

```

*****
118555 Fri Jun 29 08:54:37 2012
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
2948 do not generate 0 as random 64-bit guid
*****
_____unchanged_portion_omitted_____

779 uint64_t
780 dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
781     uint64_t flags, dmu_tx_t *tx)
782 {
783     dsl_pool_t *dp = dd->dd_pool;
784     dmu_buf_t *dbuf;
785     dsl_dataset_phys_t *dsphys;
786     uint64_t dsobj;
787     objset_t *mos = dp->dp_meta_objset;

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

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

797     dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
798         DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
799     VERIFY(0 == dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
800     dmu_buf_will_dirty(dbuf, tx);
801     dsphys = dbuf->db_data;
802     bzero(dsphys, sizeof (dsl_dataset_phys_t));
803     dsphys->ds_dir_obj = dd->dd_object;
804     dsphys->ds_flags = flags;
805     dsphys->ds_fsid_guid = unique_create();
806     do {
807 #endif /* !codereview */
808         (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
809             sizeof (dsphys->ds_guid));
810     } while (dsphys->ds_guid == 0);
811 #endif /* !codereview */
812     dsphys->ds_snapnames_zapobj =
813         zap_create_norm(mos, U8_TEXTPREP_Toupper, DMU_OT_DSL_DS_SNAP_MAP,
814             DMU_OT_NONE, 0, tx);
815     dsphys->ds_creation_time = gethrstime_sec();
816     dsphys->ds_creation_txg = tx->tx_txg == TXG_INITIAL ? 1 : tx->tx_txg;

818     if (origin == NULL) {
819         dsphys->ds_deadlist_obj = dsl_deadlist_alloc(mos, tx);
820     } else {
821         dsl_dataset_t *ohds;

823         dsphys->ds_prev_snap_obj = origin->ds_object;
824         dsphys->ds_prev_snap_txg =
825             origin->ds_phys->ds_creation_txg;
826         dsphys->ds_referenced_bytes =
827             origin->ds_phys->ds_referenced_bytes;
828         dsphys->ds_compressed_bytes =
829             origin->ds_phys->ds_compressed_bytes;
830         dsphys->ds_uncompressed_bytes =
831             origin->ds_phys->ds_uncompressed_bytes;
832         dsphys->ds_bp = origin->ds_phys->ds_bp;
833         dsphys->ds_flags |= origin->ds_phys->ds_flags;

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

```

```

838     VERIFY3U(0, ==, dsl_dataset_hold_obj(dp,
839         origin->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &ohds));
840     dsphys->ds_deadlist_obj = dsl_deadlist_clone(&ohds->ds_deadlist,
841         dsphys->ds_prev_snap_txg, dsphys->ds_prev_snap_obj, tx);
842     dsl_dataset_rele(ohds, FTAG);

844     if (spa_version(dp->dp_spa) >= SPA_VERSION_NEXT_CLONES) {
845         if (origin->ds_phys->ds_next_clones_obj == 0) {
846             origin->ds_phys->ds_next_clones_obj =
847                 zap_create(mos,
848                     DMU_OT_NEXT_CLONES, DMU_OT_NONE, 0, tx);
849         }
850         VERIFY(0 == zap_add_int(mos,
851             origin->ds_phys->ds_next_clones_obj,
852             dsobj, tx));
853     }

855     dmu_buf_will_dirty(dd->dd_dbuf, tx);
856     dd->dd_phys->dd_origin_obj = origin->ds_object;
857     if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
858         if (origin->ds_dir->dd_phys->dd_clones == 0) {
859             dmu_buf_will_dirty(origin->ds_dir->dd_dbuf, tx);
860             origin->ds_dir->dd_phys->dd_clones =
861                 zap_create(mos,
862                     DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
863         }
864         VERIFY3U(0, ==, zap_add_int(mos,
865             origin->ds_dir->dd_phys->dd_clones, dsobj, tx));
866     }
867 }

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

872     dmu_buf_rele(dbuf, FTAG);

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

877     return (dsobj);
878 }

880 uint64_t
881 dsl_dataset_create_sync(dsl_dir_t *pdd, const char *lastname,
882     dsl_dataset_t *origin, uint64_t flags, cred_t *cr, dmu_tx_t *tx)
883 {
884     dsl_pool_t *dp = pdd->dd_pool;
885     uint64_t dsobj, ddoobj;
886     dsl_dir_t *dd;

888     ASSERT(lastname[0] != '@');

890     ddoobj = dsl_dir_create_sync(dp, pdd, lastname, tx);
891     VERIFY(0 == dsl_dir_open_obj(dp, ddoobj, lastname, FTAG, &dd));

893     dsobj = dsl_dataset_create_sync_dd(dd, origin, flags, tx);

895     dsl_deleg_set_create_perms(dd, tx, cr);

897     dsl_dir_close(dd, FTAG);

899     /*
900     * If we are creating a clone, make sure we zero out any stale
901     * data from the origin snapshots zil header.
902     */
903     if (origin != NULL) {

```

```

904     dsl_dataset_t *ds;
905     objset_t *os;

907     VERIFY3U(0, ==, dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
908     VERIFY3U(0, ==, dmu_objset_from_ds(ds, &os));
909     bzero(&os->os_zil_header, sizeof(os->os_zil_header));
910     dsl_dataset_dirty(ds, tx);
911     dsl_dataset_rele(ds, FTAG);
912 }

914     return(dsobj);
915 }

917 /*
918  * The snapshots must all be in the same pool.
919  */
920 int
921 dmu_snapshots_destroy_nvlist(nvlist_t *snaps, boolean_t defer, char *failed)
922 {
923     int err;
924     dsl_sync_task_t *dst;
925     spa_t *spa;
926     nvpair_t *pair;
927     dsl_sync_task_group_t *dstg;

929     pair = nvlist_next_nvpair(snaps, NULL);
930     if (pair == NULL)
931         return(0);

933     err = spa_open(nvpair_name(pair), &spa, FTAG);
934     if (err)
935         return(err);
936     dstg = dsl_sync_task_group_create(spa_get_dsl(spa));

938     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
939          pair = nvlist_next_nvpair(snaps, pair)) {
940         dsl_dataset_t *ds;

942         err = dsl_dataset_own(nvpair_name(pair), B_TRUE, dstg, &ds);
943         if (err == 0) {
944             struct dsl_ds_destroyarg *dsda;

946             dsl_dataset_make_exclusive(ds, dstg);
947             dsda = kmem_zalloc(sizeof(struct dsl_ds_destroyarg),
948                               KM_SLEEP);
949             dsda->ds = ds;
950             dsda->defer = defer;
951             dsl_sync_task_create(dstg, dsl_dataset_destroy_check,
952                                 dsl_dataset_destroy_sync, dsda, dstg, 0);
953         } else if (err == ENOENT) {
954             err = 0;
955         } else {
956             (void) strcpy(failed, nvpair_name(pair));
957             break;
958         }
959     }

961     if (err == 0)
962         err = dsl_sync_task_group_wait(dstg);

964     for (dst = list_head(&dstg->dstg_tasks); dst;
965          dst = list_next(&dstg->dstg_tasks, dst)) {
966         struct dsl_ds_destroyarg *dsda = dst->dst_arg1;
967         dsl_dataset_t *ds = dsda->ds;

969     /*

```

```

970         * Return the file system name that triggered the error
971         */
972         if (dst->dst_err) {
973             dsl_dataset_name(ds, failed);
974         }
975         ASSERT3P(dsda->rm_origin, ==, NULL);
976         dsl_dataset_disown(ds, dstg);
977         kmem_free(dsda, sizeof(struct dsl_ds_destroyarg));
978     }

980     dsl_sync_task_group_destroy(dstg);
981     spa_close(spa, FTAG);
982     return(err);

984 }

986 static boolean_t
987 dsl_dataset_might_destroy_origin(dsl_dataset_t *ds)
988 {
989     boolean_t might_destroy = B_FALSE;

991     mutex_enter(&ds->ds_lock);
992     if (ds->ds_phys->ds_num_children == 2 && ds->ds_userrefs == 0 &&
993         DS_IS_DEFER_DESTROY(ds))
994         might_destroy = B_TRUE;
995     mutex_exit(&ds->ds_lock);

997     return(might_destroy);
998 }

1000 /*
1001  * If we're removing a clone, and these three conditions are true:
1002  *   1) the clone's origin has no other children
1003  *   2) the clone's origin has no user references
1004  *   3) the clone's origin has been marked for deferred destruction
1005  * Then, prepare to remove the origin as part of this sync task group.
1006  */
1007 static int
1008 dsl_dataset_origin_rm_prep(struct dsl_ds_destroyarg *dsda, void *tag)
1009 {
1010     dsl_dataset_t *ds = dsda->ds;
1011     dsl_dataset_t *origin = ds->ds_prev;

1013     if (dsl_dataset_might_destroy_origin(origin)) {
1014         char *name;
1015         int namelen;
1016         int error;

1018         namelen = dsl_dataset_namelen(origin) + 1;
1019         name = kmem_alloc(namelen, KM_SLEEP);
1020         dsl_dataset_name(origin, name);
1021 #ifdef _KERNEL
1022         error = zfs_unmount_snap(name, NULL);
1023         if (error) {
1024             kmem_free(name, namelen);
1025             return(error);
1026         }
1027 #endif
1028         error = dsl_dataset_own(name, B_TRUE, tag, &origin);
1029         kmem_free(name, namelen);
1030         if (error)
1031             return(error);
1032         dsda->rm_origin = origin;
1033         dsl_dataset_make_exclusive(origin, tag);
1034     }

```

```

1036     return (0);
1037 }

1039 /*
1040 * ds must be opened as OWNER. On return (whether successful or not),
1041 * ds will be closed and caller can no longer dereference it.
1042 */
1043 int
1044 dsl_dataset_destroy(dsl_dataset_t *ds, void *tag, boolean_t defer)
1045 {
1046     int err;
1047     dsl_sync_task_group_t *dstg;
1048     objset_t *os;
1049     dsl_dir_t *dd;
1050     uint64_t obj;
1051     struct dsl_ds_destroyarg dsda = { 0 };
1052     dsl_dataset_t dummy_ds = { 0 };

1054     dsda.ds = ds;

1056     if (dsl_dataset_is_snapshot(ds)) {
1057         /* Destroying a snapshot is simpler */
1058         dsl_dataset_make_exclusive(ds, tag);

1060         dsda.defer = defer;
1061         err = dsl_sync_task_do(ds->ds_dir->dd_pool,
1062             dsl_dataset_destroy_check, dsl_dataset_destroy_sync,
1063             &dsda, tag, 0);
1064         ASSERT3P(dsda.rm_origin, ==, NULL);
1065         goto out;
1066     } else if (defer) {
1067         err = EINVAL;
1068         goto out;
1069     }

1071     dd = ds->ds_dir;
1072     dummy_ds.ds_dir = dd;
1073     dummy_ds.ds_object = ds->ds_object;

1075     /*
1076     * Check for errors and mark this ds as inconsistent, in
1077     * case we crash while freeing the objects.
1078     */
1079     err = dsl_sync_task_do(dd->dd_pool, dsl_dataset_destroy_begin_check,
1080         dsl_dataset_destroy_begin_sync, ds, NULL, 0);
1081     if (err)
1082         goto out;

1084     err = dmu_objset_from_ds(ds, &os);
1085     if (err)
1086         goto out;

1088     /*
1089     * If async destruction is not enabled try to remove all objects
1090     * while in the open context so that there is less work to do in
1091     * the syncing context.
1092     */
1093     if (!spa_feature_is_enabled(dsl_dataset_get_spa(ds),
1094         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
1095         for (obj = 0; err == 0; err = dmu_object_next(os, &obj, FALSE,
1096             ds->ds_phys->ds_prev_snap_txg)) {
1097             /*
1098             * Ignore errors, if there is not enough disk space
1099             * we will deal with it in dsl_dataset_destroy_sync().
1100             */
1101             (void) dmu_free_object(os, obj);

```

```

1102     }
1103     if (err != ESRCH)
1104         goto out;
1105 }

1107     /*
1108     * Only the ZIL knows how to free log blocks.
1109     */
1110     zil_destroy(dmu_objset_zil(os), B_FALSE);

1112     /*
1113     * Sync out all in-flight IO.
1114     */
1115     txg_wait_synced(dd->dd_pool, 0);

1117     /*
1118     * If we managed to free all the objects in open
1119     * context, the user space accounting should be zero.
1120     */
1121     if (ds->ds_phys->ds_bp.blk_fill == 0 &&
1122         dmu_objset_userused_enabled(os)) {
1123         uint64_t count;

1125         ASSERT(zap_count(os, DMU_USERUSED_OBJECT, &count) != 0 ||
1126             count == 0);
1127         ASSERT(zap_count(os, DMU_GROUPUSED_OBJECT, &count) != 0 ||
1128             count == 0);
1129     }

1131     rw_enter(&dd->dd_pool->dp_config_rwlock, RW_READER);
1132     err = dsl_dir_open_obj(dd->dd_pool, dd->dd_object, NULL, FTAG, &dd);
1133     rw_exit(&dd->dd_pool->dp_config_rwlock);

1135     if (err)
1136         goto out;

1138     /*
1139     * Blow away the dsl_dir + head dataset.
1140     */
1141     dsl_dataset_make_exclusive(ds, tag);
1142     /*
1143     * If we're removing a clone, we might also need to remove its
1144     * origin.
1145     */
1146     do {
1147         dsda.need_prep = B_FALSE;
1148         if (dsl_dir_is_clone(dd)) {
1149             err = dsl_dataset_origin_rm_prep(&dsda, tag);
1150             if (err) {
1151                 dsl_dir_close(dd, FTAG);
1152                 goto out;
1153             }
1154         }

1156         dstg = dsl_sync_task_group_create(ds->ds_dir->dd_pool);
1157         dsl_sync_task_create(dstg, dsl_dataset_destroy_check,
1158             dsl_dataset_destroy_sync, &dsda, tag, 0);
1159         dsl_sync_task_create(dstg, dsl_dir_destroy_check,
1160             dsl_dir_destroy_sync, &dummy_ds, FTAG, 0);
1161         err = dsl_sync_task_group_wait(dstg);
1162         dsl_sync_task_group_destroy(dstg);

1164     /*
1165     * We could be racing against 'zfs release' or 'zfs destroy -d'
1166     * on the origin snap, in which case we can get EBUSY if we
1167     * needed to destroy the origin snap but were not ready to

```

```

1168         * do so.
1169         */
1170         if (dsda.need_prep) {
1171             ASSERT(err == EBUSY);
1172             ASSERT(dsl_dir_is_clone(dd));
1173             ASSERT(dsda.rm_origin == NULL);
1174         }
1175     } while (dsda.need_prep);
1177     if (dsda.rm_origin != NULL)
1178         dsl_dataset_disown(dsda.rm_origin, tag);
1180     /* if it is successful, dsl_dir_destroy_sync will close the dd */
1181     if (err)
1182         dsl_dir_close(dd, FTAG);
1183 out:
1184     dsl_dataset_disown(ds, tag);
1185     return (err);
1186 }
1188 blkptr_t *
1189 dsl_dataset_get_blkptr(dsl_dataset_t *ds)
1190 {
1191     return (&ds->ds_phys->ds_bp);
1192 }
1194 void
1195 dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx)
1196 {
1197     ASSERT(dmu_tx_is_syncing(tx));
1198     /* If it's the meta-objset, set dp_meta_rootbp */
1199     if (ds == NULL) {
1200         tx->tx_pool->dp_meta_rootbp = *bp;
1201     } else {
1202         dmu_buf_will_dirty(ds->ds_dbuf, tx);
1203         ds->ds_phys->ds_bp = *bp;
1204     }
1205 }
1207 spa_t *
1208 dsl_dataset_get_spa(dsl_dataset_t *ds)
1209 {
1210     return (ds->ds_dir->dd_pool->dp_spa);
1211 }
1213 void
1214 dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx)
1215 {
1216     dsl_pool_t *dp;
1218     if (ds == NULL) /* this is the meta-objset */
1219         return;
1221     ASSERT(ds->ds_objset != NULL);
1223     if (ds->ds_phys->ds_next_snap_obj != 0)
1224         panic("dirtying snapshot!");
1226     dp = ds->ds_dir->dd_pool;
1228     if (txg_list_add(&dp->dp_dirty_datasets, ds, tx->tx_txg) == 0) {
1229         /* up the hold count until we can be written out */
1230         dmu_buf_add_ref(ds->ds_dbuf, ds);
1231     }
1232 }

```

```

1234 /*
1235  * The unique space in the head dataset can be calculated by subtracting
1236  * the space used in the most recent snapshot, that is still being used
1237  * in this file system, from the space currently in use. To figure out
1238  * the space in the most recent snapshot still in use, we need to take
1239  * the total space used in the snapshot and subtract out the space that
1240  * has been freed up since the snapshot was taken.
1241  */
1242 static void
1243 dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds)
1244 {
1245     uint64_t mrs_used;
1246     uint64_t dlused, dlcomp, dluncomp;
1248     ASSERT(!dsl_dataset_is_snapshot(ds));
1250     if (ds->ds_phys->ds_prev_snap_obj != 0)
1251         mrs_used = ds->ds_prev->ds_phys->ds_referenced_bytes;
1252     else
1253         mrs_used = 0;
1255     dsl_deadlist_space(&ds->ds_deadlist, &dlused, &dlcomp, &dluncomp);
1257     ASSERT3U(dlused, <=, mrs_used);
1258     ds->ds_phys->ds_unique_bytes =
1259         ds->ds_phys->ds_referenced_bytes - (mrs_used - dlused);
1261     if (spa_version(ds->ds_dir->dd_pool->dp_spa) >=
1262         SPA_VERSION_UNIQUE_ACCURATE)
1263         ds->ds_phys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;
1264 }
1266 struct killarg {
1267     dsl_dataset_t *ds;
1268     dmu_tx_t *tx;
1269 };
1271 /* ARGSUSED */
1272 static int
1273 kill_blkptr(spa_t *spa, zillog_t *zillog, const blkptr_t *bp, arc_buf_t *pbuf,
1274     const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
1275 {
1276     struct killarg *ka = arg;
1277     dmu_tx_t *tx = ka->tx;
1279     if (bp == NULL)
1280         return (0);
1282     if (zb->zb_level == ZB_ZIL_LEVEL) {
1283         ASSERT(zillog != NULL);
1284         /*
1285          * It's a block in the intent log. It has no
1286          * accounting, so just free it.
1287          */
1288         dsl_free(ka->tx->tx_pool, ka->tx->tx_txg, bp);
1289     } else {
1290         ASSERT(zillog == NULL);
1291         ASSERT3U(bp->blk_birth, >, ka->ds->ds_phys->ds_prev_snap_txg);
1292         (void) dsl_dataset_block_kill(ka->ds, bp, tx, B_FALSE);
1293     }
1295     return (0);
1296 }
1298 /* ARGSUSED */
1299 static int

```

```

1300 dsl_dataset_destroy_begin_check(void *arg1, void *arg2, dmu_tx_t *tx)
1301 {
1302     dsl_dataset_t *ds = arg1;
1303     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
1304     uint64_t count;
1305     int err;
1307     /*
1308      * Can't delete a head dataset if there are snapshots of it.
1309      * (Except if the only snapshots are from the branch we cloned
1310      * from.)
1311      */
1312     if (ds->ds_prev != NULL &&
1313         ds->ds_prev->ds_phys->ds_next_snap_obj == ds->ds_object)
1314         return (EBUSY);
1316     /*
1317      * This is really a dsl_dir thing, but check it here so that
1318      * we'll be less likely to leave this dataset inconsistent &
1319      * nearly destroyed.
1320      */
1321     err = zap_count(mos, ds->ds_dir->dd_phys->dd_child_dir_zapobj, &count);
1322     if (err)
1323         return (err);
1324     if (count != 0)
1325         return (EEXIST);
1327     return (0);
1328 }
1330 /* ARGSUSED */
1331 static void
1332 dsl_dataset_destroy_begin_sync(void *arg1, void *arg2, dmu_tx_t *tx)
1333 {
1334     dsl_dataset_t *ds = arg1;
1335     dsl_pool_t *dp = ds->ds_dir->dd_pool;
1337     /* Mark it as inconsistent on-disk, in case we crash */
1338     dmu_buf_will_dirty(ds->ds_dbuf, tx);
1339     ds->ds_phys->ds_flags |= DS_FLAG_INCONSISTENT;
1341     spa_history_log_internal(LOG_DS_DESTROY_BEGIN, dp->dp_spa, tx,
1342         "dataset = %llu", ds->ds_object);
1343 }
1345 static int
1346 dsl_dataset_origin_check(struct dsl_ds_destroyarg *dsda, void *tag,
1347     dmu_tx_t *tx)
1348 {
1349     dsl_dataset_t *ds = dsda->ds;
1350     dsl_dataset_t *ds_prev = ds->ds_prev;
1352     if (dsl_dataset_might_destroy_origin(ds_prev)) {
1353         struct dsl_ds_destroyarg ndsda = {0};
1355         /*
1356          * If we're not prepared to remove the origin, don't remove
1357          * the clone either.
1358          */
1359         if (dsda->rm_origin == NULL) {
1360             dsda->need_prep = B_TRUE;
1361             return (EBUSY);
1362         }
1364         ndsda.ds = ds_prev;
1365         ndsda.is_origin_rm = B_TRUE;

```

```

1366         return (dsl_dataset_destroy_check(&ndsda, tag, tx));
1367     }
1369     /*
1370      * If we're not going to remove the origin after all,
1371      * undo the open context setup.
1372      */
1373     if (dsda->rm_origin != NULL) {
1374         dsl_dataset_disown(dsda->rm_origin, tag);
1375         dsda->rm_origin = NULL;
1376     }
1378     return (0);
1379 }
1381 /*
1382  * If you add new checks here, you may need to add
1383  * additional checks to the "temporary" case in
1384  * snapshot_check() in dmu_objset.c.
1385  */
1386 /* ARGSUSED */
1387 int
1388 dsl_dataset_destroy_check(void *arg1, void *arg2, dmu_tx_t *tx)
1389 {
1390     struct dsl_ds_destroyarg *dsda = arg1;
1391     dsl_dataset_t *ds = dsda->ds;
1393     /* we have an owner hold, so noone else can destroy us */
1394     ASSERT(!DSL_DATASET_IS_DESTROYED(ds));
1396     /*
1397      * Only allow deferred destroy on pools that support it.
1398      * NOTE: deferred destroy is only supported on snapshots.
1399      */
1400     if (dsda->defer) {
1401         if (spa_version(ds->ds_dir->dd_pool->dp_spa) <
1402             SPA_VERSION_USERREFS)
1403             return (ENOTSUP);
1404         ASSERT(dsl_dataset_is_snapshot(ds));
1405         return (0);
1406     }
1408     /*
1409      * Can't delete a head dataset if there are snapshots of it.
1410      * (Except if the only snapshots are from the branch we cloned
1411      * from.)
1412      */
1413     if (ds->ds_prev != NULL &&
1414         ds->ds_prev->ds_phys->ds_next_snap_obj == ds->ds_object)
1415         return (EBUSY);
1417     /*
1418      * If we made changes this txg, traverse_dsl_dataset won't find
1419      * them. Try again.
1420      */
1421     if (ds->ds_phys->ds_bp.blk_birth >= tx->tx_txg)
1422         return (EAGAIN);
1424     if (dsl_dataset_is_snapshot(ds)) {
1425         /*
1426          * If this snapshot has an elevated user reference count,
1427          * we can't destroy it yet.
1428          */
1429         if (ds->ds_userrefs > 0 && !dsda->releasing)
1430             return (EBUSY);

```

```

1432     mutex_enter(&ds->ds_lock);
1433     /*
1434      * Can't delete a branch point. However, if we're destroying
1435      * a clone and removing its origin due to it having a user
1436      * hold count of 0 and having been marked for deferred destroy,
1437      * it's OK for the origin to have a single clone.
1438      */
1439     if (ds->ds_phys->ds_num_children >
1440         (dsda->is_origin_rm ? 2 : 1)) {
1441         mutex_exit(&ds->ds_lock);
1442         return (EEXIST);
1443     }
1444     mutex_exit(&ds->ds_lock);
1445 } else if (dsl_dir_is_clone(ds->ds_dir)) {
1446     return (dsl_dataset_origin_check(dsda, arg2, tx));
1447 }
1448
1449 /* XXX we should do some i/o error checking... */
1450 return (0);
1451 }
1452
1453 struct refsarg {
1454     kmutex_t lock;
1455     boolean_t gone;
1456     kcondvar_t cv;
1457 };
1458
1459 /* ARGSUSED */
1460 static void
1461 dsl_dataset_refs_gone(dmu_buf_t *db, void *argv)
1462 {
1463     struct refsarg *arg = argv;
1464
1465     mutex_enter(&arg->lock);
1466     arg->gone = TRUE;
1467     cv_signal(&arg->cv);
1468     mutex_exit(&arg->lock);
1469 }
1470
1471 static void
1472 dsl_dataset_drain_refs(dsl_dataset_t *ds, void *tag)
1473 {
1474     struct refsarg arg;
1475
1476     mutex_init(&arg.lock, NULL, MUTEX_DEFAULT, NULL);
1477     cv_init(&arg.cv, NULL, CV_DEFAULT, NULL);
1478     arg.gone = FALSE;
1479     (void) dmu_buf_update_user(ds->ds_dbuf, ds, &arg, &ds->ds_phys,
1480         dsl_dataset_refs_gone);
1481     dmu_buf_rele(ds->ds_dbuf, tag);
1482     mutex_enter(&arg.lock);
1483     while (!arg.gone)
1484         cv_wait(&arg.cv, &arg.lock);
1485     ASSERT(arg.gone);
1486     mutex_exit(&arg.lock);
1487     ds->ds_dbuf = NULL;
1488     ds->ds_phys = NULL;
1489     mutex_destroy(&arg.lock);
1490     cv_destroy(&arg.cv);
1491 }
1492
1493 static void
1494 remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj, dmu_tx_t *tx)
1495 {
1496     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
1497     uint64_t count;

```

```

1498     int err;
1499
1500     ASSERT(ds->ds_phys->ds_num_children >= 2);
1501     err = zap_remove_int(mos, ds->ds_phys->ds_next_clones_obj, obj, tx);
1502     /*
1503      * The err should not be ENOENT, but a bug in a previous version
1504      * of the code could cause upgrade_clones_cb() to not set
1505      * ds_next_snap_obj when it should, leading to a missing entry.
1506      * If we knew that the pool was created after
1507      * SPA_VERSION_NEXT_CLONES, we could assert that it isn't
1508      * ENOENT. However, at least we can check that we don't have
1509      * too many entries in the next_clones_obj even after failing to
1510      * remove this one.
1511      */
1512     if (err != ENOENT) {
1513         VERIFY3U(err, ==, 0);
1514     }
1515     ASSERT3U(0, ==, zap_count(mos, ds->ds_phys->ds_next_clones_obj,
1516         &count));
1517     ASSERT3U(count, <=, ds->ds_phys->ds_num_children - 2);
1518 }
1519
1520 static void
1521 dsl_dataset_remove_clones_key(dsl_dataset_t *ds, uint64_t mintxg, dmu_tx_t *tx)
1522 {
1523     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
1524     zap_cursor_t zc;
1525     zap_attribute_t za;
1526
1527     /*
1528      * If it is the old version, dd_clones doesn't exist so we can't
1529      * find the clones, but deadlist_remove_key() is a no-op so it
1530      * doesn't matter.
1531      */
1532     if (ds->ds_dir->dd_phys->dd_clones == 0)
1533         return;
1534
1535     for (zap_cursor_init(&zc, mos, ds->ds_dir->dd_phys->dd_clones);
1536         zap_cursor_retrieve(&zc, &za) == 0;
1537         zap_cursor_advance(&zc)) {
1538         dsl_dataset_t *clone;
1539
1540         VERIFY3U(0, ==, dsl_dataset_hold_obj(ds->ds_dir->dd_pool,
1541             za.za_first_integer, FTAG, &clone));
1542         if (clone->ds_dir->dd_origin_txg > mintxg) {
1543             dsl_deadlist_remove_key(&clone->ds_deadlist,
1544                 mintxg, tx);
1545             dsl_dataset_remove_clones_key(clone, mintxg, tx);
1546         }
1547         dsl_dataset_rele(clone, FTAG);
1548     }
1549     zap_cursor_fini(&zc);
1550 }
1551
1552 struct process_old_arg {
1553     dsl_dataset_t *ds;
1554     dsl_dataset_t *ds_prev;
1555     boolean_t after_branch_point;
1556     zio_t *pio;
1557     uint64_t used, comp, uncomp;
1558 };
1559
1560 static int
1561 process_old_cb(void *arg, const blkptr_t *bp, dmu_tx_t *tx)
1562 {
1563     struct process_old_arg *poa = arg;

```

```

1564     dsl_pool_t *dp = poa->ds->ds_dir->dd_pool;

1566     if (bp->blk_birth <= poa->ds->ds_phys->ds_prev_snap_txg) {
1567         dsl_deadlist_insert(&poa->ds->ds_deadlist, bp, tx);
1568         if (poa->ds_prev && !poa->after_branch_point &&
1569             bp->blk_birth >
1570             poa->ds_prev->ds_phys->ds_prev_snap_txg) {
1571             poa->ds_prev->ds_phys->ds_unique_bytes +=
1572                 bp_get_dsize_sync(dp->dp_spa, bp);
1573         }
1574     } else {
1575         poa->used += bp_get_dsize_sync(dp->dp_spa, bp);
1576         poa->comp += BP_GET_PSIZE(bp);
1577         poa->uncomp += BP_GET_UCSIZE(bp);
1578         dsl_free_sync(poa->pio, dp, tx->tx_txg, bp);
1579     }
1580     return (0);
1581 }

1583 static void
1584 process_old_deadlist(dsl_dataset_t *ds, dsl_dataset_t *ds_prev,
1585     dsl_dataset_t *ds_next, boolean_t after_branch_point, dmu_tx_t *tx)
1586 {
1587     struct process_old_arg poa = { 0 };
1588     dsl_pool_t *dp = ds->ds_dir->dd_pool;
1589     objset_t *mos = dp->dp_meta_objset;

1591     ASSERT(ds->ds_deadlist.dl_oldfmt);
1592     ASSERT(ds_next->ds_deadlist.dl_oldfmt);

1594     poa.ds = ds;
1595     poa.ds_prev = ds_prev;
1596     poa.after_branch_point = after_branch_point;
1597     poa.pio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
1598     VERIFY3U(0, ==, bplib_iterate(&ds_next->ds_deadlist.dl_bplib,
1599         process_old_cb, &poa, tx));
1600     VERIFY3U(zio_wait(poa.pio), ==, 0);
1601     ASSERT3U(poa.used, ==, ds->ds_phys->ds_unique_bytes);

1603     /* change snapused */
1604     dsl_dir_diduse_space(ds->ds_dir, DD_USED_SNAP,
1605         -poa.used, -poa.comp, -poa.uncomp, tx);

1607     /* swap next's deadlist to our deadlist */
1608     dsl_deadlist_close(&ds->ds_deadlist);
1609     dsl_deadlist_close(&ds_next->ds_deadlist);
1610     SWITCH64(ds_next->ds_phys->ds_deadlist_obj,
1611         ds->ds_phys->ds_deadlist_obj);
1612     dsl_deadlist_open(&ds->ds_deadlist, mos, ds->ds_phys->ds_deadlist_obj);
1613     dsl_deadlist_open(&ds_next->ds_deadlist, mos,
1614         ds_next->ds_phys->ds_deadlist_obj);
1615 }

1617 static int
1618 old_synchronous_dataset_destroy(dsl_dataset_t *ds, dmu_tx_t *tx)
1619 {
1620     int err;
1621     struct killarg ka;

1623     /*
1624      * Free everything that we point to (that's born after
1625      * the previous snapshot, if we are a clone)
1626      *
1627      * NB: this should be very quick, because we already
1628      * freed all the objects in open context.
1629      */

```

```

1630     ka.ds = ds;
1631     ka.tx = tx;
1632     err = traverse_dataset(ds,
1633         ds->ds_phys->ds_prev_snap_txg, TRAVERSE_POST,
1634         kill_blkptr, &ka);
1635     ASSERT3U(err, ==, 0);
1636     ASSERT(!DS_UNIQUE_IS_ACCURATE(ds) || ds->ds_phys->ds_unique_bytes == 0);

1638     return (err);
1639 }

1641 void
1642 dsl_dataset_destroy_sync(void *arg1, void *tag, dmu_tx_t *tx)
1643 {
1644     struct dsl_ds_destroyarg *dsda = arg1;
1645     dsl_dataset_t *ds = dsda->ds;
1646     int err;
1647     int after_branch_point = FALSE;
1648     dsl_pool_t *dp = ds->ds_dir->dd_pool;
1649     objset_t *mos = dp->dp_meta_objset;
1650     dsl_dataset_t *ds_prev = NULL;
1651     boolean_t wont_destroy;
1652     uint64_t obj;

1654     wont_destroy = (dsda->defer &&
1655         (ds->ds_userrefs > 0 || ds->ds_phys->ds_num_children > 1));

1657     ASSERT(ds->ds_owner || wont_destroy);
1658     ASSERT(dsda->defer || ds->ds_phys->ds_num_children <= 1);
1659     ASSERT(ds->ds_prev == NULL ||
1660         ds->ds_prev->ds_phys->ds_next_snap_obj != ds->ds_object);
1661     ASSERT3U(ds->ds_phys->ds_bp.blk_birth, <=, tx->tx_txg);

1663     if (wont_destroy) {
1664         ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);
1665         dmu_buf_will_dirty(ds->ds_dbuf, tx);
1666         ds->ds_phys->ds_flags |= DS_FLAG_DEFER_DESTROY;
1667         return;
1668     }

1670     /* signal any waiters that this dataset is going away */
1671     mutex_enter(&ds->ds_lock);
1672     ds->ds_owner = dsl_reaper;
1673     cv_broadcast(&ds->ds_exclusive_cv);
1674     mutex_exit(&ds->ds_lock);

1676     /* Remove our reservation */
1677     if (ds->ds_reserved != 0) {
1678         dsl_prop_setarg_t psa;
1679         uint64_t value = 0;

1681         dsl_prop_setarg_init_uint64(&psa, "reservation",
1682             (ZPROP_SRC_NONE | ZPROP_SRC_LOCAL | ZPROP_SRC_RECEIVED),
1683             &value);
1684         psa.psa_effective_value = 0; /* predict default value */

1686         dsl_dataset_set_reservation_sync(ds, &psa, tx);
1687         ASSERT3U(ds->ds_reserved, ==, 0);
1688     }

1690     ASSERT(RW_WRITE_HELD(&dp->dp_config_rwlock));

1692     dsl_scan_ds_destroyed(ds, tx);

1694     obj = ds->ds_object;

```

```

1696     if (ds->ds_phys->ds_prev_snap_obj != 0) {
1697         if (ds->ds_prev) {
1698             ds_prev = ds->ds_prev;
1699         } else {
1700             VERIFY(0 == dsl_dataset_hold_obj(dp,
1701                 ds->ds_phys->ds_prev_snap_obj, FTAG, &ds_prev));
1702         }
1703         after_branch_point =
1704             (ds_prev->ds_phys->ds_next_snap_obj != obj);

1706     dmu_buf_will_dirty(ds_prev->ds_dbuf, tx);
1707     if (after_branch_point &&
1708         ds_prev->ds_phys->ds_next_clones_obj != 0) {
1709         remove_from_next_clones(ds_prev, obj, tx);
1710         if (ds->ds_phys->ds_next_snap_obj != 0) {
1711             VERIFY(0 == zap_add_int(mos,
1712                 ds_prev->ds_phys->ds_next_clones_obj,
1713                 ds->ds_phys->ds_next_snap_obj, tx));
1714         }
1715     }
1716     if (after_branch_point &&
1717         ds->ds_phys->ds_next_snap_obj == 0) {
1718         /* This clone is toast. */
1719         ASSERT(ds_prev->ds_phys->ds_num_children > 1);
1720         ds_prev->ds_phys->ds_num_children--;

1722         /*
1723          * If the clone's origin has no other clones, no
1724          * user holds, and has been marked for deferred
1725          * deletion, then we should have done the necessary
1726          * destroy setup for it.
1727          */
1728         if (ds_prev->ds_phys->ds_num_children == 1 &&
1729             ds_prev->ds_userrefs == 0 &&
1730             DS_IS_DEFER_DESTROY(ds_prev)) {
1731             ASSERT3P(dsda->rm_origin, !=, NULL);
1732         } else {
1733             ASSERT3P(dsda->rm_origin, ==, NULL);
1734         }
1735     } else if (!after_branch_point) {
1736         ds_prev->ds_phys->ds_next_snap_obj =
1737             ds->ds_phys->ds_next_snap_obj;
1738     }
1739 }

1741 if (dsl_dataset_is_snapshot(ds)) {
1742     dsl_dataset_t *ds_next;
1743     uint64_t old_unique;
1744     uint64_t used = 0, comp = 0, uncomp = 0;

1746     VERIFY(0 == dsl_dataset_hold_obj(dp,
1747         ds->ds_phys->ds_next_snap_obj, FTAG, &ds_next));
1748     ASSERT3U(ds_next->ds_phys->ds_prev_snap_obj, ==, obj);

1750     old_unique = ds_next->ds_phys->ds_unique_bytes;

1752     dmu_buf_will_dirty(ds_next->ds_dbuf, tx);
1753     ds_next->ds_phys->ds_prev_snap_obj =
1754         ds->ds_phys->ds_prev_snap_obj;
1755     ds_next->ds_phys->ds_prev_snap_txg =
1756         ds->ds_phys->ds_prev_snap_txg;
1757     ASSERT3U(ds->ds_phys->ds_prev_snap_txg, ==,
1758         ds_prev ? ds_prev->ds_phys->ds_creation_txg : 0);

1761     if (ds_next->ds_deadlist.dl_oldfmt) {

```

```

1762         process_old_deadlist(ds, ds_prev, ds_next,
1763             after_branch_point, tx);
1764     } else {
1765         /* Adjust prev's unique space. */
1766         if (ds_prev && !after_branch_point) {
1767             dsl_deadlist_space_range(&ds_next->ds_deadlist,
1768                 ds_prev->ds_phys->ds_prev_snap_txg,
1769                 ds->ds_phys->ds_prev_snap_txg,
1770                 &used, &comp, &uncomp);
1771             ds_prev->ds_phys->ds_unique_bytes += used;
1772         }

1774         /* Adjust snapused. */
1775         dsl_deadlist_space_range(&ds_next->ds_deadlist,
1776             ds->ds_phys->ds_prev_snap_txg, UINT64_MAX,
1777             &used, &comp, &uncomp);
1778         dsl_dir_diduse_space(ds->ds_dir, DD_USED_SNAP,
1779             -used, -comp, -uncomp, tx);

1781         /* Move blocks to be freed to pool's free list. */
1782         dsl_deadlist_move_bproj(&ds_next->ds_deadlist,
1783             &dp->dp_free_bproj, ds->ds_phys->ds_prev_snap_txg,
1784             tx);
1785         dsl_dir_diduse_space(tx->tx_pool->dp_free_dir,
1786             DD_USED_HEAD, used, comp, uncomp, tx);

1788         /* Merge our deadlist into next's and free it. */
1789         dsl_deadlist_merge(&ds_next->ds_deadlist,
1790             ds->ds_phys->ds_deadlist_obj, tx);
1791     }
1792     dsl_deadlist_close(&ds->ds_deadlist);
1793     dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);

1795     /* Collapse range in clone heads */
1796     dsl_dataset_remove_clones_key(ds,
1797         ds->ds_phys->ds_creation_txg, tx);

1799     if (dsl_dataset_is_snapshot(ds_next)) {
1800         dsl_dataset_t *ds_nextnext;

1802         /*
1803          * Update next's unique to include blocks which
1804          * were previously shared by only this snapshot
1805          * and it. Those blocks will be born after the
1806          * prev snap and before this snap, and will have
1807          * died after the next snap and before the one
1808          * after that (ie. be on the snap after next's
1809          * deadlist).
1810          */
1811         VERIFY(0 == dsl_dataset_hold_obj(dp,
1812             ds_next->ds_phys->ds_next_snap_obj,
1813             FTAG, &ds_nextnext));
1814         dsl_deadlist_space_range(&ds_nextnext->ds_deadlist,
1815             ds->ds_phys->ds_prev_snap_txg,
1816             ds->ds_phys->ds_creation_txg,
1817             &used, &comp, &uncomp);
1818         ds_next->ds_phys->ds_unique_bytes += used;
1819         dsl_dataset_rele(ds_nextnext, FTAG);
1820         ASSERT3P(ds_next->ds_prev, ==, NULL);

1822         /* Collapse range in this head. */
1823         dsl_dataset_t *hds;
1824         VERIFY3U(0, ==, dsl_dataset_hold_obj(dp,
1825             ds->ds_dir->dd_phys->dd_head_dataset_obj,
1826             FTAG, &hds));
1827         dsl_deadlist_remove_key(&hds->ds_deadlist,

```

```

1828         ds->ds_phys->ds_creation_txg, tx);
1829         dsl_dataset_rele(hds, FTAG);

1831     } else {
1832         ASSERT3P(ds_next->ds_prev, ==, ds);
1833         dsl_dataset_drop_ref(ds_next->ds_prev, ds_next);
1834         ds_next->ds_prev = NULL;
1835         if (ds_prev) {
1836             VERIFY(0 == dsl_dataset_get_ref(dp,
1837                 ds->ds_phys->ds_prev_snap_obj,
1838                 ds_next, &ds_next->ds_prev));
1839         }

1841         dsl_dataset_recalc_head_uniq(ds_next);

1843         /*
1844          * Reduce the amount of our unconsmmed reservation
1845          * being charged to our parent by the amount of
1846          * new unique data we have gained.
1847          */
1848         if (old_unique < ds_next->ds_reserved) {
1849             int64_t mrsdelta;
1850             uint64_t new_unique =
1851                 ds_next->ds_phys->ds_unique_bytes;

1853             ASSERT(old_unique <= new_unique);
1854             mrsdelta = MIN(new_unique - old_unique,
1855                 ds_next->ds_reserved - old_unique);
1856             dsl_dir_diduse_space(ds->ds_dir,
1857                 DD_USED_REFRSRV, -mrsdelta, 0, 0, tx);
1858         }
1859     }
1860     dsl_dataset_rele(ds_next, FTAG);
1861 } else {
1862     zfeature_info_t *async_destroy =
1863         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY];

1865     /*
1866      * There's no next snapshot, so this is a head dataset.
1867      * Destroy the deadlist. Unless it's a clone, the
1868      * deadlist should be empty. (If it's a clone, it's
1869      * safe to ignore the deadlist contents.)
1870      */
1871     dsl_deadlist_close(&ds->ds_deadlist);
1872     dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);
1873     ds->ds_phys->ds_deadlist_obj = 0;

1875     if (!spa_feature_is_enabled(dp->dp_spa, async_destroy)) {
1876         err = old_synchronous_dataset_destroy(ds, tx);
1877     } else {
1878         /*
1879          * Move the bptree into the pool's list of trees to
1880          * clean up and update space accounting information.
1881          */
1882         uint64_t used, comp, uncomp;

1884         ASSERT(err == 0 || err == EBUSY);
1885         if (!spa_feature_is_active(dp->dp_spa, async_destroy)) {
1886             spa_feature_incr(dp->dp_spa, async_destroy, tx);
1887             dp->dp_bptree_obj = bptree_alloc(
1888                 dp->dp_meta_objset, tx);
1889             VERIFY(zap_add(DIRECTORY_OBJECT,
1890                 DMU_POOL_DIRECTORY_OBJECT,
1891                 DMU_POOL_BPTREE_OBJ, sizeof(uint64_t), 1,
1892                 &dp->dp_bptree_obj, tx) == 0);
1893         }

```

```

1895         used = ds->ds_dir->dd_phys->dd_used_bytes;
1896         comp = ds->ds_dir->dd_phys->dd_compressed_bytes;
1897         uncomp = ds->ds_dir->dd_phys->dd_uncompressed_bytes;

1899         ASSERT(!DS_UNIQUE_IS_ACCURATE(ds) ||
1900             ds->ds_phys->ds_unique_bytes == used);

1902         bptree_add(dp->dp_meta_objset, dp->dp_bptree_obj,
1903             &ds->ds_phys->ds_bp, ds->ds_phys->ds_prev_snap_txg,
1904             used, comp, uncomp, tx);
1905         dsl_dir_diduse_space(ds->ds_dir, DD_USED_HEAD,
1906             -used, -comp, -uncomp, tx);
1907         dsl_dir_diduse_space(dp->dp_free_dir, DD_USED_HEAD,
1908             used, comp, uncomp, tx);
1909     }

1911     if (ds->ds_prev != NULL) {
1912         if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
1913             VERIFY3U(0, ==, zap_remove_int(mos,
1914                 ds->ds_prev->ds_dir->dd_phys->dd_clones,
1915                 ds->ds_object, tx));
1916         }
1917         dsl_dataset_rele(ds->ds_prev, ds);
1918         ds->ds_prev = ds_prev = NULL;
1919     }
1920 }

1922     /*
1923      * This must be done after the dsl_traverse(), because it will
1924      * re-open the objset.
1925      */
1926     if (ds->ds_objset) {
1927         dmu_objset_evict(ds->ds_objset);
1928         ds->ds_objset = NULL;
1929     }

1931     if (ds->ds_dir->dd_phys->dd_head_dataset_obj == ds->ds_object) {
1932         /* Erase the link in the dir */
1933         dmu_buf_will_dirty(ds->ds_dir->dd_dbuf, tx);
1934         ds->ds_dir->dd_phys->dd_head_dataset_obj = 0;
1935         ASSERT(ds->ds_phys->ds_snapnames_zapobj != 0);
1936         err = zap_destroy(mos, ds->ds_phys->ds_snapnames_zapobj, tx);
1937         ASSERT(err == 0);
1938     } else {
1939         /* remove from snapshot namespace */
1940         dsl_dataset_t *ds_head;
1941         ASSERT(ds->ds_phys->ds_snapnames_zapobj == 0);
1942         VERIFY(0 == dsl_dataset_hold_obj(dp,
1943             ds->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &ds_head));
1944         VERIFY(0 == dsl_dataset_get_snapname(ds));
1945 #ifndef ZFS_DEBUG
1946         {
1947             uint64_t val;

1949             err = dsl_dataset_snap_lookup(ds_head,
1950                 ds->ds_snapname, &val);
1951             ASSERT3U(err, ==, 0);
1952             ASSERT3U(val, ==, obj);
1953         }
1954 #endif
1955         err = dsl_dataset_snap_remove(ds_head, ds->ds_snapname, tx);
1956         ASSERT(err == 0);
1957         dsl_dataset_rele(ds_head, FTAG);
1958     }

```

```

1960     if (ds_prev && ds->ds_prev != ds_prev)
1961         dsl_dataset_rele(ds_prev, FTAG);

1963     spa_prop_clear_bootfs(dp->dp_spa, ds->ds_object, tx);
1964     spa_history_log_internal(LOG_DS_DESTROY, dp->dp_spa, tx,
1965         "dataset = %llu", ds->ds_object);

1967     if (ds->ds_phys->ds_next_clones_obj != 0) {
1968         uint64_t count;
1969         ASSERT(0 == zap_count(mos,
1970             ds->ds_phys->ds_next_clones_obj, &count) && count == 0);
1971         VERIFY(0 == dmu_object_free(mos,
1972             ds->ds_phys->ds_next_clones_obj, tx));
1973     }
1974     if (ds->ds_phys->ds_props_obj != 0)
1975         VERIFY(0 == zap_destroy(mos, ds->ds_phys->ds_props_obj, tx));
1976     if (ds->ds_phys->ds_userrefs_obj != 0)
1977         VERIFY(0 == zap_destroy(mos, ds->ds_phys->ds_userrefs_obj, tx));
1978     dsl_dir_close(ds->ds_dir, ds);
1979     ds->ds_dir = NULL;
1980     dsl_dataset_drain_refs(ds, tag);
1981     VERIFY(0 == dmu_object_free(mos, obj, tx));

1983     if (dsda->rm_origin) {
1984         /*
1985          * Remove the origin of the clone we just destroyed.
1986          */
1987         struct dsl_ds_destroyarg ndsda = {0};

1989         ndsda.ds = dsda->rm_origin;
1990         dsl_dataset_destroy_sync(&ndsda, tag, tx);
1991     }
1992 }

1994 static int
1995 dsl_dataset_snapshot_reserve_space(dsl_dataset_t *ds, dmu_tx_t *tx)
1996 {
1997     uint64_t asize;

1999     if (!dmu_tx_is_syncing(tx))
2000         return (0);

2002     /*
2003      * If there's an fs-only reservation, any blocks that might become
2004      * owned by the snapshot dataset must be accommodated by space
2005      * outside of the reservation.
2006      */
2007     ASSERT(ds->ds_reserved == 0 || DS_UNIQUE_IS_ACCURATE(ds));
2008     asize = MIN(ds->ds_phys->ds_unique_bytes, ds->ds_reserved);
2009     if (asize > dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE))
2010         return (ENOSPC);

2012     /*
2013      * Propagate any reserved space for this snapshot to other
2014      * snapshot checks in this sync group.
2015      */
2016     if (asize > 0)
2017         dsl_dir_willuse_space(ds->ds_dir, asize, tx);

2019     return (0);
2020 }

2022 int
2023 dsl_dataset_snapshot_check(void *arg1, void *arg2, dmu_tx_t *tx)
2024 {
2025     dsl_dataset_t *ds = arg1;

```

```

2026     const char *snapname = arg2;
2027     int err;
2028     uint64_t value;

2030     /*
2031      * We don't allow multiple snapshots of the same txg.  If there
2032      * is already one, try again.
2033      */
2034     if (ds->ds_phys->ds_prev_snap_txg >= tx->tx_txg)
2035         return (EAGAIN);

2037     /*
2038      * Check for conflicting name snapshot name.
2039      */
2040     err = dsl_dataset_snap_lookup(ds, snapname, &value);
2041     if (err == 0)
2042         return (EEXIST);
2043     if (err != ENOENT)
2044         return (err);

2046     /*
2047      * Check that the dataset's name is not too long.  Name consists
2048      * of the dataset's length + 1 for the @-sign + snapshot name's length
2049      */
2050     if (dsl_dataset_namelen(ds) + 1 + strlen(snapname) >= MAXNAMELEN)
2051         return (ENAMETOOLONG);

2053     err = dsl_dataset_snapshot_reserve_space(ds, tx);
2054     if (err)
2055         return (err);

2057     ds->ds_trysnap_txg = tx->tx_txg;
2058     return (0);
2059 }

2061 void
2062 dsl_dataset_snapshot_sync(void *arg1, void *arg2, dmu_tx_t *tx)
2063 {
2064     dsl_dataset_t *ds = arg1;
2065     const char *snapname = arg2;
2066     dsl_pool_t *dp = ds->ds_dir->dd_pool;
2067     dmu_buf_t *dbuf;
2068     dsl_dataset_phys_t *dsphys;
2069     uint64_t dsobj, crttxg;
2070     objset_t *mos = dp->dp_meta_objset;
2071     int err;

2073     ASSERT(RW_WRITE_HELD(&dp->dp_config_rwlock));

2075     /*
2076      * The origin's ds_creation_txg has to be < TXG_INITIAL
2077      */
2078     if (strcmp(snapname, ORIGIN_DIR_NAME) == 0)
2079         crttxg = 1;
2080     else
2081         crttxg = tx->tx_txg;

2083     dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
2084         DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
2085     VERIFY(0 == dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
2086     dmu_buf_will_dirty(dbuf, tx);
2087     dsphys = dbuf->db_data;
2088     bzero(dsphys, sizeof (dsl_dataset_phys_t));
2089     dsphys->ds_dir_obj = ds->ds_dir->dd_object;
2090     dsphys->ds_fsid_guid = unique_create();
2091     do {

```

```

2092 #endif /* ! codereview */
2093         (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
2094             sizeof(dsphys->ds_guid));
2095     } while (dsphys->ds_guid == 0);
2096 #endif /* ! codereview */
2097     dsphys->ds_prev_snap_obj = ds->ds_phys->ds_prev_snap_obj;
2098     dsphys->ds_prev_snap_txg = ds->ds_phys->ds_prev_snap_txg;
2099     dsphys->ds_next_snap_obj = ds->ds_object;
2100     dsphys->ds_num_children = 1;
2101     dsphys->ds_creation_time = gethrstime_sec();
2102     dsphys->ds_creation_txg = crtxg;
2103     dsphys->ds_deadlist_obj = ds->ds_phys->ds_deadlist_obj;
2104     dsphys->ds_referenced_bytes = ds->ds_phys->ds_referenced_bytes;
2105     dsphys->ds_compressed_bytes = ds->ds_phys->ds_compressed_bytes;
2106     dsphys->ds_uncompressed_bytes = ds->ds_phys->ds_uncompressed_bytes;
2107     dsphys->ds_flags = ds->ds_phys->ds_flags;
2108     dsphys->ds_bp = ds->ds_phys->ds_bp;
2109     dmu_buf_rele(dbuf, FTAG);

2111     ASSERT3U(ds->ds_prev != 0, ==, ds->ds_phys->ds_prev_snap_obj != 0);
2112     if (ds->ds_prev) {
2113         uint64_t next_clones_obj =
2114             ds->ds_prev->ds_phys->ds_next_clones_obj;
2115         ASSERT(ds->ds_prev->ds_phys->ds_next_snap_obj ==
2116             ds->ds_object ||
2117             ds->ds_prev->ds_phys->ds_num_children > 1);
2118         if (ds->ds_prev->ds_phys->ds_next_snap_obj == ds->ds_object) {
2119             dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
2120             ASSERT3U(ds->ds_phys->ds_prev_snap_txg, ==,
2121                 ds->ds_prev->ds_phys->ds_creation_txg);
2122             ds->ds_prev->ds_phys->ds_next_snap_obj = dsobj;
2123         } else if (next_clones_obj != 0) {
2124             remove_from_next_clones(ds->ds_prev,
2125                 dsphys->ds_next_snap_obj, tx);
2126             VERIFY3U(0, ==, zap_add_int(mos,
2127                 next_clones_obj, dsobj, tx));
2128         }
2129     }

2131     /*
2132     * If we have a reference-reservation on this dataset, we will
2133     * need to increase the amount of reservation being charged
2134     * since our unique space is going to zero.
2135     */
2136     if (ds->ds_reserved) {
2137         int64_t delta;
2138         ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
2139         delta = MIN(ds->ds_phys->ds_unique_bytes, ds->ds_reserved);
2140         dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV,
2141             delta, 0, 0, tx);
2142     }

2144     dmu_buf_will_dirty(ds->ds_dbuf, tx);
2145     zfs_dbgmsg("taking snapshot %s@%s/%llu",
2146         ds->ds_dir->dd_myname, snapname, dsobj,
2147         ds->ds_phys->ds_prev_snap_txg);
2148     ds->ds_phys->ds_deadlist_obj = dsl_deadlist_clone(&ds->ds_deadlist,
2149         UINT64_MAX, ds->ds_phys->ds_prev_snap_obj, tx);
2150     dsl_deadlist_close(&ds->ds_deadlist);
2151     dsl_deadlist_open(&ds->ds_deadlist, mos, ds->ds_phys->ds_deadlist_obj);
2152     dsl_deadlist_add_key(&ds->ds_deadlist,
2153         ds->ds_phys->ds_prev_snap_txg, tx);

2155     ASSERT3U(ds->ds_phys->ds_prev_snap_txg, <, tx->tx_txg);
2156     ds->ds_phys->ds_prev_snap_obj = dsobj;
2157     ds->ds_phys->ds_prev_snap_txg = crtxg;

```

```

2158     ds->ds_phys->ds_unique_bytes = 0;
2159     if (spa_version(dp->dp_spa) >= SPA_VERSION_UNIQUE_ACCURATE)
2160         ds->ds_phys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;

2162     err = zap_add(mos, ds->ds_phys->ds_snapnames_zapobj,
2163         snapname, 8, 1, &dsobj, tx);
2164     ASSERT(err == 0);

2166     if (ds->ds_prev)
2167         dsl_dataset_drop_ref(ds->ds_prev, ds);
2168     VERIFY(0 == dsl_dataset_get_ref(dp,
2169         ds->ds_phys->ds_prev_snap_obj, ds, &ds->ds_prev));

2171     dsl_scan_ds_snapshotted(ds, tx);

2173     dsl_dir_snap_cmtime_update(ds->ds_dir);

2175     spa_history_log_internal(LOG_DS_SNAPSHOT, dp->dp_spa, tx,
2176         "dataset = %llu", dsobj);
2177 }

2179 void
2180 dsl_dataset_sync(dsl_dataset_t *ds, zio_t *zio, dmu_tx_t *tx)
2181 {
2182     ASSERT(dmu_tx_is_syncing(tx));
2183     ASSERT(ds->ds_objset != NULL);
2184     ASSERT(ds->ds_phys->ds_next_snap_obj == 0);

2186     /*
2187     * in case we had to change ds_fsid_guid when we opened it,
2188     * sync it out now.
2189     */
2190     dmu_buf_will_dirty(ds->ds_dbuf, tx);
2191     ds->ds_phys->ds_fsid_guid = ds->ds_fsid_guid;

2193     dsl_dir_dirty(ds->ds_dir, tx);
2194     dmu_objset_sync(ds->ds_objset, zio, tx);
2195 }

2197 static void
2198 get_clones_stat(dsl_dataset_t *ds, nvlist_t *nv)
2199 {
2200     uint64_t count = 0;
2201     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
2202     zap_cursor_t zc;
2203     zap_attribute_t za;
2204     nvlist_t *propval;
2205     nvlist_t *val;

2207     rw_enter(&ds->ds_dir->dd_pool->dp_config_rwlock, RW_READER);
2208     VERIFY(nvlist_alloc(&propval, NV_UNIQUE_NAME, KM_SLEEP) == 0);
2209     VERIFY(nvlist_alloc(&val, NV_UNIQUE_NAME, KM_SLEEP) == 0);

2211     /*
2212     * There may be missing entries in ds_next_clones_obj
2213     * due to a bug in a previous version of the code.
2214     * Only trust it if it has the right number of entries.
2215     */
2216     if (ds->ds_phys->ds_next_clones_obj != 0) {
2217         ASSERT3U(0, ==, zap_count(mos, ds->ds_phys->ds_next_clones_obj,
2218             &count));
2219     }
2220     if (count != ds->ds_phys->ds_num_children - 1) {
2221         goto fail;
2222     }
2223     for (zap_cursor_init(&zc, mos, ds->ds_phys->ds_next_clones_obj);

```

```

2224     zap_cursor_retrieve(&zcz, &za) == 0;
2225     zap_cursor_advance(&zcz) {
2226         dsl_dataset_t *clone;
2227         char buf[ZFS_MAXNAMELEN];
2228         /*
2229          * Even though we hold the dp_config_rwlock, the dataset
2230          * may fail to open, returning ENOENT. If there is a
2231          * thread concurrently attempting to destroy this
2232          * dataset, it will have the ds_rwlock held for
2233          * RW_WRITER. Our call to dsl_dataset_hold_obj() ->
2234          * dsl_dataset_hold_ref() will fail its
2235          * rw_tryenter(&ds->ds_rwlock, RW_READER), drop the
2236          * dp_config_rwlock, and wait for the destroy progress
2237          * and signal ds_exclusive_cv. If the destroy was
2238          * successful, we will see that
2239          * DSL_DATASET_IS_DESTROYED(), and return ENOENT.
2240          */
2241         if (dsl_dataset_hold_obj(ds->ds_dir->dd_pool,
2242             za.za_first_integer, FTAG, &clone) != 0)
2243             continue;
2244         dsl_dir_name(clone->ds_dir, buf);
2245         VERIFY(nvlist_add_boolean(val, buf) == 0);
2246         dsl_dataset_rele(clone, FTAG);
2247     }
2248     zap_cursor_fini(&zcz);
2249     VERIFY(nvlist_add_nvlist(propval, ZPROP_VALUE, val) == 0);
2250     VERIFY(nvlist_add_nvlist(nv, zfs_prop_to_name(ZFS_PROP_CLONES),
2251         propval) == 0);
2252 fail:
2253     nvlist_free(val);
2254     nvlist_free(propval);
2255     rw_exit(&ds->ds_dir->dd_pool->dp_config_rwlock);
2256 }

2258 void
2259 dsl_dataset_stats(dsl_dataset_t *ds, nvlist_t *nv)
2260 {
2261     uint64_t refd, avail, uobjjs, aobjjs, ratio;

2263     dsl_dir_stats(ds->ds_dir, nv);

2265     dsl_dataset_space(ds, &refd, &avail, &uobjjs, &aobjjs);
2266     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_AVAILABLE, avail);
2267     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFERENCED, refd);

2269     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATION,
2270         ds->ds_phys->ds_creation_time);
2271     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATETXG,
2272         ds->ds_phys->ds_creation_txg);
2273     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFQUOTA,
2274         ds->ds_quota);
2275     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRESERVATION,
2276         ds->ds_reserved);
2277     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_GUID,
2278         ds->ds_phys->ds_guid);
2279     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_UNIQUE,
2280         ds->ds_phys->ds_unique_bytes);
2281     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_OBJSETID,
2282         ds->ds_object);
2283     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERREFS,
2284         ds->ds_userrefs);
2285     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_DEFER_DESTROY,
2286         DS_IS_DEFER_DESTROY(ds) ? 1 : 0);

2288     if (ds->ds_phys->ds_prev_snap_obj != 0) {
2289         uint64_t written, comp, uncomp;

```

```

2290         dsl_pool_t *dp = ds->ds_dir->dd_pool;
2291         dsl_dataset_t *prev;

2293         rw_enter(&dp->dp_config_rwlock, RW_READER);
2294         int err = dsl_dataset_hold_obj(dp,
2295             ds->ds_phys->ds_prev_snap_obj, FTAG, &prev);
2296         rw_exit(&dp->dp_config_rwlock);
2297         if (err == 0) {
2298             err = dsl_dataset_space_written(prev, ds, &written,
2299                 &comp, &uncomp);
2300             dsl_dataset_rele(prev, FTAG);
2301             if (err == 0) {
2302                 dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_WRITTEN,
2303                     written);
2304             }
2305         }
2306     }

2308     ratio = ds->ds_phys->ds_compressed_bytes == 0 ? 100 :
2309         (ds->ds_phys->ds_uncompressed_bytes * 100 /
2310             ds->ds_phys->ds_compressed_bytes);
2311     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRATIO, ratio);

2313     if (ds->ds_phys->ds_next_snap_obj) {
2314         /*
2315          * This is a snapshot; override the dd's space used with
2316          * our unique space and compression ratio.
2317          */
2318         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USED,
2319             ds->ds_phys->ds_unique_bytes);
2320         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_COMPRESSRATIO, ratio);

2322         get_clones_stat(ds, nv);
2323     }
2324 }

2326 void
2327 dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat)
2328 {
2329     stat->dds_creation_txg = ds->ds_phys->ds_creation_txg;
2330     stat->dds_inconsistent = ds->ds_phys->ds_flags & DS_FLAG_INCONSISTENT;
2331     stat->dds_guid = ds->ds_phys->ds_guid;
2332     if (ds->ds_phys->ds_next_snap_obj) {
2333         stat->dds_is_snapshot = B_TRUE;
2334         stat->dds_num_clones = ds->ds_phys->ds_num_children - 1;
2335     } else {
2336         stat->dds_is_snapshot = B_FALSE;
2337         stat->dds_num_clones = 0;
2338     }

2340     /* clone origin is really a dsl_dir thing... */
2341     rw_enter(&ds->ds_dir->dd_pool->dp_config_rwlock, RW_READER);
2342     if (dsl_dir_is_clone(ds->ds_dir)) {
2343         dsl_dataset_t *ods;

2345         VERIFY(0 == dsl_dataset_get_ref(ds->ds_dir->dd_pool,
2346             ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &ods));
2347         dsl_dataset_name(ods, stat->dds_origin);
2348         dsl_dataset_drop_ref(ods, FTAG);
2349     } else {
2350         stat->dds_origin[0] = '\0';
2351     }
2352     rw_exit(&ds->ds_dir->dd_pool->dp_config_rwlock);
2353 }

2355 uint64_t

```

```

2356 dsl_dataset_fsid_guid(dsl_dataset_t *ds)
2357 {
2358     return (ds->ds_fsid_guid);
2359 }

2361 void
2362 dsl_dataset_space(dsl_dataset_t *ds,
2363     uint64_t *refdbbytesp, uint64_t *availbytesp,
2364     uint64_t *usedobjsp, uint64_t *availobjsp)
2365 {
2366     *refdbbytesp = ds->ds_phys->ds_referenced_bytes;
2367     *availbytesp = dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE);
2368     if (ds->ds_reserved > ds->ds_phys->ds_unique_bytes)
2369         *availbytesp += ds->ds_reserved - ds->ds_phys->ds_unique_bytes;
2370     if (ds->ds_quota != 0) {
2371         /*
2372          * Adjust available bytes according to refquota
2373          */
2374         if (*refdbbytesp < ds->ds_quota)
2375             *availbytesp = MIN(*availbytesp,
2376                 ds->ds_quota - *refdbbytesp);
2377         else
2378             *availbytesp = 0;
2379     }
2380     *usedobjsp = ds->ds_phys->ds_bp.blk_fill;
2381     *availobjsp = DN_MAX_OBJECT - *usedobjsp;
2382 }

2384 boolean_t
2385 dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds)
2386 {
2387     dsl_pool_t *dp = ds->ds_dir->dd_pool;

2389     ASSERT(RW_LOCK_HELD(&dp->dp_config_rwlock) ||
2390         dsl_pool_sync_context(dp));
2391     if (ds->ds_prev == NULL)
2392         return (B_FALSE);
2393     if (ds->ds_phys->ds_bp.blk_birth >
2394         ds->ds_prev->ds_phys->ds_creation_txg) {
2395         objset_t *os, *os_prev;
2396         /*
2397          * It may be that only the ZIL differs, because it was
2398          * reset in the head. Don't count that as being
2399          * modified.
2400          */
2401         if (dmu_objset_from_ds(ds, &os) != 0)
2402             return (B_TRUE);
2403         if (dmu_objset_from_ds(ds->ds_prev, &os_prev) != 0)
2404             return (B_TRUE);
2405         return (bcmp(&os->os_phys->os_meta_dnode,
2406             &os_prev->os_phys->os_meta_dnode,
2407             sizeof (os->os_phys->os_meta_dnode)) != 0);
2408     }
2409     return (B_FALSE);
2410 }

2412 /* ARGSUSED */
2413 static int
2414 dsl_dataset_snapshot_rename_check(void *arg1, void *arg2, dmu_tx_t *tx)
2415 {
2416     dsl_dataset_t *ds = arg1;
2417     char *newsnapname = arg2;
2418     dsl_dir_t *dd = ds->ds_dir;
2419     dsl_dataset_t *hds;
2420     uint64_t val;
2421     int err;

```

```

2423     err = dsl_dataset_hold_obj(dd->dd_pool,
2424         dd->dd_phys->dd_head_dataset_obj, FTAG, &hds);
2425     if (err)
2426         return (err);

2428     /* new name better not be in use */
2429     err = dsl_dataset_snap_lookup(hds, newsnapname, &val);
2430     dsl_dataset_rele(hds, FTAG);

2432     if (err == 0)
2433         err = EEXIST;
2434     else if (err == ENOENT)
2435         err = 0;

2437     /* dataset name + 1 for the "@" + the new snapshot name must fit */
2438     if (dsl_dir_namelen(ds->ds_dir) + 1 + strlen(newsnapname) >= MAXNAMELEN)
2439         err = ENAMETOOLONG;

2441     return (err);
2442 }

2444 static void
2445 dsl_dataset_snapshot_rename_sync(void *arg1, void *arg2, dmu_tx_t *tx)
2446 {
2447     dsl_dataset_t *ds = arg1;
2448     const char *newsnapname = arg2;
2449     dsl_dir_t *dd = ds->ds_dir;
2450     objset_t *mos = dd->dd_pool->dp_meta_objset;
2451     dsl_dataset_t *hds;
2452     int err;

2454     ASSERT(ds->ds_phys->ds_next_snap_obj != 0);

2456     VERIFY(0 == dsl_dataset_hold_obj(dd->dd_pool,
2457         dd->dd_phys->dd_head_dataset_obj, FTAG, &hds));

2459     VERIFY(0 == dsl_dataset_get_snapname(ds));
2460     err = dsl_dataset_snap_remove(hds, ds->ds_snapname, tx);
2461     ASSERT3U(err, ==, 0);
2462     mutex_enter(&ds->ds_lock);
2463     (void) strcpy(ds->ds_snapname, newsnapname);
2464     mutex_exit(&ds->ds_lock);
2465     err = zap_add(mos, hds->ds_phys->ds_snapnames_zapobj,
2466         ds->ds_snapname, 8, 1, &ds->ds_object, tx);
2467     ASSERT3U(err, ==, 0);

2469     spa_history_log_internal(LOG_DS_RENAME, dd->dd_pool->dp_spa, tx,
2470         "dataset = %llu", ds->ds_object);
2471     dsl_dataset_rele(hds, FTAG);
2472 }

2474 struct renamesnaparg {
2475     dsl_sync_task_group_t *dstg;
2476     char failed[MAXPATHLEN];
2477     char *oldsnap;
2478     char *newsnap;
2479 };

2481 static int
2482 dsl_snapshot_rename_one(const char *name, void *arg)
2483 {
2484     struct renamesnaparg *ra = arg;
2485     dsl_dataset_t *ds = NULL;
2486     char *snapname;
2487     int err;

```

```

2489     snapname = kmem_asprintf("%s@%s", name, ra->oldsnap);
2490     (void) strncpy(ra->failed, snapname, sizeof (ra->failed));

2492     /*
2493      * For recursive snapshot renames the parent won't be changing
2494      * so we just pass name for both the to/from argument.
2495      */
2496     err = zfs_secpolicy_rename_perms(snapname, snapname, CRED());
2497     if (err != 0) {
2498         strfree(snapname);
2499         return (err == ENOENT ? 0 : err);
2500     }

2502 #ifdef _KERNEL
2503     /*
2504      * For all filesystems undergoing rename, we'll need to unmount it.
2505      */
2506     (void) zfs_unmount_snap(snapname, NULL);
2507 #endif
2508     err = dsl_dataset_hold(snapname, ra->dstg, &ds);
2509     strfree(snapname);
2510     if (err != 0)
2511         return (err == ENOENT ? 0 : err);

2513     dsl_sync_task_create(ra->dstg, dsl_dataset_snapshot_rename_check,
2514         dsl_dataset_snapshot_rename_sync, ds, ra->newsnap, 0);

2516     return (0);
2517 }

2519 static int
2520 dsl_recursive_rename(char *oldname, const char *newname)
2521 {
2522     int err;
2523     struct renamesnaparg *ra;
2524     dsl_sync_task_t *dst;
2525     spa_t *spa;
2526     char *cp, *fsname = spa_strdup(oldname);
2527     int len = strlen(oldname) + 1;

2529     /* truncate the snapshot name to get the fsname */
2530     cp = strchr(fsname, '@');
2531     *cp = '\0';

2533     err = spa_open(fsname, &spa, FTAG);
2534     if (err) {
2535         kmem_free(fsname, len);
2536         return (err);
2537     }
2538     ra = kmem_alloc(sizeof (struct renamesnaparg), KM_SLEEP);
2539     ra->dstg = dsl_sync_task_group_create(spa_get_dsl(spa));

2541     ra->oldsnap = strchr(oldname, '@') + 1;
2542     ra->newsnap = strchr(newname, '@') + 1;
2543     *ra->failed = '\0';

2545     err = dmu_objset_find(fsname, dsl_snapshot_rename_one, ra,
2546         DS_FIND_CHILDREN);
2547     kmem_free(fsname, len);

2549     if (err == 0) {
2550         err = dsl_sync_task_group_wait(ra->dstg);
2551     }

2553     for (dst = list_head(&ra->dstg->dstg_tasks); dst;

```

```

2554         dst = list_next(&ra->dstg->dstg_tasks, dst)) {
2555         dsl_dataset_t *ds = dst->dst_arg1;
2556         if (dst->dst_err) {
2557             dsl_dir_name(ds->ds_dir, ra->failed);
2558             (void) strlcat(ra->failed, "@", sizeof (ra->failed));
2559             (void) strlcat(ra->failed, ra->newsnap,
2560                 sizeof (ra->failed));
2561         }
2562         dsl_dataset_rele(ds, ra->dstg);
2563     }

2565     if (err)
2566         (void) strncpy(oldname, ra->failed, sizeof (ra->failed));

2568     dsl_sync_task_group_destroy(ra->dstg);
2569     kmem_free(ra, sizeof (struct renamesnaparg));
2570     spa_close(spa, FTAG);
2571     return (err);
2572 }

2574 static int
2575 dsl_valid_rename(const char *oldname, void *arg)
2576 {
2577     int delta = *(int *)arg;

2579     if (strlen(oldname) + delta >= MAXNAMELEN)
2580         return (ENAMETOOLONG);

2582     return (0);
2583 }

2585 #pragma weak dmu_objset_rename = dsl_dataset_rename
2586 int
2587 dsl_dataset_rename(char *oldname, const char *newname, boolean_t recursive)
2588 {
2589     dsl_dir_t *dd;
2590     dsl_dataset_t *ds;
2591     const char *tail;
2592     int err;

2594     err = dsl_dir_open(oldname, FTAG, &dd, &tail);
2595     if (err)
2596         return (err);

2598     if (tail == NULL) {
2599         int delta = strlen(newname) - strlen(oldname);

2601         /* if we're growing, validate child name lengths */
2602         if (delta > 0)
2603             err = dmu_objset_find(oldname, dsl_valid_rename,
2604                 &delta, DS_FIND_CHILDREN | DS_FIND_SNAPSHOTS);

2606         if (err == 0)
2607             err = dsl_dir_rename(dd, newname);
2608         dsl_dir_close(dd, FTAG);
2609         return (err);
2610     }

2612     if (tail[0] != '@') {
2613         /* the name ended in a nonexistent component */
2614         dsl_dir_close(dd, FTAG);
2615         return (ENOENT);
2616     }

2618     dsl_dir_close(dd, FTAG);

```

```

2620 /* new name must be snapshot in same filesystem */
2621 tail = strchr(newname, '@');
2622 if (tail == NULL)
2623     return (EINVAL);
2624 tail++;
2625 if (strncmp(oldname, newname, tail - newname) != 0)
2626     return (EXDEV);
2627
2628 if (recursive) {
2629     err = dsl_recursive_rename(oldname, newname);
2630 } else {
2631     err = dsl_dataset_hold(oldname, FTAG, &ds);
2632     if (err)
2633         return (err);
2634
2635     err = dsl_sync_task_do(ds->ds_dir->dd_pool,
2636         dsl_dataset_snapshot_rename_check,
2637         dsl_dataset_snapshot_rename_sync, ds, (char *)tail, 1);
2638
2639     dsl_dataset_rele(ds, FTAG);
2640 }
2641
2642 return (err);
2643 }
2644
2645 struct promotenode {
2646     list_node_t link;
2647     dsl_dataset_t *ds;
2648 };
2649
2650 struct promotearg {
2651     list_t shared_snaps, origin_snaps, clone_snaps;
2652     dsl_dataset_t *origin_origin;
2653     uint64_t used, comp, uncomp, unique, cloneusedsnap, originusedsnap;
2654     char *err_ds;
2655 };
2656
2657 static int snaplist_space(list_t *l, uint64_t mintxg, uint64_t *spacep);
2658 static boolean_t snaplist_unstable(list_t *l);
2659
2660 static int
2661 dsl_dataset_promote_check(void *arg1, void *arg2, dmu_tx_t *tx)
2662 {
2663     dsl_dataset_t *hds = arg1;
2664     struct promotearg *pa = arg2;
2665     struct promotenode *snap = list_head(&pa->shared_snaps);
2666     dsl_dataset_t *origin_ds = snap->ds;
2667     int err;
2668     uint64_t unused;
2669
2670     /* Check that it is a real clone */
2671     if (!dsl_dir_is_clone(hds->ds_dir))
2672         return (EINVAL);
2673
2674     /* Since this is so expensive, don't do the preliminary check */
2675     if (!dmu_tx_is_syncing(tx))
2676         return (0);
2677
2678     if (hds->ds_phys->ds_flags & DS_FLAG_NOPROMOTE)
2679         return (EXDEV);
2680
2681     /* compute origin's new unique space */
2682     snap = list_tail(&pa->clone_snaps);
2683     ASSERT3U(snap->ds->ds_phys->ds_prev_snap_obj, ==, origin_ds->ds_object);
2684     dsl_deadlist_space_range(&snap->ds->ds_deadlist,
2685         origin_ds->ds_phys->ds_prev_snap_txg, UINT64_MAX,

```

```

2686     &pa->unique, &unused, &unused);
2687
2688     /*
2689     * Walk the snapshots that we are moving
2690     *
2691     * Compute space to transfer. Consider the incremental changes
2692     * to used for each snapshot:
2693     * (my used) = (prev's used) + (blocks born) - (blocks killed)
2694     * So each snapshot gave birth to:
2695     * (blocks born) = (my used) - (prev's used) + (blocks killed)
2696     * So a sequence would look like:
2697     * (uN - u(N-1) + kN) + ... + (u1 - u0 + k1) + (u0 - 0 + k0)
2698     * Which simplifies to:
2699     * uN + kN + kN-1 + ... + k1 + k0
2700     * Note however, if we stop before we reach the ORIGIN we get:
2701     * uN + kN + kN-1 + ... + kM - uM-1
2702     */
2703     pa->used = origin_ds->ds_phys->ds_referenced_bytes;
2704     pa->comp = origin_ds->ds_phys->ds_compressed_bytes;
2705     pa->uncomp = origin_ds->ds_phys->ds_uncompressed_bytes;
2706     for (snap = list_head(&pa->shared_snaps); snap;
2707         snap = list_next(&pa->shared_snaps, snap)) {
2708         uint64_t val, dlused, dlcomp, dluncomp;
2709         dsl_dataset_t *ds = snap->ds;
2710
2711         /* Check that the snapshot name does not conflict */
2712         VERIFY(0 == dsl_dataset_get_snapname(ds));
2713         err = dsl_dataset_snap_lookup(hds, ds->ds_snapname, &val);
2714         if (err == 0) {
2715             err = EEXIST;
2716             goto out;
2717         }
2718         if (err != ENOENT)
2719             goto out;
2720
2721         /* The very first snapshot does not have a deadlist */
2722         if (ds->ds_phys->ds_prev_snap_obj == 0)
2723             continue;
2724
2725         dsl_deadlist_space(&ds->ds_deadlist,
2726             &dlused, &dlcomp, &dluncomp);
2727         pa->used += dlused;
2728         pa->comp += dlcomp;
2729         pa->uncomp += dluncomp;
2730     }
2731
2732     /*
2733     * If we are a clone of a clone then we never reached ORIGIN,
2734     * so we need to subtract out the clone origin's used space.
2735     */
2736     if (pa->origin_origin) {
2737         pa->used -= pa->origin_origin->ds_phys->ds_referenced_bytes;
2738         pa->comp -= pa->origin_origin->ds_phys->ds_compressed_bytes;
2739         pa->uncomp -= pa->origin_origin->ds_phys->ds_uncompressed_bytes;
2740     }
2741
2742     /* Check that there is enough space here */
2743     err = dsl_dir_transfer_possible(origin_ds->ds_dir, hds->ds_dir,
2744         pa->used);
2745     if (err)
2746         return (err);
2747
2748     /*
2749     * Compute the amounts of space that will be used by snapshots
2750     * after the promotion (for both origin and clone). For each,
2751     * it is the amount of space that will be on all of their

```

```

2752     * deadlists (that was not born before their new origin).
2753     */
2754     if (hds->ds_dir->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
2755         uint64_t space;

2757         /*
2758          * Note, typically this will not be a clone of a clone,
2759          * so dd_origin_tngx will be < TXG_INITIAL, so
2760          * these snaplist_space() -> dsl_deadlist_space_range()
2761          * calls will be fast because they do not have to
2762          * iterate over all bps.
2763          */
2764         snap = list_head(&pa->origin_snaps);
2765         err = snaplist_space(&pa->shared_snaps,
2766             snap->ds->ds_dir->dd_origin_tngx, &pa->cloneusedsnap);
2767         if (err)
2768             return (err);

2770         err = snaplist_space(&pa->clone_snaps,
2771             snap->ds->ds_dir->dd_origin_tngx, &space);
2772         if (err)
2773             return (err);
2774         pa->cloneusedsnap += space;
2775     }
2776     if (origin_ds->ds_dir->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
2777         err = snaplist_space(&pa->origin_snaps,
2778             origin_ds->ds_phys->ds_creation_tngx, &pa->originusedsnap);
2779         if (err)
2780             return (err);
2781     }

2783     return (0);
2784 out:
2785     pa->err_ds = snap->ds->ds_snapname;
2786     return (err);
2787 }

2789 static void
2790 dsl_dataset_promote_sync(void *arg1, void *arg2, dmu_tx_t *tx)
2791 {
2792     dsl_dataset_t *hds = arg1;
2793     struct promotearg *pa = arg2;
2794     struct promotenode *snap = list_head(&pa->shared_snaps);
2795     dsl_dataset_t *origin_ds = snap->ds;
2796     dsl_dataset_t *origin_head;
2797     dsl_dir_t *dd = hds->ds_dir;
2798     dsl_pool_t *dp = hds->ds_dir->dd_pool;
2799     dsl_dir_t *odd = NULL;
2800     uint64_t oldnext_obj;
2801     int64_t delta;

2803     ASSERT(0 == (hds->ds_phys->ds_flags & DS_FLAG_NOPROMOTE));

2805     snap = list_head(&pa->origin_snaps);
2806     origin_head = snap->ds;

2808     /*
2809      * We need to explicitly open odd, since origin_ds's dd will be
2810      * changing.
2811      */
2812     VERIFY(0 == dsl_dir_open_obj(dp, origin_ds->ds_dir->dd_object,
2813         NULL, FTAG, &odd));

2815     /* change origin's next snap */
2816     dmu_buf_will_dirty(origin_ds->ds_dbuf, tx);
2817     oldnext_obj = origin_ds->ds_phys->ds_next_snap_obj;

```

```

2818     snap = list_tail(&pa->clone_snaps);
2819     ASSERT3U(snap->ds->ds_phys->ds_prev_snap_obj, ==, origin_ds->ds_object);
2820     origin_ds->ds_phys->ds_next_snap_obj = snap->ds->ds_object;

2822     /* change the origin's next clone */
2823     if (origin_ds->ds_phys->ds_next_clones_obj) {
2824         remove_from_next_clones(origin_ds, snap->ds->ds_object, tx);
2825         VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
2826             origin_ds->ds_phys->ds_next_clones_obj,
2827             oldnext_obj, tx));
2828     }

2830     /* change origin */
2831     dmu_buf_will_dirty(dd->dd_dbuf, tx);
2832     ASSERT3U(dd->dd_phys->dd_origin_obj, ==, origin_ds->ds_object);
2833     dd->dd_phys->dd_origin_obj = odd->dd_phys->dd_origin_obj;
2834     dd->dd_origin_tngx = origin_head->ds_dir->dd_origin_tngx;
2835     dmu_buf_will_dirty(odd->dd_dbuf, tx);
2836     odd->dd_phys->dd_origin_obj = origin_ds->ds_object;
2837     origin_head->ds_dir->dd_origin_tngx =
2838         origin_ds->ds_phys->ds_creation_tngx;

2840     /* change dd_clone entries */
2841     if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2842         VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
2843             odd->dd_phys->dd_clones, hds->ds_object, tx));
2844         VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
2845             pa->origin_origin->ds_dir->dd_phys->dd_clones,
2846             hds->ds_object, tx));

2848         VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
2849             pa->origin_origin->ds_dir->dd_phys->dd_clones,
2850             origin_head->ds_object, tx));
2851         if (dd->dd_phys->dd_clones == 0) {
2852             dd->dd_phys->dd_clones = zap_create(dp->dp_meta_objset,
2853                 DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
2854         }
2855         VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
2856             dd->dd_phys->dd_clones, origin_head->ds_object, tx));
2858     }

2860     /* move snapshots to this dir */
2861     for (snap = list_head(&pa->shared_snaps); snap;
2862         snap = list_next(&pa->shared_snaps, snap)) {
2863         dsl_dataset_t *ds = snap->ds;

2865         /* unregister props as dsl_dir is changing */
2866         if (ds->ds_objset) {
2867             dmu_objset_evict(ds->ds_objset);
2868             ds->ds_objset = NULL;
2869         }
2870         /* move snap name entry */
2871         VERIFY(0 == dsl_dataset_get_snapname(ds));
2872         VERIFY(0 == dsl_dataset_snap_remove(origin_head,
2873             ds->ds_snapname, tx));
2874         VERIFY(0 == zap_add(dp->dp_meta_objset,
2875             hds->ds_phys->ds_snapnames_zapobj, ds->ds_snapname,
2876             8, 1, &ds->ds_object, tx));

2878         /* change containing dsl_dir */
2879         dmu_buf_will_dirty(ds->ds_dbuf, tx);
2880         ASSERT3U(ds->ds_phys->ds_dir_obj, ==, odd->dd_object);
2881         ds->ds_phys->ds_dir_obj = dd->dd_object;
2882         ASSERT3P(ds->ds_dir, ==, odd);
2883         dsl_dir_close(ds->ds_dir, ds);

```

```

2884     VERIFY(0 == dsl_dir_open_obj(dp, dd->dd_object,
2885     NULL, ds, &ds->ds_dir));

2887     /* move any clone references */
2888     if (ds->ds_phys->ds_next_clones_obj &&
2889     spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2890         zap_cursor_t zc;
2891         zap_attribute_t za;

2893         for (zap_cursor_init(&zc, dp->dp_meta_objset,
2894         ds->ds_phys->ds_next_clones_obj);
2895         zap_cursor_retrieve(&zc, &za) == 0;
2896         zap_cursor_advance(&zc)) {
2897             dsl_dataset_t *cnds;
2898             uint64_t o;

2900             if (za.za_first_integer == oldnext_obj) {
2901                 /*
2902                  * We've already moved the
2903                  * origin's reference.
2904                  */
2905                 continue;
2906             }

2908             VERIFY3U(0, ==, dsl_dataset_hold_obj(dp,
2909             za.za_first_integer, FTAG, &cnds));
2910             o = cnds->ds_dir->dd_phys->dd_head_dataset_obj;

2912             VERIFY3U(zap_remove_int(dp->dp_meta_objset,
2913             odd->dd_phys->dd_clones, o, tx), ==, 0);
2914             VERIFY3U(zap_add_int(dp->dp_meta_objset,
2915             dd->dd_phys->dd_clones, o, tx), ==, 0);
2916             dsl_dataset_rele(cnds, FTAG);
2917         }
2918         zap_cursor_fini(&zc);
2919     }

2921     ASSERT3U(dsl_prop_numcb(ds), ==, 0);
2922 }

2924 /*
2925  * Change space accounting.
2926  * Note, pa->usedsnap and dd_used_breakdown[SNAP] will either
2927  * both be valid, or both be 0 (resulting in delta == 0). This
2928  * is true for each of {clone,origin} independently.
2929  */

2931     delta = pa->cloneusedsnap -
2932     dd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
2933     ASSERT3S(delta, >=, 0);
2934     ASSERT3U(pa->used, >=, delta);
2935     dsl_dir_diduse_space(dd, DD_USED_SNAP, delta, 0, 0, tx);
2936     dsl_dir_diduse_space(dd, DD_USED_HEAD,
2937     pa->used - delta, pa->comp, pa->uncomp, tx);

2939     delta = pa->originusedsnap -
2940     odd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
2941     ASSERT3S(delta, <=, 0);
2942     ASSERT3U(pa->used, >=, -delta);
2943     dsl_dir_diduse_space(odd, DD_USED_SNAP, delta, 0, 0, tx);
2944     dsl_dir_diduse_space(odd, DD_USED_HEAD,
2945     -pa->used - delta, -pa->comp, -pa->uncomp, tx);

2947     origin_ds->ds_phys->ds_unique_bytes = pa->unique;

2949     /* log history record */

```

```

2950     spa_history_log_internal(LOG_DS_PROMOTE, dd->dd_pool->dp_spa, tx,
2951     "dataset = %llu", hds->ds_object);

2953     dsl_dir_close(odd, FTAG);
2954 }

2956 static char *snaplist_tag = "snaplist";
2957 /*
2958  * Make a list of dsl_dataset_t's for the snapshots between first_obj
2959  * (exclusive) and last_obj (inclusive). The list will be in reverse
2960  * order (last_obj will be the list_head()). If first_obj == 0, do all
2961  * snapshots back to this dataset's origin.
2962  */
2963 static int
2964 snaplist_make(dsl_pool_t *dp, boolean_t own,
2965     uint64_t first_obj, uint64_t last_obj, list_t *l)
2966 {
2967     uint64_t obj = last_obj;

2969     ASSERT(RW_LOCK_HELD(&dp->dp_config_rwlock));

2971     list_create(l, sizeof (struct promotenode),
2972     offsetof(struct promotenode, link));

2974     while (obj != first_obj) {
2975         dsl_dataset_t *ds;
2976         struct promotenode *snap;
2977         int err;

2979         if (own) {
2980             err = dsl_dataset_own_obj(dp, obj,
2981             0, snaplist_tag, &ds);
2982             if (err == 0)
2983                 dsl_dataset_make_exclusive(ds, snaplist_tag);
2984         } else {
2985             err = dsl_dataset_hold_obj(dp, obj, snaplist_tag, &ds);
2986         }
2987         if (err == ENOENT) {
2988             /* lost race with snapshot destroy */
2989             struct promotenode *last = list_tail(l);
2990             ASSERT(obj != last->ds->ds_phys->ds_prev_snap_obj);
2991             obj = last->ds->ds_phys->ds_prev_snap_obj;
2992             continue;
2993         } else if (err) {
2994             return (err);
2995         }

2997         if (first_obj == 0)
2998             first_obj = ds->ds_dir->dd_phys->dd_origin_obj;

3000         snap = kmem_alloc(sizeof (struct promotenode), KM_SLEEP);
3001         snap->ds = ds;
3002         list_insert_tail(l, snap);
3003         obj = ds->ds_phys->ds_prev_snap_obj;
3004     }

3006     return (0);
3007 }

3009 static int
3010 snaplist_space(list_t *l, uint64_t mintxg, uint64_t *spacep)
3011 {
3012     struct promotenode *snap;

3014     *spacep = 0;
3015     for (snap = list_head(l); snap; snap = list_next(l, snap)) {

```

```

3016         uint64_t used, comp, uncomp;
3017         dsl_deadlist_space_range(&snap->ds->ds_deadlist,
3018             mintxg, UINIT64_MAX, &used, &comp, &uncomp);
3019         *spacep += used;
3020     }
3021     return (0);
3022 }

3024 static void
3025 snaplist_destroy(list_t *l, boolean_t own)
3026 {
3027     struct promotenode *snap;

3029     if (!l || !list_link_active(&l->list_head))
3030         return;

3032     while ((snap = list_tail(l)) != NULL) {
3033         list_remove(l, snap);
3034         if (own)
3035             dsl_dataset_disown(snap->ds, snaplist_tag);
3036         else
3037             dsl_dataset_rele(snap->ds, snaplist_tag);
3038         kmem_free(snap, sizeof (struct promotenode));
3039     }
3040     list_destroy(l);
3041 }

3043 /*
3044  * Promote a clone.  Nomenclature note:
3045  * "clone" or "cds": the original clone which is being promoted
3046  * "origin" or "ods": the snapshot which is originally clone's origin
3047  * "origin head" or "ohds": the dataset which is the head
3048  * (filesystem/volume) for the origin
3049  * "origin origin": the origin of the origin's filesystem (typically
3050  * NULL, indicating that the clone is not a clone of a clone).
3051  */
3052 int
3053 dsl_dataset_promote(const char *name, char *conflsnap)
3054 {
3055     dsl_dataset_t *ds;
3056     dsl_dir_t *dd;
3057     dsl_pool_t *dp;
3058     dmu_object_info_t doi;
3059     struct promotearg pa = { 0 };
3060     struct promotenode *snap;
3061     int err;

3063     err = dsl_dataset_hold(name, FTAG, &ds);
3064     if (err)
3065         return (err);
3066     dd = ds->ds_dir;
3067     dp = dd->dd_pool;

3069     err = dmu_object_info(dp->dp_meta_objset,
3070         ds->ds_phys->ds_snapnames_zapobj, &doi);
3071     if (err) {
3072         dsl_dataset_rele(ds, FTAG);
3073         return (err);
3074     }

3076     if (dsl_dataset_is_snapshot(ds) || dd->dd_phys->dd_origin_obj == 0) {
3077         dsl_dataset_rele(ds, FTAG);
3078         return (EINVAL);
3079     }

3081     /*

```

```

3082     * We are going to inherit all the snapshots taken before our
3083     * origin (i.e., our new origin will be our parent's origin).
3084     * Take ownership of them so that we can rename them into our
3085     * namespace.
3086     */
3087     rw_enter(&dp->dp_config_rwlock, RW_READER);

3089     err = snaplist_make(dp, B_TRUE, 0, dd->dd_phys->dd_origin_obj,
3090         &pa.shared_snaps);
3091     if (err != 0)
3092         goto out;

3094     err = snaplist_make(dp, B_FALSE, 0, ds->ds_object, &pa.clone_snaps);
3095     if (err != 0)
3096         goto out;

3098     snap = list_head(&pa.shared_snaps);
3099     ASSERT3U(snap->ds->ds_object, ==, dd->dd_phys->dd_origin_obj);
3100     err = snaplist_make(dp, B_FALSE, dd->dd_phys->dd_origin_obj,
3101         snap->ds->ds_dir->dd_phys->dd_head_dataset_obj, &pa.origin_snaps);
3102     if (err != 0)
3103         goto out;

3105     if (snap->ds->ds_dir->dd_phys->dd_origin_obj != 0) {
3106         err = dsl_dataset_hold_obj(dp,
3107             snap->ds->ds_dir->dd_phys->dd_origin_obj,
3108             FTAG, &pa.origin_origin);
3109         if (err != 0)
3110             goto out;
3111     }

3113 out:
3114     rw_exit(&dp->dp_config_rwlock);

3116     /*
3117     * Add in 128x the snapnames zapobj size, since we will be moving
3118     * a bunch of snapnames to the promoted ds, and dirtying their
3119     * bonus buffers.
3120     */
3121     if (err == 0) {
3122         err = dsl_sync_task_do(dp, dsl_dataset_promote_check,
3123             dsl_dataset_promote_sync, ds, &pa,
3124             2 + 2 * doi.doi_physical_blocks_512);
3125         if (err && pa.err_ds && conflsnap)
3126             (void) strncpy(conflsnap, pa.err_ds, MAXNAMELEN);
3127     }

3129     snaplist_destroy(&pa.shared_snaps, B_TRUE);
3130     snaplist_destroy(&pa.clone_snaps, B_FALSE);
3131     snaplist_destroy(&pa.origin_snaps, B_FALSE);
3132     if (pa.origin_origin)
3133         dsl_dataset_rele(pa.origin_origin, FTAG);
3134     dsl_dataset_rele(ds, FTAG);
3135     return (err);
3136 }

3138 struct cloneswaparg {
3139     dsl_dataset_t *cds; /* clone dataset */
3140     dsl_dataset_t *ohds; /* origin's head dataset */
3141     boolean_t force;
3142     int64_t unused_refres_delta; /* change in unconsumed reservation */
3143 };

3145 /* ARGSUSED */
3146 static int
3147 dsl_dataset_clone_swap_check(void *arg1, void *arg2, dmu_tx_t *tx)

```

```

3148 {
3149     struct cloneswaparg *csa = arg1;

3151     /* they should both be heads */
3152     if (dsl_dataset_is_snapshot(csa->cds) ||
3153         dsl_dataset_is_snapshot(csa->ohds))
3154         return (EINVAL);

3156     /* the branch point should be just before them */
3157     if (csa->cds->ds_prev != csa->ohds->ds_prev)
3158         return (EINVAL);

3160     /* cds should be the clone (unless they are unrelated) */
3161     if (csa->cds->ds_prev != NULL &&
3162         csa->cds->ds_prev != csa->cds->ds_dir->dd_pool->dp_origin_snap &&
3163         csa->ohds->ds_object !=
3164         csa->cds->ds_prev->ds_phys->ds_next_snap_obj)
3165         return (EINVAL);

3167     /* the clone should be a child of the origin */
3168     if (csa->cds->ds_dir->dd_parent != csa->ohds->ds_dir)
3169         return (EINVAL);

3171     /* ohds shouldn't be modified unless 'force' */
3172     if (!csa->force && dsl_dataset_modified_since_lastsnap(csa->ohds))
3173         return (ETXTBSY);

3175     /* adjust amount of any unconsumed refreservation */
3176     csa->unused_refres_delta =
3177         (int64_t)MIN(csa->ohds->ds_reserved,
3178             csa->ohds->ds_phys->ds_unique_bytes) -
3179         (int64_t)MIN(csa->ohds->ds_reserved,
3180             csa->cds->ds_phys->ds_unique_bytes);

3182     if (csa->unused_refres_delta > 0 &&
3183         csa->unused_refres_delta >
3184         dsl_dir_space_available(csa->ohds->ds_dir, NULL, 0, TRUE))
3185         return (ENOSPC);

3187     if (csa->ohds->ds_quota != 0 &&
3188         csa->cds->ds_phys->ds_unique_bytes > csa->ohds->ds_quota)
3189         return (EDQUOT);

3191     return (0);
3192 }

3194 /* ARGSUSED */
3195 static void
3196 dsl_dataset_clone_swap_sync(void *arg1, void *arg2, dmu_tx_t *tx)
3197 {
3198     struct cloneswaparg *csa = arg1;
3199     dsl_pool_t *dp = csa->cds->ds_dir->dd_pool;

3201     ASSERT(csa->cds->ds_reserved == 0);
3202     ASSERT(csa->ohds->ds_quota == 0 ||
3203         csa->cds->ds_phys->ds_unique_bytes <= csa->ohds->ds_quota);

3205     dmu_buf_will_dirty(csa->cds->ds_dbuf, tx);
3206     dmu_buf_will_dirty(csa->ohds->ds_dbuf, tx);

3208     if (csa->cds->ds_objset != NULL) {
3209         dmu_objset_evict(csa->cds->ds_objset);
3210         csa->cds->ds_objset = NULL;
3211     }

3213     if (csa->ohds->ds_objset != NULL) {

```

```

3214         dmu_objset_evict(csa->ohds->ds_objset);
3215         csa->ohds->ds_objset = NULL;
3216     }

3218     /*
3219     * Reset origin's unique bytes, if it exists.
3220     */
3221     if (csa->cds->ds_prev) {
3222         dsl_dataset_t *origin = csa->cds->ds_prev;
3223         uint64_t comp, uncomp;

3225         dmu_buf_will_dirty(origin->ds_dbuf, tx);
3226         dsl_deadlist_space_range(&csa->cds->ds_deadlist,
3227             origin->ds_phys->ds_prev_snap_txg, UINT64_MAX,
3228             &origin->ds_phys->ds_unique_bytes, &comp, &uncomp);
3229     }

3231     /* swap blkptrs */
3232     {
3233         blkptr_t tmp;
3234         tmp = csa->ohds->ds_phys->ds_bp;
3235         csa->ohds->ds_phys->ds_bp = csa->cds->ds_phys->ds_bp;
3236         csa->cds->ds_phys->ds_bp = tmp;
3237     }

3239     /* set dd_*_bytes */
3240     {
3241         int64_t dused, dcomp, duncomp;
3242         uint64_t cdl_used, cdl_comp, cdl_uncomp;
3243         uint64_t odl_used, odl_comp, odl_uncomp;

3245         ASSERT3U(csa->cds->ds_dir->dd_phys->
3246             dd_used_breakdown[DD_USED_SNAP], ==, 0);

3248         dsl_deadlist_space(&csa->cds->ds_deadlist,
3249             &cdl_used, &cdl_comp, &cdl_uncomp);
3250         dsl_deadlist_space(&csa->ohds->ds_deadlist,
3251             &odl_used, &odl_comp, &odl_uncomp);

3253         dused = csa->cds->ds_phys->ds_referenced_bytes + cdl_used -
3254             (csa->ohds->ds_phys->ds_referenced_bytes + odl_used);
3255         dcomp = csa->cds->ds_phys->ds_compressed_bytes + cdl_comp -
3256             (csa->ohds->ds_phys->ds_compressed_bytes + odl_comp);
3257         duncomp = csa->cds->ds_phys->ds_uncompressed_bytes +
3258             cdl_uncomp -
3259             (csa->ohds->ds_phys->ds_uncompressed_bytes + odl_uncomp);

3261         dsl_dir_diduse_space(csa->ohds->ds_dir, DD_USED_HEAD,
3262             dused, dcomp, duncomp, tx);
3263         dsl_dir_diduse_space(csa->cds->ds_dir, DD_USED_HEAD,
3264             -dused, -dcomp, -duncomp, tx);

3266     /*
3267     * The difference in the space used by snapshots is the
3268     * difference in snapshot space due to the head's
3269     * deadlist (since that's the only thing that's
3270     * changing that affects the snapshot).
3271     */
3272     dsl_deadlist_space_range(&csa->cds->ds_deadlist,
3273         csa->ohds->ds_dir->dd_origin_txg, UINT64_MAX,
3274         &cdl_used, &cdl_comp, &cdl_uncomp);
3275     dsl_deadlist_space_range(&csa->ohds->ds_deadlist,
3276         csa->ohds->ds_dir->dd_origin_txg, UINT64_MAX,
3277         &odl_used, &odl_comp, &odl_uncomp);
3278     dsl_dir_transfer_space(csa->ohds->ds_dir, cdl_used - odl_used,
3279         DD_USED_HEAD, DD_USED_SNAP, tx);

```

```

3280     }
3282     /* swap ds_*_bytes */
3283     SWITCH64(csa->ohds->ds_phys->ds_referenced_bytes,
3284             csa->cds->ds_phys->ds_referenced_bytes);
3285     SWITCH64(csa->ohds->ds_phys->ds_compressed_bytes,
3286             csa->cds->ds_phys->ds_compressed_bytes);
3287     SWITCH64(csa->ohds->ds_phys->ds_uncompressed_bytes,
3288             csa->cds->ds_phys->ds_uncompressed_bytes);
3289     SWITCH64(csa->ohds->ds_phys->ds_unique_bytes,
3290             csa->cds->ds_phys->ds_unique_bytes);
3292     /* apply any parent delta for change in unconsumed refreservation */
3293     dsl_dir_diduse_space(csa->ohds->ds_dir, DD_USED_REFRSRV,
3294             csa->unused_refres_delta, 0, 0, tx);
3296     /*
3297     * Swap deadlists.
3298     */
3299     dsl_deadlist_close(&csa->cds->ds_deadlist);
3300     dsl_deadlist_close(&csa->ohds->ds_deadlist);
3301     SWITCH64(csa->ohds->ds_phys->ds_deadlist_obj,
3302             csa->cds->ds_phys->ds_deadlist_obj);
3303     dsl_deadlist_open(&csa->cds->ds_deadlist, dp->dp_meta_objset,
3304             csa->cds->ds_phys->ds_deadlist_obj);
3305     dsl_deadlist_open(&csa->ohds->ds_deadlist, dp->dp_meta_objset,
3306             csa->ohds->ds_phys->ds_deadlist_obj);
3308     dsl_scan_ds_clone_swapped(csa->ohds, csa->cds, tx);
3309 }
3311 /*
3312  * Swap 'clone' with its origin head datasets. Used at the end of "zfs
3313  * recv" into an existing fs to swizzle the file system to the new
3314  * version, and by "zfs rollback". Can also be used to swap two
3315  * independent head datasets if neither has any snapshots.
3316  */
3317 int
3318 dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
3319     boolean_t force)
3320 {
3321     struct cloneswaparg csa;
3322     int error;
3324     ASSERT(clone->ds_owner);
3325     ASSERT(origin_head->ds_owner);
3326     retry:
3327     /*
3328     * Need exclusive access for the swap. If we're swapping these
3329     * datasets back after an error, we already hold the locks.
3330     */
3331     if (!RW_WRITE_HELD(&clone->ds_rwlock))
3332         rw_enter(&clone->ds_rwlock, RW_WRITER);
3333     if (!RW_WRITE_HELD(&origin_head->ds_rwlock) &&
3334         !rw_tryenter(&origin_head->ds_rwlock, RW_WRITER)) {
3335         rw_exit(&clone->ds_rwlock);
3336         rw_enter(&origin_head->ds_rwlock, RW_WRITER);
3337         if (!rw_tryenter(&clone->ds_rwlock, RW_WRITER)) {
3338             rw_exit(&origin_head->ds_rwlock);
3339             goto retry;
3340         }
3341     }
3342     csa.cds = clone;
3343     csa.ohds = origin_head;
3344     csa.force = force;
3345     error = dsl_sync_task_do(clone->ds_dir->dd_pool,

```

```

3346         dsl_dataset_clone_swap_check,
3347         dsl_dataset_clone_swap_sync, &csa, NULL, 9);
3348     return (error);
3349 }
3351 /*
3352  * Given a pool name and a dataset object number in that pool,
3353  * return the name of that dataset.
3354  */
3355 int
3356 dsl_dsoobj_to_dsname(char *pname, uint64_t obj, char *buf)
3357 {
3358     spa_t *spa;
3359     dsl_pool_t *dp;
3360     dsl_dataset_t *ds;
3361     int error;
3363     if ((error = spa_open(pname, &spa, FTAG)) != 0)
3364         return (error);
3365     dp = spa_get_dsl(spa);
3366     rw_enter(&dp->dp_config_rwlock, RW_READER);
3367     if ((error = dsl_dataset_hold_obj(dp, obj, FTAG, &ds)) == 0) {
3368         dsl_dataset_name(ds, buf);
3369         dsl_dataset_rele(ds, FTAG);
3370     }
3371     rw_exit(&dp->dp_config_rwlock);
3372     spa_close(spa, FTAG);
3374     return (error);
3375 }
3377 int
3378 dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
3379     uint64_t asize, uint64_t inflight, uint64_t *used, uint64_t *ref_rsrv)
3380 {
3381     int error = 0;
3383     ASSERT3S(asize, >, 0);
3385     /*
3386     * *ref_rsrv is the portion of asize that will come from any
3387     * unconsumed refreservation space.
3388     */
3389     *ref_rsrv = 0;
3391     mutex_enter(&ds->ds_lock);
3392     /*
3393     * Make a space adjustment for reserved bytes.
3394     */
3395     if (ds->ds_reserved > ds->ds_phys->ds_unique_bytes) {
3396         ASSERT3U(*used, >=,
3397             ds->ds_reserved - ds->ds_phys->ds_unique_bytes);
3398         *used -= (ds->ds_reserved - ds->ds_phys->ds_unique_bytes);
3399         *ref_rsrv =
3400             asize - MIN(asize, parent_delta(ds, asize + inflight));
3401     }
3403     if (!check_quota || ds->ds_quota == 0) {
3404         mutex_exit(&ds->ds_lock);
3405         return (0);
3406     }
3407     /*
3408     * If they are requesting more space, and our current estimate
3409     * is over quota, they get to try again unless the actual
3410     * on-disk is over quota and there are no pending changes (which
3411     * may free up space for us).

```

```

3412     */
3413     if (ds->ds_phys->ds_referenced_bytes + inflight >= ds->ds_quota) {
3414         if (inflight > 0 ||
3415             ds->ds_phys->ds_referenced_bytes < ds->ds_quota)
3416             error = ERESTART;
3417         else
3418             error = EDQUOT;
3419     }
3420     mutex_exit(&ds->ds_lock);
3422     return (error);
3423 }

3425 /* ARGSUSED */
3426 static int
3427 dsl_dataset_set_quota_check(void *arg1, void *arg2, dmu_tx_t *tx)
3428 {
3429     dsl_dataset_t *ds = arg1;
3430     dsl_prop_setarg_t *psa = arg2;
3431     int err;

3433     if (spa_version(ds->ds_dir->dd_pool->dp_spa) < SPA_VERSION_REFQUOTA)
3434         return (ENOTSUP);

3436     if ((err = dsl_prop_predict_sync(ds->ds_dir, psa)) != 0)
3437         return (err);

3439     if (psa->psa_effective_value == 0)
3440         return (0);

3442     if (psa->psa_effective_value < ds->ds_phys->ds_referenced_bytes ||
3443         psa->psa_effective_value < ds->ds_reserved)
3444         return (ENOSPC);

3446     return (0);
3447 }

3449 extern void dsl_prop_set_sync(void *, void *, dmu_tx_t *);

3451 void
3452 dsl_dataset_set_quota_sync(void *arg1, void *arg2, dmu_tx_t *tx)
3453 {
3454     dsl_dataset_t *ds = arg1;
3455     dsl_prop_setarg_t *psa = arg2;
3456     uint64_t effective_value = psa->psa_effective_value;

3458     dsl_prop_set_sync(ds, psa, tx);
3459     DSL_PROP_CHECK_PREDICTION(ds->ds_dir, psa);

3461     if (ds->ds_quota != effective_value) {
3462         dmu_buf_will_dirty(ds->ds_dbuf, tx);
3463         ds->ds_quota = effective_value;

3465         spa_history_log_internal(LOG_DS_REFQUOTA,
3466             ds->ds_dir->dd_pool->dp_spa, tx, "%lld dataset = %llu ",
3467             (longlong_t)ds->ds_quota, ds->ds_object);
3468     }
3469 }

3471 int
3472 dsl_dataset_set_quota(const char *dsname, zprop_source_t source, uint64_t quota)
3473 {
3474     dsl_dataset_t *ds;
3475     dsl_prop_setarg_t psa;
3476     int err;

```

```

3478     dsl_prop_setarg_init_uint64(&psa, "refquota", source, &quota);

3480     err = dsl_dataset_hold(dsname, FTAG, &ds);
3481     if (err)
3482         return (err);

3484     /*
3485      * If someone removes a file, then tries to set the quota, we
3486      * want to make sure the file freeing takes effect.
3487      */
3488     txg_wait_open(ds->ds_dir->dd_pool, 0);

3490     err = dsl_sync_task_do(ds->ds_dir->dd_pool,
3491         dsl_dataset_set_quota_check, dsl_dataset_set_quota_sync,
3492         ds, &psa, 0);

3494     dsl_dataset_rele(ds, FTAG);
3495     return (err);
3496 }

3498 static int
3499 dsl_dataset_set_reservation_check(void *arg1, void *arg2, dmu_tx_t *tx)
3500 {
3501     dsl_dataset_t *ds = arg1;
3502     dsl_prop_setarg_t *psa = arg2;
3503     uint64_t effective_value;
3504     uint64_t unique;
3505     int err;

3507     if (spa_version(ds->ds_dir->dd_pool->dp_spa) <
3508         SPA_VERSION_REFRESERVATION)
3509         return (ENOTSUP);

3511     if (dsl_dataset_is_snapshot(ds))
3512         return (EINVAL);

3514     if ((err = dsl_prop_predict_sync(ds->ds_dir, psa)) != 0)
3515         return (err);

3517     effective_value = psa->psa_effective_value;

3519     /*
3520      * If we are doing the preliminary check in open context, the
3521      * space estimates may be inaccurate.
3522      */
3523     if (!dmu_tx_is_syncing(tx))
3524         return (0);

3526     mutex_enter(&ds->ds_lock);
3527     if (!DS_UNIQUE_IS_ACCURATE(ds))
3528         dsl_dataset_recalc_head_uniq(ds);
3529     unique = ds->ds_phys->ds_unique_bytes;
3530     mutex_exit(&ds->ds_lock);

3532     if (MAX(unique, effective_value) > MAX(unique, ds->ds_reserved)) {
3533         uint64_t delta = MAX(unique, effective_value) -
3534             MAX(unique, ds->ds_reserved);

3536         if (delta > dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE))
3537             return (ENOSPC);
3538         if (ds->ds_quota > 0 &&
3539             effective_value > ds->ds_quota)
3540             return (ENOSPC);
3541     }

3543     return (0);

```

```

3544 }

3546 static void
3547 dsl_dataset_set_reservation_sync(void *arg1, void *arg2, dmu_tx_t *tx)
3548 {
3549     dsl_dataset_t *ds = arg1;
3550     dsl_prop_setarg_t *psa = arg2;
3551     uint64_t effective_value = psa->psa_effective_value;
3552     uint64_t unique;
3553     int64_t delta;

3555     dsl_prop_set_sync(ds, psa, tx);
3556     DSL_PROP_CHECK_PREDICTION(ds->ds_dir, psa);

3558     dmu_buf_will_dirty(ds->ds_dbuf, tx);

3560     mutex_enter(&ds->ds_dir->dd_lock);
3561     mutex_enter(&ds->ds_lock);
3562     ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
3563     unique = ds->ds_phys->ds_unique_bytes;
3564     delta = MAX(0, (int64_t)(effective_value - unique)) -
3565             MAX(0, (int64_t)(ds->ds_reserved - unique));
3566     ds->ds_reserved = effective_value;
3567     mutex_exit(&ds->ds_lock);

3569     dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV, delta, 0, 0, tx);
3570     mutex_exit(&ds->ds_dir->dd_lock);

3572     spa_history_log_internal(LOG_DS_REFRESERV,
3573         ds->ds_dir->dd_pool->dp_spa, tx, "%lld dataset = %llu",
3574         (longlong_t)effective_value, ds->ds_object);
3575 }

3577 int
3578 dsl_dataset_set_reservation(const char *dsname, zprop_source_t source,
3579     uint64_t reservation)
3580 {
3581     dsl_dataset_t *ds;
3582     dsl_prop_setarg_t psa;
3583     int err;

3585     dsl_prop_setarg_init_uint64(&psa, "refreservation", source,
3586         &reservation);

3588     err = dsl_dataset_hold(dsname, FTAG, &ds);
3589     if (err)
3590         return (err);

3592     err = dsl_sync_task_do(ds->ds_dir->dd_pool,
3593         dsl_dataset_set_reservation_check,
3594         dsl_dataset_set_reservation_sync, ds, &psa, 0);

3596     dsl_dataset_rele(ds, FTAG);
3597     return (err);
3598 }

3600 typedef struct zfs_hold_cleanup_arg {
3601     dsl_pool_t *dp;
3602     uint64_t dsobj;
3603     char htag[MAXNAMELEN];
3604 } zfs_hold_cleanup_arg_t;

3606 static void
3607 dsl_dataset_user_release_onexit(void *arg)
3608 {
3609     zfs_hold_cleanup_arg_t *ca = arg;

```

```

3611     (void) dsl_dataset_user_release_tmp(ca->dp, ca->dsobj, ca->htag,
3612         B_TRUE);
3613     kmem_free(ca, sizeof (zfs_hold_cleanup_arg_t));
3614 }

3616 void
3617 dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
3618     minor_t minor)
3619 {
3620     zfs_hold_cleanup_arg_t *ca;

3622     ca = kmem_alloc(sizeof (zfs_hold_cleanup_arg_t), KM_SLEEP);
3623     ca->dp = ds->ds_dir->dd_pool;
3624     ca->dsobj = ds->ds_object;
3625     (void) strncpy(ca->htag, htag, sizeof (ca->htag));
3626     VERIFY3U(0, ==, zfs_onexit_add_cb(minor,
3627         dsl_dataset_user_release_onexit, ca, NULL));
3628 }

3630 /*
3631  * If you add new checks here, you may need to add
3632  * additional checks to the "temporary" case in
3633  * snapshot_check() in dmu_objset.c.
3634  */
3635 static int
3636 dsl_dataset_user_hold_check(void *arg1, void *arg2, dmu_tx_t *tx)
3637 {
3638     dsl_dataset_t *ds = arg1;
3639     struct dsl_ds_holdarg *ha = arg2;
3640     char *htag = ha->htag;
3641     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
3642     int error = 0;

3644     if (spa_version(ds->ds_dir->dd_pool->dp_spa) < SPA_VERSION_USERREFS)
3645         return (ENOTSUP);

3647     if (!dsl_dataset_is_snapshot(ds))
3648         return (EINVAL);

3650     /* tags must be unique */
3651     mutex_enter(&ds->ds_lock);
3652     if (ds->ds_phys->ds_userrefs_obj) {
3653         error = zap_lookup(mos, ds->ds_phys->ds_userrefs_obj, htag,
3654             8, 1, tx);
3655         if (error == 0)
3656             error = EEXIST;
3657         else if (error == ENOENT)
3658             error = 0;
3659     }
3660     mutex_exit(&ds->ds_lock);

3662     if (error == 0 && ha->temphold &&
3663         strlen(htag) + MAX_TAG_PREFIX_LEN >= MAXNAMELEN)
3664         error = E2BIG;

3666     return (error);
3667 }

3669 void
3670 dsl_dataset_user_hold_sync(void *arg1, void *arg2, dmu_tx_t *tx)
3671 {
3672     dsl_dataset_t *ds = arg1;
3673     struct dsl_ds_holdarg *ha = arg2;
3674     char *htag = ha->htag;
3675     dsl_pool_t *dp = ds->ds_dir->dd_pool;

```

```

3676     objset_t *mos = dp->dp_meta_objset;
3677     uint64_t now = gethrestime_sec();
3678     uint64_t zapobj;

3680     mutex_enter(&ds->ds_lock);
3681     if (ds->ds_phys->ds_userrefs_obj == 0) {
3682         /*
3683          * This is the first user hold for this dataset. Create
3684          * the userrefs zap object.
3685          */
3686         dmu_buf_will_dirty(ds->ds_dbuf, tx);
3687         zapobj = ds->ds_phys->ds_userrefs_obj =
3688             zap_create(mos, DMU_OT_USERREFS, DMU_OT_NONE, 0, tx);
3689     } else {
3690         zapobj = ds->ds_phys->ds_userrefs_obj;
3691     }
3692     ds->ds_userrefs++;
3693     mutex_exit(&ds->ds_lock);

3695     VERIFY(0 == zap_add(mos, zapobj, htag, 8, 1, &now, tx));

3697     if (ha->temphold) {
3698         VERIFY(0 == dsl_pool_user_hold(dp, ds->ds_object,
3699             htag, &now, tx));
3700     }

3702     spa_history_log_internal(LOG_DS_USER_HOLD,
3703         dp->dp_spa, tx, "<%s> temp = %d dataset = %llu", htag,
3704         (int)ha->temphold, ds->ds_object);
3705 }

3707 static int
3708 dsl_dataset_user_hold_one(const char *dsname, void *arg)
3709 {
3710     struct dsl_ds_holdarg *ha = arg;
3711     dsl_dataset_t *ds;
3712     int error;
3713     char *name;

3715     /* alloc a buffer to hold dsname@snapname plus terminating NULL */
3716     name = kmem_asprintf("%s@%s", dsname, ha->snapname);
3717     error = dsl_dataset_hold(name, ha->dstg, &ds);
3718     strfree(name);
3719     if (error == 0) {
3720         ha->gotone = B_TRUE;
3721         dsl_sync_task_create(ha->dstg, dsl_dataset_user_hold_check,
3722             dsl_dataset_user_hold_sync, ds, ha, 0);
3723     } else if (error == ENOENT && ha->recursive) {
3724         error = 0;
3725     } else {
3726         (void) strncpy(ha->failed, dsname, sizeof (ha->failed));
3727     }
3728     return (error);
3729 }

3731 int
3732 dsl_dataset_user_hold_for_send(dsl_dataset_t *ds, char *htag,
3733     boolean_t temphold)
3734 {
3735     struct dsl_ds_holdarg *ha;
3736     int error;

3738     ha = kmem_zalloc(sizeof (struct dsl_ds_holdarg), KM_SLEEP);
3739     ha->htag = htag;
3740     ha->temphold = temphold;
3741     error = dsl_sync_task_do(ds->ds_dir->dd_pool,

```

```

3742         dsl_dataset_user_hold_check, dsl_dataset_user_hold_sync,
3743         ds, ha, 0);
3744     kmem_free(ha, sizeof (struct dsl_ds_holdarg));

3746     return (error);
3747 }

3749 int
3750 dsl_dataset_user_hold(char *dsname, char *snapname, char *htag,
3751     boolean_t recursive, boolean_t temphold, int cleanup_fd)
3752 {
3753     struct dsl_ds_holdarg *ha;
3754     dsl_sync_task_t *dst;
3755     spa_t *spa;
3756     int error;
3757     minor_t minor = 0;

3759     if (cleanup_fd != -1) {
3760         /* Currently we only support cleanup-on-exit of tempholds. */
3761         if (!temphold)
3762             return (EINVAL);
3763         error = zfs_onexit_fd_hold(cleanup_fd, &minor);
3764         if (error)
3765             return (error);
3766     }

3768     ha = kmem_zalloc(sizeof (struct dsl_ds_holdarg), KM_SLEEP);
3770     (void) strncpy(ha->failed, dsname, sizeof (ha->failed));

3772     error = spa_open(dsname, &spa, FTAG);
3773     if (error) {
3774         kmem_free(ha, sizeof (struct dsl_ds_holdarg));
3775         if (cleanup_fd != -1)
3776             zfs_onexit_fd_rele(cleanup_fd);
3777         return (error);
3778     }

3780     ha->dstg = dsl_sync_task_group_create(spa_get_dsl(spa));
3781     ha->htag = htag;
3782     ha->snapname = snapname;
3783     ha->recursive = recursive;
3784     ha->temphold = temphold;

3786     if (recursive) {
3787         error = dmu_objset_find(dsname, dsl_dataset_user_hold_one,
3788             ha, DS_FIND_CHILDREN);
3789     } else {
3790         error = dsl_dataset_user_hold_one(dsname, ha);
3791     }
3792     if (error == 0)
3793         error = dsl_sync_task_group_wait(ha->dstg);

3795     for (dst = list_head(&ha->dstg->dstg_tasks); dst;
3796         dst = list_next(&ha->dstg->dstg_tasks, dst)) {
3797         dsl_dataset_t *ds = dst->dst_arg1;

3799         if (dst->dst_err) {
3800             dsl_dataset_name(ds, ha->failed);
3801             *strchr(ha->failed, '@') = '\0';
3802         } else if (error == 0 && minor != 0 && temphold) {
3803             /*
3804              * If this hold is to be released upon process exit,
3805              * register that action now.
3806              */
3807             dsl_register_onexit_hold_cleanup(ds, htag, minor);

```

```

3808     }
3809     dsl_dataset_rele(ds, ha->dstg);
3810 }
3811
3812 if (error == 0 && recursive && !ha->gotone)
3813     error = ENOENT;
3814
3815 if (error)
3816     (void) strcpy(dsname, ha->failed, sizeof (ha->failed));
3817
3818 dsl_sync_task_group_destroy(ha->dstg);
3819
3820 kmem_free(ha, sizeof (struct dsl_ds_holdarg));
3821 spa_close(spa, FTAG);
3822 if (cleanup_fd != -1)
3823     zfs_onexit_fd_rele(cleanup_fd);
3824 return (error);
3825 }
3826
3827 struct dsl_ds_releasearg {
3828     dsl_dataset_t *ds;
3829     const char *htag;
3830     boolean_t own;          /* do we own or just hold ds? */
3831 };
3832
3833 static int
3834 dsl_dataset_release_might_destroy(dsl_dataset_t *ds, const char *htag,
3835     boolean_t *might_destroy)
3836 {
3837     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
3838     uint64_t zapobj;
3839     uint64_t tmp;
3840     int error;
3841
3842     *might_destroy = B_FALSE;
3843
3844     mutex_enter(&ds->ds_lock);
3845     zapobj = ds->ds_phys->ds_userrefs_obj;
3846     if (zapobj == 0) {
3847         /* The tag can't possibly exist */
3848         mutex_exit(&ds->ds_lock);
3849         return (ESRCH);
3850     }
3851
3852     /* Make sure the tag exists */
3853     error = zap_lookup(mos, zapobj, htag, 8, 1, &tmp);
3854     if (error) {
3855         mutex_exit(&ds->ds_lock);
3856         if (error == ENOENT)
3857             error = ESRCH;
3858         return (error);
3859     }
3860
3861     if (ds->ds_userrefs == 1 && ds->ds_phys->ds_num_children == 1 &&
3862         DS_IS_DEFER_DESTROY(ds))
3863         *might_destroy = B_TRUE;
3864
3865     mutex_exit(&ds->ds_lock);
3866     return (0);
3867 }
3868
3869 static int
3870 dsl_dataset_user_release_check(void *arg1, void *tag, dmu_tx_t *tx)
3871 {
3872     struct dsl_ds_releasearg *ra = arg1;
3873     dsl_dataset_t *ds = ra->ds;

```

```

3874     boolean_t might_destroy;
3875     int error;
3876
3877     if (spa_version(ds->ds_dir->dd_pool->dp_spa) < SPA_VERSION_USERREFS)
3878         return (ENOTSUP);
3879
3880     error = dsl_dataset_release_might_destroy(ds, ra->htag, &might_destroy);
3881     if (error)
3882         return (error);
3883
3884     if (might_destroy) {
3885         struct dsl_ds_destroyarg dsda = {0};
3886
3887         if (dmu_tx_is_syncing(tx)) {
3888             /*
3889              * If we're not prepared to remove the snapshot,
3890              * we can't allow the release to happen right now.
3891              */
3892             if (!ra->own)
3893                 return (EBUSY);
3894         }
3895         dsda.ds = ds;
3896         dsda.releasing = B_TRUE;
3897         return (dsl_dataset_destroy_check(&dsda, tag, tx));
3898     }
3899
3900     return (0);
3901 }
3902
3903 static void
3904 dsl_dataset_user_release_sync(void *arg1, void *tag, dmu_tx_t *tx)
3905 {
3906     struct dsl_ds_releasearg *ra = arg1;
3907     dsl_dataset_t *ds = ra->ds;
3908     dsl_pool_t *dp = ds->ds_dir->dd_pool;
3909     objset_t *mos = dp->dp_meta_objset;
3910     uint64_t zapobj;
3911     uint64_t dsobj = ds->ds_object;
3912     uint64_t refs;
3913     int error;
3914
3915     mutex_enter(&ds->ds_lock);
3916     ds->ds_userrefs--;
3917     refs = ds->ds_userrefs;
3918     mutex_exit(&ds->ds_lock);
3919     error = dsl_pool_user_release(dp, ds->ds_object, ra->htag, tx);
3920     VERIFY(error == 0 || error == ENOENT);
3921     zapobj = ds->ds_phys->ds_userrefs_obj;
3922     VERIFY(0 == zap_remove(mos, zapobj, ra->htag, tx));
3923     if (ds->ds_userrefs == 0 && ds->ds_phys->ds_num_children == 1 &&
3924         DS_IS_DEFER_DESTROY(ds)) {
3925         struct dsl_ds_destroyarg dsda = {0};
3926
3927         ASSERT(ra->own);
3928         dsda.ds = ds;
3929         dsda.releasing = B_TRUE;
3930         /* We already did the destroy check */
3931         dsl_dataset_destroy_sync(&dsda, tag, tx);
3932     }
3933
3934     spa_history_log_internal(LOG_DS_USER_RELEASE,
3935         dp->dp_spa, tx, "<%s> %lld dataset = %llu",
3936         ra->htag, (longlong_t)refs, dsobj);
3937 }
3938
3939 static int

```

```

3940 dsl_dataset_user_release_one(const char *dsname, void *arg)
3941 {
3942     struct dsl_ds_holdarg *ha = arg;
3943     struct dsl_ds_releasearg *ra;
3944     dsl_dataset_t *ds;
3945     int error;
3946     void *dtag = ha->dtag;
3947     char *name;
3948     boolean_t own = B_FALSE;
3949     boolean_t might_destroy;

3951     /* alloc a buffer to hold dsname@snapname, plus the terminating NULL */
3952     name = kmem_asprintf("%s@%s", dsname, ha->snapname);
3953     error = dsl_dataset_hold(name, dtag, &ds);
3954     strfree(name);
3955     if (error == ENOENT && ha->recursive)
3956         return (0);
3957     (void) strncpy(ha->failed, dsname, sizeof(ha->failed));
3958     if (error)
3959         return (error);

3961     ha->gotone = B_TRUE;

3963     ASSERT(dsl_dataset_is_snapshot(ds));

3965     error = dsl_dataset_release_might_destroy(ds, ha->htag, &might_destroy);
3966     if (error) {
3967         dsl_dataset_rele(ds, dtag);
3968         return (error);
3969     }

3971     if (might_destroy) {
3972 #ifdef _KERNEL
3973         name = kmem_asprintf("%s@%s", dsname, ha->snapname);
3974         error = zfs_unmount_snap(name, NULL);
3975         strfree(name);
3976         if (error) {
3977             dsl_dataset_rele(ds, dtag);
3978             return (error);
3979         }
3980 #endif
3981         if (!dsl_dataset_tryown(ds, B_TRUE, dtag)) {
3982             dsl_dataset_rele(ds, dtag);
3983             return (EBUSY);
3984         } else {
3985             own = B_TRUE;
3986             dsl_dataset_make_exclusive(ds, dtag);
3987         }
3988     }

3990     ra = kmem_alloc(sizeof(struct dsl_ds_releasearg), KM_SLEEP);
3991     ra->ds = ds;
3992     ra->htag = ha->htag;
3993     ra->own = own;
3994     dsl_sync_task_create(ha->dtag, dsl_dataset_user_release_check,
3995         dsl_dataset_user_release_sync, ra, dtag, 0);

3997     return (0);
3998 }

4000 int
4001 dsl_dataset_user_release(char *dsname, char *snapname, char *htag,
4002     boolean_t recursive)
4003 {
4004     struct dsl_ds_holdarg *ha;
4005     dsl_sync_task_t *dst;

```

```

4006     spa_t *spa;
4007     int error;

4009 top:
4010     ha = kmem_zalloc(sizeof(struct dsl_ds_holdarg), KM_SLEEP);

4012     (void) strncpy(ha->failed, dsname, sizeof(ha->failed));

4014     error = spa_open(dsname, &spa, FTAG);
4015     if (error) {
4016         kmem_free(ha, sizeof(struct dsl_ds_holdarg));
4017         return (error);
4018     }

4020     ha->dtag = dsl_sync_task_group_create(spa_get_dsl(spa));
4021     ha->htag = htag;
4022     ha->snapname = snapname;
4023     ha->recursive = recursive;
4024     if (recursive) {
4025         error = dmub_objset_find(dsname, dsl_dataset_user_release_one,
4026             ha, DS_FIND_CHILDREN);
4027     } else {
4028         error = dsl_dataset_user_release_one(dsname, ha);
4029     }
4030     if (error == 0)
4031         error = dsl_sync_task_group_wait(ha->dtag);

4033     for (dst = list_head(&ha->dtag->dtag_tasks); dst;
4034         dst = list_next(&ha->dtag->dtag_tasks, dst)) {
4035         struct dsl_ds_releasearg *ra = dst->dst_arg1;
4036         dsl_dataset_t *ds = ra->ds;

4038         if (dst->dst_err)
4039             dsl_dataset_name(ds, ha->failed);

4041         if (ra->own)
4042             dsl_dataset_disown(ds, ha->dtag);
4043         else
4044             dsl_dataset_rele(ds, ha->dtag);

4046         kmem_free(ra, sizeof(struct dsl_ds_releasearg));
4047     }

4049     if (error == 0 && recursive && !ha->gotone)
4050         error = ENOENT;

4052     if (error && error != EBUSY)
4053         (void) strncpy(dsname, ha->failed, sizeof(ha->failed));

4055     dsl_sync_task_group_destroy(ha->dtag);
4056     kmem_free(ha, sizeof(struct dsl_ds_holdarg));
4057     spa_close(spa, FTAG);

4059     /*
4060      * We can get EBUSY if we were racing with deferred destroy and
4061      * dsl_dataset_user_release_check() hadn't done the necessary
4062      * open context setup. We can also get EBUSY if we're racing
4063      * with destroy and that thread is the ds_owner. Either way
4064      * the busy condition should be transient, and we should retry
4065      * the release operation.
4066      */
4067     if (error == EBUSY)
4068         goto top;

4070     return (error);
4071 }

```

```

4073 /*
4074  * Called at spa_load time (with retry == B_FALSE) to release a stale
4075  * temporary user hold. Also called by the onexit code (with retry == B_TRUE).
4076  */
4077 int
4078 dsl_dataset_user_release_tmp(dsl_pool_t *dp, uint64_t dsobj, char *htag,
4079     boolean_t retry)
4080 {
4081     dsl_dataset_t *ds;
4082     char *snap;
4083     char *name;
4084     int namelen;
4085     int error;
4086
4087     do {
4088         rw_enter(&dp->dp_config_rwlock, RW_READER);
4089         error = dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds);
4090         rw_exit(&dp->dp_config_rwlock);
4091         if (error)
4092             return (error);
4093         namelen = dsl_dataset_namelen(ds)+1;
4094         name = kmem_alloc(namelen, KM_SLEEP);
4095         dsl_dataset_name(ds, name);
4096         dsl_dataset_rele(ds, FTAG);
4097
4098         snap = strchr(name, '@');
4099         *snap = '\0';
4100         ++snap;
4101         error = dsl_dataset_user_release(name, snap, htag, B_FALSE);
4102         kmem_free(name, namelen);
4103
4104         /*
4105          * The object can't have been destroyed because we have a hold,
4106          * but it might have been renamed, resulting in ENOENT. Retry
4107          * if we've been requested to do so.
4108          *
4109          * It would be nice if we could use the dsobj all the way
4110          * through and avoid ENOENT entirely. But we might need to
4111          * unmount the snapshot, and there's currently no way to lookup
4112          * a vfsp using a ZFS object id.
4113          */
4114     } while ((error == ENOENT) && retry);
4115
4116     return (error);
4117 }
4118
4119 int
4120 dsl_dataset_get_holds(const char *dsname, nvlist_t **nvp)
4121 {
4122     dsl_dataset_t *ds;
4123     int err;
4124
4125     err = dsl_dataset_hold(dsname, FTAG, &ds);
4126     if (err)
4127         return (err);
4128
4129     VERIFY(0 == nvlist_alloc(nvp, NV_UNIQUE_NAME, KM_SLEEP));
4130     if (ds->ds_phys->ds_userrefs_obj != 0) {
4131         zap_attribute_t *za;
4132         zap_cursor_t zc;
4133
4134         za = kmem_alloc(sizeof(zap_attribute_t), KM_SLEEP);
4135         for (zap_cursor_init(&zc, ds->ds_dir->dd_pool->dp_meta_objset,
4136             ds->ds_phys->ds_userrefs_obj);
4137             zap_cursor_retrieve(&zc, za) == 0;

```

```

4138         zap_cursor_advance(&zc)) {
4139             VERIFY(0 == nvlist_add_uint64(*nvp, za->za_name,
4140                 za->za_first_integer));
4141         }
4142         zap_cursor_fini(&zc);
4143         kmem_free(za, sizeof(zap_attribute_t));
4144     }
4145     dsl_dataset_rele(ds, FTAG);
4146     return (0);
4147 }
4148
4149 /*
4150  * Note, this function is used as the callback for dmu_objset_find(). We
4151  * always return 0 so that we will continue to find and process
4152  * inconsistent datasets, even if we encounter an error trying to
4153  * process one of them.
4154  */
4155 /* ARGSUSED */
4156 int
4157 dsl_destroy_inconsistent(const char *dsname, void *arg)
4158 {
4159     dsl_dataset_t *ds;
4160
4161     if (dsl_dataset_own(dsname, B_TRUE, FTAG, &ds) == 0) {
4162         if (DS_IS_INCONSISTENT(ds))
4163             (void) dsl_dataset_destroy(ds, FTAG, B_FALSE);
4164         else
4165             dsl_dataset_disown(ds, FTAG);
4166     }
4167     return (0);
4168 }
4169
4170 /*
4171  * Return (in *usedp) the amount of space written in new that is not
4172  * present in oldsnap. New may be a snapshot or the head. Old must be
4173  * a snapshot before new, in new's filesystem (or its origin). If not then
4174  * fail and return EINVAL.
4175  *
4176  * The written space is calculated by considering two components: First, we
4177  * ignore any freed space, and calculate the written as new's used space
4178  * minus old's used space. Next, we add in the amount of space that was freed
4179  * between the two snapshots, thus reducing new's used space relative to old's.
4180  * Specifically, this is the space that was born before old->ds_creation_txg,
4181  * and freed before new (ie. on new's deadlist or a previous deadlist).
4182  *
4183  * space freed          [-----]
4184  * snapshots          ---0-----0-----0-----
4185  *                    oldsnap                new
4186  */
4187 int
4188 dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
4189     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
4190 {
4191     int err = 0;
4192     uint64_t snapobj;
4193     dsl_pool_t *dp = new->ds_dir->dd_pool;
4194
4195     *usedp = 0;
4196     *usedp += new->ds_phys->ds_referenced_bytes;
4197     *usedp -= oldsnap->ds_phys->ds_referenced_bytes;
4198
4199     *compp = 0;
4200     *compp += new->ds_phys->ds_compressed_bytes;
4201     *compp -= oldsnap->ds_phys->ds_compressed_bytes;
4202
4203     *uncompp = 0;

```

```

4204 *uncompp += new->ds_phys->ds_uncompressed_bytes;
4205 *uncompp -= oldsnap->ds_phys->ds_uncompressed_bytes;

4207 rw_enter(&dp->dp_config_rwlock, RW_READER);
4208 snapobj = new->ds_object;
4209 while (snapobj != oldsnap->ds_object) {
4210     dsl_dataset_t *snap;
4211     uint64_t used, comp, uncompp;

4213     if (snapobj == new->ds_object) {
4214         snap = new;
4215     } else {
4216         err = dsl_dataset_hold_obj(dp, snapobj, FTAG, &snap);
4217         if (err != 0)
4218             break;
4219     }

4221     if (snap->ds_phys->ds_prev_snap_txg ==
4222         oldsnap->ds_phys->ds_creation_txg) {
4223         /*
4224          * The blocks in the deadlist can not be born after
4225          * ds_prev_snap_txg, so get the whole deadlist space,
4226          * which is more efficient (especially for old-format
4227          * deadlists). Unfortunately the deadlist code
4228          * doesn't have enough information to make this
4229          * optimization itself.
4230          */
4231         dsl_deadlist_space(&snap->ds_deadlist,
4232             &used, &comp, &uncompp);
4233     } else {
4234         dsl_deadlist_space_range(&snap->ds_deadlist,
4235             0, oldsnap->ds_phys->ds_creation_txg,
4236             &used, &comp, &uncompp);
4237     }
4238     *usedp += used;
4239     *compp += comp;
4240     *uncompp += uncompp;

4242     /*
4243      * If we get to the beginning of the chain of snapshots
4244      * (ds_prev_snap_obj == 0) before oldsnap, then oldsnap
4245      * was not a snapshot of/before new.
4246      */
4247     snapobj = snap->ds_phys->ds_prev_snap_obj;
4248     if (snap != new)
4249         dsl_dataset_rele(snap, FTAG);
4250     if (snapobj == 0) {
4251         err = EINVAL;
4252         break;
4253     }
4255 }
4256 rw_exit(&dp->dp_config_rwlock);
4257 return (err);
4258 }

4260 /*
4261 * Return (in *usedp) the amount of space that will be reclaimed if firstsnap,
4262 * lastsnap, and all snapshots in between are deleted.
4263 *
4264 * blocks that would be freed          [-----]
4265 * snapshots          ---O-----O-----O-----O-----
4266 *                    firstsnap      lastsnap
4267 *
4268 * This is the set of blocks that were born after the snap before firstsnap,
4269 * (birth > firstsnap->prev_snap_txg) and died before the snap after the

```

```

4270 * last snap (ie, is on lastsnap->ds_next->ds_deadlist or an earlier deadlist).
4271 * We calculate this by iterating over the relevant deadlists (from the snap
4272 * after lastsnap, backward to the snap after firstsnap), summing up the
4273 * space on the deadlist that was born after the snap before firstsnap.
4274 */
4275 int
4276 dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap,
4277     dsl_dataset_t *lastsnap,
4278     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
4279 {
4280     int err = 0;
4281     uint64_t snapobj;
4282     dsl_pool_t *dp = firstsnap->ds_dir->dd_pool;

4284     ASSERT(dsl_dataset_is_snapshot(firstsnap));
4285     ASSERT(dsl_dataset_is_snapshot(lastsnap));

4287     /*
4288      * Check that the snapshots are in the same dsl_dir, and firstsnap
4289      * is before lastsnap.
4290      */
4291     if (firstsnap->ds_dir != lastsnap->ds_dir ||
4292         firstsnap->ds_phys->ds_creation_txg >
4293         lastsnap->ds_phys->ds_creation_txg)
4294         return (EINVAL);

4296     *usedp = *compp = *uncompp = 0;

4298     rw_enter(&dp->dp_config_rwlock, RW_READER);
4299     snapobj = lastsnap->ds_phys->ds_next_snap_obj;
4300     while (snapobj != firstsnap->ds_object) {
4301         dsl_dataset_t *ds;
4302         uint64_t used, comp, uncompp;

4304         err = dsl_dataset_hold_obj(dp, snapobj, FTAG, &ds);
4305         if (err != 0)
4306             break;

4308         dsl_deadlist_space_range(&ds->ds_deadlist,
4309             firstsnap->ds_phys->ds_prev_snap_txg, UINT64_MAX,
4310             &used, &comp, &uncompp);
4311         *usedp += used;
4312         *compp += comp;
4313         *uncompp += uncompp;

4315         snapobj = ds->ds_phys->ds_prev_snap_obj;
4316         ASSERT3U(snapobj, !=, 0);
4317         dsl_dataset_rele(ds, FTAG);
4318     }
4319     rw_exit(&dp->dp_config_rwlock);
4320     return (err);
4321 }

```