

\*\*\*\*\*

80627 Thu May 16 14:21:05 2013

new/usr/src/uts/common/fs/zfs/dsl\_dataset.c

3743 zfs needs a refcount audit

Submitted by: Will Andrews <willa@spectrallogic.com>

Submitted by: Justin Gibbs <justing@spectrallogic.com>

Reviewed by: Matthew Ahrens <mahrens@delphix.com>

Reviewed by: Eric Schrock <eric.schrock@delphix.com>

Reviewed by: George Wilson <george.wilson@delphix.com>

\*\*\*\*\*

\_\_\_\_\_unchanged\_portion\_omitted\_\_\_\_\_

```

341 int
342 dsl_dataset_hold_obj(dsl_pool_t *dp, uint64_t dsobj, void *tag,
343     dsl_dataset_t **dsp)
344 {
345     objset_t *mos = dp->dp_meta_objset;
346     dmu_buf_t *dbuf;
347     dsl_dataset_t *ds;
348     int err;
349     dmu_object_info_t doi;
350
351     ASSERT(dsl_pool_config_held(dp));
352
353     err = dmu_bonus_hold(mos, dsobj, tag, &dbuf);
354     if (err != 0)
355         return (err);
356
357     /* Make sure dsobj has the correct object type. */
358     dmu_object_info_from_db(dbuf, &doi);
359     if (doi.doi_type != DMU_OT_DSL_DATASET) {
360         dmu_buf_rele(dbuf, tag);
361         if (doi.doi_type != DMU_OT_DSL_DATASET)
362             return (SET_ERROR(EINVAL));
363     }
364 #endif /* ! codereview */
365
366     ds = dmu_buf_get_user(dbuf);
367     if (ds == NULL) {
368         dsl_dataset_t *winner = NULL;
369
370         ds = kmem_zalloc(sizeof (dsl_dataset_t), KM_SLEEP);
371         ds->ds_dbuf = dbuf;
372         ds->ds_object = dsobj;
373         ds->ds_phys = dbuf->db_data;
374
375         mutex_init(&ds->ds_lock, NULL, MUTEX_DEFAULT, NULL);
376         mutex_init(&ds->ds_opening_lock, NULL, MUTEX_DEFAULT, NULL);
377         mutex_init(&ds->ds_sendstream_lock, NULL, MUTEX_DEFAULT, NULL);
378         refcount_create(&ds->ds_longholds);
379
380         bplist_create(&ds->ds_pending_deadlist);
381         dsl_deadlist_open(&ds->ds_deadlist,
382             mos, ds->ds_phys->ds_deadlist_obj);
383
384         list_create(&ds->ds_sendstreams, sizeof (dmu_sendarg_t),
385             offsetof(dmu_sendarg_t, dsa_link));
386
387         if (err == 0) {
388             err = dsl_dir_hold_obj(dp,
389                 ds->ds_phys->ds_dir_obj, NULL, ds, &ds->ds_dir);
390         }
391         if (err != 0) {
392             mutex_destroy(&ds->ds_lock);
393             mutex_destroy(&ds->ds_opening_lock);
394             refcount_destroy(&ds->ds_longholds);

```

```

394         bplist_destroy(&ds->ds_pending_deadlist);
395         dsl_deadlist_close(&ds->ds_deadlist);
396         kmem_free(ds, sizeof (dsl_dataset_t));
397         dmu_buf_rele(dbuf, tag);
398         return (err);
399     }
400
401     if (!dsl_dataset_is_snapshot(ds)) {
402         ds->ds_snapname[0] = '\0';
403         if (ds->ds_phys->ds_prev_snap_obj != 0) {
404             err = dsl_dataset_hold_obj(dp,
405                 ds->ds_phys->ds_prev_snap_obj,
406                 ds, &ds->ds_prev);
407         }
408     } else {
409         if (zfs_flags & ZFS_DEBUG_SNAPNAMES)
410             err = dsl_dataset_get_snapname(ds);
411         if (err == 0 && ds->ds_phys->ds_userrefs_obj != 0) {
412             err = zap_count(
413                 ds->ds_dir->dd_pool->dp_meta_objset,
414                 ds->ds_phys->ds_userrefs_obj,
415                 &ds->ds_userrefs);
416         }
417     }
418
419     if (err == 0 && !dsl_dataset_is_snapshot(ds)) {
420         err = dsl_prop_get_int(ds,
421             zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
422             &ds->ds_reserved);
423         if (err == 0) {
424             err = dsl_prop_get_int(ds,
425                 zfs_prop_to_name(ZFS_PROP_REFQUOTA),
426                 &ds->ds_quota);
427         }
428     } else {
429         ds->ds_reserved = ds->ds_quota = 0;
430     }
431
432     if (err != 0 || (winner = dmu_buf_set_user_ie(dbuf, ds,
433         &ds->ds_phys, dsl_dataset_evict)) != NULL) {
434         bplist_destroy(&ds->ds_pending_deadlist);
435         dsl_deadlist_close(&ds->ds_deadlist);
436         if (ds->ds_prev)
437             dsl_dataset_rele(ds->ds_prev, ds);
438         dsl_dir_rele(ds->ds_dir, ds);
439         mutex_destroy(&ds->ds_lock);
440         mutex_destroy(&ds->ds_opening_lock);
441         refcount_destroy(&ds->ds_longholds);
442         kmem_free(ds, sizeof (dsl_dataset_t));
443         if (err != 0) {
444             dmu_buf_rele(dbuf, tag);
445             return (err);
446         }
447         ds = winner;
448     } else {
449         ds->ds_fsid_guid =
450             unique_insert(ds->ds_phys->ds_fsid_guid);
451     }
452
453     ASSERT3P(ds->ds_dbuf, ==, dbuf);
454     ASSERT3P(ds->ds_phys, ==, dbuf->db_data);
455     ASSERT(ds->ds_phys->ds_prev_snap_obj != 0 ||
456         spa_version(dp->dp_spa) < SPA_VERSION_ORIGIN ||
457         dp->dp_origin_snap == NULL || ds == dp->dp_origin_snap);
458     *dsp = ds;
459     return (0);

```

```

460 }

462 int
463 dsl_dataset_hold(dsl_pool_t *dp, const char *name,
464 void *tag, dsl_dataset_t **dsp)
465 {
466     dsl_dir_t *dd;
467     const char *snapname;
468     uint64_t obj;
469     int err = 0;

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

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

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

486         if (*snapname++ != '@') {
487             dsl_dataset_rele(*dsp, tag);
488             dsl_dir_rele(dd, FTAG);
489             return (SET_ERROR(ENOENT));
490         }

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

498         if (err == 0) {
499             mutex_enter(&ds->ds_lock);
500             if (ds->ds_snapname[0] == 0)
501                 (void) strcpy(ds->ds_snapname, snapname,
502                     sizeof (ds->ds_snapname));
503             mutex_exit(&ds->ds_lock);
504             *dsp = ds;
505         }
506     }

508     dsl_dir_rele(dd, FTAG);
509     return (err);
510 }

512 int
513 dsl_dataset_own_obj(dsl_pool_t *dp, uint64_t dsobj,
514 void *tag, dsl_dataset_t **dsp)
515 {
516     int err = dsl_dataset_hold_obj(dp, dsobj, tag, dsp);
517     if (err != 0)
518         return (err);
519     if (!dsl_dataset_tryown(*dsp, tag)) {
520         dsl_dataset_rele(*dsp, tag);
521         *dsp = NULL;
522         return (SET_ERROR(EBUSY));
523     }
524     return (0);
525 }

```

```

527 int
528 dsl_dataset_own(dsl_pool_t *dp, const char *name,
529 void *tag, dsl_dataset_t **dsp)
530 {
531     int err = dsl_dataset_hold(dp, name, tag, dsp);
532     if (err != 0)
533         return (err);
534     if (!dsl_dataset_tryown(*dsp, tag)) {
535         dsl_dataset_rele(*dsp, tag);
536         return (SET_ERROR(EBUSY));
537     }
538     return (0);
539 }

541 /*
542  * See the comment above dsl_pool_hold() for details. In summary, a long
543  * hold is used to prevent destruction of a dataset while the pool hold
544  * is dropped, allowing other concurrent operations (e.g. spa_sync()).
545  *
546  * The dataset and pool must be held when this function is called. After it
547  * is called, the pool hold may be released while the dataset is still held
548  * and accessed.
549  */
550 void
551 dsl_dataset_long_hold(dsl_dataset_t *ds, void *tag)
552 {
553     ASSERT(dsl_pool_config_held(ds->ds_dir->dd_pool));
554     (void) refcount_add(&ds->ds_longholds, tag);
555 }

557 void
558 dsl_dataset_long_rele(dsl_dataset_t *ds, void *tag)
559 {
560     (void) refcount_remove(&ds->ds_longholds, tag);
561 }

563 /* Return B_TRUE if there are any long holds on this dataset. */
564 boolean_t
565 dsl_dataset_long_held(dsl_dataset_t *ds)
566 {
567     return (!refcount_is_zero(&ds->ds_longholds));
568 }

570 void
571 dsl_dataset_name(dsl_dataset_t *ds, char *name)
572 {
573     if (ds == NULL) {
574         (void) strcpy(name, "mos");
575     } else {
576         dsl_dir_name(ds->ds_dir, name);
577         VERIFY0(dsl_dataset_get_snapname(ds));
578         if (ds->ds_snapname[0]) {
579             (void) strcat(name, "@");
580             /*
581              * We use a "recursive" mutex so that we
582              * can call dprintf_ds() with ds_lock held.
583              */
584             if (!MUTEX_HELD(&ds->ds_lock)) {
585                 mutex_enter(&ds->ds_lock);
586                 (void) strcat(name, ds->ds_snapname);
587                 mutex_exit(&ds->ds_lock);
588             } else {
589                 (void) strcat(name, ds->ds_snapname);
590             }
591         }
592     }

```

```

592     }
593 }

595 static int
596 dsl_dataset_namelen(dsl_dataset_t *ds)
597 {
598     int result;

600     if (ds == NULL) {
601         result = 3;      /* "mos" */
602     } else {
603         result = dsl_dir_namelen(ds->ds_dir);
604         VERIFY0(dsl_dataset_get_snapname(ds));
605         if (ds->ds_snapname[0]) {
606             ++result;    /* adding one for the @-sign */
607             if (!MUTEX_HELD(&ds->ds_lock)) {
608                 mutex_enter(&ds->ds_lock);
609                 result += strlen(ds->ds_snapname);
610                 mutex_exit(&ds->ds_lock);
611             } else {
612                 result += strlen(ds->ds_snapname);
613             }
614         }
615     }

617     return (result);
618 }

620 void
621 dsl_dataset_rele(dsl_dataset_t *ds, void *tag)
622 {
623     dmu_buf_rele(ds->ds_dbuf, tag);
624 }

626 void
627 dsl_dataset_disown(dsl_dataset_t *ds, void *tag)
628 {
629     ASSERT(ds->ds_owner == tag && ds->ds_dbuf != NULL);

631     mutex_enter(&ds->ds_lock);
632     ds->ds_owner = NULL;
633     mutex_exit(&ds->ds_lock);
634     dsl_dataset_long_rele(ds, tag);
635     if (ds->ds_dbuf != NULL)
636         dsl_dataset_rele(ds, tag);
637     else
638         dsl_dataset_evict(NULL, ds);
639 }

641 boolean_t
642 dsl_dataset_tryown(dsl_dataset_t *ds, void *tag)
643 {
644     boolean_t gotit = FALSE;

646     mutex_enter(&ds->ds_lock);
647     if (ds->ds_owner == NULL && !DS_IS_INCONSISTENT(ds)) {
648         ds->ds_owner = tag;
649         dsl_dataset_long_hold(ds, tag);
650         gotit = TRUE;
651     }
652     mutex_exit(&ds->ds_lock);
653     return (gotit);
654 }

656 uint64_t
657 dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,

```

```

658     uint64_t flags, dmu_tx_t *tx)
659 {
660     dsl_pool_t *dp = dd->dd_pool;
661     dmu_buf_t *dbuf;
662     dsl_dataset_phys_t *dsphys;
663     uint64_t dsobj;
664     objset_t *mos = dp->dp_meta_objset;

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

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

674     dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
675                             DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
676     VERIFY0(dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
677     dmu_buf_will_dirty(dbuf, tx);
678     dsphys = dbuf->db_data;
679     bzero(dsphys, sizeof (dsl_dataset_phys_t));
680     dsphys->ds_dir_obj = dd->dd_object;
681     dsphys->ds_flags = flags;
682     dsphys->ds_fsuid_guid = unique_create();
683     (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
684                                   sizeof (dsphys->ds_guid));
685     dsphys->ds_snapnames_zapobj =
686         zap_create_norm(mos, U8_TEXTPREP_Toupper, DMU_OT_DSL_DS_SNAP_MAP,
687                        DMU_OT_NONE, 0, tx);
688     dsphys->ds_creation_time = gethrtime_sec();
689     dsphys->ds_creation_txg = tx->tx_txg == TXG_INITIAL ? 1 : tx->tx_txg;

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

696         dsphys->ds_prev_snap_obj = origin->ds_object;
697         dsphys->ds_prev_snap_txg =
698             origin->ds_phys->ds_creation_txg;
699         dsphys->ds_referenced_bytes =
700             origin->ds_phys->ds_referenced_bytes;
701         dsphys->ds_compressed_bytes =
702             origin->ds_phys->ds_compressed_bytes;
703         dsphys->ds_uncompressed_bytes =
704             origin->ds_phys->ds_uncompressed_bytes;
705         dsphys->ds_bp = origin->ds_phys->ds_bp;
706         dsphys->ds_flags |= origin->ds_phys->ds_flags;

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

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

717         if (spa_version(dp->dp_spa) >= SPA_VERSION_NEXT_CLONES) {
718             if (origin->ds_phys->ds_next_clones_obj == 0) {
719                 origin->ds_phys->ds_next_clones_obj =
720                     zap_create(mos,
721                                DMU_OT_NEXT_CLONES, DMU_OT_NONE, 0, tx);
722             }
723             VERIFY0(zap_add_int(mos,

```

```

724         origin->ds_phys->ds_next_clones_obj, dsobj, tx));
725     }
726
727     dmu_buf_will_dirty(dd->dd_dbuf, tx);
728     dd->dd_phys->dd_origin_obj = origin->ds_object;
729     if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
730         if (origin->ds_dir->dd_phys->dd_clones == 0) {
731             dmu_buf_will_dirty(origin->ds_dir->dd_dbuf, tx);
732             origin->ds_dir->dd_phys->dd_clones =
733                 zap_create(mos,
734                     DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
735         }
736         VERIFY0(zap_add_int(mos,
737             origin->ds_dir->dd_phys->dd_clones, dsobj, tx));
738     }
739 }
740
741 if (spa_version(dp->dp_spa) >= SPA_VERSION_UNIQUE_ACCURATE)
742     dsphys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;
743
744 dmu_buf_rele(dbuf, FTAG);
745
746 dmu_buf_will_dirty(dd->dd_dbuf, tx);
747 dd->dd_phys->dd_head_dataset_obj = dsobj;
748
749 return (dsobj);
750 }
751
752 static void
753 dsl_dataset_zero_zil(dsl_dataset_t *ds, dmu_tx_t *tx)
754 {
755     objset_t *os;
756
757     VERIFY0(dmu_objset_from_ds(ds, &os));
758     bzero(&os->os_zil_header, sizeof (os->os_zil_header));
759     dsl_dataset_dirty(ds, tx);
760 }
761
762 uint64_t
763 dsl_dataset_create_sync(dsl_dir_t *pdd, const char *lastname,
764     dsl_dataset_t *origin, uint64_t flags, cred_t *cr, dmu_tx_t *tx)
765 {
766     dsl_pool_t *dp = pdd->dd_pool;
767     uint64_t dsobj, ddoobj;
768     dsl_dir_t *dd;
769
770     ASSERT(dmu_tx_is_syncing(tx));
771     ASSERT(lastname[0] != '@');
772
773     ddoobj = dsl_dir_create_sync(dp, pdd, lastname, tx);
774     VERIFY0(dsl_dir_hold_obj(dp, ddoobj, lastname, FTAG, &dd));
775
776     dsobj = dsl_dataset_create_sync_dd(dd, origin,
777         flags & ~DS_CREATE_FLAG_NODIRTY, tx);
778
779     dsl_deleg_set_create_perms(dd, tx, cr);
780
781     dsl_dir_rele(dd, FTAG);
782
783     /*
784      * If we are creating a clone, make sure we zero out any stale
785      * data from the origin snapshots zil header.
786      */
787     if (origin != NULL && !(flags & DS_CREATE_FLAG_NODIRTY)) {
788         dsl_dataset_t *ds;

```

```

790         VERIFY0(dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
791         dsl_dataset_zero_zil(ds, tx);
792         dsl_dataset_rele(ds, FTAG);
793     }
794
795     return (dsobj);
796 }
797
798 /*
799  * The unique space in the head dataset can be calculated by subtracting
800  * the space used in the most recent snapshot, that is still being used
801  * in this file system, from the space currently in use. To figure out
802  * the space in the most recent snapshot still in use, we need to take
803  * the total space used in the snapshot and subtract out the space that
804  * has been freed up since the snapshot was taken.
805  */
806 void
807 dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds)
808 {
809     uint64_t mrs_used;
810     uint64_t dlused, dlcomp, dluncomp;
811
812     ASSERT(!dsl_dataset_is_snapshot(ds));
813
814     if (ds->ds_phys->ds_prev_snap_obj != 0)
815         mrs_used = ds->ds_prev->ds_phys->ds_referenced_bytes;
816     else
817         mrs_used = 0;
818
819     dsl_deadlist_space(&ds->ds_deadlist, &dlused, &dlcomp, &dluncomp);
820
821     ASSERT3U(dlused, <=, mrs_used);
822     ds->ds_phys->ds_unique_bytes =
823         ds->ds_phys->ds_referenced_bytes - (mrs_used - dlused);
824
825     if (spa_version(ds->ds_dir->dd_pool->dp_spa) >=
826         SPA_VERSION_UNIQUE_ACCURATE)
827         ds->ds_phys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;
828 }
829
830 void
831 dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
832     dmu_tx_t *tx)
833 {
834     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
835     uint64_t count;
836     int err;
837
838     ASSERT(ds->ds_phys->ds_num_children >= 2);
839     err = zap_remove_int(mos, ds->ds_phys->ds_next_clones_obj, obj, tx);
840     /*
841      * The err should not be ENOENT, but a bug in a previous version
842      * of the code could cause upgrade_clones_cb() to not set
843      * ds_next_snap_obj when it should, leading to a missing entry.
844      * If we knew that the pool was created after
845      * SPA_VERSION_NEXT_CLONES, we could assert that it isn't
846      * ENOENT. However, at least we can check that we don't have
847      * too many entries in the next_clones_obj even after failing to
848      * remove this one.
849      */
850     if (err != ENOENT)
851         VERIFY0(err);
852     ASSERT0(zap_count(mos, ds->ds_phys->ds_next_clones_obj,
853         &count));
854     ASSERT3U(count, <=, ds->ds_phys->ds_num_children - 2);
855 }

```

```

858 blkptr_t *
859 dsl_dataset_get_blkptr(dsl_dataset_t *ds)
860 {
861     return (&ds->ds_phys->ds_bp);
862 }

864 void
865 dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx)
866 {
867     ASSERT(dmu_tx_is_syncing(tx));
868     /* If it's the meta-objset, set dp_meta_rootbp */
869     if (ds == NULL) {
870         tx->tx_pool->dp_meta_rootbp = *bp;
871     } else {
872         dmu_buf_will_dirty(ds->ds_dbuf, tx);
873         ds->ds_phys->ds_bp = *bp;
874     }
875 }

877 spa_t *
878 dsl_dataset_get_spa(dsl_dataset_t *ds)
879 {
880     return (ds->ds_dir->dd_pool->dp_spa);
881 }

883 void
884 dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx)
885 {
886     dsl_pool_t *dp;

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

891     ASSERT(ds->ds_objset != NULL);

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

896     dp = ds->ds_dir->dd_pool;

898     if (txg_list_add(&dp->dp_dirty_datasets, ds, tx->tx_txg)) {
899         /* up the hold count until we can be written out */
900         dmu_buf_add_ref(ds->ds_dbuf, ds);
901     }
902 }

904 boolean_t
905 dsl_dataset_is_dirty(dsl_dataset_t *ds)
906 {
907     for (int t = 0; t < TXG_SIZE; t++) {
908         if (txg_list_member(&ds->ds_dir->dd_pool->dp_dirty_datasets,
909             ds, t))
910             return (B_TRUE);
911     }
912     return (B_FALSE);
913 }

915 static int
916 dsl_dataset_snapshot_reserve_space(dsl_dataset_t *ds, dmu_tx_t *tx)
917 {
918     uint64_t asize;

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

```

```

923     /*
924      * If there's an fs-only reservation, any blocks that might become
925      * owned by the snapshot dataset must be accommodated by space
926      * outside of the reservation.
927      */
928     ASSERT(ds->ds_reserved == 0 || DS_UNIQUE_IS_ACCURATE(ds));
929     asize = MIN(ds->ds_phys->ds_unique_bytes, ds->ds_reserved);
930     if (asize > dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE))
931         return (SET_ERROR(ENOSPC));

933     /*
934      * Propagate any reserved space for this snapshot to other
935      * snapshot checks in this sync group.
936      */
937     if (asize > 0)
938         dsl_dir_willuse_space(ds->ds_dir, asize, tx);

940     return (0);
941 }

943 typedef struct dsl_dataset_snapshot_arg {
944     nvlist_t *ddsa_snaps;
945     nvlist_t *ddsa_props;
946     nvlist_t *ddsa_errors;
947 } dsl_dataset_snapshot_arg_t;

949 int
950 dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
951     dmu_tx_t *tx)
952 {
953     int error;
954     uint64_t value;

956     ds->ds_trysnap_txg = tx->tx_txg;

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

961     /*
962      * We don't allow multiple snapshots of the same txg. If there
963      * is already one, try again.
964      */
965     if (ds->ds_phys->ds_prev_snap_txg >= tx->tx_txg)
966         return (SET_ERROR(EAGAIN));

968     /*
969      * Check for conflicting snapshot name.
970      */
971     error = dsl_dataset_snap_lookup(ds, snapname, &value);
972     if (error == 0)
973         return (SET_ERROR(EEXIST));
974     if (error != ENOENT)
975         return (error);

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

981     return (0);
982 }

984 static int
985 dsl_dataset_snapshot_check(void *arg, dmu_tx_t *tx)
986 {
987     dsl_dataset_snapshot_arg_t *ddsa = arg;

```

```

988     dsl_pool_t *dp = dmu_tx_pool(tx);
989     nvpair_t *pair;
990     int rv = 0;

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

999         name = nvpair_name(pair);
1000         if (strlen(name) >= MAXNAMELEN)
1001             error = SET_ERROR(ENAMETOOLONG);
1002         if (error == 0) {
1003             atp = strchr(name, '@');
1004             if (atp == NULL)
1005                 error = SET_ERROR(EINVAL);
1006             if (error == 0)
1007                 (void) strncpy(dsname, name, atp - name + 1);
1008         }
1009         if (error == 0)
1010             error = dsl_dataset_hold(dp, dsname, FTAG, &ds);
1011         if (error == 0) {
1012             error = dsl_dataset_snapshot_check_impl(ds,
1013                 atp + 1, tx);
1014             dsl_dataset_rele(ds, FTAG);
1015         }

1017         if (error != 0) {
1018             if (ddsa->ddsa_errors != NULL) {
1019                 fnvlist_add_int32(ddsa->ddsa_errors,
1020                     name, error);
1021             }
1022             rv = error;
1023         }
1024     }
1025     return (rv);
1026 }

1028 void
1029 dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,
1030     dmu_tx_t *tx)
1031 {
1032     static zil_header_t zero_zil;

1034     dsl_pool_t *dp = ds->ds_dir->dd_pool;
1035     dmu_buf_t *dbuf;
1036     dsl_dataset_phys_t *dsphys;
1037     uint64_t dsobj, crtngx;
1038     objset_t *mos = dp->dp_meta_objset;
1039     objset_t *os;

1041     ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

1043     /*
1044      * If we are on an old pool, the zil must not be active, in which
1045      * case it will be zeroed. Usually zil_suspend() accomplishes this.
1046      */
1047     ASSERT(spa_version(dmu_tx_pool(tx)->dp_spa) >= SPA_VERSION_FAST_SNAP ||
1048         dmu_objset_from_ds(ds, &os) != 0 ||
1049         bcmp(&os->os_phys->os_zil_header, &zero_zil,
1050             sizeof (zero_zil)) == 0);

1053     /*

```

```

1054     * The origin's ds_creation_txg has to be < TXG_INITIAL
1055     */
1056     if (strcmp(snapname, ORIGIN_DIR_NAME) == 0)
1057         crtngx = 1;
1058     else
1059         crtngx = tx->tx_txg;

1061     dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
1062         DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
1063     VERIFY0(dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
1064     dmu_buf_will_dirty(dbuf, tx);
1065     dsphys = dbuf->db_data;
1066     bzero(dsphys, sizeof (dsl_dataset_phys_t));
1067     dsphys->ds_dir_obj = ds->ds_dir->dd_object;
1068     dsphys->ds_fsuid_guid = unique_create();
1069     (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
1070         sizeof (dsphys->ds_guid));
1071     dsphys->ds_prev_snap_obj = ds->ds_phys->ds_prev_snap_obj;
1072     dsphys->ds_prev_snap_txg = ds->ds_phys->ds_prev_snap_txg;
1073     dsphys->ds_next_snap_obj = ds->ds_object;
1074     dsphys->ds_num_children = 1;
1075     dsphys->ds_creation_time = gethrestime_sec();
1076     dsphys->ds_creation_txg = crtngx;
1077     dsphys->ds_deadlist_obj = ds->ds_phys->ds_deadlist_obj;
1078     dsphys->ds_referenced_bytes = ds->ds_phys->ds_referenced_bytes;
1079     dsphys->ds_compressed_bytes = ds->ds_phys->ds_compressed_bytes;
1080     dsphys->ds_uncompressed_bytes = ds->ds_phys->ds_uncompressed_bytes;
1081     dsphys->ds_flags = ds->ds_phys->ds_flags;
1082     dsphys->ds_bp = ds->ds_phys->ds_bp;
1083     dmu_buf_rele(dbuf, FTAG);

1085     ASSERT3U(ds->ds_prev != 0, ==, ds->ds_phys->ds_prev_snap_obj != 0);
1086     if (ds->ds_prev) {
1087         uint64_t next_clones_obj =
1088             ds->ds_prev->ds_phys->ds_next_clones_obj;
1089         ASSERT(ds->ds_prev->ds_phys->ds_next_snap_obj ==
1090             ds->ds_object ||
1091             ds->ds_prev->ds_phys->ds_num_children > 1);
1092         if (ds->ds_prev->ds_phys->ds_next_snap_obj == ds->ds_object) {
1093             dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
1094             ASSERT3U(ds->ds_phys->ds_prev_snap_txg, ==,
1095                 ds->ds_prev->ds_phys->ds_creation_txg);
1096             ds->ds_prev->ds_phys->ds_next_snap_obj = dsobj;
1097         } else if (next_clones_obj != 0) {
1098             dsl_dataset_remove_from_next_clones(ds->ds_prev,
1099                 dsphys->ds_next_snap_obj, tx);
1100             VERIFY0(zap_add_int(mos,
1101                 next_clones_obj, dsobj, tx));
1102         }
1103     }

1105     /*
1106      * If we have a reference-reservation on this dataset, we will
1107      * need to increase the amount of refreservation being charged
1108      * since our unique space is going to zero.
1109      */
1110     if (ds->ds_reserved) {
1111         int64_t delta;
1112         ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
1113         delta = MIN(ds->ds_phys->ds_unique_bytes, ds->ds_reserved);
1114         dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV,
1115             delta, 0, 0, tx);
1116     }

1118     dmu_buf_will_dirty(ds->ds_dbuf, tx);
1119     ds->ds_phys->ds_deadlist_obj = dsl_deadlist_clone(&ds->ds_deadlist,

```

```

1120     UINT64_MAX, ds->ds_phys->ds_prev_snap_obj, tx);
1121     dsl_deadlist_close(&ds->ds_deadlist);
1122     dsl_deadlist_open(&ds->ds_deadlist, mos, ds->ds_phys->ds_deadlist_obj);
1123     dsl_deadlist_add_key(&ds->ds_deadlist,
1124         ds->ds_phys->ds_prev_snap_txg, tx);
1125
1126     ASSERT3U(ds->ds_phys->ds_prev_snap_txg, <, tx->tx_txg);
1127     ds->ds_phys->ds_prev_snap_obj = dsobj;
1128     ds->ds_phys->ds_prev_snap_txg = crtxg;
1129     ds->ds_phys->ds_unique_bytes = 0;
1130     if (spa_version(dp->dp_spa) >= SPA_VERSION_UNIQUE_ACCURATE)
1131         ds->ds_phys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;
1132
1133     VERIFY0(zap_add(mos, ds->ds_phys->ds_snapnames_zapobj,
1134         snapname, 8, 1, &dsobj, tx));
1135
1136     if (ds->ds_prev)
1137         dsl_dataset_rele(ds->ds_prev, ds);
1138     VERIFY0(dsl_dataset_hold_obj(dp,
1139         ds->ds_phys->ds_prev_snap_obj, ds, &ds->ds_prev));
1140
1141     dsl_scan_ds_snapshotted(ds, tx);
1142
1143     dsl_dir_snap_cmtime_update(ds->ds_dir);
1144
1145     spa_history_log_internal_ds(ds->ds_prev, "snapshot", tx, "");
1146 }
1147
1148 static void
1149 dsl_dataset_snapshot_sync(void *arg, dmu_tx_t *tx)
1150 {
1151     dsl_dataset_snapshot_arg_t *ddsa = arg;
1152     dsl_pool_t *dp = dmu_tx_pool(tx);
1153     nvpair_t *pair;
1154
1155     for (pair = nvlist_next_nvpair(ddsa->ddsa_snaps, NULL);
1156         pair != NULL; pair = nvlist_next_nvpair(ddsa->ddsa_snaps, pair)) {
1157         dsl_dataset_t *ds;
1158         char *name, *atp;
1159         char dsname[MAXNAMELEN];
1160
1161         name = nvpair_name(pair);
1162         atp = strchr(name, '@');
1163         (void) strcpy(dsname, name, atp - name + 1);
1164         VERIFY0(dsl_dataset_hold(dp, dsname, FTAG, &ds));
1165
1166         dsl_dataset_snapshot_sync_impl(ds, atp + 1, tx);
1167         if (ddsa->ddsa_props != NULL) {
1168             dsl_props_set_sync_impl(ds->ds_prev,
1169                 ZPROP_SRC_LOCAL, ddsa->ddsa_props, tx);
1170         }
1171         dsl_dataset_rele(ds, FTAG);
1172     }
1173 }
1174
1175 /*
1176  * The snapshots must all be in the same pool.
1177  * All-or-nothing: if there are any failures, nothing will be modified.
1178  */
1179 int
1180 dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors)
1181 {
1182     dsl_dataset_snapshot_arg_t ddsa;
1183     nvpair_t *pair;
1184     boolean_t needsuspend;
1185     int error;

```

```

1186     spa_t *spa;
1187     char *firstname;
1188     nvlist_t *suspended = NULL;
1189
1190     pair = nvlist_next_nvpair(snaps, NULL);
1191     if (pair == NULL)
1192         return (0);
1193     firstname = nvpair_name(pair);
1194
1195     error = spa_open(firstname, &spa, FTAG);
1196     if (error != 0)
1197         return (error);
1198     needsuspend = (spa_version(spa) < SPA_VERSION_FAST_SNAP);
1199     spa_close(spa, FTAG);
1200
1201     if (needsuspend) {
1202         suspended = fnvlist_alloc();
1203         for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
1204             pair = nvlist_next_nvpair(snaps, pair)) {
1205             char fsname[MAXNAMELEN];
1206             char *snapname = nvpair_name(pair);
1207             char *atp;
1208             void *cookie;
1209
1210             atp = strchr(snapname, '@');
1211             if (atp == NULL) {
1212                 error = SET_ERROR(EINVAL);
1213                 break;
1214             }
1215             (void) strcpy(fsname, snapname, atp - snapname + 1);
1216
1217             error = zil_suspend(fsname, &cookie);
1218             if (error != 0)
1219                 break;
1220             fnvlist_add_uint64(suspended, fsname,
1221                 (uintptr_t)cookie);
1222         }
1223     }
1224
1225     ddsa.ddsa_snaps = snaps;
1226     ddsa.ddsa_props = props;
1227     ddsa.ddsa_errors = errors;
1228
1229     if (error == 0) {
1230         error = dsl_sync_task(firstname, dsl_dataset_snapshot_check,
1231             dsl_dataset_snapshot_sync, &ddsa,
1232             fnvlist_num_pairs(snaps) * 3);
1233     }
1234
1235     if (suspended != NULL) {
1236         for (pair = nvlist_next_nvpair(suspended, NULL); pair != NULL;
1237             pair = nvlist_next_nvpair(suspended, pair)) {
1238             zil_resume((void *) (uintptr_t)
1239                 fnvpair_value_uint64(pair));
1240         }
1241         fnvlist_free(suspended);
1242     }
1243
1244     return (error);
1245 }
1246
1247 typedef struct dsl_dataset_snapshot_tmp_arg {
1248     const char *ddsta_fsname;
1249     const char *ddsta_snapname;
1250     minor_t ddsta_cleanup_minor;
1251     const char *ddsta_htag;

```

```

1252 } dsl_dataset_snapshot_tmp_arg_t;

1254 static int
1255 dsl_dataset_snapshot_tmp_check(void *arg, dmu_tx_t *tx)
1256 {
1257     dsl_dataset_snapshot_tmp_arg_t *ddsta = arg;
1258     dsl_pool_t *dp = dmu_tx_pool(tx);
1259     dsl_dataset_t *ds;
1260     int error;

1262     error = dsl_dataset_hold(dp, ddsta->ddsta_fsname, FTAG, &ds);
1263     if (error != 0)
1264         return (error);

1266     error = dsl_dataset_snapshot_check_impl(ds, ddsta->ddsta_snapname, tx);
1267     if (error != 0) {
1268         dsl_dataset_rele(ds, FTAG);
1269         return (error);
1270     }

1272     if (spa_version(dp->dp_spa) < SPA_VERSION_USERREFS) {
1273         dsl_dataset_rele(ds, FTAG);
1274         return (SET_ERROR(ENOTSUP));
1275     }
1276     error = dsl_dataset_user_hold_check_one(NULL, ddsta->ddsta_htag,
1277         B_TRUE, tx);
1278     if (error != 0) {
1279         dsl_dataset_rele(ds, FTAG);
1280         return (error);
1281     }

1283     dsl_dataset_rele(ds, FTAG);
1284     return (0);
1285 }

1287 static void
1288 dsl_dataset_snapshot_tmp_sync(void *arg, dmu_tx_t *tx)
1289 {
1290     dsl_dataset_snapshot_tmp_arg_t *ddsta = arg;
1291     dsl_pool_t *dp = dmu_tx_pool(tx);
1292     dsl_dataset_t *ds;

1294     VERIFY0(dsl_dataset_hold(dp, ddsta->ddsta_fsname, FTAG, &ds));

1296     dsl_dataset_snapshot_sync_impl(ds, ddsta->ddsta_snapname, tx);
1297     dsl_dataset_user_hold_sync_one(ds->ds_prev, ddsta->ddsta_htag,
1298         ddsta->ddsta_cleanup_minor, gethrestime_sec(), tx);
1299     dsl_destroy_snapshot_sync_impl(ds->ds_prev, B_TRUE, tx);

1301     dsl_dataset_rele(ds, FTAG);
1302 }

1304 int
1305 dsl_dataset_snapshot_tmp(const char *fsname, const char *snapname,
1306     minor_t cleanup_minor, const char *htag)
1307 {
1308     dsl_dataset_snapshot_tmp_arg_t ddsta;
1309     int error;
1310     spa_t *spa;
1311     boolean_t needsuspend;
1312     void *cookie;

1314     ddsta.ddsta_fsname = fsname;
1315     ddsta.ddsta_snapname = snapname;
1316     ddsta.ddsta_cleanup_minor = cleanup_minor;
1317     ddsta.ddsta_htag = htag;

```

```

1319     error = spa_open(fsname, &spa, FTAG);
1320     if (error != 0)
1321         return (error);
1322     needsuspend = (spa_version(spa) < SPA_VERSION_FAST_SNAP);
1323     spa_close(spa, FTAG);

1325     if (needsuspend) {
1326         error = zil_suspend(fsname, &cookie);
1327         if (error != 0)
1328             return (error);
1329     }

1331     error = dsl_sync_task(fsname, dsl_dataset_snapshot_tmp_check,
1332         dsl_dataset_snapshot_tmp_sync, &ddsta, 3);

1334     if (needsuspend)
1335         zil_resume(cookie);
1336     return (error);
1337 }

1340 void
1341 dsl_dataset_sync(dsl_dataset_t *ds, zio_t *zio, dmu_tx_t *tx)
1342 {
1343     ASSERT(dmu_tx_is_syncing(tx));
1344     ASSERT(ds->ds_objset != NULL);
1345     ASSERT(ds->ds_phys->ds_next_snap_obj == 0);

1347     /*
1348      * in case we had to change ds_fsid_guid when we opened it,
1349      * sync it out now.
1350      */
1351     dmu_buf_will_dirty(ds->ds_dbuf, tx);
1352     ds->ds_phys->ds_fsid_guid = ds->ds_fsid_guid;

1354     dmu_objset_sync(ds->ds_objset, zio, tx);
1355 }

1357 static void
1358 get_clones_stat(dsl_dataset_t *ds, nvlist_t *nv)
1359 {
1360     uint64_t count = 0;
1361     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
1362     zap_cursor_t zc;
1363     zap_attribute_t za;
1364     nvlist_t *propval = fnvlist_alloc();
1365     nvlist_t *val = fnvlist_alloc();

1367     ASSERT(dsl_pool_config_held(ds->ds_dir->dd_pool));

1369     /*
1370      * There may be missing entries in ds_next_clones_obj
1371      * due to a bug in a previous version of the code.
1372      * Only trust it if it has the right number of entries.
1373      */
1374     if (ds->ds_phys->ds_next_clones_obj != 0) {
1375         ASSERT0(zap_count(mos, ds->ds_phys->ds_next_clones_obj,
1376             &count));
1377     }
1378     if (count != ds->ds_phys->ds_num_children - 1)
1379         goto fail;
1380     for (zap_cursor_init(&zc, mos, ds->ds_phys->ds_next_clones_obj);
1381         zap_cursor_retrieve(&zc, &za) == 0;
1382         zap_cursor_advance(&zc)) {
1383         dsl_dataset_t *clone;

```



```

1384         char buf[ZFS_MAXNAMELEN];
1385         VERIFY0(dsl_dataset_hold_obj(ds->ds_dir->dd_pool,
1386             za.za_first_integer, FTAG, &clone));
1387         dsl_dir_name(clone->ds_dir, buf);
1388         fnvlist_add_boolean(val, buf);
1389         dsl_dataset_rele(clone, FTAG);
1390     }
1391     zap_cursor_fini(&ztc);
1392     fnvlist_add_nvlist(propval, ZPROP_VALUE, val);
1393     fnvlist_add_nvlist(nv, zfs_prop_to_name(ZFS_PROP_CLONES), propval);
1394 fail:
1395     nvlist_free(val);
1396     nvlist_free(propval);
1397 }

1399 void
1400 dsl_dataset_stats(dsl_dataset_t *ds, nvlist_t *nv)
1401 {
1402     dsl_pool_t *dp = ds->ds_dir->dd_pool;
1403     uint64_t refd, avail, uobjjs, aobjjs, ratio;

1405     ASSERT(dsl_pool_config_held(dp));

1407     ratio = ds->ds_phys->ds_compressed_bytes == 0 ? 100 :
1408         (ds->ds_phys->ds_uncompressed_bytes * 100 /
1409             ds->ds_phys->ds_compressed_bytes);

1411     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRATIO, ratio);
1412     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_LOGICALREFERENCED,
1413         ds->ds_phys->ds_uncompressed_bytes);

1415     if (dsl_dataset_is_snapshot(ds)) {
1416         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_COMPRESSRATIO, ratio);
1417         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USED,
1418             ds->ds_phys->ds_unique_bytes);
1419         get_clones_stat(ds, nv);
1420     } else {
1421         dsl_dir_stats(ds->ds_dir, nv);
1422     }

1424     dsl_dataset_space(ds, &refd, &avail, &uobjjs, &aobjjs);
1425     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_AVAILABLE, avail);
1426     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFERENCED, refd);

1428     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATION,
1429         ds->ds_phys->ds_creation_time);
1430     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATETXG,
1431         ds->ds_phys->ds_creation_txg);
1432     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFQUOTA,
1433         ds->ds_quota);
1434     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRESERVATION,
1435         ds->ds_reserved);
1436     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_GUID,
1437         ds->ds_phys->ds_guid);
1438     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_UNIQUE,
1439         ds->ds_phys->ds_unique_bytes);
1440     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_OBJSETID,
1441         ds->ds_object);
1442     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERREFS,
1443         ds->ds_userrefs);
1444     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_DEFER_DESTROY,
1445         DS_IS_DEFER_DESTROY(ds) ? 1 : 0);

1447     if (ds->ds_phys->ds_prev_snap_obj != 0) {
1448         uint64_t written, comp, uncomp;
1449         dsl_pool_t *dp = ds->ds_dir->dd_pool;

```

```

1450         dsl_dataset_t *prev;

1452     int err = dsl_dataset_hold_obj(dp,
1453         ds->ds_phys->ds_prev_snap_obj, FTAG, &prev);
1454     if (err == 0) {
1455         err = dsl_dataset_space_written(prev, ds, &written,
1456             &comp, &uncomp);
1457         dsl_dataset_rele(prev, FTAG);
1458         if (err == 0) {
1459             dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_WRITTEN,
1460                 written);
1461         }
1462     }
1463 }

1466 void
1467 dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat)
1468 {
1469     dsl_pool_t *dp = ds->ds_dir->dd_pool;
1470     ASSERT(dsl_pool_config_held(dp));

1472     stat->dds_creation_txg = ds->ds_phys->ds_creation_txg;
1473     stat->dds_inconsistent = ds->ds_phys->ds_flags & DS_FLAG_INCONSISTENT;
1474     stat->dds_guid = ds->ds_phys->ds_guid;
1475     stat->dds_origin[0] = '\0';
1476     if (dsl_dataset_is_snapshot(ds)) {
1477         stat->dds_is_snapshot = B_TRUE;
1478         stat->dds_num_clones = ds->ds_phys->ds_num_children - 1;
1479     } else {
1480         stat->dds_is_snapshot = B_FALSE;
1481         stat->dds_num_clones = 0;

1483         if (dsl_dir_is_clone(ds->ds_dir)) {
1484             dsl_dataset_t *ods;

1486             VERIFY0(dsl_dataset_hold_obj(dp,
1487                 ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &ods));
1488             dsl_dataset_name(ods, stat->dds_origin);
1489             dsl_dataset_rele(ods, FTAG);
1490         }
1491     }
1492 }

1494 uint64_t
1495 dsl_dataset_fsid_guid(dsl_dataset_t *ds)
1496 {
1497     return (ds->ds_fsid_guid);
1498 }

1500 void
1501 dsl_dataset_space(dsl_dataset_t *ds,
1502     uint64_t *refdbytesp, uint64_t *availbytesp,
1503     uint64_t *usedobjjsp, uint64_t *availobjjsp)
1504 {
1505     *refdbytesp = ds->ds_phys->ds_referenced_bytes;
1506     *availbytesp = dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE);
1507     if (ds->ds_reserved > ds->ds_phys->ds_unique_bytes)
1508         *availbytesp += ds->ds_reserved - ds->ds_phys->ds_unique_bytes;
1509     if (ds->ds_quota != 0) {
1510         /*
1511          * Adjust available bytes according to refquota
1512          */
1513         if (*refdbytesp < ds->ds_quota)
1514             *availbytesp = MIN(*availbytesp,
1515                 ds->ds_quota - *refdbytesp);

```

```

1516         else
1517             *availbytesp = 0;
1518     }
1519     *usedobjsp = ds->ds_phys->ds_bp.blk_fill;
1520     *availobjsp = DN_MAX_OBJECT - *usedobjsp;
1521 }

1523 boolean_t
1524 dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds)
1525 {
1526     dsl_pool_t *dp = ds->ds_dir->dd_pool;

1528     ASSERT(dsl_pool_config_held(dp));
1529     if (ds->ds_prev == NULL)
1530         return (B_FALSE);
1531     if (ds->ds_phys->ds_bp.blk_birth >
1532         ds->ds_prev->ds_phys->ds_creation_txg) {
1533         objset_t *os, *os_prev;
1534         /*
1535          * It may be that only the ZIL differs, because it was
1536          * reset in the head. Don't count that as being
1537          * modified.
1538          */
1539         if (dmu_objset_from_ds(ds, &os) != 0)
1540             return (B_TRUE);
1541         if (dmu_objset_from_ds(ds->ds_prev, &os_prev) != 0)
1542             return (B_TRUE);
1543         return (bcmp(&os->os_phys->os_meta_dnode,
1544             &os_prev->os_phys->os_meta_dnode,
1545             sizeof (os->os_phys->os_meta_dnode)) != 0);
1546     }
1547     return (B_FALSE);
1548 }

1550 typedef struct dsl_dataset_rename_snapshot_arg {
1551     const char *ddrsa_fsname;
1552     const char *ddrsa_oldsnapname;
1553     const char *ddrsa_newsnapname;
1554     boolean_t ddrsa_recursive;
1555     dmu_tx_t *ddrsa_tx;
1556 } dsl_dataset_rename_snapshot_arg_t;

1558 /* ARGSUSED */
1559 static int
1560 dsl_dataset_rename_snapshot_check_impl(dsl_pool_t *dp,
1561     dsl_dataset_t *hds, void *arg)
1562 {
1563     dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
1564     int error;
1565     uint64_t val;

1567     error = dsl_dataset_snap_lookup(hds, ddrsa->ddrsa_oldsnapname, &val);
1568     if (error != 0) {
1569         /* ignore nonexistent snapshots */
1570         return (error == ENOENT ? 0 : error);
1571     }

1573     /* new name should not exist */
1574     error = dsl_dataset_snap_lookup(hds, ddrsa->ddrsa_newsnapname, &val);
1575     if (error == 0)
1576         error = SET_ERROR(EEXIST);
1577     else if (error == ENOENT)
1578         error = 0;

1580     /* dataset name + 1 for the "@" + the new snapshot name must fit */
1581     if (dsl_dir_namelen(hds->ds_dir) + 1 +

```

```

1582         strlen(ddrsa->ddrsa_newsnapname) >= MAXNAMELEN)
1583         error = SET_ERROR(ENAMETOOLONG);

1585     return (error);
1586 }

1588 static int
1589 dsl_dataset_rename_snapshot_check(void *arg, dmu_tx_t *tx)
1590 {
1591     dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
1592     dsl_pool_t *dp = dmu_tx_pool(tx);
1593     dsl_dataset_t *hds;
1594     int error;

1596     error = dsl_dataset_hold(dp, ddrsa->ddrsa_fsname, FTAG, &hds);
1597     if (error != 0)
1598         return (error);

1600     if (ddrsa->ddrsa_recursive) {
1601         error = dmu_objset_find_dp(dp, hds->ds_dir->dd_object,
1602             dsl_dataset_rename_snapshot_check_impl, ddrsa,
1603             DS_FIND_CHILDREN);
1604     } else {
1605         error = dsl_dataset_rename_snapshot_check_impl(dp, hds, ddrsa);
1606     }
1607     dsl_dataset_rele(hds, FTAG);
1608     return (error);
1609 }

1611 static int
1612 dsl_dataset_rename_snapshot_sync_impl(dsl_pool_t *dp,
1613     dsl_dataset_t *hds, void *arg)
1614 {
1615     dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
1616     dsl_dataset_t *ds;
1617     uint64_t val;
1618     dmu_tx_t *tx = ddrsa->ddrsa_tx;
1619     int error;

1621     error = dsl_dataset_snap_lookup(hds, ddrsa->ddrsa_oldsnapname, &val);
1622     ASSERT(error == 0 || error == ENOENT);
1623     if (error == ENOENT) {
1624         /* ignore nonexistent snapshots */
1625         return (0);
1626     }

1628     VERIFY0(dsl_dataset_hold_obj(dp, val, FTAG, &ds));

1630     /* log before we change the name */
1631     spa_history_log_internal_ds(ds, "rename", tx,
1632         "-> %s", ddrsa->ddrsa_newsnapname);

1634     VERIFY0(dsl_dataset_snap_remove(hds, ddrsa->ddrsa_oldsnapname, tx));
1635     mutex_enter(&ds->ds_lock);
1636     (void) strcpy(ds->ds_snapname, ddrsa->ddrsa_newsnapname);
1637     mutex_exit(&ds->ds_lock);
1638     VERIFY0(zap_add(dp->dp_meta_objset, hds->ds_phys->ds_snapnames_zapobj,
1639         ds->ds_snapname, 8, 1, &ds->ds_object, tx));

1641     dsl_dataset_rele(ds, FTAG);
1642     return (0);
1643 }

1645 static void
1646 dsl_dataset_rename_snapshot_sync(void *arg, dmu_tx_t *tx)
1647 {

```

```

1648     dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
1649     dsl_pool_t *dp = dmu_tx_pool(tx);
1650     dsl_dataset_t *hds;

1652     VERIFY0(dsl_dataset_hold(dp, ddrsa->ddrsa_fsname, FTAG, &hds));
1653     ddrsa->ddrsa_tx = tx;
1654     if (ddrsa->ddrsa_recursive) {
1655         VERIFY0(dmu_objset_find_dp(dp, hds->ds_dir->dd_object,
1656             dsl_dataset_rename_snapshot_sync_impl, ddrsa,
1657             DS_FIND_CHILDREN));
1658     } else {
1659         VERIFY0(dsl_dataset_rename_snapshot_sync_impl(dp, hds, ddrsa));
1660     }
1661     dsl_dataset_rele(hds, FTAG);
1662 }

1664 int
1665 dsl_dataset_rename_snapshot(const char *fsname,
1666     const char *oldsnapname, const char *newsnapname, boolean_t recursive)
1667 {
1668     dsl_dataset_rename_snapshot_arg_t ddrsa;

1670     ddrsa.ddrsa_fsname = fsname;
1671     ddrsa.ddrsa_oldsnapname = oldsnapname;
1672     ddrsa.ddrsa_newsnapname = newsnapname;
1673     ddrsa.ddrsa_recursive = recursive;

1675     return (dsl_sync_task(fsname, dsl_dataset_rename_snapshot_check,
1676         dsl_dataset_rename_snapshot_sync, &ddrsa, 1));
1677 }

1679 static int
1680 dsl_dataset_rollback_check(void *arg, dmu_tx_t *tx)
1681 {
1682     const char *fsname = arg;
1683     dsl_pool_t *dp = dmu_tx_pool(tx);
1684     dsl_dataset_t *ds;
1685     int64_t unused_refres_delta;
1686     int error;

1688     error = dsl_dataset_hold(dp, fsname, FTAG, &ds);
1689     if (error != 0)
1690         return (error);

1692     /* must not be a snapshot */
1693     if (dsl_dataset_is_snapshot(ds)) {
1694         dsl_dataset_rele(ds, FTAG);
1695         return (SET_ERROR(EINVAL));
1696     }

1698     /* must have a most recent snapshot */
1699     if (ds->ds_phys->ds_prev_snap_txg < TXG_INITIAL) {
1700         dsl_dataset_rele(ds, FTAG);
1701         return (SET_ERROR(EINVAL));
1702     }

1704     if (dsl_dataset_long_held(ds)) {
1705         dsl_dataset_rele(ds, FTAG);
1706         return (SET_ERROR(EBUSY));
1707     }

1709     /*
1710      * Check if the snap we are rolling back to uses more than
1711      * the refquota.
1712      */
1713     if (ds->ds_quota != 0 &&

```

```

1714         ds->ds_prev->ds_phys->ds_referenced_bytes > ds->ds_quota) {
1715             dsl_dataset_rele(ds, FTAG);
1716             return (SET_ERROR(EDQUOT));
1717         }

1719     /*
1720      * When we do the clone swap, we will temporarily use more space
1721      * due to the refreservation (the head will no longer have any
1722      * unique space, so the entire amount of the refreservation will need
1723      * to be free). We will immediately destroy the clone, freeing
1724      * this space, but the freeing happens over many txg's.
1725      */
1726     unused_refres_delta = (int64_t)MIN(ds->ds_reserved,
1727         ds->ds_phys->ds_unique_bytes);

1729     if (unused_refres_delta > 0 &&
1730         unused_refres_delta >
1731         dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE)) {
1732         dsl_dataset_rele(ds, FTAG);
1733         return (SET_ERROR(ENOSPC));
1734     }

1736     dsl_dataset_rele(ds, FTAG);
1737     return (0);
1738 }

1740 static void
1741 dsl_dataset_rollback_sync(void *arg, dmu_tx_t *tx)
1742 {
1743     const char *fsname = arg;
1744     dsl_pool_t *dp = dmu_tx_pool(tx);
1745     dsl_dataset_t *ds, *clone;
1746     uint64_t cloneobj;

1748     VERIFY0(dsl_dataset_hold(dp, fsname, FTAG, &ds));

1750     cloneobj = dsl_dataset_create_sync(ds->ds_dir, "%rollback",
1751         ds->ds_prev, DS_CREATE_FLAG_NODIRTY, kcred, tx);

1753     VERIFY0(dsl_dataset_hold_obj(dp, cloneobj, FTAG, &clone));

1755     dsl_dataset_clone_swap_sync_impl(clone, ds, tx);
1756     dsl_dataset_zero zil(ds, tx);

1758     dsl_destroy_head_sync_impl(clone, tx);

1760     dsl_dataset_rele(clone, FTAG);
1761     dsl_dataset_rele(ds, FTAG);
1762 }

1764 int
1765 dsl_dataset_rollback(const char *fsname)
1766 {
1767     return (dsl_sync_task(fsname, dsl_dataset_rollback_check,
1768         dsl_dataset_rollback_sync, (void *)fsname, 1));
1769 }

1771 struct promotenode {
1772     list_node_t link;
1773     dsl_dataset_t *ds;
1774 };

1776 typedef struct dsl_dataset_promote_arg {
1777     const char *ddpa_clonename;
1778     dsl_dataset_t *ddpa_clone;
1779     list_t shared_snaps, origin_snaps, clone_snaps;

```

```

1780     dsl_dataset_t *origin_origin; /* origin of the origin */
1781     uint64_t used, comp, uncomp, unique, cloneusedsnap, originusedsnap;
1782     char *err_ds;
1783 } dsl_dataset_promote_arg_t;

1785 static int snaplist_space(list_t *l, uint64_t mintxg, uint64_t *spacep);
1786 static int promote_hold(dsl_dataset_promote_arg_t *ddpa, dsl_pool_t *dp,
1787     void *tag);
1788 static void promote_rele(dsl_dataset_promote_arg_t *ddpa, void *tag);

1790 static int
1791 dsl_dataset_promote_check(void *arg, dmu_tx_t *tx)
1792 {
1793     dsl_dataset_promote_arg_t *ddpa = arg;
1794     dsl_pool_t *dp = dmu_tx_pool(tx);
1795     dsl_dataset_t *hds;
1796     struct promotenode *snap;
1797     dsl_dataset_t *origin_ds;
1798     int err;
1799     uint64_t unused;

1801     err = promote_hold(ddpa, dp, FTAG);
1802     if (err != 0)
1803         return (err);

1805     hds = ddpa->ddpa_clone;

1807     if (hds->ds_phys->ds_flags & DS_FLAG_NOPROMOTE) {
1808         promote_rele(ddpa, FTAG);
1809         return (SET_ERROR(EXDEV));
1810     }

1812     /*
1813      * Compute and check the amount of space to transfer. Since this is
1814      * so expensive, don't do the preliminary check.
1815      */
1816     if (!dmu_tx_is_syncing(tx)) {
1817         promote_rele(ddpa, FTAG);
1818         return (0);
1819     }

1821     snap = list_head(&ddpa->shared_snaps);
1822     origin_ds = snap->ds;

1824     /* compute origin's new unique space */
1825     snap = list_tail(&ddpa->clone_snaps);
1826     ASSERT3U(snap->ds->ds_phys->ds_prev_snap_obj, ==, origin_ds->ds_object);
1827     dsl_deadlist_space_range(&snap->ds->ds_deadlist,
1828         origin_ds->ds_phys->ds_prev_snap_txg, UINT64_MAX,
1829         &ddpa->unique, &unused, &unused);

1831     /*
1832      * Walk the snapshots that we are moving
1833      *
1834      * Compute space to transfer. Consider the incremental changes
1835      * to used by each snapshot:
1836      * (my used) = (prev's used) + (blocks born) - (blocks killed)
1837      * So each snapshot gave birth to:
1838      * (blocks born) = (my used) - (prev's used) + (blocks killed)
1839      * So a sequence would look like:
1840      * (uN - u(N-1) + kN) + ... + (u1 - u0 + k1) + (u0 - 0 + k0)
1841      * Which simplifies to:
1842      * uN + kN + kN-1 + ... + k1 + k0
1843      * Note however, if we stop before we reach the ORIGIN we get:
1844      * uN + kN + kN-1 + ... + kM - uM-1
1845      */

```

```

1846     ddpa->used = origin_ds->ds_phys->ds_referenced_bytes;
1847     ddpa->comp = origin_ds->ds_phys->ds_compressed_bytes;
1848     ddpa->uncomp = origin_ds->ds_phys->ds_uncompressed_bytes;
1849     for (snap = list_head(&ddpa->shared_snaps); snap;
1850         snap = list_next(&ddpa->shared_snaps, snap)) {
1851         uint64_t val, dlused, dlcomp, dluncomp;
1852         dsl_dataset_t *ds = snap->ds;

1854         /*
1855          * If there are long holds, we won't be able to evict
1856          * the objset.
1857          */
1858         if (dsl_dataset_long_held(ds)) {
1859             err = SET_ERROR(EBUSY);
1860             goto out;
1861         }

1863         /* Check that the snapshot name does not conflict */
1864         VERIFY0(dsl_dataset_get_snapname(ds));
1865         err = dsl_dataset_snap_lookup(hds, ds->ds_snapname, &val);
1866         if (err == 0) {
1867             (void) strcpy(ddpa->err_ds, snap->ds->ds_snapname);
1868             err = SET_ERROR(EXIST);
1869             goto out;
1870         }
1871         if (err != ENOENT)
1872             goto out;

1874         /* The very first snapshot does not have a deadlist */
1875         if (ds->ds_phys->ds_prev_snap_obj == 0)
1876             continue;

1878         dsl_deadlist_space(&ds->ds_deadlist,
1879             &dlused, &dlcomp, &dluncomp);
1880         ddpa->used += dlused;
1881         ddpa->comp += dlcomp;
1882         ddpa->uncomp += dluncomp;
1883     }

1885     /*
1886      * If we are a clone of a clone then we never reached ORIGIN,
1887      * so we need to subtract out the clone origin's used space.
1888      */
1889     if (ddpa->origin_origin) {
1890         ddpa->used -= ddpa->origin_origin->ds_phys->ds_referenced_bytes;
1891         ddpa->comp -= ddpa->origin_origin->ds_phys->ds_compressed_bytes;
1892         ddpa->uncomp -=
1893             ddpa->origin_origin->ds_phys->ds_uncompressed_bytes;
1894     }

1896     /* Check that there is enough space here */
1897     err = dsl_dir_transfer_possible(origin_ds->ds_dir, hds->ds_dir,
1898         ddpa->used);
1899     if (err != 0)
1900         goto out;

1902     /*
1903      * Compute the amounts of space that will be used by snapshots
1904      * after the promotion (for both origin and clone). For each,
1905      * it is the amount of space that will be on all of their
1906      * deadlists (that was not born before their new origin).
1907      */
1908     if (hds->ds_dir->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
1909         uint64_t space;

1911         /*

```

```

1912     * Note, typically this will not be a clone of a clone,
1913     * so dd_origin_txg will be < TXG_INITIAL, so
1914     * these snaplist_space() -> dsl_deadlist_space_range()
1915     * calls will be fast because they do not have to
1916     * iterate over all bps.
1917     */
1918     snap = list_head(&ddpa->origin_snaps);
1919     err = snaplist_space(&ddpa->shared_snaps,
1920         snap->ds->ds_dir->dd_origin_txg, &ddpa->cloneusedsnap);
1921     if (err != 0)
1922         goto out;

1924     err = snaplist_space(&ddpa->clone_snaps,
1925         snap->ds->ds_dir->dd_origin_txg, &space);
1926     if (err != 0)
1927         goto out;
1928     ddpa->cloneusedsnap += space;
1929 }
1930 if (origin_ds->ds_dir->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
1931     err = snaplist_space(&ddpa->origin_snaps,
1932         origin_ds->ds_phys->ds_creation_txg, &ddpa->originusedsnap);
1933     if (err != 0)
1934         goto out;
1935 }

1937 out:
1938     promote_rele(ddpa, FTAG);
1939     return (err);
1940 }

1942 static void
1943 dsl_dataset_promote_sync(void *arg, dmu_tx_t *tx)
1944 {
1945     dsl_dataset_promote_arg_t *ddpa = arg;
1946     dsl_pool_t *dp = dmu_tx_pool(tx);
1947     dsl_dataset_t *hds;
1948     struct promotenode *snap;
1949     dsl_dataset_t *origin_ds;
1950     dsl_dataset_t *origin_head;
1951     dsl_dir_t *dd;
1952     dsl_dir_t *odd = NULL;
1953     uint64_t oldnext_obj;
1954     int64_t delta;

1956     VERIFY0(promote_hold(ddpa, dp, FTAG));
1957     hds = ddpa->ddpa_clone;

1959     ASSERT0(hds->ds_phys->ds_flags & DS_FLAG_NOPROMOTE);

1961     snap = list_head(&ddpa->shared_snaps);
1962     origin_ds = snap->ds;
1963     dd = hds->ds_dir;

1965     snap = list_head(&ddpa->origin_snaps);
1966     origin_head = snap->ds;

1968     /*
1969     * We need to explicitly open odd, since origin_ds's dd will be
1970     * changing.
1971     */
1972     VERIFY0(dsl_dir_hold_obj(dp, origin_ds->ds_dir->dd_object,
1973         NULL, FTAG, &odd));

1975     /* change origin's next snap */
1976     dmu_buf_will_dirty(origin_ds->ds_dbuf, tx);
1977     oldnext_obj = origin_ds->ds_phys->ds_next_snap_obj;

```

```

1978     snap = list_tail(&ddpa->clone_snaps);
1979     ASSERT3U(snap->ds->ds_phys->ds_prev_snap_obj, ==, origin_ds->ds_object);
1980     origin_ds->ds_phys->ds_next_snap_obj = snap->ds->ds_object;

1982     /* change the origin's next clone */
1983     if (origin_ds->ds_phys->ds_next_clones_obj) {
1984         dsl_dataset_remove_from_next_clones(origin_ds,
1985             snap->ds->ds_object, tx);
1986         VERIFY0(zap_add_int(dp->dp_meta_objset,
1987             origin_ds->ds_phys->ds_next_clones_obj,
1988             oldnext_obj, tx));
1989     }

1991     /* change origin */
1992     dmu_buf_will_dirty(dd->dd_dbuf, tx);
1993     ASSERT3U(dd->dd_phys->dd_origin_obj, ==, origin_ds->ds_object);
1994     dd->dd_phys->dd_origin_obj = odd->dd_phys->dd_origin_obj;
1995     dd->dd_origin_txg = origin_head->ds_dir->dd_origin_txg;
1996     dmu_buf_will_dirty(odd->dd_dbuf, tx);
1997     odd->dd_phys->dd_origin_obj = origin_ds->ds_object;
1998     origin_head->ds_dir->dd_origin_txg =
1999         origin_ds->ds_phys->ds_creation_txg;

2001     /* change dd_clone entries */
2002     if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2003         VERIFY0(zap_remove_int(dp->dp_meta_objset,
2004             odd->dd_phys->dd_clones, hds->ds_object, tx));
2005         VERIFY0(zap_add_int(dp->dp_meta_objset,
2006             ddpa->origin_origin->ds_dir->dd_phys->dd_clones,
2007             hds->ds_object, tx));

2009         VERIFY0(zap_remove_int(dp->dp_meta_objset,
2010             ddpa->origin_origin->ds_dir->dd_phys->dd_clones,
2011             origin_head->ds_object, tx));
2012         if (dd->dd_phys->dd_clones == 0) {
2013             dd->dd_phys->dd_clones = zap_create(dp->dp_meta_objset,
2014                 DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
2015         }
2016         VERIFY0(zap_add_int(dp->dp_meta_objset,
2017             dd->dd_phys->dd_clones, origin_head->ds_object, tx));
2018     }

2020     /* move snapshots to this dir */
2021     for (snap = list_head(&ddpa->shared_snaps); snap;
2022         snap = list_next(&ddpa->shared_snaps, snap)) {
2023         dsl_dataset_t *ds = snap->ds;

2025         /*
2026         * Property callbacks are registered to a particular
2027         * dsl_dir. Since ours is changing, evict the objset
2028         * so that they will be unregistered from the old dsl_dir.
2029         */
2030         if (ds->ds_objset) {
2031             dmu_objset_evict(ds->ds_objset);
2032             ds->ds_objset = NULL;
2033         }

2035         /* move snap name entry */
2036         VERIFY0(dsl_dataset_get_snapname(ds));
2037         VERIFY0(dsl_dataset_snap_remove(origin_head,
2038             ds->ds_snapname, tx));
2039         VERIFY0(zap_add(dp->dp_meta_objset,
2040             hds->ds_phys->ds_snapnames_zapobj, ds->ds_snapname,
2041             8, 1, &ds->ds_object, tx));

2043         /* change containing dsl_dir */

```

```

2044     dmu_buf_will_dirty(ds->ds_dbuf, tx);
2045     ASSERT3U(ds->ds_phys->ds_dir_obj, ==, odd->dd_object);
2046     ds->ds_phys->ds_dir_obj = dd->dd_object;
2047     ASSERT3P(ds->ds_dir, ==, odd);
2048     dsl_dir_rele(ds->ds_dir, ds);
2049     VERIFY0(dsl_dir_hold_obj(dp, dd->dd_object,
2050         NULL, ds, &ds->ds_dir));
2051
2052     /* move any clone references */
2053     if (ds->ds_phys->ds_next_clones_obj &&
2054         spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2055         zap_cursor_t zc;
2056         zap_attribute_t za;
2057
2058         for (zap_cursor_init(&zc, dp->dp_meta_objset,
2059             ds->ds_phys->ds_next_clones_obj);
2060             zap_cursor_retrieve(&zc, &za) == 0;
2061             zap_cursor_advance(&zc)) {
2062             dsl_dataset_t *cnds;
2063             uint64_t o;
2064
2065             if (za.za_first_integer == oldnext_obj) {
2066                 /*
2067                  * We've already moved the
2068                  * origin's reference.
2069                  */
2070                 continue;
2071             }
2072
2073             VERIFY0(dsl_dataset_hold_obj(dp,
2074                 za.za_first_integer, FTAG, &cnds));
2075             o = cnds->ds_dir->dd_phys->dd_head_dataset_obj;
2076
2077             VERIFY0(zap_remove_int(dp->dp_meta_objset,
2078                 odd->dd_phys->dd_clones, o, tx));
2079             VERIFY0(zap_add_int(dp->dp_meta_objset,
2080                 dd->dd_phys->dd_clones, o, tx));
2081             dsl_dataset_rele(cnds, FTAG);
2082         }
2083         zap_cursor_fini(&zc);
2084     }
2085
2086     ASSERT(!dsl_prop_hascb(ds));
2087 }
2088
2089 /*
2090  * Change space accounting.
2091  * Note, pa->usedsnap and dd_used_breakdown[SNAP] will either
2092  * both be valid, or both be 0 (resulting in delta == 0). This
2093  * is true for each of {clone,origin} independently.
2094  */
2095
2096 delta = ddpa->cloneusedsnap -
2097     dd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
2098 ASSERT3S(delta, >=, 0);
2099 ASSERT3U(ddpa->used, >=, delta);
2100 dsl_dir_diduse_space(dd, DD_USED_SNAP, delta, 0, 0, tx);
2101 dsl_dir_diduse_space(dd, DD_USED_HEAD,
2102     ddpa->used - delta, ddpa->comp, ddpa->uncomp, tx);
2103
2104 delta = ddpa->originusedsnap -
2105     odd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
2106 ASSERT3S(delta, <=, 0);
2107 ASSERT3U(ddpa->used, >=, -delta);
2108 dsl_dir_diduse_space(odd, DD_USED_SNAP, delta, 0, 0, tx);
2109 dsl_dir_diduse_space(odd, DD_USED_HEAD,

```

```

2110     -ddpa->used - delta, -ddpa->comp, -ddpa->uncomp, tx);
2111
2112     origin_ds->ds_phys->ds_unique_bytes = ddpa->unique;
2113
2114     /* log history record */
2115     spa_history_log_internal(ds, "promote", tx, "");
2116
2117     dsl_dir_rele(odd, FTAG);
2118     promote_rele(ddpa, FTAG);
2119 }
2120
2121 /*
2122  * Make a list of dsl_dataset_t's for the snapshots between first_obj
2123  * (exclusive) and last_obj (inclusive). The list will be in reverse
2124  * order (last_obj will be the list_head()). If first_obj == 0, do all
2125  * snapshots back to this dataset's origin.
2126  */
2127 static int
2128 snaplist_make(dsl_pool_t *dp,
2129     uint64_t first_obj, uint64_t last_obj, list_t *l, void *tag)
2130 {
2131     uint64_t obj = last_obj;
2132
2133     list_create(l, sizeof (struct promotenode),
2134         offsetof(struct promotenode, link));
2135
2136     while (obj != first_obj) {
2137         dsl_dataset_t *ds;
2138         struct promotenode *snap;
2139         int err;
2140
2141         err = dsl_dataset_hold_obj(dp, obj, tag, &ds);
2142         ASSERT(err != ENOENT);
2143         if (err != 0)
2144             return (err);
2145
2146         if (first_obj == 0)
2147             first_obj = ds->ds_dir->dd_phys->dd_origin_obj;
2148
2149         snap = kmem_alloc(sizeof (*snap), KM_SLEEP);
2150         snap->ds = ds;
2151         list_insert_tail(l, snap);
2152         obj = ds->ds_phys->ds_prev_snap_obj;
2153     }
2154
2155     return (0);
2156 }
2157
2158 static int
2159 snaplist_space(list_t *l, uint64_t mintxg, uint64_t *spacep)
2160 {
2161     struct promotenode *snap;
2162
2163     *spacep = 0;
2164     for (snap = list_head(l); snap; snap = list_next(l, snap)) {
2165         uint64_t used, comp, uncomp;
2166         dsl_deadlist_space_range(&snap->ds->ds_deadlist,
2167             mintxg, UINT64_MAX, &used, &comp, &uncomp);
2168         *spacep += used;
2169     }
2170     return (0);
2171 }
2172
2173 static void
2174 snaplist_destroy(list_t *l, void *tag)
2175 {

```

```

2176     struct promotenode *snap;
2177
2178     if (l == NULL || !list_link_active(&l->list_head))
2179         return;
2180
2181     while ((snap = list_tail(l)) != NULL) {
2182         list_remove(l, snap);
2183         dsl_dataset_rele(snap->ds, tag);
2184         kmem_free(snap, sizeof (*snap));
2185     }
2186     list_destroy(l);
2187 }
2188
2189 static int
2190 promote_hold(dsl_dataset_promote_arg_t *ddpa, dsl_pool_t *dp, void *tag)
2191 {
2192     int error;
2193     dsl_dir_t *dd;
2194     struct promotenode *snap;
2195
2196     error = dsl_dataset_hold(dp, ddpa->ddpa_clonename, tag,
2197         &ddpa->ddpa_clone);
2198     if (error != 0)
2199         return (error);
2200     dd = ddpa->ddpa_clone->ds_dir;
2201
2202     if (dsl_dataset_is_snapshot(ddpa->ddpa_clone) ||
2203         !dsl_dir_is_clone(dd)) {
2204         dsl_dataset_rele(ddpa->ddpa_clone, tag);
2205         return (SET_ERROR(EINVAL));
2206     }
2207
2208     error = snaplist_make(dp, 0, dd->dd_phys->dd_origin_obj,
2209         &ddpa->shared_snaps, tag);
2210     if (error != 0)
2211         goto out;
2212
2213     error = snaplist_make(dp, 0, ddpa->ddpa_clone->ds_object,
2214         &ddpa->clone_snaps, tag);
2215     if (error != 0)
2216         goto out;
2217
2218     snap = list_head(&ddpa->shared_snaps);
2219     ASSERT3U(snap->ds->ds_object, ==, dd->dd_phys->dd_origin_obj);
2220     error = snaplist_make(dp, dd->dd_phys->dd_origin_obj,
2221         snap->ds->ds_dir->dd_phys->dd_head_dataset_obj,
2222         &ddpa->origin_snaps, tag);
2223     if (error != 0)
2224         goto out;
2225
2226     if (snap->ds->ds_dir->dd_phys->dd_origin_obj != 0) {
2227         error = dsl_dataset_hold_obj(dp,
2228             snap->ds->ds_dir->dd_phys->dd_origin_obj,
2229             tag, &ddpa->origin_origin);
2230         if (error != 0)
2231             goto out;
2232     }
2233 out:
2234     if (error != 0)
2235         promote_rele(ddpa, tag);
2236     return (error);
2237 }
2238
2239 static void
2240 promote_rele(dsl_dataset_promote_arg_t *ddpa, void *tag)
2241 {

```

```

2242     snaplist_destroy(&ddpa->shared_snaps, tag);
2243     snaplist_destroy(&ddpa->clone_snaps, tag);
2244     snaplist_destroy(&ddpa->origin_snaps, tag);
2245     if (ddpa->origin_origin != NULL)
2246         dsl_dataset_rele(ddpa->origin_origin, tag);
2247     dsl_dataset_rele(ddpa->ddpa_clone, tag);
2248 }
2249
2250 /*
2251  * Promote a clone.
2252  *
2253  * If it fails due to a conflicting snapshot name, "conflsnap" will be filled
2254  * in with the name. (It must be at least MAXNAMELEN bytes long.)
2255  */
2256 int
2257 dsl_dataset_promote(const char *name, char *conflsnap)
2258 {
2259     dsl_dataset_promote_arg_t ddpa = { 0 };
2260     uint64_t numsnaps;
2261     int error;
2262     objset_t *os;
2263
2264     /*
2265      * We will modify space proportional to the number of
2266      * snapshots. Compute numsnaps.
2267      */
2268     error = dmu_objset_hold(name, FTAG, &os);
2269     if (error != 0)
2270         return (error);
2271     error = zap_count(dmu_objset_pool(os)->dp_meta_objset,
2272         dmu_objset_ds(os)->ds_phys->ds_snapnames_zapobj, &numsnaps);
2273     dmu_objset_rele(os, FTAG);
2274     if (error != 0)
2275         return (error);
2276
2277     ddpa.ddpa_clonename = name;
2278     ddpa.err_ds = conflsnap;
2279
2280     return (dsl_sync_task(name, dsl_dataset_promote_check,
2281         dsl_dataset_promote_sync, &ddpa, 2 + numsnaps));
2282 }
2283
2284 int
2285 dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
2286     dsl_dataset_t *origin_head, boolean_t force)
2287 {
2288     int64_t unused_refres_delta;
2289
2290     /* they should both be heads */
2291     if (dsl_dataset_is_snapshot(clone) ||
2292         dsl_dataset_is_snapshot(origin_head))
2293         return (SET_ERROR(EINVAL));
2294
2295     /* the branch point should be just before them */
2296     if (clone->ds_prev != origin_head->ds_prev)
2297         return (SET_ERROR(EINVAL));
2298
2299     /* clone should be the clone (unless they are unrelated) */
2300     if (clone->ds_prev != NULL &&
2301         clone->ds_prev != clone->ds_dir->dd_pool->dp_origin_snap &&
2302         origin_head->ds_object !=
2303         clone->ds_prev->ds_phys->ds_next_snap_obj)
2304         return (SET_ERROR(EINVAL));
2305
2306     /* the clone should be a child of the origin */
2307     if (clone->ds_dir->dd_parent != origin_head->ds_dir)

```

```

2308         return (SET_ERROR(EINVAL));

2310     /* origin_head shouldn't be modified unless 'force' */
2311     if (!force && dsl_dataset_modified_since_lastsnap(origin_head))
2312         return (SET_ERROR(ETXTBSY));

2314     /* origin_head should have no long holds (e.g. is not mounted) */
2315     if (dsl_dataset_long_held(origin_head))
2316         return (SET_ERROR(EBUSY));

2318     /* check amount of any unconsumed refreservation */
2319     unused_refres_delta =
2320         (int64_t)MIN(origin_head->ds_reserved,
2321             origin_head->ds_phys->ds_unique_bytes) -
2322         (int64_t)MIN(origin_head->ds_reserved,
2323             clone->ds_phys->ds_unique_bytes);

2325     if (unused_refres_delta > 0 &&
2326         unused_refres_delta >
2327         dsl_dir_space_available(origin_head->ds_dir, NULL, 0, TRUE))
2328         return (SET_ERROR(ENOSPC));

2330     /* clone can't be over the head's refquota */
2331     if (origin_head->ds_quota != 0 &&
2332         clone->ds_phys->ds_referenced_bytes > origin_head->ds_quota)
2333         return (SET_ERROR(EDQUOT));

2335     return (0);
2336 }

2338 void
2339 dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
2340     dsl_dataset_t *origin_head, dmu_tx_t *tx)
2341 {
2342     dsl_pool_t *dp = dmu_tx_pool(tx);
2343     int64_t unused_refres_delta;

2345     ASSERT(clone->ds_reserved == 0);
2346     ASSERT(origin_head->ds_quota == 0 ||
2347         clone->ds_phys->ds_unique_bytes <= origin_head->ds_quota);

2349     dmu_buf_will_dirty(clone->ds_dbuf, tx);
2350     dmu_buf_will_dirty(origin_head->ds_dbuf, tx);

2352     if (clone->ds_objset != NULL) {
2353         dmu_objset_evict(clone->ds_objset);
2354         clone->ds_objset = NULL;
2355     }

2357     if (origin_head->ds_objset != NULL) {
2358         dmu_objset_evict(origin_head->ds_objset);
2359         origin_head->ds_objset = NULL;
2360     }

2362     unused_refres_delta =
2363         (int64_t)MIN(origin_head->ds_reserved,
2364             origin_head->ds_phys->ds_unique_bytes) -
2365         (int64_t)MIN(origin_head->ds_reserved,
2366             clone->ds_phys->ds_unique_bytes);

2368     /*
2369      * Reset origin's unique bytes, if it exists.
2370      */
2371     if (clone->ds_prev) {
2372         dsl_dataset_t *origin = clone->ds_prev;
2373         uint64_t comp, uncomp;

```

```

2375         dmu_buf_will_dirty(origin->ds_dbuf, tx);
2376         dsl_deadlist_space_range(&clone->ds_deadlist,
2377             origin->ds_phys->ds_prev_snap_txg, UINT64_MAX,
2378             &origin->ds_phys->ds_unique_bytes, &comp, &uncomp);
2379     }

2381     /* swap blkptrs */
2382     {
2383         blkptr_t tmp;
2384         tmp = origin_head->ds_phys->ds_bp;
2385         origin_head->ds_phys->ds_bp = clone->ds_phys->ds_bp;
2386         clone->ds_phys->ds_bp = tmp;
2387     }

2389     /* set dd_*_bytes */
2390     {
2391         int64_t dused, dcomp, duncomp;
2392         uint64_t cdl_used, cdl_comp, cdl_uncomp;
2393         uint64_t odl_used, odl_comp, odl_uncomp;

2395         ASSERT3U(clone->ds_dir->dd_phys->
2396             dd_used_breakdown[DD_USED_SNAP], ==, 0);

2398         dsl_deadlist_space(&clone->ds_deadlist,
2399             &cdl_used, &cdl_comp, &cdl_uncomp);
2400         dsl_deadlist_space(&origin_head->ds_deadlist,
2401             &odl_used, &odl_comp, &odl_uncomp);

2403         dused = clone->ds_phys->ds_referenced_bytes + cdl_used -
2404             (origin_head->ds_phys->ds_referenced_bytes + odl_used);
2405         dcomp = clone->ds_phys->ds_compressed_bytes + cdl_comp -
2406             (origin_head->ds_phys->ds_compressed_bytes + odl_comp);
2407         duncomp = clone->ds_phys->ds_uncompressed_bytes +
2408             cdl_uncomp -
2409             (origin_head->ds_phys->ds_uncompressed_bytes + odl_uncomp);

2411         dsl_dir_diduse_space(origin_head->ds_dir, DD_USED_HEAD,
2412             dused, dcomp, duncomp, tx);
2413         dsl_dir_diduse_space(clone->ds_dir, DD_USED_HEAD,
2414             -dused, -dcomp, -duncomp, tx);

2416         /*
2417          * The difference in the space used by snapshots is the
2418          * difference in snapshot space due to the head's
2419          * deadlist (since that's the only thing that's
2420          * changing that affects the snapused).
2421          */
2422         dsl_deadlist_space_range(&clone->ds_deadlist,
2423             origin_head->ds_dir->dd_origin_txg, UINT64_MAX,
2424             &cdl_used, &cdl_comp, &cdl_uncomp);
2425         dsl_deadlist_space_range(&origin_head->ds_deadlist,
2426             origin_head->ds_dir->dd_origin_txg, UINT64_MAX,
2427             &odl_used, &odl_comp, &odl_uncomp);
2428         dsl_dir_transfer_space(origin_head->ds_dir, cdl_used - odl_used,
2429             DD_USED_HEAD, DD_USED_SNAP, tx);
2430     }

2432     /* swap ds_*_bytes */
2433     SWITCH64(origin_head->ds_phys->ds_referenced_bytes,
2434         clone->ds_phys->ds_referenced_bytes);
2435     SWITCH64(origin_head->ds_phys->ds_compressed_bytes,
2436         clone->ds_phys->ds_compressed_bytes);
2437     SWITCH64(origin_head->ds_phys->ds_uncompressed_bytes,
2438         clone->ds_phys->ds_uncompressed_bytes);
2439     SWITCH64(origin_head->ds_phys->ds_unique_bytes,

```



```

2440     clone->ds_phys->ds_unique_bytes);

2442     /* apply any parent delta for change in unconsumed refreservation */
2443     dsl_dir_diduse_space(origin_head->ds_dir, DD_USED_REFRSRV,
2444         unused_refres_delta, 0, 0, tx);

2446     /*
2447      * Swap deadlists.
2448      */
2449     dsl_deadlist_close(&clone->ds_deadlist);
2450     dsl_deadlist_close(&origin_head->ds_deadlist);
2451     SWITCH64(origin_head->ds_phys->ds_deadlist_obj,
2452         clone->ds_phys->ds_deadlist_obj);
2453     dsl_deadlist_open(&clone->ds_deadlist, dp->dp_meta_objset,
2454         clone->ds_phys->ds_deadlist_obj);
2455     dsl_deadlist_open(&origin_head->ds_deadlist, dp->dp_meta_objset,
2456         origin_head->ds_phys->ds_deadlist_obj);

2458     dsl_scan_ds_clone_swapped(origin_head, clone, tx);

2460     spa_history_log_internal_ds(clone, "clone swap", tx,
2461         "parent=%s", origin_head->ds_dir->dd_myname);
2462 }

2464 /*
2465  * Given a pool name and a dataset object number in that pool,
2466  * return the name of that dataset.
2467  */
2468 int
2469 dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf)
2470 {
2471     dsl_pool_t *dp;
2472     dsl_dataset_t *ds;
2473     int error;

2475     error = dsl_pool_hold(pname, FTAG, &dp);
2476     if (error != 0)
2477         return (error);

2479     error = dsl_dataset_hold_obj(dp, obj, FTAG, &ds);
2480     if (error == 0) {
2481         dsl_dataset_name(ds, buf);
2482         dsl_dataset_rele(ds, FTAG);
2483     }
2484     dsl_pool_rele(dp, FTAG);

2486     return (error);
2487 }

2489 int
2490 dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
2491     uint64_t asize, uint64_t inflight, uint64_t *used, uint64_t *ref_rsrv)
2492 {
2493     int error = 0;

2495     ASSERT3S(asize, >, 0);

2497     /*
2498      * *ref_rsrv is the portion of asize that will come from any
2499      * unconsumed refreservation space.
2500      */
2501     *ref_rsrv = 0;

2503     mutex_enter(&ds->ds_lock);
2504     /*
2505      * Make a space adjustment for reserved bytes.

```

```

2506     /*
2507     if (ds->ds_reserved > ds->ds_phys->ds_unique_bytes) {
2508         ASSERT3U(*used, >=,
2509             ds->ds_reserved - ds->ds_phys->ds_unique_bytes);
2510         *used -= (ds->ds_reserved - ds->ds_phys->ds_unique_bytes);
2511         *ref_rsrv =
2512             asize - MIN(asize, parent_delta(ds, asize + inflight));
2513     }

2515     if (!check_quota || ds->ds_quota == 0) {
2516         mutex_exit(&ds->ds_lock);
2517         return (0);
2518     }
2519     /*
2520     * If they are requesting more space, and our current estimate
2521     * is over quota, they get to try again unless the actual
2522     * on-disk is over quota and there are no pending changes (which
2523     * may free up space for us).
2524     */
2525     if (ds->ds_phys->ds_referenced_bytes + inflight >= ds->ds_quota) {
2526         if (inflight > 0 ||
2527             ds->ds_phys->ds_referenced_bytes < ds->ds_quota)
2528             error = SET_ERROR(ERESTART);
2529         else
2530             error = SET_ERROR(EDQUOT);
2531     }
2532     mutex_exit(&ds->ds_lock);

2534     return (error);
2535 }

2537 typedef struct dsl_dataset_set_qr_arg {
2538     const char *ddsqra_name;
2539     zprop_source_t ddsqra_source;
2540     uint64_t ddsqra_value;
2541 } dsl_dataset_set_qr_arg_t;

2544 /* ARGSUSED */
2545 static int
2546 dsl_dataset_set_refquota_check(void *arg, dmu_tx_t *tx)
2547 {
2548     dsl_dataset_set_qr_arg_t *ddsqra = arg;
2549     dsl_pool_t *dp = dmu_tx_pool(tx);
2550     dsl_dataset_t *ds;
2551     int error;
2552     uint64_t newval;

2554     if (spa_version(dp->dp_spa) < SPA_VERSION_REFQUOTA)
2555         return (SET_ERROR(ENOTSUP));

2557     error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
2558     if (error != 0)
2559         return (error);

2561     if (dsl_dataset_is_snapshot(ds)) {
2562         dsl_dataset_rele(ds, FTAG);
2563         return (SET_ERROR(EINVAL));
2564     }

2566     error = dsl_prop_predict(ds->ds_dir,
2567         zfs_prop_to_name(ZFS_PROP_REFQUOTA),
2568         ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
2569     if (error != 0) {
2570         dsl_dataset_rele(ds, FTAG);
2571         return (error);

```

```

2572     }

2574     if (newval == 0) {
2575         dsl_dataset_rele(ds, FTAG);
2576         return (0);
2577     }

2579     if (newval < ds->ds_phys->ds_referenced_bytes ||
2580         newval < ds->ds_reserved) {
2581         dsl_dataset_rele(ds, FTAG);
2582         return (SET_ERROR(ENOSPC));
2583     }

2585     dsl_dataset_rele(ds, FTAG);
2586     return (0);
2587 }

2589 static void
2590 dsl_dataset_set_refquota_sync(void *arg, dmu_tx_t *tx)
2591 {
2592     dsl_dataset_set_qr_arg_t *ddsqra = arg;
2593     dsl_pool_t *dp = dmu_tx_pool(tx);
2594     dsl_dataset_t *ds;
2595     uint64_t newval;

2597     VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds));

2599     dsl_prop_set_sync_impl(ds,
2600         zfs_prop_to_name(ZFS_PROP_REFQUOTA),
2601         ddsqra->ddsqra_source, sizeof (ddsqra->ddsqra_value), 1,
2602         &ddsqra->ddsqra_value, tx);

2604     VERIFY0(dsl_prop_get_int_ds(ds,
2605         zfs_prop_to_name(ZFS_PROP_REFQUOTA), &newval));

2607     if (ds->ds_quota != newval) {
2608         dmu_buf_will_dirty(ds->ds_dbuf, tx);
2609         ds->ds_quota = newval;
2610     }
2611     dsl_dataset_rele(ds, FTAG);
2612 }

2614 int
2615 dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
2616     uint64_t refquota)
2617 {
2618     dsl_dataset_set_qr_arg_t ddsqra;

2620     ddsqra.ddsqra_name = dsname;
2621     ddsqra.ddsqra_source = source;
2622     ddsqra.ddsqra_value = refquota;

2624     return (dsl_sync_task(dsname, dsl_dataset_set_refquota_check,
2625         dsl_dataset_set_refquota_sync, &ddsqra, 0));
2626 }

2628 static int
2629 dsl_dataset_set_refreservation_check(void *arg, dmu_tx_t *tx)
2630 {
2631     dsl_dataset_set_qr_arg_t *ddsqra = arg;
2632     dsl_pool_t *dp = dmu_tx_pool(tx);
2633     dsl_dataset_t *ds;
2634     int error;
2635     uint64_t newval, unique;

2637     if (spa_version(dp->dp_spa) < SPA_VERSION_REFRESERVATION)

```

```

2638         return (SET_ERROR(ENOTSUP));

2640     error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
2641     if (error != 0)
2642         return (error);

2644     if (dsl_dataset_is_snapshot(ds)) {
2645         dsl_dataset_rele(ds, FTAG);
2646         return (SET_ERROR(EINVAL));
2647     }

2649     error = dsl_prop_predict(ds->ds_dir,
2650         zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
2651         ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
2652     if (error != 0) {
2653         dsl_dataset_rele(ds, FTAG);
2654         return (error);
2655     }

2657     /*
2658      * If we are doing the preliminary check in open context, the
2659      * space estimates may be inaccurate.
2660      */
2661     if (!dmu_tx_is_syncing(tx)) {
2662         dsl_dataset_rele(ds, FTAG);
2663         return (0);
2664     }

2666     mutex_enter(&ds->ds_lock);
2667     if (!DS_UNIQUE_IS_ACCURATE(ds))
2668         dsl_dataset_recalc_head_uniq(ds);
2669     unique = ds->ds_phys->ds_unique_bytes;
2670     mutex_exit(&ds->ds_lock);

2672     if (MAX(unique, newval) > MAX(unique, ds->ds_reserved)) {
2673         uint64_t delta = MAX(unique, newval) -
2674             MAX(unique, ds->ds_reserved);

2676         if (delta >
2677             dsl_dir_space_available(ds->ds_dir, NULL, 0, B_TRUE) ||
2678             (ds->ds_quota > 0 && newval > ds->ds_quota)) {
2679             dsl_dataset_rele(ds, FTAG);
2680             return (SET_ERROR(ENOSPC));
2681         }
2682     }

2684     dsl_dataset_rele(ds, FTAG);
2685     return (0);
2686 }

2688 void
2689 dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
2690     zprop_source_t source, uint64_t value, dmu_tx_t *tx)
2691 {
2692     uint64_t newval;
2693     uint64_t unique;
2694     int64_t delta;

2696     dsl_prop_set_sync_impl(ds, zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
2697         source, sizeof (value), 1, &value, tx);

2699     VERIFY0(dsl_prop_get_int_ds(ds,
2700         zfs_prop_to_name(ZFS_PROP_REFRESERVATION), &newval));

2702     dmu_buf_will_dirty(ds->ds_dbuf, tx);
2703     mutex_enter(&ds->ds_dir->dd_lock);

```

```

2704     mutex_enter(&ds->ds_lock);
2705     ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
2706     unique = ds->ds_phys->ds_unique_bytes;
2707     delta = MAX(0, (int64_t)(newval - unique)) -
2708         MAX(0, (int64_t)(ds->ds_reserved - unique));
2709     ds->ds_reserved = newval;
2710     mutex_exit(&ds->ds_lock);

2712     dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV, delta, 0, 0, tx);
2713     mutex_exit(&ds->ds_dir->dd_lock);
2714 }

2716 static void
2717 dsl_dataset_set_refreservation_sync(void *arg, dmu_tx_t *tx)
2718 {
2719     dsl_dataset_set_qr_arg_t *ddsgra = arg;
2720     dsl_pool_t *dp = dmu_tx_pool(tx);
2721     dsl_dataset_t *ds;

2723     VERIFY0(dsl_dataset_hold(dp, ddsgra->ddsgra_name, FTAG, &ds));
2724     dsl_dataset_set_refreservation_sync_impl(ds,
2725         ddsgra->ddsgra_source, ddsgra->ddsgra_value, tx);
2726     dsl_dataset_rele(ds, FTAG);
2727 }

2729 int
2730 dsl_dataset_set_refreservation(const char *dsname, zprop_source_t source,
2731     uint64_t refreservation)
2732 {
2733     dsl_dataset_set_qr_arg_t ddsgra;

2735     ddsgra.ddsgra_name = dsname;
2736     ddsgra.ddsgra_source = source;
2737     ddsgra.ddsgra_value = refreservation;

2739     return (dsl_sync_task(dsname, dsl_dataset_set_refreservation_check,
2740         dsl_dataset_set_refreservation_sync, &ddsgra, 0));
2741 }

2743 /*
2744  * Return (in *usedp) the amount of space written in new that is not
2745  * present in oldsnap. New may be a snapshot or the head. Old must be
2746  * a snapshot before new, in new's filesystem (or its origin). If not then
2747  * fail and return EINVAL.
2748  *
2749  * The written space is calculated by considering two components: First, we
2750  * ignore any freed space, and calculate the written as new's used space
2751  * minus old's used space. Next, we add in the amount of space that was freed
2752  * between the two snapshots, thus reducing new's used space relative to old's.
2753  * Specifically, this is the space that was born before old->ds_creation_txg,
2754  * and freed before new (ie. on new's deadlist or a previous deadlist).
2755  *
2756  * space freed          [-----]
2757  * snapshots          ---O-----O-----O-----
2758  *                      oldsnap          new
2759  */
2760 int
2761 dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
2762     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
2763 {
2764     int err = 0;
2765     uint64_t snapobj;
2766     dsl_pool_t *dp = new->ds_dir->dd_pool;

2768     ASSERT(dsl_pool_config_held(dp));

```

```

2770     *usedp = 0;
2771     *usedp += new->ds_phys->ds_referenced_bytes;
2772     *usedp -= oldsnap->ds_phys->ds_referenced_bytes;

2774     *compp = 0;
2775     *compp += new->ds_phys->ds_compressed_bytes;
2776     *compp -= oldsnap->ds_phys->ds_compressed_bytes;

2778     *uncompp = 0;
2779     *uncompp += new->ds_phys->ds_uncompressed_bytes;
2780     *uncompp -= oldsnap->ds_phys->ds_uncompressed_bytes;

2782     snapobj = new->ds_object;
2783     while (snapobj != oldsnap->ds_object) {
2784         dsl_dataset_t *snap;
2785         uint64_t used, comp, uncomp;

2787         if (snapobj == new->ds_object) {
2788             snap = new;
2789         } else {
2790             err = dsl_dataset_hold_obj(dp, snapobj, FTAG, &snap);
2791             if (err != 0)
2792                 break;
2793         }

2795         if (snap->ds_phys->ds_prev_snap_txg ==
2796             oldsnap->ds_phys->ds_creation_txg) {
2797             /*
2798              * The blocks in the deadlist can not be born after
2799              * ds_prev_snap_txg, so get the whole deadlist space,
2800              * which is more efficient (especially for old-format
2801              * deadlists). Unfortunately the deadlist code
2802              * doesn't have enough information to make this
2803              * optimization itself.
2804              */
2805             dsl_deadlist_space(&snap->ds_deadlist,
2806                 &used, &comp, &uncomp);
2807         } else {
2808             dsl_deadlist_space_range(&snap->ds_deadlist,
2809                 0, oldsnap->ds_phys->ds_creation_txg,
2810                 &used, &comp, &uncomp);
2811         }
2812         *usedp += used;
2813         *compp += comp;
2814         *uncompp += uncomp;

2816         /*
2817          * If we get to the beginning of the chain of snapshots
2818          * (ds_prev_snap_obj == 0) before oldsnap, then oldsnap
2819          * was not a snapshot of/before new.
2820          */
2821         snapobj = snap->ds_phys->ds_prev_snap_obj;
2822         if (snap != new)
2823             dsl_dataset_rele(snap, FTAG);
2824         if (snapobj == 0) {
2825             err = SET_ERROR(EINVAL);
2826             break;
2827         }

2829     }
2830     return (err);
2831 }

2833 /*
2834  * Return (in *usedp) the amount of space that will be reclaimed if firstsnap,
2835  * lastsnap, and all snapshots in between are deleted.

```

```

2836 *
2837 * blocks that would be freed      [-----]
2838 * snapshots      ---0-----0-----0-----0
2839 *                firstsnap      lastsnap
2840 *
2841 * This is the set of blocks that were born after the snap before firstsnap,
2842 * (birth > firstsnap->prev_snap_txg) and died before the snap after the
2843 * last snap (ie, is on lastsnap->ds_next->ds_deadlist or an earlier deadlist).
2844 * We calculate this by iterating over the relevant deadlists (from the snap
2845 * after lastsnap, backward to the snap after firstsnap), summing up the
2846 * space on the deadlist that was born after the snap before firstsnap.
2847 */
2848 int
2849 dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap,
2850     dsl_dataset_t *lastsnap,
2851     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
2852 {
2853     int err = 0;
2854     uint64_t snapobj;
2855     dsl_pool_t *dp = firstsnap->ds_dir->dd_pool;
2856
2857     ASSERT(dsl_dataset_is_snapshot(firstsnap));
2858     ASSERT(dsl_dataset_is_snapshot(lastsnap));
2859
2860     /*
2861      * Check that the snapshots are in the same dsl_dir, and firstsnap
2862      * is before lastsnap.
2863      */
2864     if (firstsnap->ds_dir != lastsnap->ds_dir ||
2865         firstsnap->ds_phys->ds_creation_txg >
2866         lastsnap->ds_phys->ds_creation_txg)
2867         return (SET_ERROR(EINVAL));
2868
2869     *usedp = *compp = *uncompp = 0;
2870
2871     snapobj = lastsnap->ds_phys->ds_next_snap_obj;
2872     while (snapobj != firstsnap->ds_object) {
2873         dsl_dataset_t *ds;
2874         uint64_t used, comp, uncomp;
2875
2876         err = dsl_dataset_hold_obj(dp, snapobj, FTAG, &ds);
2877         if (err != 0)
2878             break;
2879
2880         dsl_deadlist_space_range(&ds->ds_deadlist,
2881             firstsnap->ds_phys->ds_prev_snap_txg, UINT64_MAX,
2882             &used, &comp, &uncomp);
2883         *usedp += used;
2884         *compp += comp;
2885         *uncompp += uncomp;
2886
2887         snapobj = ds->ds_phys->ds_prev_snap_obj;
2888         ASSERT3U(snapobj, !=, 0);
2889         dsl_dataset_rele(ds, FTAG);
2890     }
2891     return (err);
2892 }
2893
2894 /*
2895 * Return TRUE if 'earlier' is an earlier snapshot in 'later's timeline.
2896 * For example, they could both be snapshots of the same filesystem, and
2897 * 'earlier' is before 'later'. Or 'earlier' could be the origin of
2898 * 'later's filesystem. Or 'earlier' could be an older snapshot in the origin's
2899 * filesystem. Or 'earlier' could be the origin's origin.
2900 */
2901 boolean_t

```

```

2902 dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier)
2903 {
2904     dsl_pool_t *dp = later->ds_dir->dd_pool;
2905     int error;
2906     boolean_t ret;
2907
2908     ASSERT(dsl_pool_config_held(dp));
2909
2910     if (earlier->ds_phys->ds_creation_txg >=
2911         later->ds_phys->ds_creation_txg)
2912         return (B_FALSE);
2913
2914     if (later->ds_dir == earlier->ds_dir)
2915         return (B_TRUE);
2916     if (!dsl_dir_is_clone(later->ds_dir))
2917         return (B_FALSE);
2918
2919     if (later->ds_dir->dd_phys->dd_origin_obj == earlier->ds_object)
2920         return (B_TRUE);
2921     dsl_dataset_t *origin;
2922     error = dsl_dataset_hold_obj(dp,
2923         later->ds_dir->dd_phys->dd_origin_obj, FTAG, &origin);
2924     if (error != 0)
2925         return (B_FALSE);
2926     ret = dsl_dataset_is_before(origin, earlier);
2927     dsl_dataset_rele(origin, FTAG);
2928     return (ret);
2929 }

```

new/usr/src/uts/common/fs/zfs/spa\_errlog.c

1

```
*****
11322 Thu May 16 14:21:06 2013
new/usr/src/uts/common/fs/zfs/spa_errlog.c
3743 zfs needs a refcount audit
Submitted by: Will Andrews <willa@spectrallogic.com>
Submitted by: Justin Gibbs <justing@spectrallogic.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Eric Schrock <eric.schrock@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
*****
_____unchanged_portion_omitted_____

162 #ifdef _KERNEL
163 static int
164 process_error_log(spa_t *spa, uint64_t obj, void *addr, size_t *count)
165 {
166     zap_cursor_t zc;
167     zap_attribute_t za;
168     zbookmark_t zb;

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

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

177         if (*count == 0) {
178             zap_cursor_fini(&zc);
179             return (SET_ERROR(ENOMEM));
180         }

182         name_to_bookmark(za.za_name, &zb);

184         if (copyout(&zb, (char *)addr +
185                    (*count - 1) * sizeof (zbookmark_t),
186                    sizeof (zbookmark_t)) != 0) {
187             zap_cursor_fini(&zc);
188             return (SET_ERROR(EFAULT));
189         }
190     }
191 #endif /* ! codereview */

192     *count -= 1;
193 }

195 zap_cursor_fini(&zc);

197 return (0);
198 }

200 static int
201 process_error_list(avl_tree_t *list, void *addr, size_t *count)
202 {
203     spa_error_entry_t *se;

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

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

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

new/usr/src/uts/common/fs/zfs/spa\_errlog.c

2

```
215         *count -= 1;
216     }

218     return (0);
219 }
220 #endif

222 /*
223  * Copy all known errors to userland as an array of bookmarks. This is
224  * actually a union of the on-disk last log and current log, as well as any
225  * pending error requests.
226  *
227  * Because the act of reading the on-disk log could cause errors to be
228  * generated, we have two separate locks: one for the error log and one for the
229  * in-core error lists. We only need the error list lock to log and error, so
230  * we grab the error log lock while we read the on-disk logs, and only pick up
231  * the error list lock when we are finished.
232  */
233 int
234 spa_get_errlog(spa_t *spa, void *uaddr, size_t *count)
235 {
236     int ret = 0;

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

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

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

247     mutex_enter(&spa->spa_errlist_lock);
248     if (!ret)
249         ret = process_error_list(&spa->spa_errlist_scrub, uaddr,
250                                count);
251     if (!ret)
252         ret = process_error_list(&spa->spa_errlist_last, uaddr,
253                                count);
254     mutex_exit(&spa->spa_errlist_lock);

256     mutex_exit(&spa->spa_errlog_lock);
257 #endif

259     return (ret);
260 }

262 /*
263  * Called when a scrub completes. This simply set a bit which tells which AVL
264  * tree to add new errors. spa_errlog_sync() is responsible for actually
265  * syncing the changes to the underlying objects.
266  */
267 void
268 spa_errlog_rotate(spa_t *spa)
269 {
270     mutex_enter(&spa->spa_errlist_lock);
271     spa->spa_scrub_finished = B_TRUE;
272     mutex_exit(&spa->spa_errlist_lock);
273 }

275 /*
276  * Discard any pending errors from the spa_t. Called when unloading a faulted
277  * pool, as the errors encountered during the open cannot be synced to disk.
278  */
279 void
280 spa_errlog_drain(spa_t *spa)
```

```

281 {
282     spa_error_entry_t *se;
283     void *cookie;
284
285     mutex_enter(&spa->spa_errlist_lock);
286
287     cookie = NULL;
288     while ((se = avl_destroy_nodes(&spa->spa_errlist_last,
289     &cookie)) != NULL)
290         kmem_free(se, sizeof (spa_error_entry_t));
291     cookie = NULL;
292     while ((se = avl_destroy_nodes(&spa->spa_errlist_scrub,
293     &cookie)) != NULL)
294         kmem_free(se, sizeof (spa_error_entry_t));
295
296     mutex_exit(&spa->spa_errlist_lock);
297 }
298
299 /*
300  * Process a list of errors into the current on-disk log.
301  */
302 static void
303 sync_error_list(spa_t *spa, avl_tree_t *t, uint64_t *obj, dmu_tx_t *tx)
304 {
305     spa_error_entry_t *se;
306     char buf[64];
307     void *cookie;
308
309     if (avl_numnodes(t) != 0) {
310         /* create log if necessary */
311         if (*obj == 0)
312             *obj = zap_create(spa->spa_meta_objset,
313             DMU_OT_ERROR_LOG, DMU_OT_NONE,
314             0, tx);
315
316         /* add errors to the current log */
317         for (se = avl_first(t); se != NULL; se = AVL_NEXT(t, se)) {
318             char *name = se->se_name ? se->se_name : "";
319
320             bookmark_to_name(&se->se_bookmark, buf, sizeof (buf));
321
322             (void) zap_update(spa->spa_meta_objset,
323             *obj, buf, 1, strlen(name) + 1, name, tx);
324         }
325
326         /* purge the error list */
327         cookie = NULL;
328         while ((se = avl_destroy_nodes(t, &cookie)) != NULL)
329             kmem_free(se, sizeof (spa_error_entry_t));
330     }
331 }
332
333 /*
334  * Sync the error log out to disk. This is a little tricky because the act of
335  * writing the error log requires the spa_errlist_lock. So, we need to lock the
336  * error lists, take a copy of the lists, and then reinitialize them. Then, we
337  * drop the error list lock and take the error log lock, at which point we
338  * do the errlog processing. Then, if we encounter an I/O error during this
339  * process, we can successfully add the error to the list. Note that this will
340  * result in the perpetual recycling of errors, but it is an unlikely situation
341  * and not a performance critical operation.
342  */
343 void
344 spa_errlog_sync(spa_t *spa, uint64_t txg)
345 {
346     dmu_tx_t *tx;

```

```

347     avl_tree_t scrub, last;
348     int scrub_finished;
349
350     mutex_enter(&spa->spa_errlist_lock);
351
352     /*
353      * Bail out early under normal circumstances.
354      */
355     if (avl_numnodes(&spa->spa_errlist_scrub) == 0 &&
356     avl_numnodes(&spa->spa_errlist_last) == 0 &&
357     !spa->spa_scrub_finished) {
358         mutex_exit(&spa->spa_errlist_lock);
359         return;
360     }
361
362     spa_get_errlists(spa, &last, &scrub);
363     scrub_finished = spa->spa_scrub_finished;
364     spa->spa_scrub_finished = B_FALSE;
365
366     mutex_exit(&spa->spa_errlist_lock);
367     mutex_enter(&spa->spa_errlog_lock);
368
369     tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);
370
371     /*
372      * Sync out the current list of errors.
373      */
374     sync_error_list(spa, &last, &spa->spa_errlog_last, tx);
375
376     /*
377      * Rotate the log if necessary.
378      */
379     if (scrub_finished) {
380         if (spa->spa_errlog_last != 0)
381             VERIFY(dmu_object_free(spa->spa_meta_objset,
382             spa->spa_errlog_last, tx) == 0);
383         spa->spa_errlog_last = spa->spa_errlog_scrub;
384         spa->spa_errlog_scrub = 0;
385
386         sync_error_list(spa, &scrub, &spa->spa_errlog_last, tx);
387     }
388
389     /*
390      * Sync out any pending scrub errors.
391      */
392     sync_error_list(spa, &scrub, &spa->spa_errlog_scrub, tx);
393
394     /*
395      * Update the MOS to reflect the new values.
396      */
397     (void) zap_update(spa->spa_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
398     DMU_POOL_ERRLOG_LAST, sizeof (uint64_t), 1,
399     &spa->spa_errlog_last, tx);
400     (void) zap_update(spa->spa_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
401     DMU_POOL_ERRLOG_SCRUB, sizeof (uint64_t), 1,
402     &spa->spa_errlog_scrub, tx);
403
404     dmu_tx_commit(tx);
405
406     mutex_exit(&spa->spa_errlog_lock);
407 }

```

```

*****
33995 Thu May 16 14:21:06 2013
new/usr/src/uts/common/fs/zfs/zap.c
3743 zfs needs a refcount audit
Submitted by: Will Andrews <willa@spectrallogic.com>
Submitted by: Justin Gibbs <justing@spectrallogic.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Eric Schrock <eric.schrock@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
*****
_____unchanged_portion_omitted_____

267 static int
268 zap_table_load(zap_t *zap, zap_table_phys_t *tbl, uint64_t idx, uint64_t *valp)
269 {
270     uint64_t blk, off;
271     int err;
272     dmu_buf_t *db;
273     int bs = FZAP_BLOCK_SHIFT(zap);

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

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

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

287     if (tbl->zt_nextblk != 0) {
288         /*
289          * read the nextblk for the sake of i/o error checking,
290          * so that zap_table_load() will catch errors for
291          * zap_table_store.
292          */
293         blk = (idx*2) >> (bs-3);

295         err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
296             (tbl->zt_nextblk + blk) << bs, FTAG, &db,
297             DMU_READ_NO_PREFETCH);
298         if (err == 0)
299             #endif /* ! codereview */
300             dmu_buf_rele(db, FTAG);
301     }
302     return (err);
303 }

305 /*
306  * Routines for growing the ptrtbl.
307  */

309 static void
310 zap_ptrtbl_transfer(const uint64_t *src, uint64_t *dst, int n)
311 {
312     int i;
313     for (i = 0; i < n; i++) {
314         uint64_t lb = src[i];
315         dst[2*i+0] = lb;
316         dst[2*i+1] = lb;
317     }
318 }

320 static int

```

```

321 zap_grow_ptrtbl(zap_t *zap, dmu_tx_t *tx)
322 {
323     /*
324      * The pointer table should never use more hash bits than we
325      * have (otherwise we'd be using useless zero bits to index it).
326      * If we are within 2 bits of running out, stop growing, since
327      * this is already an aberrant condition.
328      */
329     if (zap->zap_f.zap_phys->zap_ptrtbl.zt_shift >= zap_hashbits(zap) - 2)
330         return (SET_ERROR(ENOSPC));

332     if (zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks == 0) {
333         /*
334          * We are outgrowing the "embedded" ptrtbl (the one
335          * stored in the header block). Give it its own entire
336          * block, which will double the size of the ptrtbl.
337          */
338         uint64_t newblk;
339         dmu_buf_t *db_new;
340         int err;

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

346         newblk = zap_allocate_blocks(zap, 1);
347         err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
348             newblk << FZAP_BLOCK_SHIFT(zap), FTAG, &db_new,
349             DMU_READ_NO_PREFETCH);
350         if (err)
351             return (err);
352         dmu_buf_will_dirty(db_new, tx);
353         zap_ptrtbl_transfer(&ZAP_EMBEDDED_PTRTBL_ENT(zap, 0),
354             db_new->db_data, 1 << ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
355         dmu_buf_rele(db_new, FTAG);

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

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

365         return (0);
366     } else {
367         return (zap_table_grow(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
368             zap_ptrtbl_transfer, tx));
369     }
370 }

372 static void
373 zap_increment_num_entries(zap_t *zap, int delta, dmu_tx_t *tx)
374 {
375     dmu_buf_will_dirty(zap->zap_dbuf, tx);
376     mutex_enter(&zap->zap_f.zap_num_entries_mtx);
377     ASSERT(delta > 0 || zap->zap_f.zap_phys->zap_num_entries >= -delta);
378     zap->zap_f.zap_phys->zap_num_entries += delta;
379     mutex_exit(&zap->zap_f.zap_num_entries_mtx);
380 }

382 static uint64_t
383 zap_allocate_blocks(zap_t *zap, int nblocks)
384 {
385     uint64_t newblk;
386     ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));

```

```

387     newblk = zap->zap_phys->zap_freeblk;
388     zap->zap_f.zap_phys->zap_freeblk += nblocks;
389     return (newblk);
390 }

392 static zap_leaf_t *
393 zap_create_leaf(zap_t *zap, dmu_tx_t *tx)
394 {
395     void *winner;
396     zap_leaf_t *l = kmem_alloc(sizeof (zap_leaf_t), KM_SLEEP);

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

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

406     VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
407         l->l_blkid << FZAP_BLOCK_SHIFT(zap), NULL, &l->l_dbuf,
408         DMU_READ_NO_PREFETCH));
409     winner = dmu_buf_set_user(l->l_dbuf, 1, &l->l_phys, zap_leaf_pageout);
410     ASSERT(winner == NULL);
411     dmu_buf_will_dirty(l->l_dbuf, tx);

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

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

417     return (l);
418 }

420 int
421 fzap_count(zap_t *zap, uint64_t *count)
422 {
423     ASSERT(!zap->zap_ismicro);
424     mutex_enter(&zap->zap_f.zap_num_entries_mtx); /* unnecessary */
425     *count = zap->zap_f.zap_phys->zap_num_entries;
426     mutex_exit(&zap->zap_f.zap_num_entries_mtx);
427     return (0);
428 }

430 /*
431  * Routines for obtaining zap_leaf_t's
432  */

434 void
435 zap_put_leaf(zap_leaf_t *l)
436 {
437     rw_exit(&l->l_rwlock);
438     dmu_buf_rele(l->l_dbuf, NULL);
439 }

441 _NOTE(ARGUSED(0))
442 static void
443 zap_leaf_pageout(dmu_buf_t *db, void *vl)
444 {
445     zap_leaf_t *l = vl;

447     rw_destroy(&l->l_rwlock);
448     kmem_free(l, sizeof (zap_leaf_t));
449 }

451 static zap_leaf_t *
452 zap_open_leaf(uint64_t blkid, dmu_buf_t *db)

```

```

453 {
454     zap_leaf_t *l, *winner;

456     ASSERT(blkid != 0);

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

466     winner = dmu_buf_set_user(db, 1, &l->l_phys, zap_leaf_pageout);

468     rw_exit(&l->l_rwlock);
469     if (winner != NULL) {
470         /* someone else set it first */
471         zap_leaf_pageout(NULL, l);
472         l = winner;
473     }

475     /*
476      * lhr_pad was previously used for the next leaf in the leaf
477      * chain. There should be no chained leafs (as we have removed
478      * support for them).
479      */
480     ASSERT0(l->l_phys->l_hdr.lh_pad1);

482     /*
483      * There should be more hash entries than there can be
484      * chunks to put in the hash table
485      */
486     ASSERT3U(ZAP_LEAF_HASH_NUMENTRIES(1), >, ZAP_LEAF_NUMCHUNKS(1) / 3);

488     /* The chunks should begin at the end of the hash table */
489     ASSERT3P(&ZAP_LEAF_CHUNK(1, 0), ==,
490         &l->l_phys->l_hash[ZAP_LEAF_HASH_NUMENTRIES(1)]);

492     /* The chunks should end at the end of the block */
493     ASSERT3U((uintptr_t)&ZAP_LEAF_CHUNK(1, ZAP_LEAF_NUMCHUNKS(1)) -
494         (uintptr_t)l->l_phys, ==, l->l_dbuf->db_size);

496     return (l);
497 }

499 static int
500 zap_get_leaf_byblk(zap_t *zap, uint64_t blkid, dmu_tx_t *tx, krw_t lt,
501     zap_leaf_t **lp)
502 {
503     dmu_buf_t *db;
504     zap_leaf_t *l;
505     int bs = FZAP_BLOCK_SHIFT(zap);
506     int err;

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

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

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

```



```

520     l = dmu_buf_get_user(db);

522     if (l == NULL)
523         l = zap_open_leaf(blkid, db);

525     rw_enter(&l->l_rwlock, lt);
526     /*
527      * Must lock before dirtying, otherwise l->l_phys could change,
528      * causing ASSERT below to fail.
529      */
530     if (lt == RW_WRITER)
531         dmu_buf_will_dirty(db, tx);
532     ASSERT3U(l->l_blkid, ==, blkid);
533     ASSERT3P(l->l_dbuf, ==, db);
534     ASSERT3P(l->l_phys, ==, l->l_dbuf->db_data);
535     ASSERT3U(l->l_phys->l_hdr.lh_block_type, ==, ZBT_LEAF);
536     ASSERT3U(l->l_phys->l_hdr.lh_magic, ==, ZAP_LEAF_MAGIC);

538     *lp = l;
539     return (0);
540 }

542 static int
543 zap_idx_to_blk(zap_t *zap, uint64_t idx, uint64_t *valp)
544 {
545     ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));

547     if (zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks == 0) {
548         ASSERT3U(idx, <,
549             (1ULL << zap->zap_f.zap_phys->zap_ptrtbl.zt_shift));
550         *valp = ZAP_EMBEDDED_PTRTBL_ENT(zap, idx);
551         return (0);
552     } else {
553         return (zap_table_load(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
554             idx, valp));
555     }
556 }

558 static int
559 zap_set_idx_to_blk(zap_t *zap, uint64_t idx, uint64_t blk, dmu_tx_t *tx)
560 {
561     ASSERT(tx != NULL);
562     ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));

564     if (zap->zap_f.zap_phys->zap_ptrtbl.zt_blk == 0) {
565         ZAP_EMBEDDED_PTRTBL_ENT(zap, idx) = blk;
566         return (0);
567     } else {
568         return (zap_table_store(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
569             idx, blk, tx));
570     }
571 }

573 static int
574 zap_deref_leaf(zap_t *zap, uint64_t h, dmu_tx_t *tx, krw_t lt, zap_leaf_t **lp)
575 {
576     uint64_t idx, blk;
577     int err;

579     ASSERT(zap->zap_dbuf == NULL ||
580         zap->zap_f.zap_phys == zap->zap_dbuf->db_data);
581     ASSERT3U(zap->zap_f.zap_phys->zap_magic, ==, ZAP_MAGIC);
582     idx = ZAP_HASH_IDX(h, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
583     err = zap_idx_to_blk(zap, idx, &blk);
584     if (err != 0)

```

```

585         return (err);
586     err = zap_get_leaf_byblk(zap, blk, tx, lt, lp);

588     ASSERT(err || ZAP_HASH_IDX(h, (*lp)->l_phys->l_hdr.lh_prefix_len) ==
589         (*lp)->l_phys->l_hdr.lh_prefix);
590     return (err);
591 }

593 static int
594 zap_expand_leaf(zap_name_t *zn, zap_leaf_t *l, dmu_tx_t *tx, zap_leaf_t **lp)
595 {
596     zap_t *zap = zn->zn_zap;
597     uint64_t hash = zn->zn_hash;
598     zap_leaf_t *nl;
599     int prefix_diff, i, err;
600     uint64_t sibling;
601     int old_prefix_len = l->l_phys->l_hdr.lh_prefix_len;

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

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

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

615         zap_put_leaf(l);
616         zap_unlockdir(zap);
617         err = zap_lockdir(os, object, tx, RW_WRITER,
618             FALSE, FALSE, &zn->zn_zap);
619         zap = zn->zn_zap;
620         if (err)
621             return (err);
622         ASSERT(!zap->zap_ismicro);

624         while (old_prefix_len ==
625             zap->zap_f.zap_phys->zap_ptrtbl.zt_shift) {
626             err = zap_grow_ptrtbl(zap, tx);
627             if (err)
628                 return (err);
629         }

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

635         if (l->l_phys->l_hdr.lh_prefix_len != old_prefix_len) {
636             /* it split while our locks were down */
637             *lp = l;
638             return (0);
639         }
640     }
641     ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
642     ASSERT3U(old_prefix_len, <, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
643     ASSERT3U(ZAP_HASH_IDX(hash, old_prefix_len), ==,
644         l->l_phys->l_hdr.lh_prefix);

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

650     /* check for i/o errors before doing zap_leaf_split */

```

```

651     for (i = 0; i < (1ULL<<prefix_diff); i++) {
652         uint64_t blk;
653         err = zap_idx_to_blk(zap, sibling+i, &blk);
654         if (err)
655             return (err);
656         ASSERT3U(blk, ==, l->l_blkid);
657     }

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

662     /* set sibling pointers */
663     for (i = 0; i < (1ULL<<prefix_diff); i++) {
664         err = zap_set_idx_to_blk(zap, sibling+i, nl->l_blkid, tx);
665         ASSERT0(err); /* we checked for i/o errors above */
666     }

668     if (hash & (1ULL<<(64 - l->l_phys->l_hdr.lh_prefix_len))) {
669         /* we want the sibling */
670         zap_put_leaf(l);
671         *lp = nl;
672     } else {
673         zap_put_leaf(nl);
674         *lp = l;
675     }

677     return (0);
678 }

680 static void
681 zap_put_leaf_maybe_grow_ptrtbl(zap_name_t *zn, zap_leaf_t *l, dmu_tx_t *tx)
682 {
683     zap_t *zap = zn->zn_zap;
684     int shift = zap->zap_f.zap_phys->zap_ptrtbl.zt_shift;
685     int leaffull = (l->l_phys->l_hdr.lh_prefix_len == shift &&
686         l->l_phys->l_hdr.lh_nfree < ZAP_LEAF_LOW_WATER);

688     zap_put_leaf(l);

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

693         /*
694          * We are in the middle of growing the pointer table, or
695          * this leaf will soon make us grow it.
696          */
697         if (zap_tryupgradedir(zap, tx) == 0) {
698             objset_t *os = zap->zap_objset;
699             uint64_t zapobj = zap->zap_object;

701             zap_unlockdir(zap);
702             err = zap_lockdir(os, zapobj, tx,
703                 RW_WRITER, FALSE, FALSE, &zn->zn_zap);
704             zap = zn->zn_zap;
705             if (err)
706                 return;
707         }

709         /* could have finished growing while our locks were down */
710         if (zap->zap_f.zap_phys->zap_ptrtbl.zt_shift == shift)
711             (void) zap_grow_ptrtbl(zap, tx);
712     }
713 }

715 static int
716 fzap_checkname(zap_name_t *zn)

```

```

717 {
718     if (zn->zn_key_orig_numints * zn->zn_key_intlen > ZAP_MAXNAMELEN)
719         return (SET_ERROR(ENAMETOOLONG));
720     return (0);
721 }

723 static int
724 fzap_checksize(uint64_t integer_size, uint64_t num_integers)
725 {
726     /* Only integer sizes supported by C */
727     switch (integer_size) {
728     case 1:
729     case 2:
730     case 4:
731     case 8:
732         break;
733     default:
734         return (SET_ERROR(EINVAL));
735     }

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

740     return (0);
741 }

743 static int
744 fzap_check(zap_name_t *zn, uint64_t integer_size, uint64_t num_integers)
745 {
746     int err;

748     if ((err = fzap_checkname(zn)) != 0)
749         return (err);
750     return (fzap_checksize(integer_size, num_integers));
751 }

753 /*
754  * Routines for manipulating attributes.
755  */
756 int
757 fzap_lookup(zap_name_t *zn,
758     uint64_t integer_size, uint64_t num_integers, void *buf,
759     char *realname, int rn_len, boolean_t *ncp)
760 {
761     zap_leaf_t *l;
762     int err;
763     zap_entry_handle_t zeh;

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

768     err = zap_deref_leaf(zn->zn_zap, zn->zn_hash, NULL, RW_READER, &l);
769     if (err != 0)
770         return (err);
771     err = zap_leaf_lookup(l, zn, &zeh);
772     if (err == 0) {
773         if ((err = fzap_checksize(integer_size, num_integers)) != 0) {
774             zap_put_leaf(l);
775             return (err);
776         }

778         err = zap_entry_read(&zeh, integer_size, num_integers, buf);
779         (void) zap_entry_read_name(zn->zn_zap, &zeh, rn_len, realname);
780         if (ncp) {
781             *ncp = zap_entry_normalization_conflict(&zeh,
782                 zn, NULL, zn->zn_zap);

```

```

783     }
784 }

786     zap_put_leaf(l);
787     return (err);
788 }

790 int
791 fzap_add_cd(zap_name_t *zn,
792     uint64_t integer_size, uint64_t num_integers,
793     const void *val, uint32_t cd, dmu_tx_t *tx)
794 {
795     zap_leaf_t *l;
796     int err;
797     zap_entry_handle_t zeh;
798     zap_t *zap = zn->zn_zap;

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

804     err = zap_deref_leaf(zap, zn->zn_hash, tx, RW_WRITER, &l);
805     if (err != 0)
806         return (err);
807 retry:
808     err = zap_leaf_lookup(l, zn, &zeh);
809     if (err == 0) {
810         err = SET_ERROR(EEXIST);
811         goto out;
812     }
813     if (err != ENOENT)
814         goto out;

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

819     if (err == 0) {
820         zap_increment_num_entries(zap, 1, tx);
821     } else if (err == EAGAIN) {
822         err = zap_expand_leaf(zn, l, tx, &l);
823         zap = zn->zn_zap; /* zap_expand_leaf() may change zap */
824         if (err == 0)
825             goto retry;
826     }

828 out:
829     if (zap != NULL)
830         zap_put_leaf_maybe_grow_ptrtbl(zn, l, tx);
831     return (err);
832 }

834 int
835 fzap_add(zap_name_t *zn,
836     uint64_t integer_size, uint64_t num_integers,
837     const void *val, dmu_tx_t *tx)
838 {
839     int err = fzap_check(zn, integer_size, num_integers);
840     if (err != 0)
841         return (err);

843     return (fzap_add_cd(zn, integer_size, num_integers,
844         val, ZAP_NEED_CD, tx));
845 }

847 int
848 fzap_update(zap_name_t *zn,

```

```

849     int integer_size, uint64_t num_integers, const void *val, dmu_tx_t *tx)
850 {
851     zap_leaf_t *l;
852     int err, create;
853     zap_entry_handle_t zeh;
854     zap_t *zap = zn->zn_zap;

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

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

869     if (create) {
870         err = zap_entry_create(l, zn, ZAP_NEED_CD,
871             integer_size, num_integers, val, &zeh);
872         if (err == 0)
873             zap_increment_num_entries(zap, 1, tx);
874     } else {
875         err = zap_entry_update(&zeh, integer_size, num_integers, val);
876     }

878     if (err == EAGAIN) {
879         err = zap_expand_leaf(zn, l, tx, &l);
880         zap = zn->zn_zap; /* zap_expand_leaf() may change zap */
881         if (err == 0)
882             goto retry;
883     }

885     if (zap != NULL)
886         zap_put_leaf_maybe_grow_ptrtbl(zn, l, tx);
887     return (err);
888 }

890 int
891 fzap_length(zap_name_t *zn,
892     uint64_t *integer_size, uint64_t *num_integers)
893 {
894     zap_leaf_t *l;
895     int err;
896     zap_entry_handle_t zeh;

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

905     if (integer_size)
906         *integer_size = zeh.zeh_integer_size;
907     if (num_integers)
908         *num_integers = zeh.zeh_num_integers;
909 out:
910     zap_put_leaf(l);
911     return (err);
912 }

914 int

```

```

915 fzap_remove(zap_name_t *zn, dmu_tx_t *tx)
916 {
917     zap_leaf_t *l;
918     int err;
919     zap_entry_handle_t zeh;
920
921     err = zap_deref_leaf(zn->zn_zap, zn->zn_hash, tx, RW_WRITER, &l);
922     if (err != 0)
923         return (err);
924     err = zap_leaf_lookup(l, zn, &zeh);
925     if (err == 0) {
926         zap_entry_remove(&zeh);
927         zap_increment_num_entries(zn->zn_zap, -1, tx);
928     }
929     zap_put_leaf(l);
930     return (err);
931 }
932
933 void
934 fzap_prefetch(zap_name_t *zn)
935 {
936     uint64_t idx, blk;
937     zap_t *zap = zn->zn_zap;
938     int bs;
939
940     idx = ZAP_HASH_IDX(zn->zn_hash,
941         zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
942     if (zap_idx_to_blk(zap, idx, &blk) != 0)
943         return;
944     bs = FZAP_BLOCK_SHIFT(zap);
945     dmu_prefetch(zap->zap_objset, zap->zap_object, blk << bs, 1 << bs);
946 }
947
948 /*
949  * Helper functions for consumers.
950  */
951
952 uint64_t
953 zap_create_link(objset_t *os, dmu_object_type_t ot, uint64_t parent_obj,
954     const char *name, dmu_tx_t *tx)
955 {
956     uint64_t new_obj;
957
958     VERIFY((new_obj = zap_create(os, ot, DMU_OT_NONE, 0, tx)) > 0);
959     VERIFY(zap_add(os, parent_obj, name, sizeof (uint64_t), 1, &new_obj,
960         tx) == 0);
961
962     return (new_obj);
963 }
964
965 int
966 zap_value_search(objset_t *os, uint64_t zapobj, uint64_t value, uint64_t mask,
967     char *name)
968 {
969     zap_cursor_t zc;
970     zap_attribute_t *za;
971     int err;
972
973     if (mask == 0)
974         mask = -1ULL;
975
976     za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
977     for (zap_cursor_init(&zc, os, zapobj);
978         (err = zap_cursor_retrieve(&zc, za)) == 0;
979         zap_cursor_advance(&zc)) {
980         if ((za->za_first_integer & mask) == (value & mask)) {

```

```

981         (void) strcpy(name, za->za_name);
982         break;
983     }
984 }
985 zap_cursor_fini(&zc);
986 kmem_free(za, sizeof (zap_attribute_t));
987 return (err);
988 }
989
990 int
991 zap_join(objset_t *os, uint64_t fromobj, uint64_t intoobj, dmu_tx_t *tx)
992 {
993     zap_cursor_t zc;
994     zap_attribute_t za;
995     int err;
996
997     err = 0;
998 #endif /* ! codereview */
999     for (zap_cursor_init(&zc, os, fromobj);
1000         zap_cursor_retrieve(&zc, &za) == 0;
1001         (void) zap_cursor_advance(&zc)) {
1002         if (za.za_integer_length != 8 || za.za_num_integers != 1) {
1003             err = SET_ERROR(EINVAL);
1004             break;
1005         }
1006         if (za.za_integer_length != 8 || za.za_num_integers != 1)
1007             return (SET_ERROR(EINVAL));
1008         err = zap_add(os, intoobj, za.za_name,
1009             8, 1, &za.za_first_integer, tx);
1010         if (err)
1011             break;
1012         return (err);
1013     }
1014     zap_cursor_fini(&zc);
1015     return (err);
1016 }
1017
1018 int
1019 zap_join_key(objset_t *os, uint64_t fromobj, uint64_t intoobj,
1020     uint64_t value, dmu_tx_t *tx)
1021 {
1022     zap_cursor_t zc;
1023     zap_attribute_t za;
1024     int err;
1025
1026     err = 0;
1027 #endif /* ! codereview */
1028     for (zap_cursor_init(&zc, os, fromobj);
1029         zap_cursor_retrieve(&zc, &za) == 0;
1030         (void) zap_cursor_advance(&zc)) {
1031         if (za.za_integer_length != 8 || za.za_num_integers != 1) {
1032             err = SET_ERROR(EINVAL);
1033             break;
1034         }
1035         if (za.za_integer_length != 8 || za.za_num_integers != 1)
1036             return (SET_ERROR(EINVAL));
1037         err = zap_add(os, intoobj, za.za_name,
1038             8, 1, &value, tx);
1039         if (err)
1040             break;
1041         return (err);
1042     }
1043     zap_cursor_fini(&zc);
1044     return (err);
1045 }

```

```

1039 }
1041 int
1042 zap_join_increment(objset_t *os, uint64_t fromobj, uint64_t intoobj,
1043     dmu_tx_t *tx)
1044 {
1045     zap_cursor_t zc;
1046     zap_attribute_t za;
1047     int err;

1049     err = 0;
1050 #endif /* ! codereview */
1051     for (zap_cursor_init(&zc, os, fromobj);
1052         zap_cursor_retrieve(&zc, &za) == 0;
1053         (void) zap_cursor_advance(&zc)) {
1054         uint64_t delta = 0;

1056         if (za.za_integer_length != 8 || za.za_num_integers != 1) {
1057             err = SET_ERROR(EINVAL);
1058             break;
1059         }
1060         if (za.za_integer_length != 8 || za.za_num_integers != 1)
1061             return (SET_ERROR(EINVAL));

1062         err = zap_lookup(os, intoobj, za.za_name, 8, 1, &delta);
1063         if (err != 0 && err != ENOENT)
1064             break;
1065         return (err);
1066         delta += za.za_first_integer;
1067         err = zap_update(os, intoobj, za.za_name, 8, 1, &delta, tx);
1068         if (err)
1069             break;
1070         return (err);
1071     }
1072     zap_cursor_fini(&zc);
1073     return (err);
1074 }
1075
1076 unchanged_portion_omitted

```