used) + (blocks killed)
         * So a sequence would look like:
         * (uN - u(N-1) + kN) + ... + (u1 - u0 + k1) + (u0 - 0 + k0)
         * Which simplifies to:
         * uN + kN + kN-1 + ... + k1 + k0
         * Note however, if we stop before we reach the ORIGIN we get:
         * uN + kN + kN-1 + ... + kM - uM-1
         */
        ddpa->used = origin_ds->ds_phys->ds_referenced_bytes;
        ddpa->comp = origin_ds->ds_phys->ds_compressed_bytes;
```

```
        ddpa->uncomp = origin_ds->ds_phys->ds_uncompressed_bytes;
        for (snap = list_head(&ddpa->shared_snaps); snap;
            snap = list_next(&ddpa->shared_snaps, snap)) {
                uint64_t val, dlused, dlcomp, dluncomp;
                dsl_dataset_t *ds = snap->ds;

                /*
                 * If there are long holds, we won't be able to evict
                 * the objset.
                 */
                if (dsl_dataset_long_held(ds)) {
                        err = SET_ERROR(EBUSY);
                        goto out;
                }

                /* Check that the snapshot name does not conflict */
                VERIFY0(dsl_dataset_get_snapname(ds));
                err = dsl_dataset_snap_lookup(hds, ds->ds_snapname, &val);
                if (err == 0) {
                        (void) strcpy(ddpa->err_ds, snap->ds->ds_snapname);
                        err = SET_ERROR(EEXIST);
                        goto out;
                }
                if (err != ENOENT)
                        goto out;

                /* The very first snapshot does not have a deadlist */
                if (ds->ds_phys->ds_prev_snap_obj == 0)
                        continue;

                dsl_deadlist_space(&ds->ds_deadlist,
                    &dlused, &dlcomp, &dluncomp);
                ddpa->used += dlused;
                ddpa->comp += dlcomp;
                ddpa->uncomp += dluncomp;
        }

        /*
         * If we are a clone of a clone then we never reached ORIGIN,
         * so we need to subtract out the clone origin's used space.
         */
        if (ddpa->origin_origin) {
                ddpa->used -= ddpa->origin_origin->ds_phys->ds_referenced_bytes;
                ddpa->comp -= ddpa->origin_origin->ds_phys->ds_compressed_bytes;
                ddpa->uncomp -=
                    ddpa->origin_origin->ds_phys->ds_uncompressed_bytes;
        }

        /* Check that there is enough space here */
        err = dsl_dir_transfer_possible(origin_ds->ds_dir, hds->ds_dir,
            ddpa->used);
        if (err != 0)
                goto out;

        /*
         * Compute the amounts of space that will be used by snapshots
         * after the promotion (for both origin and clone).  For each,
         * it is the amount of space that will be on all of their
         * deadlists (that was not born before their new origin).
         */
        if (hds->ds_dir->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
                uint64_t space;

                /*
                 * Note, typically this will not be a clone of a clone,
                 * so dd_origin_txg will be < TXG_INITIAL, so
                 * these snaplist_space() -> dsl_deadlist_space_range()
                 * calls will be fast because they do not have to
                 * iterate over all bps.
                 */
                snap = list_head(&ddpa->origin_snaps);
                err = snaplist_space(&ddpa->shared_snaps,
                    snap->ds->ds_dir->dd_origin_txg, &ddpa->cloneusedsnap);
                if (err != 0)
                        goto out;

                err = snaplist_space(&ddpa->clone_snaps,
                    snap->ds->ds_dir->dd_origin_txg, &space);
                if (err != 0)
                        goto out;
                ddpa->cloneusedsnap += space;
        }
        if (origin_ds->ds_dir->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
                err = snaplist_space(&ddpa->origin_snaps,
                    origin_ds->ds_phys->ds_creation_txg, &ddpa->originusedsnap);
                if (err != 0)
                        goto out;
        }

out:
        promote_rele(ddpa, FTAG);
        return (err);
}

static void
dsl_dataset_promote_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_promote_arg_t *ddpa = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *hds;
        struct promotenode *snap;
        dsl_dataset_t *origin_ds;
        dsl_dataset_t *origin_head;
        dsl_dir_t *dd;
        dsl_dir_t *odd = NULL;
        uint64_t oldnext_obj;
        int64_t delta;

        VERIFY0(promote_hold(ddpa, dp, FTAG));
        hds = ddpa->ddpa_clone;

        ASSERT0(hds->ds_phys->ds_flags & DS_FLAG_NOPROMOTE);

        snap = list_head(&ddpa->shared_snaps);
        origin_ds = snap->ds;
        dd = hds->ds_dir;

        snap = list_head(&ddpa->origin_snaps);
        origin_head = snap->ds;

        /*
         * We need to explicitly open odd, since origin_ds’s dd will be
         * changing.
         */
        VERIFY0(dsl_dir_hold_obj(dp, origin_ds->ds_dir->dd_object,
            NULL, FTAG, &odd));

        /* change origin’s next snap */
        dmu_buf_will_dirty(origin_ds->ds_dbuf, tx);
        oldnext_obj = origin_ds->ds_phys->ds_next_snap_obj;
        snap = list_tail(&ddpa->clone_snaps);
        ASSERT3U(snap->ds->ds_phys->ds_prev_snap_obj, ==, origin_ds->ds_object);
```

```
        origin_ds->ds_phys->ds_next_snap_obj = snap->ds->ds_object;

        /* change the origin’s next clone */
        if (origin_ds->ds_phys->ds_next_clones_obj) {
                dsl_dataset_remove_from_next_clones(origin_ds,
                    snap->ds->ds_object, tx);
                VERIFY0(zap_add_int(dp->dp_meta_objset,
                    origin_ds->ds_phys->ds_next_clones_obj,
                    oldnext_obj, tx));
        }

        /* change origin */
        dmu_buf_will_dirty(dd->dd_dbuf, tx);
        ASSERT3U(dd->dd_phys->dd_origin_obj, ==, origin_ds->ds_object);
        dd->dd_phys->dd_origin_obj = odd->dd_phys->dd_origin_obj;
        dd->dd_origin_txg = origin_head->ds_dir->dd_origin_txg;
        dmu_buf_will_dirty(odd->dd_dbuf, tx);
        odd->dd_phys->dd_origin_obj = origin_ds->ds_object;
        origin_head->ds_dir->dd_origin_txg =
            origin_ds->ds_phys->ds_creation_txg;

        /* change dd_clone entries */
        if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
                VERIFY0(zap_remove_int(dp->dp_meta_objset,
                    odd->dd_phys->dd_clones, hds->ds_object, tx));
                VERIFY0(zap_add_int(dp->dp_meta_objset,
                    ddpa->origin_origin->ds_dir->dd_phys->dd_clones,
                    hds->ds_object, tx));

                VERIFY0(zap_remove_int(dp->dp_meta_objset,
                    ddpa->origin_origin->ds_dir->dd_phys->dd_clones,
                    origin_head->ds_object, tx));
                if (dd->dd_phys->dd_clones == 0) {
                        dd->dd_phys->dd_clones = zap_create(dp->dp_meta_objset,
                            DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
                }
                VERIFY0(zap_add_int(dp->dp_meta_objset,
                    dd->dd_phys->dd_clones, origin_head->ds_object, tx));
        }

        /* move snapshots to this dir */
        for (snap = list_head(&ddpa->shared_snaps); snap;
            snap = list_next(&ddpa->shared_snaps, snap)) {
                dsl_dataset_t *ds = snap->ds;

                /*
                 * Property callbacks are registered to a particular
                 * dsl_dir.  Since ours is changing, evict the objset
                 * so that they will be unregistered from the old dsl_dir.
                 */
                if (ds->ds_objset) {
                        dmu_objset_evict(ds->ds_objset);
                        ds->ds_objset = NULL;
                }

                /* move snap name entry */
                VERIFY0(dsl_dataset_get_snapname(ds));
                VERIFY0(dsl_dataset_snap_remove(origin_head,
                    ds->ds_snapname, tx));
                VERIFY0(zap_add(dp->dp_meta_objset,
                    hds->ds_phys->ds_snapnames_zapobj, ds->ds_snapname,
                    8, 1, &ds->ds_object, tx));

                /* change containing dsl_dir */
                dmu_buf_will_dirty(ds->ds_dbuf, tx);
                ASSERT3U(ds->ds_phys->ds_dir_obj, ==, odd->dd_object);
                ds->ds_phys->ds_dir_obj = dd->dd_object;
                ASSERT3P(ds->ds_dir, ==, odd);
                dsl_dir_rele(ds->ds_dir, ds);
                VERIFY0(dsl_dir_hold_obj(dp, dd->dd_object,
                    NULL, ds, &ds->ds_dir));

                /* move any clone references */
                if (ds->ds_phys->ds_next_clones_obj &&
                    spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
                        zap_cursor_t zc;
                        zap_attribute_t za;

                        for (zap_cursor_init(&zc, dp->dp_meta_objset,
                            ds->ds_phys->ds_next_clones_obj);
                            zap_cursor_retrieve(&zc, &za) == 0;
                            zap_cursor_advance(&zc)) {
                                dsl_dataset_t *cnds;
                                uint64_t o;

                                if (za.za_first_integer == oldnext_obj) {
                                        /*
                                         * We’ve already moved the
                                         * origin’s reference.
                                         */
                                        continue;
                                }

                                VERIFY0(dsl_dataset_hold_obj(dp,
                                    za.za_first_integer, FTAG, &cnds));
                                o = cnds->ds_dir->dd_phys->dd_head_dataset_obj;

                                VERIFY0(zap_remove_int(dp->dp_meta_objset,
                                    odd->dd_phys->dd_clones, o, tx));
                                VERIFY0(zap_add_int(dp->dp_meta_objset,
                                    dd->dd_phys->dd_clones, o, tx));
                                dsl_dataset_rele(cnds, FTAG);
                        }
                        zap_cursor_fini(&zc);
                }

                ASSERT(!dsl_prop_hascb(ds));
        }

        /*
         * Change space accounting.
         * Note, pa->*usedsnap and dd_used_breakdown[SNAP] will either
         * both be valid, or both be 0 (resulting in delta == 0).  This
         * is true for each of {clone,origin} independently.
         */

        delta = ddpa->cloneusedsnap -
            dd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
        ASSERT3S(delta, >=, 0);
        ASSERT3U(ddpa->used, >=, delta);
        dsl_dir_diduse_space(dd, DD_USED_SNAP, delta, 0, 0, tx);
        dsl_dir_diduse_space(dd, DD_USED_HEAD,
            ddpa->used - delta, ddpa->comp, ddpa->uncomp, tx);

        delta = ddpa->originusedsnap -
            odd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
        ASSERT3S(delta, <=, 0);
        ASSERT3U(ddpa->used, >=, -delta);
        dsl_dir_diduse_space(odd, DD_USED_SNAP, delta, 0, 0, tx);
        dsl_dir_diduse_space(odd, DD_USED_HEAD,
            -ddpa->used - delta, -ddpa->comp, -ddpa->uncomp, tx);
```

```
        origin_ds->ds_phys->ds_unique_bytes = ddpa->unique;

        /* log history record */
        spa_history_log_internal_ds(hds, "promote", tx, "");

        dsl_dir_rele(odd, FTAG);
        promote_rele(ddpa, FTAG);
}

/*
 * Make a list of dsl_dataset_t’s for the snapshots between first_obj
 * (exclusive) and last_obj (inclusive).  The list will be in reverse
 * order (last_obj will be the list_head()).  If first_obj == 0, do all
 * snapshots back to this dataset’s origin.
 */
static int
snaplist_make(dsl_pool_t *dp,
    uint64_t first_obj, uint64_t last_obj, list_t *l, void *tag)
{
        uint64_t obj = last_obj;

        list_create(l, sizeof (struct promotenode),
            offsetof(struct promotenode, link));

        while (obj != first_obj) {
                dsl_dataset_t *ds;
                struct promotenode *snap;
                int err;

                err = dsl_dataset_hold_obj(dp, obj, tag, &ds);
                ASSERT(err != ENOENT);
                if (err != 0)
                        return (err);

                if (first_obj == 0)
                        first_obj = ds->ds_dir->dd_phys->dd_origin_obj;

                snap = kmem_alloc(sizeof (*snap), KM_SLEEP);
                snap->ds = ds;
                list_insert_tail(l, snap);
                obj = ds->ds_phys->ds_prev_snap_obj;
        }

        return (0);
}

static int
snaplist_space(list_t *l, uint64_t mintxg, uint64_t *spacep)
{
        struct promotenode *snap;

        *spacep = 0;
        for (snap = list_head(l); snap; snap = list_next(l, snap)) {
                uint64_t used, comp, uncomp;
                dsl_deadlist_space_range(&snap->ds->ds_deadlist,
                    mintxg, UINT64_MAX, &used, &comp, &uncomp);
                *spacep += used;
        }
        return (0);
}

static void
snaplist_destroy(list_t *l, void *tag)
{
        struct promotenode *snap;
        if (l == NULL || !list_link_active(&l->list_head))
                return;

        while ((snap = list_tail(l)) != NULL) {
                list_remove(l, snap);
                dsl_dataset_rele(snap->ds, tag);
                kmem_free(snap, sizeof (*snap));
        }
        list_destroy(l);
}

static int
promote_hold(dsl_dataset_promote_arg_t *ddpa, dsl_pool_t *dp, void *tag)
{
        int error;
        dsl_dir_t *dd;
        struct promotenode *snap;

        error = dsl_dataset_hold(dp, ddpa->ddpa_clonename, tag,
            &ddpa->ddpa_clone);
        if (error != 0)
                return (error);
        dd = ddpa->ddpa_clone->ds_dir;

        if (dsl_dataset_is_snapshot(ddpa->ddpa_clone) ||
            !dsl_dir_is_clone(dd)) {
                dsl_dataset_rele(ddpa->ddpa_clone, tag);
                return (SET_ERROR(EINVAL));
        }

        error = snaplist_make(dp, 0, dd->dd_phys->dd_origin_obj,
            &ddpa->shared_snaps, tag);
        if (error != 0)
                goto out;

        error = snaplist_make(dp, 0, ddpa->ddpa_clone->ds_object,
            &ddpa->clone_snaps, tag);
        if (error != 0)
                goto out;

        snap = list_head(&ddpa->shared_snaps);
        ASSERT3U(snap->ds->ds_object, ==, dd->dd_phys->dd_origin_obj);
        error = snaplist_make(dp, dd->dd_phys->dd_origin_obj,
            snap->ds->ds_dir->dd_phys->dd_head_dataset_obj,
            &ddpa->origin_snaps, tag);
        if (error != 0)
                goto out;

        if (snap->ds->ds_dir->dd_phys->dd_origin_obj != 0) {
                error = dsl_dataset_hold_obj(dp,
                    snap->ds->ds_dir->dd_phys->dd_origin_obj,
                    tag, &ddpa->origin_origin);
                if (error != 0)
                        goto out;
        }
out:
        if (error != 0)
                promote_rele(ddpa, tag);
        return (error);
}

static void
promote_rele(dsl_dataset_promote_arg_t *ddpa, void *tag)
{
        snaplist_destroy(&ddpa->shared_snaps, tag);
        snaplist_destroy(&ddpa->clone_snaps, tag);
```

```
        snaplist_destroy(&ddpa->origin_snaps, tag);
        if (ddpa->origin_origin != NULL)
                dsl_dataset_rele(ddpa->origin_origin, tag);
        dsl_dataset_rele(ddpa->ddpa_clone, tag);
}

/*
 * Promote a clone.
 *
 * If it fails due to a conflicting snapshot name, "conflsnap" will be filled
 * in with the name.  (It must be at least MAXNAMELEN bytes long.)
 */
int
dsl_dataset_promote(const char *name, char *conflsnap)
{
        dsl_dataset_promote_arg_t ddpa = { 0 };
        uint64_t numsnaps;
        int error;
        objset_t *os;

        /*
         * We will modify space proportional to the number of
         * snapshots.  Compute numsnaps.
         */
        error = dmu_objset_hold(name, FTAG, &os);
        if (error != 0)
                return (error);
        error = zap_count(dmu_objset_pool(os)->dp_meta_objset,
            dmu_objset_ds(os)->ds_phys->ds_snapnames_zapobj, &numsnaps);
        dmu_objset_rele(os, FTAG);
        if (error != 0)
                return (error);

        ddpa.ddpa_clonename = name;
        ddpa.err_ds = conflsnap;

        return (dsl_sync_task(name, dsl_dataset_promote_check,
            dsl_dataset_promote_sync, &ddpa, 2 + numsnaps));
}

int
dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
    dsl_dataset_t *origin_head, boolean_t force)
{
        int64_t unused_refres_delta;

        /* they should both be heads */
        if (dsl_dataset_is_snapshot(clone) ||
            dsl_dataset_is_snapshot(origin_head))
                return (SET_ERROR(EINVAL));

        /* the branch point should be just before them */
        if (clone->ds_prev != origin_head->ds_prev)
                return (SET_ERROR(EINVAL));

        /* clone should be the clone (unless they are unrelated) */
        if (clone->ds_prev != NULL &&
            clone->ds_prev != clone->ds_dir->dd_pool->dp_origin_snap &&
            origin_head->ds_object !=
            clone->ds_prev->ds_phys->ds_next_snap_obj)
                return (SET_ERROR(EINVAL));

        /* the clone should be a child of the origin */
        if (clone->ds_dir->dd_parent != origin_head->ds_dir)
                return (SET_ERROR(EINVAL));
        /* origin_head shouldn’t be modified unless ’force’ */
        if (!force && dsl_dataset_modified_since_lastsnap(origin_head))
                return (SET_ERROR(ETXTBSY));

        /* origin_head should have no long holds (e.g. is not mounted) */
        if (dsl_dataset_long_held(origin_head))
                return (SET_ERROR(EBUSY));

        /* check amount of any unconsumed refreservation */
        unused_refres_delta =
            (int64_t)MIN(origin_head->ds_reserved,
            origin_head->ds_phys->ds_unique_bytes) -
            (int64_t)MIN(origin_head->ds_reserved,
            clone->ds_phys->ds_unique_bytes);

        if (unused_refres_delta > 0 &&
            unused_refres_delta >
            dsl_dir_space_available(origin_head->ds_dir, NULL, 0, TRUE))
                return (SET_ERROR(ENOSPC));

        /* clone can’t be over the head’s refquota */
        if (origin_head->ds_quota != 0 &&
            clone->ds_phys->ds_referenced_bytes > origin_head->ds_quota)
                return (SET_ERROR(EDQUOT));

        return (0);
}

void
dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
    dsl_dataset_t *origin_head, dmu_tx_t *tx)
{
        dsl_pool_t *dp = dmu_tx_pool(tx);
        int64_t unused_refres_delta;

        ASSERT(clone->ds_reserved == 0);
        ASSERT(origin_head->ds_quota == 0 ||
            clone->ds_phys->ds_unique_bytes <= origin_head->ds_quota);

        dmu_buf_will_dirty(clone->ds_dbuf, tx);
        dmu_buf_will_dirty(origin_head->ds_dbuf, tx);

        if (clone->ds_objset != NULL) {
                dmu_objset_evict(clone->ds_objset);
                clone->ds_objset = NULL;
        }

        if (origin_head->ds_objset != NULL) {
                dmu_objset_evict(origin_head->ds_objset);
                origin_head->ds_objset = NULL;
        }

        unused_refres_delta =
            (int64_t)MIN(origin_head->ds_reserved,
            origin_head->ds_phys->ds_unique_bytes) -
            (int64_t)MIN(origin_head->ds_reserved,
            clone->ds_phys->ds_unique_bytes);

        /*
         * Reset origin’s unique bytes, if it exists.
         */
        if (clone->ds_prev) {
                dsl_dataset_t *origin = clone->ds_prev;
                uint64_t comp, uncomp;

                dmu_buf_will_dirty(origin->ds_dbuf, tx);
```

```
                dsl_deadlist_space_range(&clone->ds_deadlist,
                    origin->ds_phys->ds_prev_snap_txg, UINT64_MAX,
                    &origin->ds_phys->ds_unique_bytes, &comp, &uncomp);
        }

        /* swap blkptrs */
        {
                blkptr_t tmp;
                tmp = origin_head->ds_phys->ds_bp;
                origin_head->ds_phys->ds_bp = clone->ds_phys->ds_bp;
                clone->ds_phys->ds_bp = tmp;
        }

        /* set dd_*_bytes */
        {
                int64_t dused, dcomp, duncomp;
                uint64_t cdl_used, cdl_comp, cdl_uncomp;
                uint64_t odl_used, odl_comp, odl_uncomp;

                ASSERT3U(clone->ds_dir->dd_phys->
                    dd_used_breakdown[DD_USED_SNAP], ==, 0);

                dsl_deadlist_space(&clone->ds_deadlist,
                    &cdl_used, &cdl_comp, &cdl_uncomp);
                dsl_deadlist_space(&origin_head->ds_deadlist,
                    &odl_used, &odl_comp, &odl_uncomp);

                dused = clone->ds_phys->ds_referenced_bytes + cdl_used -
                    (origin_head->ds_phys->ds_referenced_bytes + odl_used);
                dcomp = clone->ds_phys->ds_compressed_bytes + cdl_comp -
                    (origin_head->ds_phys->ds_compressed_bytes + odl_comp);
                duncomp = clone->ds_phys->ds_uncompressed_bytes +
                    cdl_uncomp -
                    (origin_head->ds_phys->ds_uncompressed_bytes + odl_uncomp);

                dsl_dir_diduse_space(origin_head->ds_dir, DD_USED_HEAD,
                    dused, dcomp, duncomp, tx);
                dsl_dir_diduse_space(clone->ds_dir, DD_USED_HEAD,
                    -dused, -dcomp, -duncomp, tx);

                /*
                 * The difference in the space used by snapshots is the
                 * difference in snapshot space due to the head’s
                 * deadlist (since that’s the only thing that’s
                 * changing that affects the snapused).
                 */
                dsl_deadlist_space_range(&clone->ds_deadlist,
                    origin_head->ds_dir->dd_origin_txg, UINT64_MAX,
                    &cdl_used, &cdl_comp, &cdl_uncomp);
                dsl_deadlist_space_range(&origin_head->ds_deadlist,
                    origin_head->ds_dir->dd_origin_txg, UINT64_MAX,
                    &odl_used, &odl_comp, &odl_uncomp);
                dsl_dir_transfer_space(origin_head->ds_dir, cdl_used - odl_used,
                    DD_USED_HEAD, DD_USED_SNAP, tx);
        }

        /* swap ds_*_bytes */
        SWITCH64(origin_head->ds_phys->ds_referenced_bytes,
            clone->ds_phys->ds_referenced_bytes);
        SWITCH64(origin_head->ds_phys->ds_compressed_bytes,
            clone->ds_phys->ds_compressed_bytes);
        SWITCH64(origin_head->ds_phys->ds_uncompressed_bytes,
            clone->ds_phys->ds_uncompressed_bytes);
        SWITCH64(origin_head->ds_phys->ds_unique_bytes,
            clone->ds_phys->ds_unique_bytes);
        /* apply any parent delta for change in unconsumed refreservation */
        dsl_dir_diduse_space(origin_head->ds_dir, DD_USED_REFRSRV,
            unused_refres_delta, 0, 0, tx);

        /*
         * Swap deadlists.
         */
        dsl_deadlist_close(&clone->ds_deadlist);
        dsl_deadlist_close(&origin_head->ds_deadlist);
        SWITCH64(origin_head->ds_phys->ds_deadlist_obj,
            clone->ds_phys->ds_deadlist_obj);
        dsl_deadlist_open(&clone->ds_deadlist, dp->dp_meta_objset,
            clone->ds_phys->ds_deadlist_obj);
        dsl_deadlist_open(&origin_head->ds_deadlist, dp->dp_meta_objset,
            origin_head->ds_phys->ds_deadlist_obj);

        dsl_scan_ds_clone_swapped(origin_head, clone, tx);

        spa_history_log_internal_ds(clone, "clone swap", tx,
            "parent=%s", origin_head->ds_dir->dd_myname);
}

/*
 * Given a pool name and a dataset object number in that pool,
 * return the name of that dataset.
 */
int
dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf)
{
        dsl_pool_t *dp;
        dsl_dataset_t *ds;
        int error;

        error = dsl_pool_hold(pname, FTAG, &dp);
        if (error != 0)
                return (error);

        error = dsl_dataset_hold_obj(dp, obj, FTAG, &ds);
        if (error == 0) {
                dsl_dataset_name(ds, buf);
                dsl_dataset_rele(ds, FTAG);
        }
        dsl_pool_rele(dp, FTAG);

        return (error);
}

int
dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
    uint64_t asize, uint64_t inflight, uint64_t *used, uint64_t *ref_rsrv)
{
        int error = 0;

        ASSERT3S(asize, >, 0);

        /*
         * *ref_rsrv is the portion of asize that will come from any
         * unconsumed refreservation space.
         */
        *ref_rsrv = 0;

        mutex_enter(&ds->ds_lock);
        /*
         * Make a space adjustment for reserved bytes.
         */
        if (ds->ds_reserved > ds->ds_phys->ds_unique_bytes) {
```

```
                ASSERT3U(*used, >=,
                    ds->ds_reserved - ds->ds_phys->ds_unique_bytes);
                *used -= (ds->ds_reserved - ds->ds_phys->ds_unique_bytes);
                *ref_rsrv =
                    asize - MIN(asize, parent_delta(ds, asize + inflight));
        }

        if (!check_quota || ds->ds_quota == 0) {
                mutex_exit(&ds->ds_lock);
                return (0);
        }
        /*
         * If they are requesting more space, and our current estimate
         * is over quota, they get to try again unless the actual
         * on-disk is over quota and there are no pending changes (which
         * may free up space for us).
         */
        if (ds->ds_phys->ds_referenced_bytes + inflight >= ds->ds_quota) {
                if (inflight > 0 ||
                    ds->ds_phys->ds_referenced_bytes < ds->ds_quota)
                        error = SET_ERROR(ERESTART);
                else
                        error = SET_ERROR(EDQUOT);
        }
        mutex_exit(&ds->ds_lock);

        return (error);
}

typedef struct dsl_dataset_set_qr_arg {
        const char *ddsqra_name;
        zprop_source_t ddsqra_source;
        uint64_t ddsqra_value;
} dsl_dataset_set_qr_arg_t;


/* ARGSUSED */
static int
dsl_dataset_set_refquota_check(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        int error;
        uint64_t newval;

        if (spa_version(dp->dp_spa) < SPA_VERSION_REFQUOTA)
                return (SET_ERROR(ENOTSUP));

        error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
        if (error != 0)
                return (error);

        if (dsl_dataset_is_snapshot(ds)) {
                dsl_dataset_rele(ds, FTAG);
                return (SET_ERROR(EINVAL));
        }

        error = dsl_prop_predict(ds->ds_dir,
            zfs_prop_to_name(ZFS_PROP_REFQUOTA),
            ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
        if (error != 0) {
                dsl_dataset_rele(ds, FTAG);
                return (error);
        }
        if (newval == 0) {
                dsl_dataset_rele(ds, FTAG);
                return (0);
        }

        if (newval < ds->ds_phys->ds_referenced_bytes ||
            newval < ds->ds_reserved) {
                dsl_dataset_rele(ds, FTAG);
                return (SET_ERROR(ENOSPC));
        }

        dsl_dataset_rele(ds, FTAG);
        return (0);
}

static void
dsl_dataset_set_refquota_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        uint64_t newval;

        VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds));

        dsl_prop_set_sync_impl(ds,
            zfs_prop_to_name(ZFS_PROP_REFQUOTA),
            ddsqra->ddsqra_source, sizeof (ddsqra->ddsqra_value), 1,
            &ddsqra->ddsqra_value, tx);

        VERIFY0(dsl_prop_get_int_ds(ds,
            zfs_prop_to_name(ZFS_PROP_REFQUOTA), &newval));

        if (ds->ds_quota != newval) {
                dmu_buf_will_dirty(ds->ds_dbuf, tx);
                ds->ds_quota = newval;
        }
        dsl_dataset_rele(ds, FTAG);
}

int
dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
    uint64_t refquota)
{
        dsl_dataset_set_qr_arg_t ddsqra;

        ddsqra.ddsqra_name = dsname;
        ddsqra.ddsqra_source = source;
        ddsqra.ddsqra_value = refquota;

        return (dsl_sync_task(dsname, dsl_dataset_set_refquota_check,
            dsl_dataset_set_refquota_sync, &ddsqra, 0));
}

static int
dsl_dataset_set_refreservation_check(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        int error;
        uint64_t newval, unique;

        if (spa_version(dp->dp_spa) < SPA_VERSION_REFRESERVATION)
                return (SET_ERROR(ENOTSUP));
```

```
        error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
        if (error != 0)
                return (error);

        if (dsl_dataset_is_snapshot(ds)) {
                dsl_dataset_rele(ds, FTAG);
                return (SET_ERROR(EINVAL));
        }

        error = dsl_prop_predict(ds->ds_dir,
            zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
            ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
        if (error != 0) {
                dsl_dataset_rele(ds, FTAG);
                return (error);
        }

        /*
         * If we are doing the preliminary check in open context, the
         * space estimates may be inaccurate.
         */
        if (!dmu_tx_is_syncing(tx)) {
                dsl_dataset_rele(ds, FTAG);
                return (0);
        }

        mutex_enter(&ds->ds_lock);
        if (!DS_UNIQUE_IS_ACCURATE(ds))
                dsl_dataset_recalc_head_uniq(ds);
        unique = ds->ds_phys->ds_unique_bytes;
        mutex_exit(&ds->ds_lock);

        if (MAX(unique, newval) > MAX(unique, ds->ds_reserved)) {
                uint64_t delta = MAX(unique, newval) -
                    MAX(unique, ds->ds_reserved);

                if (delta >
                    dsl_dir_space_available(ds->ds_dir, NULL, 0, B_TRUE) ||
                    (ds->ds_quota > 0 && newval > ds->ds_quota)) {
                        dsl_dataset_rele(ds, FTAG);
                        return (SET_ERROR(ENOSPC));
                }
        }

        dsl_dataset_rele(ds, FTAG);
        return (0);
}

void
dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
    zprop_source_t source, uint64_t value, dmu_tx_t *tx)
{
        uint64_t newval;
        uint64_t unique;
        int64_t delta;

        dsl_prop_set_sync_impl(ds, zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
            source, sizeof (value), 1, &value, tx);

        VERIFY0(dsl_prop_get_int_ds(ds,
            zfs_prop_to_name(ZFS_PROP_REFRESERVATION), &newval));

        dmu_buf_will_dirty(ds->ds_dbuf, tx);
        mutex_enter(&ds->ds_dir->dd_lock);
        mutex_enter(&ds->ds_lock);
        ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
        unique = ds->ds_phys->ds_unique_bytes;
        delta = MAX(0, (int64_t)(newval - unique)) -
            MAX(0, (int64_t)(ds->ds_reserved - unique));
        ds->ds_reserved = newval;
        mutex_exit(&ds->ds_lock);

        dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV, delta, 0, 0, tx);
        mutex_exit(&ds->ds_dir->dd_lock);
}

static void
dsl_dataset_set_refreservation_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;

        VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds));
        dsl_dataset_set_refreservation_sync_impl(ds,
            ddsqra->ddsqra_source, ddsqra->ddsqra_value, tx);
        dsl_dataset_rele(ds, FTAG);
}

int
dsl_dataset_set_refreservation(const char *dsname, zprop_source_t source,
    uint64_t refreservation)
{
        dsl_dataset_set_qr_arg_t ddsqra;

        ddsqra.ddsqra_name = dsname;
        ddsqra.ddsqra_source = source;
        ddsqra.ddsqra_value = refreservation;

        return (dsl_sync_task(dsname, dsl_dataset_set_refreservation_check,
            dsl_dataset_set_refreservation_sync, &ddsqra, 0));
}

/*
 * Return (in *usedp) the amount of space written in new that is not
 * present in oldsnap.  New may be a snapshot or the head.  Old must be
 * a snapshot before new, in new’s filesystem (or its origin).  If not then
 * fail and return EINVAL.
 *
 * The written space is calculated by considering two components:  First, we
 * ignore any freed space, and calculate the written as new’s used space
 * minus old’s used space.  Next, we add in the amount of space that was freed
 * between the two snapshots, thus reducing new’s used space relative to old’s.
 * Specifically, this is the space that was born before old->ds_creation_txg,
 * and freed before new (ie. on new’s deadlist or a previous deadlist).
 *
 * space freed                         [---------------------]
 * snapshots                       ---O-------O--------O-------O------
 *                                         oldsnap            new
 */
int
dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
    uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
{
        int err = 0;
        uint64_t snapobj;
        dsl_pool_t *dp = new->ds_dir->dd_pool;

        ASSERT(dsl_pool_config_held(dp));

        *usedp = 0;
        *usedp += new->ds_phys->ds_referenced_bytes;
```

```
        *usedp -= oldsnap->ds_phys->ds_referenced_bytes;

        *compp = 0;
        *compp += new->ds_phys->ds_compressed_bytes;
        *compp -= oldsnap->ds_phys->ds_compressed_bytes;

        *uncompp = 0;
        *uncompp += new->ds_phys->ds_uncompressed_bytes;
        *uncompp -= oldsnap->ds_phys->ds_uncompressed_bytes;

        snapobj = new->ds_object;
        while (snapobj != oldsnap->ds_object) {
                dsl_dataset_t *snap;
                uint64_t used, comp, uncomp;

                if (snapobj == new->ds_object) {
                        snap = new;
                } else {
                        err = dsl_dataset_hold_obj(dp, snapobj, FTAG, &snap);
                        if (err != 0)
                                break;
                }

                if (snap->ds_phys->ds_prev_snap_txg ==
                    oldsnap->ds_phys->ds_creation_txg) {
                        /*
                         * The blocks in the deadlist can not be born after
                         * ds_prev_snap_txg, so get the whole deadlist space,
                         * which is more efficient (especially for old-format
                         * deadlists).  Unfortunately the deadlist code
                         * doesn’t have enough information to make this
                         * optimization itself.
                         */
                        dsl_deadlist_space(&snap->ds_deadlist,
                            &used, &comp, &uncomp);
                } else {
                        dsl_deadlist_space_range(&snap->ds_deadlist,
                            0, oldsnap->ds_phys->ds_creation_txg,
                            &used, &comp, &uncomp);
                }
                *usedp += used;
                *compp += comp;
                *uncompp += uncomp;

                /*
                 * If we get to the beginning of the chain of snapshots
                 * (ds_prev_snap_obj == 0) before oldsnap, then oldsnap
                 * was not a snapshot of/before new.
                 */
                snapobj = snap->ds_phys->ds_prev_snap_obj;
                if (snap != new)
                        dsl_dataset_rele(snap, FTAG);
                if (snapobj == 0) {
                        err = SET_ERROR(EINVAL);
                        break;
                }

        }
        return (err);
}

/*
 * Return (in *usedp) the amount of space that will be reclaimed if firstsnap,
 * lastsnap, and all snapshots in between are deleted.
 *
 * blocks that would be freed            [---------------------------]
 * snapshots                       ---O-------O--------O-------O--------O
 *                                        firstsnap        lastsnap
 *
 * This is the set of blocks that were born after the snap before firstsnap,
 * (birth > firstsnap->prev_snap_txg) and died before the snap after the
 * last snap (ie, is on lastsnap->ds_next->ds_deadlist or an earlier deadlist).
 * We calculate this by iterating over the relevant deadlists (from the snap
 * after lastsnap, backward to the snap after firstsnap), summing up the
 * space on the deadlist that was born after the snap before firstsnap.
 */
int
dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap,
    dsl_dataset_t *lastsnap,
    uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
{
        int err = 0;
        uint64_t snapobj;
        dsl_pool_t *dp = firstsnap->ds_dir->dd_pool;

        ASSERT(dsl_dataset_is_snapshot(firstsnap));
        ASSERT(dsl_dataset_is_snapshot(lastsnap));

        /*
         * Check that the snapshots are in the same dsl_dir, and firstsnap
         * is before lastsnap.
         */
        if (firstsnap->ds_dir != lastsnap->ds_dir ||
            firstsnap->ds_phys->ds_creation_txg >
            lastsnap->ds_phys->ds_creation_txg)
                return (SET_ERROR(EINVAL));

        *usedp = *compp = *uncompp = 0;

        snapobj = lastsnap->ds_phys->ds_next_snap_obj;
        while (snapobj != firstsnap->ds_object) {
                dsl_dataset_t *ds;
                uint64_t used, comp, uncomp;

                err = dsl_dataset_hold_obj(dp, snapobj, FTAG, &ds);
                if (err != 0)
                        break;

                dsl_deadlist_space_range(&ds->ds_deadlist,
                    firstsnap->ds_phys->ds_prev_snap_txg, UINT64_MAX,
                    &used, &comp, &uncomp);
                *usedp += used;
                *compp += comp;
                *uncompp += uncomp;

                snapobj = ds->ds_phys->ds_prev_snap_obj;
                ASSERT3U(snapobj, !=, 0);
                dsl_dataset_rele(ds, FTAG);
        }
        return (err);
}

/*
 * Return TRUE if 'earlier' is an earlier snapshot in 'later's timeline.
 * For example, they could both be snapshots of the same filesystem, and
 * 'earlier' is before 'later'.  Or 'earlier' could be the origin of
 * 'later's filesystem.  Or 'earlier' could be an older snapshot in the origin's
 * filesystem.  Or 'earlier' could be the origin's origin.
 */
boolean_t
dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier)
{
```

```
        dsl_pool_t *dp = later->ds_dir->dd_pool;
        int error;
        boolean_t ret;

        ASSERT(dsl_pool_config_held(dp));

        if (earlier->ds_phys->ds_creation_txg >=
            later->ds_phys->ds_creation_txg)
                return (B_FALSE);

        if (later->ds_dir == earlier->ds_dir)
                return (B_TRUE);
        if (!dsl_dir_is_clone(later->ds_dir))
                return (B_FALSE);

        if (later->ds_dir->dd_phys->dd_origin_obj == earlier->ds_object)
                return (B_TRUE);
        dsl_dataset_t *origin;
        error = dsl_dataset_hold_obj(dp,
            later->ds_dir->dd_phys->dd_origin_obj, FTAG, &origin);
        if (error != 0)
                return (B_FALSE);
        ret = dsl_dataset_is_before(origin, earlier);
        dsl_dataset_rele(origin, FTAG);
        return (ret);
}
```

```
************************************************************
   11322 Tue Apr 23 15:35:05 2013
new/usr/src/uts/common/fs/zfs/spa_errlog.c
3743 zfs needs a refcount audit
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
************************************************************
______unchanged_portion_omitted_
```

```
#ifdef _KERNEL
static int
process_error_log(spa_t *spa, uint64_t obj, void *addr, size_t *count)
{
        zap_cursor_t zc;
        zap_attribute_t za;
        zbookmark_t zb;

        if (obj == 0)
                return (0);

        for (zap_cursor_init(&zc, spa->spa_meta_objset, obj);
            zap_cursor_retrieve(&zc, &za) == 0;
            zap_cursor_advance(&zc)) {

                if (*count == 0) {
                        zap_cursor_fini(&zc);
                        return (SET_ERROR(ENOMEM));
                }

                name_to_bookmark(za.za_name, &zb);

                if (copyout(&zb, (char *)addr +
                    (*count - 1) * sizeof (zbookmark_t),
                    sizeof (zbookmark_t)) != 0) {
                        zap_cursor_fini(&zc);
                    sizeof (zbookmark_t)) != 0)
                        return (SET_ERROR(EFAULT));
                }
#endif /* ! codereview */

                *count -= 1;
        }

        zap_cursor_fini(&zc);

        return (0);
}

static int
process_error_list(avl_tree_t *list, void *addr, size_t *count)
{
        spa_error_entry_t *se;

        for (se = avl_first(list); se != NULL; se = AVL_NEXT(list, se)) {

                if (*count == 0)
                        return (SET_ERROR(ENOMEM));

                if (copyout(&se->se_bookmark, (char *)addr +
                    (*count - 1) * sizeof (zbookmark_t),
                    sizeof (zbookmark_t)) != 0)
                        return (SET_ERROR(EFAULT));

                *count -= 1;
        }
```

```
        return (0);
}
#endif

/*
 * Copy all known errors to userland as an array of bookmarks.  This is
 * actually a union of the on-disk last log and current log, as well as any
 * pending error requests.
 *
 * Because the act of reading the on-disk log could cause errors to be
 * generated, we have two separate locks: one for the error log and one for the
 * in-core error lists.  We only need the error list lock to log and error, so
 * we grab the error log lock while we read the on-disk logs, and only pick up
 * the error list lock when we are finished.
 */
int
spa_get_errlog(spa_t *spa, void *uaddr, size_t *count)
{
        int ret = 0;

#ifdef _KERNEL
        mutex_enter(&spa->spa_errlog_lock);

        ret = process_error_log(spa, spa->spa_errlog_scrub, uaddr, count);

        if (!ret && !spa->spa_scrub_finished)
                ret = process_error_log(spa, spa->spa_errlog_last, uaddr,
                    count);

        mutex_enter(&spa->spa_errlist_lock);
        if (!ret)
                ret = process_error_list(&spa->spa_errlist_scrub, uaddr,
                    count);
        if (!ret)
                ret = process_error_list(&spa->spa_errlist_last, uaddr,
                    count);
        mutex_exit(&spa->spa_errlist_lock);

        mutex_exit(&spa->spa_errlog_lock);
#endif

        return (ret);
}

/*
 * Called when a scrub completes.  This simply set a bit which tells which AVL
 * tree to add new errors.  spa_errlog_sync() is responsible for actually
 * syncing the changes to the underlying objects.
 */
void
spa_errlog_rotate(spa_t *spa)
{
        mutex_enter(&spa->spa_errlist_lock);
        spa->spa_scrub_finished = B_TRUE;
        mutex_exit(&spa->spa_errlist_lock);
}

/*
 * Discard any pending errors from the spa_t.  Called when unloading a faulted
 * pool, as the errors encountered during the open cannot be synced to disk.
 */
void
spa_errlog_drain(spa_t *spa)
{
        spa_error_entry_t *se;
        void *cookie;

        mutex_enter(&spa->spa_errlist_lock);

        cookie = NULL;
        while ((se = avl_destroy_nodes(&spa->spa_errlist_last,
            &cookie)) != NULL)
                kmem_free(se, sizeof (spa_error_entry_t));
        cookie = NULL;
        while ((se = avl_destroy_nodes(&spa->spa_errlist_scrub,
            &cookie)) != NULL)
                kmem_free(se, sizeof (spa_error_entry_t));

        mutex_exit(&spa->spa_errlist_lock);
}

/*
 * Process a list of errors into the current on-disk log.
 */
static void
sync_error_list(spa_t *spa, avl_tree_t *t, uint64_t *obj, dmu_tx_t *tx)
{
        spa_error_entry_t *se;
        char buf[64];
        void *cookie;

        if (avl_numnodes(t) != 0) {
                /* create log if necessary */
                if (*obj == 0)
                        *obj = zap_create(spa->spa_meta_objset,
                            DMU_OT_ERROR_LOG, DMU_OT_NONE,
                            0, tx);

                /* add errors to the current log */
                for (se = avl_first(t); se != NULL; se = AVL_NEXT(t, se)) {
                        char *name = se->se_name ? se->se_name : "";

                        bookmark_to_name(&se->se_bookmark, buf, sizeof (buf));

                        (void) zap_update(spa->spa_meta_objset,
                            *obj, buf, 1, strlen(name) + 1, name, tx);
                }

                /* purge the error list */
                cookie = NULL;
                while ((se = avl_destroy_nodes(t, &cookie)) != NULL)
                        kmem_free(se, sizeof (spa_error_entry_t));
        }
}

/*
 * Sync the error log out to disk.  This is a little tricky because the act of
 * writing the error log requires the spa_errlist_lock.  So, we need to lock the
 * error lists, take a copy of the lists, and then reinitialize them.  Then, we
 * drop the error list lock and take the error log lock, at which point we
 * do the errlog processing.  Then, if we encounter an I/O error during this
 * process, we can successfully add the error to the list.  Note that this will
 * result in the perpetual recycling of errors, but it is an unlikely situation
 * and not a performance critical operation.
 */
void
spa_errlog_sync(spa_t *spa, uint64_t txg)
{
        dmu_tx_t *tx;
        avl_tree_t scrub, last;
        int scrub_finished;
```

```

        mutex_enter(&spa->spa_errlist_lock);

        /*
         * Bail out early under normal circumstances.
         */
        if (avl_numnodes(&spa->spa_errlist_scrub) == 0 &&
            avl_numnodes(&spa->spa_errlist_last) == 0 &&
            !spa->spa_scrub_finished) {
                mutex_exit(&spa->spa_errlist_lock);
                return;
        }

        spa_get_errlists(spa, &last, &scrub);
        scrub_finished = spa->spa_scrub_finished;
        spa->spa_scrub_finished = B_FALSE;

        mutex_exit(&spa->spa_errlist_lock);
        mutex_enter(&spa->spa_errlog_lock);

        tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);

        /*
         * Sync out the current list of errors.
         */
        sync_error_list(spa, &last, &spa->spa_errlog_last, tx);

        /*
         * Rotate the log if necessary.
         */
        if (scrub_finished) {
                if (spa->spa_errlog_last != 0)
                        VERIFY(dmu_object_free(spa->spa_meta_objset,
                            spa->spa_errlog_last, tx) == 0);
                spa->spa_errlog_last = spa->spa_errlog_scrub;
                spa->spa_errlog_scrub = 0;

                sync_error_list(spa, &scrub, &spa->spa_errlog_last, tx);
        }

        /*
         * Sync out any pending scrub errors.
         */
        sync_error_list(spa, &scrub, &spa->spa_errlog_scrub, tx);

        /*
         * Update the MOS to reflect the new values.
         */
        (void) zap_update(spa->spa_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
            DMU_POOL_ERRLOG_LAST, sizeof (uint64_t), 1,
            &spa->spa_errlog_last, tx);
        (void) zap_update(spa->spa_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
            DMU_POOL_ERRLOG_SCRUB, sizeof (uint64_t), 1,
            &spa->spa_errlog_scrub, tx);

        dmu_tx_commit(tx);

        mutex_exit(&spa->spa_errlog_lock);
}
```

```
**********************************************************
   33995 Tue Apr 23 15:35:05 2013
new/usr/src/uts/common/fs/zfs/zap.c
3743 zfs needs a refcount audit
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
static int
zap_table_load(zap_t *zap, zap_table_phys_t *tbl, uint64_t idx, uint64_t *valp)
{
        uint64_t blk, off;
        int err;
        dmu_buf_t *db;
        int bs = FZAP_BLOCK_SHIFT(zap);

        ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));

        blk = idx >> (bs-3);
        off = idx & ((1<<(bs-3))-1);

        err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
            (tbl->zt_blk + blk) << bs, FTAG, &db, DMU_READ_NO_PREFETCH);
        if (err)
                return (err);
        *valp = ((uint64_t *)db->db_data)[off];
        dmu_buf_rele(db, FTAG);

        if (tbl->zt_nextblk != 0) {
                /*
                 * read the nextblk for the sake of i/o error checking,
                 * so that zap_table_load() will catch errors for
                 * zap_table_store.
                 */
                blk = (idx*2) >> (bs-3);

                err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
                    (tbl->zt_nextblk + blk) << bs, FTAG, &db,
                    DMU_READ_NO_PREFETCH);
                if (err == 0)
#endif /* ! codereview */
                        dmu_buf_rele(db, FTAG);
        }
        return (err);
}

/*
 * Routines for growing the ptrtbl.
 */

static void
zap_ptrtbl_transfer(const uint64_t *src, uint64_t *dst, int n)
{
        int i;
        for (i = 0; i < n; i++) {
                uint64_t lb = src[i];
                dst[2*i+0] = lb;
                dst[2*i+1] = lb;
        }
}

static int
zap_grow_ptrtbl(zap_t *zap, dmu_tx_t *tx)
{
```

```
        /*
         * The pointer table should never use more hash bits than we
         * have (otherwise we'd be using useless zero bits to index it).
         * If we are within 2 bits of running out, stop growing, since
         * this is already an aberrant condition.
         */
        if (zap->zap_f.zap_phys->zap_ptrtbl.zt_shift >= zap_hashbits(zap) - 2)
                return (SET_ERROR(ENOSPC));

        if (zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks == 0) {
                /*
                 * We are outgrowing the "embedded" ptrtbl (the one
                 * stored in the header block).  Give it its own entire
                 * block, which will double the size of the ptrtbl.
                 */
                uint64_t newblk;
                dmu_buf_t *db_new;
                int err;

                ASSERT3U(zap->zap_f.zap_phys->zap_ptrtbl.zt_shift, ==,
                    ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
                ASSERT0(zap->zap_f.zap_phys->zap_ptrtbl.zt_blk);

                newblk = zap_allocate_blocks(zap, 1);
                err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
                    newblk << FZAP_BLOCK_SHIFT(zap), FTAG, &db_new,
                    DMU_READ_NO_PREFETCH);
                if (err)
                        return (err);
                dmu_buf_will_dirty(db_new, tx);
                zap_ptrtbl_transfer(&ZAP_EMBEDDED_PTRTBL_ENT(zap, 0),
                    db_new->db_data, 1 << ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
                dmu_buf_rele(db_new, FTAG);

                zap->zap_f.zap_phys->zap_ptrtbl.zt_blk = newblk;
                zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks = 1;
                zap->zap_f.zap_phys->zap_ptrtbl.zt_shift++;

                ASSERT3U(1ULL << zap->zap_f.zap_phys->zap_ptrtbl.zt_shift, ==,
                    zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks <<
                    (FZAP_BLOCK_SHIFT(zap)-3));

                return (0);
        } else {
                return (zap_table_grow(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
                    zap_ptrtbl_transfer, tx));
        }
}

static void
zap_increment_num_entries(zap_t *zap, int delta, dmu_tx_t *tx)
{
        dmu_buf_will_dirty(zap->zap_dbuf, tx);
        mutex_enter(&zap->zap_f.zap_num_entries_mtx);
        ASSERT(delta > 0 || zap->zap_f.zap_phys->zap_num_entries >= -delta);
        zap->zap_f.zap_phys->zap_num_entries += delta;
        mutex_exit(&zap->zap_f.zap_num_entries_mtx);
}

static uint64_t
zap_allocate_blocks(zap_t *zap, int nblocks)
{
        uint64_t newblk;
        ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
        newblk = zap->zap_f.zap_phys->zap_freeblk;
        zap->zap_f.zap_phys->zap_freeblk += nblocks;
        return (newblk);
}

static zap_leaf_t *
zap_create_leaf(zap_t *zap, dmu_tx_t *tx)
{
        void *winner;
        zap_leaf_t *l = kmem_alloc(sizeof (zap_leaf_t), KM_SLEEP);

        ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));

        rw_init(&l->l_rwlock, 0, 0, 0);
        rw_enter(&l->l_rwlock, RW_WRITER);
        l->l_blkid = zap_allocate_blocks(zap, 1);
        l->l_dbuf = NULL;
        l->l_phys = NULL;

        VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
            l->l_blkid << FZAP_BLOCK_SHIFT(zap), NULL, &l->l_dbuf,
            DMU_READ_NO_PREFETCH));
        winner = dmu_buf_set_user(l->l_dbuf, l, &l->l_phys, zap_leaf_pageout);
        ASSERT(winner == NULL);
        dmu_buf_will_dirty(l->l_dbuf, tx);

        zap_leaf_init(l, zap->zap_normflags != 0);

        zap->zap_f.zap_phys->zap_num_leafs++;

        return (l);
}

int
fzap_count(zap_t *zap, uint64_t *count)
{
        ASSERT(!zap->zap_ismicro);
        mutex_enter(&zap->zap_f.zap_num_entries_mtx); /* unnecessary */
        *count = zap->zap_f.zap_phys->zap_num_entries;
        mutex_exit(&zap->zap_f.zap_num_entries_mtx);
        return (0);
}

/*
 * Routines for obtaining zap_leaf_t’s
 */

void
zap_put_leaf(zap_leaf_t *l)
{
        rw_exit(&l->l_rwlock);
        dmu_buf_rele(l->l_dbuf, NULL);
}

_NOTE(ARGSUSED(0))
static void
zap_leaf_pageout(dmu_buf_t *db, void *vl)
{
        zap_leaf_t *l = vl;

        rw_destroy(&l->l_rwlock);
        kmem_free(l, sizeof (zap_leaf_t));
}

static zap_leaf_t *
zap_open_leaf(uint64_t blkid, dmu_buf_t *db)
{
        zap_leaf_t *l, *winner;
```

```

        ASSERT(blkid != 0);

        l = kmem_alloc(sizeof (zap_leaf_t), KM_SLEEP);
        rw_init(&l->l_rwlock, 0, 0, 0);
        rw_enter(&l->l_rwlock, RW_WRITER);
        l->l_blkid = blkid;
        l->l_bs = highbit(db->db_size)-1;
        l->l_dbuf = db;
        l->l_phys = NULL;

        winner = dmu_buf_set_user(db, l, &l->l_phys, zap_leaf_pageout);

        rw_exit(&l->l_rwlock);
        if (winner != NULL) {
                /* someone else set it first */
                zap_leaf_pageout(NULL, l);
                l = winner;
        }

        /*
         * lhr_pad was previously used for the next leaf in the leaf
         * chain.  There should be no chained leafs (as we have removed
         * support for them).
         */
        ASSERT0(l->l_phys->l_hdr.lh_pad1);

        /*
         * There should be more hash entries than there can be
         * chunks to put in the hash table
         */
        ASSERT3U(ZAP_LEAF_HASH_NUMENTRIES(l), >, ZAP_LEAF_NUMCHUNKS(l) / 3);

        /* The chunks should begin at the end of the hash table */
        ASSERT3P(&ZAP_LEAF_CHUNK(l, 0), ==,
            &l->l_phys->l_hash[ZAP_LEAF_HASH_NUMENTRIES(l)]);

        /* The chunks should end at the end of the block */
        ASSERT3U((uintptr_t)&ZAP_LEAF_CHUNK(l, ZAP_LEAF_NUMCHUNKS(l)) -
            (uintptr_t)l->l_phys, ==, l->l_dbuf->db_size);

        return (l);
}

static int
zap_get_leaf_byblk(zap_t *zap, uint64_t blkid, dmu_tx_t *tx, krw_t lt,
    zap_leaf_t **lp)
{
        dmu_buf_t *db;
        zap_leaf_t *l;
        int bs = FZAP_BLOCK_SHIFT(zap);
        int err;

        ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));

        err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
            blkid << bs, NULL, &db, DMU_READ_NO_PREFETCH);
        if (err)
                return (err);

        ASSERT3U(db->db_object, ==, zap->zap_object);
        ASSERT3U(db->db_offset, ==, blkid << bs);
        ASSERT3U(db->db_size, ==, 1 << bs);
        ASSERT(blkid != 0);

        l = dmu_buf_get_user(db);
        if (l == NULL)
                l = zap_open_leaf(blkid, db);

        rw_enter(&l->l_rwlock, lt);
        /*
         * Must lock before dirtying, otherwise l->l_phys could change,
         * causing ASSERT below to fail.
         */
        if (lt == RW_WRITER)
                dmu_buf_will_dirty(db, tx);
        ASSERT3U(l->l_blkid, ==, blkid);
        ASSERT3P(l->l_dbuf, ==, db);
        ASSERT3P(l->l_phys, ==, l->l_dbuf->db_data);
        ASSERT3U(l->l_phys->l_hdr.lh_block_type, ==, ZBT_LEAF);
        ASSERT3U(l->l_phys->l_hdr.lh_magic, ==, ZAP_LEAF_MAGIC);

        *lp = l;
        return (0);
}

static int
zap_idx_to_blk(zap_t *zap, uint64_t idx, uint64_t *valp)
{
        ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));

        if (zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks == 0) {
                ASSERT3U(idx, <,
                    (1ULL << zap->zap_f.zap_phys->zap_ptrtbl.zt_shift));
                *valp = ZAP_EMBEDDED_PTRTBL_ENT(zap, idx);
                return (0);
        } else {
                return (zap_table_load(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
                    idx, valp));
        }
}

static int
zap_set_idx_to_blk(zap_t *zap, uint64_t idx, uint64_t blk, dmu_tx_t *tx)
{
        ASSERT(tx != NULL);
        ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));

        if (zap->zap_f.zap_phys->zap_ptrtbl.zt_blk == 0) {
                ZAP_EMBEDDED_PTRTBL_ENT(zap, idx) = blk;
                return (0);
        } else {
                return (zap_table_store(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
                    idx, blk, tx));
        }
}

static int
zap_deref_leaf(zap_t *zap, uint64_t h, dmu_tx_t *tx, krw_t lt, zap_leaf_t **lp)
{
        uint64_t idx, blk;
        int err;

        ASSERT(zap->zap_dbuf == NULL ||
            zap->zap_f.zap_phys == zap->zap_dbuf->db_data);
        ASSERT3U(zap->zap_f.zap_phys->zap_magic, ==, ZAP_MAGIC);
        idx = ZAP_HASH_IDX(h, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
        err = zap_idx_to_blk(zap, idx, &blk);
        if (err != 0)
                return (err);
        err = zap_get_leaf_byblk(zap, blk, tx, lt, lp);
```

```
        ASSERT(err || ZAP_HASH_IDX(h, (*lp)->l_phys->l_hdr.lh_prefix_len) ==
            (*lp)->l_phys->l_hdr.lh_prefix);
        return (err);
}

static int
zap_expand_leaf(zap_name_t *zn, zap_leaf_t *l, dmu_tx_t *tx, zap_leaf_t **lp)
{
        zap_t *zap = zn->zn_zap;
        uint64_t hash = zn->zn_hash;
        zap_leaf_t *nl;
        int prefix_diff, i, err;
        uint64_t sibling;
        int old_prefix_len = l->l_phys->l_hdr.lh_prefix_len;

        ASSERT3U(old_prefix_len, <=, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
        ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));

        ASSERT3U(ZAP_HASH_IDX(hash, old_prefix_len), ==,
            l->l_phys->l_hdr.lh_prefix);

        if (zap_tryupgradedir(zap, tx) == 0 ||
            old_prefix_len == zap->zap_f.zap_phys->zap_ptrtbl.zt_shift) {
                /* We failed to upgrade, or need to grow the pointer table */
                objset_t *os = zap->zap_objset;
                uint64_t object = zap->zap_object;

                zap_put_leaf(l);
                zap_unlockdir(zap);
                err = zap_lockdir(os, object, tx, RW_WRITER,
                    FALSE, FALSE, &zn->zn_zap);
                zap = zn->zn_zap;
                if (err)
                        return (err);
                ASSERT(!zap->zap_ismicro);

                while (old_prefix_len ==
                    zap->zap_f.zap_phys->zap_ptrtbl.zt_shift) {
                        err = zap_grow_ptrtbl(zap, tx);
                        if (err)
                                return (err);
                }

                err = zap_deref_leaf(zap, hash, tx, RW_WRITER, &l);
                if (err)
                        return (err);

                if (l->l_phys->l_hdr.lh_prefix_len != old_prefix_len) {
                        /* it split while our locks were down */
                        *lp = l;
                        return (0);
                }
        }
        ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
        ASSERT3U(old_prefix_len, <, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
        ASSERT3U(ZAP_HASH_IDX(hash, old_prefix_len), ==,
            l->l_phys->l_hdr.lh_prefix);

        prefix_diff = zap->zap_f.zap_phys->zap_ptrtbl.zt_shift -
            (old_prefix_len + 1);
        sibling = (ZAP_HASH_IDX(hash, old_prefix_len + 1) | 1) << prefix_diff;

        /* check for i/o errors before doing zap_leaf_split */
        for (i = 0; i < (1ULL<<prefix_diff); i++) {
                uint64_t blk;
                err = zap_idx_to_blk(zap, sibling+i, &blk);
                if (err)
                        return (err);
                ASSERT3U(blk, ==, l->l_blkid);
        }

        nl = zap_create_leaf(zap, tx);
        zap_leaf_split(l, nl, zap->zap_normflags != 0);

        /* set sibling pointers */
        for (i = 0; i < (1ULL << prefix_diff); i++) {
                err = zap_set_idx_to_blk(zap, sibling+i, nl->l_blkid, tx);
                ASSERT0(err); /* we checked for i/o errors above */
        }

        if (hash & (1ULL << (64 - l->l_phys->l_hdr.lh_prefix_len))) {
                /* we want the sibling */
                zap_put_leaf(l);
                *lp = nl;
        } else {
                zap_put_leaf(nl);
                *lp = l;
        }

        return (0);
}

static void
zap_put_leaf_maybe_grow_ptrtbl(zap_name_t *zn, zap_leaf_t *l, dmu_tx_t *tx)
{
        zap_t *zap = zn->zn_zap;
        int shift = zap->zap_f.zap_phys->zap_ptrtbl.zt_shift;
        int leaffull = (l->l_phys->l_hdr.lh_prefix_len == shift &&
            l->l_phys->l_hdr.lh_nfree < ZAP_LEAF_LOW_WATER);

        zap_put_leaf(l);

        if (leaffull || zap->zap_f.zap_phys->zap_ptrtbl.zt_nextblk) {
                int err;

                /*
                 * We are in the middle of growing the pointer table, or
                 * this leaf will soon make us grow it.
                 */
                if (zap_tryupgradedir(zap, tx) == 0) {
                        objset_t *os = zap->zap_objset;
                        uint64_t zapobj = zap->zap_object;

                        zap_unlockdir(zap);
                        err = zap_lockdir(os, zapobj, tx,
                            RW_WRITER, FALSE, FALSE, &zn->zn_zap);
                        zap = zn->zn_zap;
                        if (err)
                                return;
                }

                /* could have finished growing while our locks were down */
                if (zap->zap_f.zap_phys->zap_ptrtbl.zt_shift == shift)
                        (void) zap_grow_ptrtbl(zap, tx);
        }
}

static int
fzap_checkname(zap_name_t *zn)
{
        if (zn->zn_key_orig_numints * zn->zn_key_intlen > ZAP_MAXNAMELEN)
```

```
                return (SET_ERROR(ENAMETOOLONG));
        return (0);
}

static int
fzap_checksize(uint64_t integer_size, uint64_t num_integers)
{
        /* Only integer sizes supported by C */
        switch (integer_size) {
        case 1:
        case 2:
        case 4:
        case 8:
                break;
        default:
                return (SET_ERROR(EINVAL));
        }

        if (integer_size * num_integers > ZAP_MAXVALUELEN)
                return (E2BIG);

        return (0);
}

static int
fzap_check(zap_name_t *zn, uint64_t integer_size, uint64_t num_integers)
{
        int err;

        if ((err = fzap_checkname(zn)) != 0)
                return (err);
        return (fzap_checksize(integer_size, num_integers));
}

/*
 * Routines for manipulating attributes.
 */
int
fzap_lookup(zap_name_t *zn,
    uint64_t integer_size, uint64_t num_integers, void *buf,
    char *realname, int rn_len, boolean_t *ncp)
{
        zap_leaf_t *l;
        int err;
        zap_entry_handle_t zeh;

        if ((err = fzap_checkname(zn)) != 0)
                return (err);

        err = zap_deref_leaf(zn->zn_zap, zn->zn_hash, NULL, RW_READER, &l);
        if (err != 0)
                return (err);
        err = zap_leaf_lookup(l, zn, &zeh);
        if (err == 0) {
                if ((err = fzap_checksize(integer_size, num_integers)) != 0) {
                        zap_put_leaf(l);
                        return (err);
                }

                err = zap_entry_read(&zeh, integer_size, num_integers, buf);
                (void) zap_entry_read_name(zn->zn_zap, &zeh, rn_len, realname);
                if (ncp) {
                        *ncp = zap_entry_normalization_conflict(&zeh,
                            zn, NULL, zn->zn_zap);
                }
        }
        zap_put_leaf(l);
        return (err);
}

int
fzap_add_cd(zap_name_t *zn,
    uint64_t integer_size, uint64_t num_integers,
    const void *val, uint32_t cd, dmu_tx_t *tx)
{
        zap_leaf_t *l;
        int err;
        zap_entry_handle_t zeh;
        zap_t *zap = zn->zn_zap;

        ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));
        ASSERT(!zap->zap_ismicro);
        ASSERT(fzap_check(zn, integer_size, num_integers) == 0);

        err = zap_deref_leaf(zap, zn->zn_hash, tx, RW_WRITER, &l);
        if (err != 0)
                return (err);
retry:
        err = zap_leaf_lookup(l, zn, &zeh);
        if (err == 0) {
                err = SET_ERROR(EEXIST);
                goto out;
        }
        if (err != ENOENT)
                goto out;

        err = zap_entry_create(l, zn, cd,
            integer_size, num_integers, val, &zeh);

        if (err == 0) {
                zap_increment_num_entries(zap, 1, tx);
        } else if (err == EAGAIN) {
                err = zap_expand_leaf(zn, l, tx, &l);
                zap = zn->zn_zap;       /* zap_expand_leaf() may change zap */
                if (err == 0)
                        goto retry;
        }

out:
        if (zap != NULL)
                zap_put_leaf_maybe_grow_ptrtbl(zn, l, tx);
        return (err);
}

int
fzap_add(zap_name_t *zn,
    uint64_t integer_size, uint64_t num_integers,
    const void *val, dmu_tx_t *tx)
{
        int err = fzap_check(zn, integer_size, num_integers);
        if (err != 0)
                return (err);

        return (fzap_add_cd(zn, integer_size, num_integers,
            val, ZAP_NEED_CD, tx));
}

int
fzap_update(zap_name_t *zn,
    int integer_size, uint64_t num_integers, const void *val, dmu_tx_t *tx)
{
```

```
        zap_leaf_t *l;
        int err, create;
        zap_entry_handle_t zeh;
        zap_t *zap = zn->zn_zap;

        ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));
        err = fzap_check(zn, integer_size, num_integers);
        if (err != 0)
                return (err);

        err = zap_deref_leaf(zap, zn->zn_hash, tx, RW_WRITER, &l);
        if (err != 0)
                return (err);
retry:
        err = zap_leaf_lookup(l, zn, &zeh);
        create = (err == ENOENT);
        ASSERT(err == 0 || err == ENOENT);

        if (create) {
                err = zap_entry_create(l, zn, ZAP_NEED_CD,
                    integer_size, num_integers, val, &zeh);
                if (err == 0)
                        zap_increment_num_entries(zap, 1, tx);
        } else {
                err = zap_entry_update(&zeh, integer_size, num_integers, val);
        }

        if (err == EAGAIN) {
                err = zap_expand_leaf(zn, l, tx, &l);
                zap = zn->zn_zap;       /* zap_expand_leaf() may change zap */
                if (err == 0)
                        goto retry;
        }

        if (zap != NULL)
                zap_put_leaf_maybe_grow_ptrtbl(zn, l, tx);
        return (err);
}

int
fzap_length(zap_name_t *zn,
    uint64_t *integer_size, uint64_t *num_integers)
{
        zap_leaf_t *l;
        int err;
        zap_entry_handle_t zeh;

        err = zap_deref_leaf(zn->zn_zap, zn->zn_hash, NULL, RW_READER, &l);
        if (err != 0)
                return (err);
        err = zap_leaf_lookup(l, zn, &zeh);
        if (err != 0)
                goto out;

        if (integer_size)
                *integer_size = zeh.zeh_integer_size;
        if (num_integers)
                *num_integers = zeh.zeh_num_integers;
out:
        zap_put_leaf(l);
        return (err);
}

int
fzap_remove(zap_name_t *zn, dmu_tx_t *tx)
{
        zap_leaf_t *l;
        int err;
        zap_entry_handle_t zeh;

        err = zap_deref_leaf(zn->zn_zap, zn->zn_hash, tx, RW_WRITER, &l);
        if (err != 0)
                return (err);
        err = zap_leaf_lookup(l, zn, &zeh);
        if (err == 0) {
                zap_entry_remove(&zeh);
                zap_increment_num_entries(zn->zn_zap, -1, tx);
        }
        zap_put_leaf(l);
        return (err);
}

void
fzap_prefetch(zap_name_t *zn)
{
        uint64_t idx, blk;
        zap_t *zap = zn->zn_zap;
        int bs;

        idx = ZAP_HASH_IDX(zn->zn_hash,
            zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
        if (zap_idx_to_blk(zap, idx, &blk) != 0)
                return;
        bs = FZAP_BLOCK_SHIFT(zap);
        dmu_prefetch(zap->zap_objset, zap->zap_object, blk << bs, 1 << bs);
}

/*
 * Helper functions for consumers.
 */

uint64_t
zap_create_link(objset_t *os, dmu_object_type_t ot, uint64_t parent_obj,
    const char *name, dmu_tx_t *tx)
{
        uint64_t new_obj;

        VERIFY((new_obj = zap_create(os, ot, DMU_OT_NONE, 0, tx)) > 0);
        VERIFY(zap_add(os, parent_obj, name, sizeof (uint64_t), 1, &new_obj,
            tx) == 0);

        return (new_obj);
}

int
zap_value_search(objset_t *os, uint64_t zapobj, uint64_t value, uint64_t mask,
    char *name)
{
        zap_cursor_t zc;
        zap_attribute_t *za;
        int err;

        if (mask == 0)
                mask = -1ULL;

        za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
        for (zap_cursor_init(&zc, os, zapobj);
            (err = zap_cursor_retrieve(&zc, za)) == 0;
            zap_cursor_advance(&zc)) {
                if ((za->za_first_integer & mask) == (value & mask)) {
                        (void) strcpy(name, za->za_name);
                        break;
```

```
                }
        }
        zap_cursor_fini(&zc);
        kmem_free(za, sizeof (zap_attribute_t));
        return (err);
}

int
zap_join(objset_t *os, uint64_t fromobj, uint64_t intoobj, dmu_tx_t *tx)
{
        zap_cursor_t zc;
        zap_attribute_t za;
        int err;

        err = 0;
#endif /* ! codereview */
        for (zap_cursor_init(&zc, os, fromobj);
            zap_cursor_retrieve(&zc, &za) == 0;
            (void) zap_cursor_advance(&zc)) {
                if (za.za_integer_length != 8 || za.za_num_integers != 1) {
                        err = SET_ERROR(EINVAL);
                        break;
                }
                if (za.za_integer_length != 8 || za.za_num_integers != 1)
                        return (SET_ERROR(EINVAL));
                err = zap_add(os, intoobj, za.za_name,
                    8, 1, &za.za_first_integer, tx);
                if (err)
                        break;
                        return (err);
        }
        zap_cursor_fini(&zc);
        return (err);
        return (0);
}

int
zap_join_key(objset_t *os, uint64_t fromobj, uint64_t intoobj,
    uint64_t value, dmu_tx_t *tx)
{
        zap_cursor_t zc;
        zap_attribute_t za;
        int err;

        err = 0;
#endif /* ! codereview */
        for (zap_cursor_init(&zc, os, fromobj);
            zap_cursor_retrieve(&zc, &za) == 0;
            (void) zap_cursor_advance(&zc)) {
                if (za.za_integer_length != 8 || za.za_num_integers != 1) {
                        err = SET_ERROR(EINVAL);
                        break;
                }
                if (za.za_integer_length != 8 || za.za_num_integers != 1)
                        return (SET_ERROR(EINVAL));
                err = zap_add(os, intoobj, za.za_name,
                    8, 1, &value, tx);
                if (err)
                        break;
                        return (err);
        }
        zap_cursor_fini(&zc);
        return (err);
        return (0);
}
```

```
int
zap_join_increment(objset_t *os, uint64_t fromobj, uint64_t intoobj,
    dmu_tx_t *tx)
{
        zap_cursor_t zc;
        zap_attribute_t za;
        int err;

        err = 0;
#endif /* ! codereview */
        for (zap_cursor_init(&zc, os, fromobj);
            zap_cursor_retrieve(&zc, &za) == 0;
            (void) zap_cursor_advance(&zc)) {
                uint64_t delta = 0;

                if (za.za_integer_length != 8 || za.za_num_integers != 1) {
                        err = SET_ERROR(EINVAL);
                        break;
                }
                if (za.za_integer_length != 8 || za.za_num_integers != 1)
                        return (SET_ERROR(EINVAL));

                err = zap_lookup(os, intoobj, za.za_name, 8, 1, &delta);
                if (err != 0 && err != ENOENT)
                        break;
                        return (err);
                delta += za.za_first_integer;
                err = zap_update(os, intoobj, za.za_name, 8, 1, &delta, tx);
                if (err)
                        break;
                        return (err);
        }
        zap_cursor_fini(&zc);
        return (err);
        return (0);
}
```

______*unchanged_portion_omitted_*