```
**********************************************************
  100200 Fri Oct 16 11:29:50 2015
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
6314 buffer overflow in dsl_dataset_name
**********************************************************
______unchanged_portion_omitted_

 655 void
 656 dsl_dataset_name(dsl_dataset_t *ds, char *name)
 657 {
 658         if (ds == NULL) {
 659                 (void) strcpy(name, "mos");
 660         } else {
 661                 dsl_dir_name(ds->ds_dir, name);
 662                 VERIFY0(dsl_dataset_get_snapname(ds));
 663                 if (ds->ds_snapname[0]) {
 664                         (void) strcat(name, "@");
 665                         /*
 666                          * We use a "recursive" mutex so that we
 667                          * can call dprintf_ds() with ds_lock held.
 668                          */
 669                         if (!MUTEX_HELD(&ds->ds_lock)) {
 670                                 mutex_enter(&ds->ds_lock);
 671                                 VERIFY3U(strlen(name) +
 672                                         strlen(ds->ds_snapname) + 1, <=,
 673                                         ZFS_MAXNAMELEN);
 674 #endif /* ! codereview */
 675                                 (void) strcat(name, ds->ds_snapname);
 676                                 mutex_exit(&ds->ds_lock);
 677                         } else {
 678                                 VERIFY3U(strlen(name) +
 679                                         strlen(ds->ds_snapname) + 1, <=,
 680                                         ZFS_MAXNAMELEN);
 681 #endif /* ! codereview */
 682                                 (void) strcat(name, ds->ds_snapname);
 683                         }
 684                 }
 685         }
 686 }

 688 void
 689 dsl_dataset_rele(dsl_dataset_t *ds, void *tag)
 690 {
 691         dmu_buf_rele(ds->ds_dbuf, tag);
 692 }

 694 void
 695 dsl_dataset_disown(dsl_dataset_t *ds, void *tag)
 696 {
 697         ASSERT3P(ds->ds_owner, ==, tag);
 698         ASSERT(ds->ds_dbuf != NULL);

 700         mutex_enter(&ds->ds_lock);
 701         ds->ds_owner = NULL;
 702         mutex_exit(&ds->ds_lock);
 703         dsl_dataset_long_rele(ds, tag);
 704         dsl_dataset_rele(ds, tag);
 705 }

 707 boolean_t
 708 dsl_dataset_tryown(dsl_dataset_t *ds, void *tag)
 709 {
 710         boolean_t gotit = FALSE;

 712         ASSERT(dsl_pool_config_held(ds->ds_dir->dd_pool));
 713         mutex_enter(&ds->ds_lock);
```



```
 714         if (ds->ds_owner == NULL && !DS_IS_INCONSISTENT(ds)) {
 715                 ds->ds_owner = tag;
 716                 dsl_dataset_long_hold(ds, tag);
 717                 gotit = TRUE;
 718         }
 719         mutex_exit(&ds->ds_lock);
 720         return (gotit);
 721 }

 723 boolean_t
 724 dsl_dataset_has_owner(dsl_dataset_t *ds)
 725 {
 726         boolean_t rv;
 727         mutex_enter(&ds->ds_lock);
 728         rv = (ds->ds_owner != NULL);
 729         mutex_exit(&ds->ds_lock);
 730         return (rv);
 731 }

 733 static void
 734 dsl_dataset_activate_feature(uint64_t dsobj, spa_feature_t f, dmu_tx_t *tx)
 735 {
 736         spa_t *spa = dmu_tx_pool(tx)->dp_spa;
 737         objset_t *mos = dmu_tx_pool(tx)->dp_meta_objset;
 738         uint64_t zero = 0;

 740         VERIFY(spa_feature_table[f].fi_flags & ZFEATURE_FLAG_PER_DATASET);

 742         spa_feature_incr(spa, f, tx);
 743         dmu_object_zapify(mos, dsobj, DMU_OT_DSL_DATASET, tx);

 745         VERIFY0(zap_add(mos, dsobj, spa_feature_table[f].fi_guid,
 746             sizeof (zero), 1, &zero, tx));
 747 }

 749 void
 750 dsl_dataset_deactivate_feature(uint64_t dsobj, spa_feature_t f, dmu_tx_t *tx)
 751 {
 752         spa_t *spa = dmu_tx_pool(tx)->dp_spa;
 753         objset_t *mos = dmu_tx_pool(tx)->dp_meta_objset;

 755         VERIFY(spa_feature_table[f].fi_flags & ZFEATURE_FLAG_PER_DATASET);

 757         VERIFY0(zap_remove(mos, dsobj, spa_feature_table[f].fi_guid, tx));
 758         spa_feature_decr(spa, f, tx);
 759 }

 761 uint64_t
 762 dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
 763     uint64_t flags, dmu_tx_t *tx)
 764 {
 765         dsl_pool_t *dp = dd->dd_pool;
 766         dmu_buf_t *dbuf;
 767         dsl_dataset_phys_t *dsphys;
 768         uint64_t dsobj;
 769         objset_t *mos = dp->dp_meta_objset;

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

 774         ASSERT(origin == NULL || origin->ds_dir->dd_pool == dp);
 775         ASSERT(origin == NULL || dsl_dataset_phys(origin)->ds_num_children > 0);
 776         ASSERT(dmu_tx_is_syncing(tx));
 777         ASSERT(dsl_dir_phys(dd)->dd_head_dataset_obj == 0);

 779         dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
```

```
780             DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
781         VERIFY0(dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
782         dmu_buf_will_dirty(dbuf, tx);
783         dsphys = dbuf->db_data;
784         bzero(dsphys, sizeof (dsl_dataset_phys_t));
785         dsphys->ds_dir_obj = dd->dd_object;
786         dsphys->ds_flags = flags;
787         dsphys->ds_fsid_guid = unique_create();
788         (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
789             sizeof (dsphys->ds_guid));
790         dsphys->ds_snapnames_zapobj =
791             zap_create_norm(mos, U8_TEXTPREP_TOUPPER, DMU_OT_DSL_DS_SNAP_MAP,
792             DMU_OT_NONE, 0, tx);
793         dsphys->ds_creation_time = gethrestime_sec();
794         dsphys->ds_creation_txg = tx->tx_txg == TXG_INITIAL ? 1 : tx->tx_txg;

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

801                 dsphys->ds_prev_snap_obj = origin->ds_object;
802                 dsphys->ds_prev_snap_txg =
803                     dsl_dataset_phys(origin)->ds_creation_txg;
804                 dsphys->ds_referenced_bytes =
805                     dsl_dataset_phys(origin)->ds_referenced_bytes;
806                 dsphys->ds_compressed_bytes =
807                     dsl_dataset_phys(origin)->ds_compressed_bytes;
808                 dsphys->ds_uncompressed_bytes =
809                     dsl_dataset_phys(origin)->ds_uncompressed_bytes;
810                 dsphys->ds_bp = dsl_dataset_phys(origin)->ds_bp;

812                 /*
813                  * Inherit flags that describe the dataset's contents
814                  * (INCONSISTENT) or properties (Case Insensitive).
815                  */
816                 dsphys->ds_flags |= dsl_dataset_phys(origin)->ds_flags &
817                     (DS_FLAG_INCONSISTENT | DS_FLAG_CI_DATASET);

819                 for (spa_feature_t f = 0; f < SPA_FEATURES; f++) {
820                         if (origin->ds_feature_inuse[f])
821                                 dsl_dataset_activate_feature(dsobj, f, tx);
822                 }

824                 dmu_buf_will_dirty(origin->ds_dbuf, tx);
825                 dsl_dataset_phys(origin)->ds_num_children++;

827                 VERIFY0(dsl_dataset_hold_obj(dp,
828                     dsl_dir_phys(origin->ds_dir)->dd_head_dataset_obj,
829                     FTAG, &ohds));
830                 dsphys->ds_deadlist_obj = dsl_deadlist_clone(&ohds->ds_deadlist,
831                     dsphys->ds_prev_snap_txg, dsphys->ds_prev_snap_obj, tx);
832                 dsl_dataset_rele(ohds, FTAG);

834                 if (spa_version(dp->dp_spa) >= SPA_VERSION_NEXT_CLONES) {
835                         if (dsl_dataset_phys(origin)->ds_next_clones_obj == 0) {
836                                 dsl_dataset_phys(origin)->ds_next_clones_obj =
837                                     zap_create(mos,
838                                     DMU_OT_NEXT_CLONES, DMU_OT_NONE, 0, tx);
839                         }
840                         VERIFY0(zap_add_int(mos,
841                             dsl_dataset_phys(origin)->ds_next_clones_obj,
842                             dsobj, tx));
843                 }

845                 dmu_buf_will_dirty(dd->dd_dbuf, tx);
```



```
846                 dsl_dir_phys(dd)->dd_origin_obj = origin->ds_object;
847                 if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
848                         if (dsl_dir_phys(origin->ds_dir)->dd_clones == 0) {
849                                 dmu_buf_will_dirty(origin->ds_dir->dd_dbuf, tx);
850                                 dsl_dir_phys(origin->ds_dir)->dd_clones =
851                                     zap_create(mos,
852                                     DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
853                         }
854                         VERIFY0(zap_add_int(mos,
855                             dsl_dir_phys(origin->ds_dir)->dd_clones,
856                             dsobj, tx));
857                 }
858         }

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

863         dmu_buf_rele(dbuf, FTAG);

865         dmu_buf_will_dirty(dd->dd_dbuf, tx);
866         dsl_dir_phys(dd)->dd_head_dataset_obj = dsobj;

868         return (dsobj);
869 }

871 static void
872 dsl_dataset_zero_zil(dsl_dataset_t *ds, dmu_tx_t *tx)
873 {
874         objset_t *os;

876         VERIFY0(dmu_objset_from_ds(ds, &os));
877         bzero(&os->os_zil_header, sizeof (os->os_zil_header));
878         dsl_dataset_dirty(ds, tx);
879 }

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

889         ASSERT(dmu_tx_is_syncing(tx));
890         ASSERT(lastname[0] != '@');

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

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

898         dsl_deleg_set_create_perms(dd, tx, cr);

900         /*
901          * Since we're creating a new node we know it's a leaf, so we can
902          * initialize the counts if the limit feature is active.
903          */
904         if (spa_feature_is_active(dp->dp_spa, SPA_FEATURE_FS_SS_LIMIT)) {
905                 uint64_t cnt = 0;
906                 objset_t *os = dd->dd_pool->dp_meta_objset;

908                 dsl_dir_zapify(dd, tx);
909                 VERIFY0(zap_add(os, dd->dd_object, DD_FIELD_FILESYSTEM_COUNT,
910                     sizeof (cnt), 1, &cnt, tx));
911                 VERIFY0(zap_add(os, dd->dd_object, DD_FIELD_SNAPSHOT_COUNT,
```

```
912                     sizeof (cnt), 1, &cnt, tx));
913         }

915         dsl_dir_rele(dd, FTAG);

917         /*
918          * If we are creating a clone, make sure we zero out any stale
919          * data from the origin snapshots zil header.
920          */
921         if (origin != NULL && !(flags & DS_CREATE_FLAG_NODIRTY)) {
922                 dsl_dataset_t *ds;

924                 VERIFY0(dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
925                 dsl_dataset_zero_zil(ds, tx);
926                 dsl_dataset_rele(ds, FTAG);
927         }

929         return (dsobj);
930 }

932 /*
933  * The unique space in the head dataset can be calculated by subtracting
934  * the space used in the most recent snapshot, that is still being used
935  * in this file system, from the space currently in use.  To figure out
936  * the space in the most recent snapshot still in use, we need to take
937  * the total space used in the snapshot and subtract out the space that
938  * has been freed up since the snapshot was taken.
939  */
940 void
941 dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds)
942 {
943         uint64_t mrs_used;
944         uint64_t dlused, dlcomp, dluncomp;

946         ASSERT(!ds->ds_is_snapshot);

948         if (dsl_dataset_phys(ds)->ds_prev_snap_obj != 0)
949                 mrs_used = dsl_dataset_phys(ds->ds_prev)->ds_referenced_bytes;
950         else
951                 mrs_used = 0;

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

955         ASSERT3U(dlused, <=, mrs_used);
956         dsl_dataset_phys(ds)->ds_unique_bytes =
957             dsl_dataset_phys(ds)->ds_referenced_bytes - (mrs_used - dlused);

959         if (spa_version(ds->ds_dir->dd_pool->dp_spa) >=
960             SPA_VERSION_UNIQUE_ACCURATE)
961                 dsl_dataset_phys(ds)->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;
962 }

964 void
965 dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
966     dmu_tx_t *tx)
967 {
968         objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
969         uint64_t count;
970         int err;

972         ASSERT(dsl_dataset_phys(ds)->ds_num_children >= 2);
973         err = zap_remove_int(mos, dsl_dataset_phys(ds)->ds_next_clones_obj,
974             obj, tx);
975         /*
976          * The err should not be ENOENT, but a bug in a previous version
977          * of the code could cause upgrade_clones_cb() to not set
```



```
978          * ds_next_snap_obj when it should, leading to a missing entry.
979          * If we knew that the pool was created after
980          * SPA_VERSION_NEXT_CLONES, we could assert that it isn't
981          * ENOENT.  However, at least we can check that we don't have
982          * too many entries in the next_clones_obj even after failing to
983          * remove this one.
984          */
985         if (err != ENOENT)
986                 VERIFY0(err);
987         ASSERT0(zap_count(mos, dsl_dataset_phys(ds)->ds_next_clones_obj,
988             &count));
989         ASSERT3U(count, <=, dsl_dataset_phys(ds)->ds_num_children - 2);
990 }


993 blkptr_t *
994 dsl_dataset_get_blkptr(dsl_dataset_t *ds)
995 {
996         return (&dsl_dataset_phys(ds)->ds_bp);
997 }

999 void
1000 dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx)
1001 {
1002         ASSERT(dmu_tx_is_syncing(tx));
1003         /* If it's the meta-objset, set dp_meta_rootbp */
1004         if (ds == NULL) {
1005                 tx->tx_pool->dp_meta_rootbp = *bp;
1006         } else {
1007                 dmu_buf_will_dirty(ds->ds_dbuf, tx);
1008                 dsl_dataset_phys(ds)->ds_bp = *bp;
1009         }
1010 }

1012 spa_t *
1013 dsl_dataset_get_spa(dsl_dataset_t *ds)
1014 {
1015         return (ds->ds_dir->dd_pool->dp_spa);
1016 }

1018 void
1019 dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx)
1020 {
1021         dsl_pool_t *dp;

1023         if (ds == NULL) /* this is the meta-objset */
1024                 return;

1026         ASSERT(ds->ds_objset != NULL);

1028         if (dsl_dataset_phys(ds)->ds_next_snap_obj != 0)
1029                 panic("dirtying snapshot!");

1031         dp = ds->ds_dir->dd_pool;

1033         if (txg_list_add(&dp->dp_dirty_datasets, ds, tx->tx_txg)) {
1034                 /* up the hold count until we can be written out */
1035                 dmu_buf_add_ref(ds->ds_dbuf, ds);
1036         }
1037 }

1039 boolean_t
1040 dsl_dataset_is_dirty(dsl_dataset_t *ds)
1041 {
1042         for (int t = 0; t < TXG_SIZE; t++) {
1043                 if (txg_list_member(&ds->ds_dir->dd_pool->dp_dirty_datasets,
```

```
1044                     ds, t))
1045                         return (B_TRUE);
1046         }
1047         return (B_FALSE);
1048 }

1050 static int
1051 dsl_dataset_snapshot_reserve_space(dsl_dataset_t *ds, dmu_tx_t *tx)
1052 {
1053         uint64_t asize;

1055         if (!dmu_tx_is_syncing(tx))
1056                 return (0);

1058         /*
1059          * If there's an fs-only reservation, any blocks that might become
1060          * owned by the snapshot dataset must be accommodated by space
1061          * outside of the reservation.
1062          */
1063         ASSERT(ds->ds_reserved == 0 || DS_UNIQUE_IS_ACCURATE(ds));
1064         asize = MIN(dsl_dataset_phys(ds)->ds_unique_bytes, ds->ds_reserved);
1065         if (asize > dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE))
1066                 return (SET_ERROR(ENOSPC));

1068         /*
1069          * Propagate any reserved space for this snapshot to other
1070          * snapshot checks in this sync group.
1071          */
1072         if (asize > 0)
1073                 dsl_dir_willuse_space(ds->ds_dir, asize, tx);

1075         return (0);
1076 }

1078 typedef struct dsl_dataset_snapshot_arg {
1079         nvlist_t *ddsa_snaps;
1080         nvlist_t *ddsa_props;
1081         nvlist_t *ddsa_errors;
1082         cred_t *ddsa_cr;
1083 } dsl_dataset_snapshot_arg_t;

1085 int
1086 dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
1087     dmu_tx_t *tx, boolean_t recv, uint64_t cnt, cred_t *cr)
1088 {
1089         int error;
1090         uint64_t value;

1092         ds->ds_trysnap_txg = tx->tx_txg;

1094         if (!dmu_tx_is_syncing(tx))
1095                 return (0);

1097         /*
1098          * We don't allow multiple snapshots of the same txg.  If there
1099          * is already one, try again.
1100          */
1101         if (dsl_dataset_phys(ds)->ds_prev_snap_txg >= tx->tx_txg)
1102                 return (SET_ERROR(EAGAIN));

1104         /*
1105          * Check for conflicting snapshot name.
1106          */
1107         error = dsl_dataset_snap_lookup(ds, snapname, &value);
1108         if (error == 0)
1109                 return (SET_ERROR(EEXIST));
```



```
1110         if (error != ENOENT)
1111                 return (error);

1113         /*
1114          * We don't allow taking snapshots of inconsistent datasets, such as
1115          * those into which we are currently receiving.  However, if we are
1116          * creating this snapshot as part of a receive, this check will be
1117          * executed atomically with respect to the completion of the receive
1118          * itself but prior to the clearing of DS_FLAG_INCONSISTENT; in this
1119          * case we ignore this, knowing it will be fixed up for us shortly in
1120          * dmu_recv_end_sync().
1121          */
1122         if (!recv && DS_IS_INCONSISTENT(ds))
1123                 return (SET_ERROR(EBUSY));

1125         /*
1126          * Skip the check for temporary snapshots or if we have already checked
1127          * the counts in dsl_dataset_snapshot_check. This means we really only
1128          * check the count here when we're receiving a stream.
1129          */
1130         if (cnt != 0 && cr != NULL) {
1131                 error = dsl_fs_ss_limit_check(ds->ds_dir, cnt,
1132                     ZFS_PROP_SNAPSHOT_LIMIT, NULL, cr);
1133                 if (error != 0)
1134                         return (error);
1135         }

1137         error = dsl_dataset_snapshot_reserve_space(ds, tx);
1138         if (error != 0)
1139                 return (error);

1141         return (0);
1142 }

1144 static int
1145 dsl_dataset_snapshot_check(void *arg, dmu_tx_t *tx)
1146 {
1147         dsl_dataset_snapshot_arg_t *ddsa = arg;
1148         dsl_pool_t *dp = dmu_tx_pool(tx);
1149         nvpair_t *pair;
1150         int rv = 0;

1152         /*
1153          * Pre-compute how many total new snapshots will be created for each
1154          * level in the tree and below. This is needed for validating the
1155          * snapshot limit when either taking a recursive snapshot or when
1156          * taking multiple snapshots.
1157          *
1158          * The problem is that the counts are not actually adjusted when
1159          * we are checking, only when we finally sync. For a single snapshot,
1160          * this is easy, the count will increase by 1 at each node up the tree,
1161          * but its more complicated for the recursive/multiple snapshot case.
1162          *
1163          * The dsl_fs_ss_limit_check function does recursively check the count
1164          * at each level up the tree but since it is validating each snapshot
1165          * independently we need to be sure that we are validating the complete
1166          * count for the entire set of snapshots. We do this by rolling up the
1167          * counts for each component of the name into an nvlist and then
1168          * checking each of those cases with the aggregated count.
1169          *
1170          * This approach properly handles not only the recursive snapshot
1171          * case (where we get all of those on the ddsa_snaps list) but also
1172          * the sibling case (e.g. snapshot a/b and a/c so that we will also
1173          * validate the limit on 'a' using a count of 2).
1174          *
1175          * We validate the snapshot names in the third loop and only report
```

```
1176          * name errors once.
1177          */
1178         if (dmu_tx_is_syncing(tx)) {
1179                 nvlist_t *cnt_track = NULL;
1180                 cnt_track = fnvlist_alloc();
1181
1182                 /* Rollup aggregated counts into the cnt_track list */
1183                 for (pair = nvlist_next_nvpair(ddsa->ddsa_snaps, NULL);
1184                     pair != NULL;
1185                     pair = nvlist_next_nvpair(ddsa->ddsa_snaps, pair)) {
1186                         char *pdelim;
1187                         uint64_t val;
1188                         char nm[MAXPATHLEN];
1189
1190                         (void) strlcpy(nm, nvpair_name(pair), sizeof (nm));
1191                         pdelim = strchr(nm, '@');
1192                         if (pdelim == NULL)
1193                                 continue;
1194                         *pdelim = '\0';
1195
1196                         do {
1197                                 if (nvlist_lookup_uint64(cnt_track, nm,
1198                                     &val) == 0) {
1199                                         /* update existing entry */
1200                                         fnvlist_add_uint64(cnt_track, nm,
1201                                             val + 1);
1202                                 } else {
1203                                         /* add to list */
1204                                         fnvlist_add_uint64(cnt_track, nm, 1);
1205                                 }
1206
1207                                 pdelim = strrchr(nm, '/');
1208                                 if (pdelim != NULL)
1209                                         *pdelim = '\0';
1210                         } while (pdelim != NULL);
1211                 }
1212
1213                 /* Check aggregated counts at each level */
1214                 for (pair = nvlist_next_nvpair(cnt_track, NULL);
1215                     pair != NULL; pair = nvlist_next_nvpair(cnt_track, pair)) {
1216                         int error = 0;
1217                         char *name;
1218                         uint64_t cnt = 0;
1219                         dsl_dataset_t *ds;
1220
1221                         name = nvpair_name(pair);
1222                         cnt = fnvpair_value_uint64(pair);
1223                         ASSERT(cnt > 0);
1224
1225                         error = dsl_dataset_hold(dp, name, FTAG, &ds);
1226                         if (error == 0) {
1227                                 error = dsl_fs_ss_limit_check(ds->ds_dir, cnt,
1228                                     ZFS_PROP_SNAPSHOT_LIMIT, NULL,
1229                                     ddsa->ddsa_cr);
1230                                 dsl_dataset_rele(ds, FTAG);
1231                         }
1232
1233                         if (error != 0) {
1234                                 if (ddsa->ddsa_errors != NULL)
1235                                         fnvlist_add_int32(ddsa->ddsa_errors,
1236                                             name, error);
1237                                 rv = error;
1238                                 /* only report one error for this check */
1239                                 break;
1240                         }
1241                 }
```



```
1242                 nvlist_free(cnt_track);
1243         }
1244
1245         for (pair = nvlist_next_nvpair(ddsa->ddsa_snaps, NULL);
1246             pair != NULL; pair = nvlist_next_nvpair(ddsa->ddsa_snaps, pair)) {
1247                 int error = 0;
1248                 dsl_dataset_t *ds;
1249                 char *name, *atp;
1250                 char dsname[MAXNAMELEN];
1251
1252                 name = nvpair_name(pair);
1253                 if (strlen(name) >= MAXNAMELEN)
1254                         error = SET_ERROR(ENAMETOOLONG);
1255                 if (error == 0) {
1256                         atp = strchr(name, '@');
1257                         if (atp == NULL)
1258                                 error = SET_ERROR(EINVAL);
1259                         if (error == 0)
1260                                 (void) strlcpy(dsname, name, atp - name + 1);
1261                 }
1262                 if (error == 0)
1263                         error = dsl_dataset_hold(dp, dsname, FTAG, &ds);
1264                 if (error == 0) {
1265                         /* passing 0/NULL skips dsl_fs_ss_limit_check */
1266                         error = dsl_dataset_snapshot_check_impl(ds,
1267                             atp + 1, tx, B_FALSE, 0, NULL);
1268                         dsl_dataset_rele(ds, FTAG);
1269                 }
1270
1271                 if (error != 0) {
1272                         if (ddsa->ddsa_errors != NULL) {
1273                                 fnvlist_add_int32(ddsa->ddsa_errors,
1274                                     name, error);
1275                         }
1276                         rv = error;
1277                 }
1278         }
1279
1280         return (rv);
1281 }
1282
1283 void
1284 dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,
1285     dmu_tx_t *tx)
1286 {
1287         static zil_header_t zero_zil;
1288
1289         dsl_pool_t *dp = ds->ds_dir->dd_pool;
1290         dmu_buf_t *dbuf;
1291         dsl_dataset_phys_t *dsphys;
1292         uint64_t dsobj, crtxg;
1293         objset_t *mos = dp->dp_meta_objset;
1294         objset_t *os;
1295
1296         ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));
1297
1298         /*
1299          * If we are on an old pool, the zil must not be active, in which
1300          * case it will be zeroed.  Usually zil_suspend() accomplishes this.
1301          */
1302         ASSERT(spa_version(dmu_tx_pool(tx)->dp_spa) >= SPA_VERSION_FAST_SNAP ||
1303             dmu_objset_from_ds(ds, &os) != 0 ||
1304             bcmp(&os->os_phys->os_zil_header, &zero_zil,
1305             sizeof (zero_zil)) == 0);
1306
1307         dsl_fs_ss_count_adjust(ds->ds_dir, 1, DD_FIELD_SNAPSHOT_COUNT, tx);
```

```
1309         /*
1310          * The origin's ds_creation_txg has to be < TXG_INITIAL
1311          */
1312         if (strcmp(snapname, ORIGIN_DIR_NAME) == 0)
1313                 crtxg = 1;
1314         else
1315                 crtxg = tx->tx_txg;

1317         dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
1318             DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
1319         VERIFY0(dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
1320         dmu_buf_will_dirty(dbuf, tx);
1321         dsphys = dbuf->db_data;
1322         bzero(dsphys, sizeof (dsl_dataset_phys_t));
1323         dsphys->ds_dir_obj = ds->ds_dir->dd_object;
1324         dsphys->ds_fsid_guid = unique_create();
1325         (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
1326             sizeof (dsphys->ds_guid));
1327         dsphys->ds_prev_snap_obj = dsl_dataset_phys(ds)->ds_prev_snap_obj;
1328         dsphys->ds_prev_snap_txg = dsl_dataset_phys(ds)->ds_prev_snap_txg;
1329         dsphys->ds_next_snap_obj = ds->ds_object;
1330         dsphys->ds_num_children = 1;
1331         dsphys->ds_creation_time = gethrestime_sec();
1332         dsphys->ds_creation_txg = crtxg;
1333         dsphys->ds_deadlist_obj = dsl_dataset_phys(ds)->ds_deadlist_obj;
1334         dsphys->ds_referenced_bytes = dsl_dataset_phys(ds)->ds_referenced_bytes;
1335         dsphys->ds_compressed_bytes = dsl_dataset_phys(ds)->ds_compressed_bytes;
1336         dsphys->ds_uncompressed_bytes =
1337             dsl_dataset_phys(ds)->ds_uncompressed_bytes;
1338         dsphys->ds_flags = dsl_dataset_phys(ds)->ds_flags;
1339         dsphys->ds_bp = dsl_dataset_phys(ds)->ds_bp;
1340         dmu_buf_rele(dbuf, FTAG);

1342         for (spa_feature_t f = 0; f < SPA_FEATURES; f++) {
1343                 if (ds->ds_feature_inuse[f])
1344                         dsl_dataset_activate_feature(dsobj, f, tx);
1345         }

1347         ASSERT3U(ds->ds_prev != 0, ==,
1348             dsl_dataset_phys(ds)->ds_prev_snap_obj != 0);
1349         if (ds->ds_prev) {
1350                 uint64_t next_clones_obj =
1351                     dsl_dataset_phys(ds->ds_prev)->ds_next_clones_obj;
1352                 ASSERT(dsl_dataset_phys(ds->ds_prev)->ds_next_snap_obj ==
1353                     ds->ds_object ||
1354                     dsl_dataset_phys(ds->ds_prev)->ds_num_children > 1);
1355                 if (dsl_dataset_phys(ds->ds_prev)->ds_next_snap_obj ==
1356                     ds->ds_object) {
1357                         dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
1358                         ASSERT3U(dsl_dataset_phys(ds)->ds_prev_snap_txg, ==,
1359                             dsl_dataset_phys(ds->ds_prev)->ds_creation_txg);
1360                         dsl_dataset_phys(ds->ds_prev)->ds_next_snap_obj = dsobj;
1361                 } else if (next_clones_obj != 0) {
1362                         dsl_dataset_remove_from_next_clones(ds->ds_prev,
1363                             dsphys->ds_next_snap_obj, tx);
1364                         VERIFY0(zap_add_int(mos,
1365                             next_clones_obj, dsobj, tx));
1366                 }
1367         }

1369         /*
1370          * If we have a reference-reservation on this dataset, we will
1371          * need to increase the amount of refreservation being charged
1372          * since our unique space is going to zero.
1373          */
```



```
1374         if (ds->ds_reserved) {
1375                 int64_t delta;
1376                 ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
1377                 delta = MIN(dsl_dataset_phys(ds)->ds_unique_bytes,
1378                     ds->ds_reserved);
1379                 dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV,
1380                     delta, 0, 0, tx);
1381         }

1383         dmu_buf_will_dirty(ds->ds_dbuf, tx);
1384         dsl_dataset_phys(ds)->ds_deadlist_obj =
1385             dsl_deadlist_clone(&ds->ds_deadlist, UINT64_MAX,
1386             dsl_dataset_phys(ds)->ds_prev_snap_obj, tx);
1387         dsl_deadlist_close(&ds->ds_deadlist);
1388         dsl_deadlist_open(&ds->ds_deadlist, mos,
1389             dsl_dataset_phys(ds)->ds_deadlist_obj);
1390         dsl_deadlist_add_key(&ds->ds_deadlist,
1391             dsl_dataset_phys(ds)->ds_prev_snap_txg, tx);

1393         ASSERT3U(dsl_dataset_phys(ds)->ds_prev_snap_txg, <, tx->tx_txg);
1394         dsl_dataset_phys(ds)->ds_prev_snap_obj = dsobj;
1395         dsl_dataset_phys(ds)->ds_prev_snap_txg = crtxg;
1396         dsl_dataset_phys(ds)->ds_unique_bytes = 0;
1397         if (spa_version(dp->dp_spa) >= SPA_VERSION_UNIQUE_ACCURATE)
1398                 dsl_dataset_phys(ds)->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;

1400         VERIFY0(zap_add(mos, dsl_dataset_phys(ds)->ds_snapnames_zapobj,
1401             snapname, 8, 1, &dsobj, tx));

1403         if (ds->ds_prev)
1404                 dsl_dataset_rele(ds->ds_prev, ds);
1405         VERIFY0(dsl_dataset_hold_obj(dp,
1406             dsl_dataset_phys(ds)->ds_prev_snap_obj, ds, &ds->ds_prev));

1408         dsl_scan_ds_snapshotted(ds, tx);

1410         dsl_dir_snap_cmtime_update(ds->ds_dir);

1412         spa_history_log_internal_ds(ds->ds_prev, "snapshot", tx, "");
1413 }

1415 static void
1416 dsl_dataset_snapshot_sync(void *arg, dmu_tx_t *tx)
1417 {
1418         dsl_dataset_snapshot_arg_t *ddsa = arg;
1419         dsl_pool_t *dp = dmu_tx_pool(tx);
1420         nvpair_t *pair;

1422         for (pair = nvlist_next_nvpair(ddsa->ddsa_snaps, NULL);
1423             pair != NULL; pair = nvlist_next_nvpair(ddsa->ddsa_snaps, pair)) {
1424                 dsl_dataset_t *ds;
1425                 char *name, *atp;
1426                 char dsname[MAXNAMELEN];

1428                 name = nvpair_name(pair);
1429                 atp = strchr(name, '@');
1430                 (void) strlcpy(dsname, name, atp - name + 1);
1431                 VERIFY0(dsl_dataset_hold(dp, dsname, FTAG, &ds));

1433                 dsl_dataset_snapshot_sync_impl(ds, atp + 1, tx);
1434                 if (ddsa->ddsa_props != NULL) {
1435                         dsl_props_set_sync_impl(ds->ds_prev,
1436                             ZPROP_SRC_LOCAL, ddsa->ddsa_props, tx);
1437                 }
1438                 dsl_dataset_rele(ds, FTAG);
1439         }
```

```
1440 }

1442 /*
1443  * The snapshots must all be in the same pool.
1444  * All-or-nothing: if there are any failures, nothing will be modified.
1445  */
1446 int
1447 dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors)
1448 {
1449         dsl_dataset_snapshot_arg_t ddsa;
1450         nvpair_t *pair;
1451         boolean_t needsuspend;
1452         int error;
1453         spa_t *spa;
1454         char *firstname;
1455         nvlist_t *suspended = NULL;

1457         pair = nvlist_next_nvpair(snaps, NULL);
1458         if (pair == NULL)
1459                 return (0);
1460         firstname = nvpair_name(pair);

1462         error = spa_open(firstname, &spa, FTAG);
1463         if (error != 0)
1464                 return (error);
1465         needsuspend = (spa_version(spa) < SPA_VERSION_FAST_SNAP);
1466         spa_close(spa, FTAG);

1468         if (needsuspend) {
1469                 suspended = fnvlist_alloc();
1470                 for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
1471                     pair = nvlist_next_nvpair(snaps, pair)) {
1472                         char fsname[MAXNAMELEN];
1473                         char *snapname = nvpair_name(pair);
1474                         char *atp;
1475                         void *cookie;

1477                         atp = strchr(snapname, '@');
1478                         if (atp == NULL) {
1479                                 error = SET_ERROR(EINVAL);
1480                                 break;
1481                         }
1482                         (void) strlcpy(fsname, snapname, atp - snapname + 1);

1484                         error = zil_suspend(fsname, &cookie);
1485                         if (error != 0)
1486                                 break;
1487                         fnvlist_add_uint64(suspended, fsname,
1488                             (uintptr_t)cookie);
1489                 }
1490         }

1492         ddsa.ddsa_snaps = snaps;
1493         ddsa.ddsa_props = props;
1494         ddsa.ddsa_errors = errors;
1495         ddsa.ddsa_cr = CRED();

1497         if (error == 0) {
1498                 error = dsl_sync_task(firstname, dsl_dataset_snapshot_check,
1499                     dsl_dataset_snapshot_sync, &ddsa,
1500                     fnvlist_num_pairs(snaps) * 3, ZFS_SPACE_CHECK_NORMAL);
1501         }

1503         if (suspended != NULL) {
1504                 for (pair = nvlist_next_nvpair(suspended, NULL); pair != NULL;
1505                     pair = nvlist_next_nvpair(suspended, pair)) {
```



```
1506                         zil_resume((void *)(uintptr_t)
1507                             fnvpair_value_uint64(pair));
1508                 }
1509                 fnvlist_free(suspended);
1510         }

1512         return (error);
1513 }

1515 typedef struct dsl_dataset_snapshot_tmp_arg {
1516         const char *ddsta_fsname;
1517         const char *ddsta_snapname;
1518         minor_t ddsta_cleanup_minor;
1519         const char *ddsta_htag;
1520 } dsl_dataset_snapshot_tmp_arg_t;

1522 static int
1523 dsl_dataset_snapshot_tmp_check(void *arg, dmu_tx_t *tx)
1524 {
1525         dsl_dataset_snapshot_tmp_arg_t *ddsta = arg;
1526         dsl_pool_t *dp = dmu_tx_pool(tx);
1527         dsl_dataset_t *ds;
1528         int error;

1530         error = dsl_dataset_hold(dp, ddsta->ddsta_fsname, FTAG, &ds);
1531         if (error != 0)
1532                 return (error);

1534         /* NULL cred means no limit check for tmp snapshot */
1535         error = dsl_dataset_snapshot_check_impl(ds, ddsta->ddsta_snapname,
1536             tx, B_FALSE, 0, NULL);
1537         if (error != 0) {
1538                 dsl_dataset_rele(ds, FTAG);
1539                 return (error);
1540         }

1542         if (spa_version(dp->dp_spa) < SPA_VERSION_USERREFS) {
1543                 dsl_dataset_rele(ds, FTAG);
1544                 return (SET_ERROR(ENOTSUP));
1545         }
1546         error = dsl_dataset_user_hold_check_one(NULL, ddsta->ddsta_htag,
1547             B_TRUE, tx);
1548         if (error != 0) {
1549                 dsl_dataset_rele(ds, FTAG);
1550                 return (error);
1551         }

1553         dsl_dataset_rele(ds, FTAG);
1554         return (0);
1555 }

1557 static void
1558 dsl_dataset_snapshot_tmp_sync(void *arg, dmu_tx_t *tx)
1559 {
1560         dsl_dataset_snapshot_tmp_arg_t *ddsta = arg;
1561         dsl_pool_t *dp = dmu_tx_pool(tx);
1562         dsl_dataset_t *ds;

1564         VERIFY0(dsl_dataset_hold(dp, ddsta->ddsta_fsname, FTAG, &ds));

1566         dsl_dataset_snapshot_sync_impl(ds, ddsta->ddsta_snapname, tx);
1567         dsl_dataset_user_hold_sync_one(ds->ds_prev, ddsta->ddsta_htag,
1568             ddsta->ddsta_cleanup_minor, gethrestime_sec(), tx);
1569         dsl_destroy_snapshot_sync_impl(ds->ds_prev, B_TRUE, tx);

1571         dsl_dataset_rele(ds, FTAG);
```

```
}

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

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

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

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

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

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


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

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

        if (ds->ds_resume_bytes[tx->tx_txg & TXG_MASK] != 0) {
                VERIFY0(zap_update(tx->tx_pool->dp_meta_objset,
                    ds->ds_object, DS_FIELD_RESUME_OBJECT, 8, 1,
                    &ds->ds_resume_object[tx->tx_txg & TXG_MASK], tx));
                VERIFY0(zap_update(tx->tx_pool->dp_meta_objset,
                    ds->ds_object, DS_FIELD_RESUME_OFFSET, 8, 1,
                    &ds->ds_resume_offset[tx->tx_txg & TXG_MASK], tx));
                VERIFY0(zap_update(tx->tx_pool->dp_meta_objset,
                    ds->ds_object, DS_FIELD_RESUME_BYTES, 8, 1,
                    &ds->ds_resume_bytes[tx->tx_txg & TXG_MASK], tx));
                ds->ds_resume_object[tx->tx_txg & TXG_MASK] = 0;
                ds->ds_resume_offset[tx->tx_txg & TXG_MASK] = 0;
                ds->ds_resume_bytes[tx->tx_txg & TXG_MASK] = 0;
        }
```



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

        for (spa_feature_t f = 0; f < SPA_FEATURES; f++) {
                if (ds->ds_feature_activation_needed[f]) {
                        if (ds->ds_feature_inuse[f])
                                continue;
                        dsl_dataset_activate_feature(ds->ds_object, f, tx);
                        ds->ds_feature_inuse[f] = B_TRUE;
                }
        }
}

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

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

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

static void
get_receive_resume_stats(dsl_dataset_t *ds, nvlist_t *nv)
{
        dsl_pool_t *dp = ds->ds_dir->dd_pool;

        if (dsl_dataset_has_resume_receive_state(ds)) {
                char *str;
                void *packed;
                uint8_t *compressed;
                uint64_t val;
```

```
1704                nvlist_t *token_nv = fnvlist_alloc();
1705                size_t packed_size, compressed_size;

1707                if (zap_lookup(dp->dp_meta_objset, ds->ds_object,
1708                    DS_FIELD_RESUME_FROMGUID, sizeof (val), 1, &val) == 0) {
1709                        fnvlist_add_uint64(token_nv, "fromguid", val);
1710                }
1711                if (zap_lookup(dp->dp_meta_objset, ds->ds_object,
1712                    DS_FIELD_RESUME_OBJECT, sizeof (val), 1, &val) == 0) {
1713                        fnvlist_add_uint64(token_nv, "object", val);
1714                }
1715                if (zap_lookup(dp->dp_meta_objset, ds->ds_object,
1716                    DS_FIELD_RESUME_OFFSET, sizeof (val), 1, &val) == 0) {
1717                        fnvlist_add_uint64(token_nv, "offset", val);
1718                }
1719                if (zap_lookup(dp->dp_meta_objset, ds->ds_object,
1720                    DS_FIELD_RESUME_BYTES, sizeof (val), 1, &val) == 0) {
1721                        fnvlist_add_uint64(token_nv, "bytes", val);
1722                }
1723                if (zap_lookup(dp->dp_meta_objset, ds->ds_object,
1724                    DS_FIELD_RESUME_TOGUID, sizeof (val), 1, &val) == 0) {
1725                        fnvlist_add_uint64(token_nv, "toguid", val);
1726                }
1727                char buf[256];
1728                if (zap_lookup(dp->dp_meta_objset, ds->ds_object,
1729                    DS_FIELD_RESUME_TONAME, 1, sizeof (buf), buf) == 0) {
1730                        fnvlist_add_string(token_nv, "toname", buf);
1731                }
1732                if (zap_contains(dp->dp_meta_objset, ds->ds_object,
1733                    DS_FIELD_RESUME_EMBEDOK) == 0) {
1734                        fnvlist_add_boolean(token_nv, "embedok");
1735                }
1736                packed = fnvlist_pack(token_nv, &packed_size);
1737                fnvlist_free(token_nv);
1738                compressed = kmem_alloc(packed_size, KM_SLEEP);

1740                compressed_size = gzip_compress(packed, compressed,
1741                    packed_size, packed_size, 6);

1743                zio_cksum_t cksum;
1744                fletcher_4_native(compressed, compressed_size, NULL, &cksum);

1746                str = kmem_alloc(compressed_size * 2 + 1, KM_SLEEP);
1747                for (int i = 0; i < compressed_size; i++) {
1748                        (void) sprintf(str + i * 2, "%02x", compressed[i]);
1749                }
1750                str[compressed_size * 2] = '\0';
1751                char *propval = kmem_asprintf("%u-%llx-%llx-%s",
1752                    ZFS_SEND_RESUME_TOKEN_VERSION,
1753                    (longlong_t)cksum.zc_word[0],
1754                    (longlong_t)packed_size, str);
1755                dsl_prop_nvlist_add_string(nv,
1756                    ZFS_PROP_RECEIVE_RESUME_TOKEN, propval);
1757                kmem_free(packed, packed_size);
1758                kmem_free(str, compressed_size * 2 + 1);
1759                kmem_free(compressed, packed_size);
1760                strfree(propval);
1761        }
1762 }

1764 void
1765 dsl_dataset_stats(dsl_dataset_t *ds, nvlist_t *nv)
1766 {
1767        dsl_pool_t *dp = ds->ds_dir->dd_pool;
1768        uint64_t refd, avail, uobjs, aobjs, ratio;
```



```
1770        ASSERT(dsl_pool_config_held(dp));

1772        ratio = dsl_dataset_phys(ds)->ds_compressed_bytes == 0 ? 100 :
1773            (dsl_dataset_phys(ds)->ds_uncompressed_bytes * 100 /
1774            dsl_dataset_phys(ds)->ds_compressed_bytes);

1776        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRATIO, ratio);
1777        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_LOGICALREFERENCED,
1778            dsl_dataset_phys(ds)->ds_uncompressed_bytes);

1780        if (ds->ds_is_snapshot) {
1781                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_COMPRESSRATIO, ratio);
1782                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USED,
1783                    dsl_dataset_phys(ds)->ds_unique_bytes);
1784                get_clones_stat(ds, nv);
1785        } else {
1786                if (ds->ds_prev != NULL && ds->ds_prev != dp->dp_origin_snap) {
1787                        char buf[MAXNAMELEN];
1788                        dsl_dataset_name(ds->ds_prev, buf);
1789                        dsl_prop_nvlist_add_string(nv, ZFS_PROP_PREV_SNAP, buf);
1790                }

1792                dsl_dir_stats(ds->ds_dir, nv);
1793        }

1795        dsl_dataset_space(ds, &refd, &avail, &uobjs, &aobjs);
1796        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_AVAILABLE, avail);
1797        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFERENCED, refd);

1799        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATION,
1800            dsl_dataset_phys(ds)->ds_creation_time);
1801        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATETXG,
1802            dsl_dataset_phys(ds)->ds_creation_txg);
1803        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFQUOTA,
1804            ds->ds_quota);
1805        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRESERVATION,
1806            ds->ds_reserved);
1807        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_GUID,
1808            dsl_dataset_phys(ds)->ds_guid);
1809        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_UNIQUE,
1810            dsl_dataset_phys(ds)->ds_unique_bytes);
1811        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_OBJSETID,
1812            ds->ds_object);
1813        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERREFS,
1814            ds->ds_userrefs);
1815        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_DEFER_DESTROY,
1816            DS_IS_DEFER_DESTROY(ds) ? 1 : 0);

1818        if (dsl_dataset_phys(ds)->ds_prev_snap_obj != 0) {
1819                uint64_t written, comp, uncomp;
1820                dsl_pool_t *dp = ds->ds_dir->dd_pool;
1821                dsl_dataset_t *prev;

1823                int err = dsl_dataset_hold_obj(dp,
1824                    dsl_dataset_phys(ds)->ds_prev_snap_obj, FTAG, &prev);
1825                if (err == 0) {
1826                        err = dsl_dataset_space_written(prev, ds, &written,
1827                            &comp, &uncomp);
1828                        dsl_dataset_rele(prev, FTAG);
1829                        if (err == 0) {
1830                                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_WRITTEN,
1831                                    written);
1832                        }
1833                }
1834        }
```

```
1836         if (!dsl_dataset_is_snapshot(ds)) {
1837                 /*
1838                  * A failed "newfs" (e.g. full) resumable receive leaves
1839                  * the stats set on this dataset.  Check here for the prop.
1840                  */
1841                 get_receive_resume_stats(ds, nv);

1843                 /*
1844                  * A failed incremental resumable receive leaves the
1845                  * stats set on our child named "%recv".  Check the child
1846                  * for the prop.
1847                  */
1848                 char recvname[ZFS_MAXNAMELEN];
1849                 dsl_dataset_t *recv_ds;
1850                 dsl_dataset_name(ds, recvname);
1851                 (void) strcat(recvname, "/");
1852                 (void) strcat(recvname, recv_clone_name);
1853                 if (dsl_dataset_hold(dp, recvname, FTAG, &recv_ds) == 0) {
1854                         get_receive_resume_stats(recv_ds, nv);
1855                         dsl_dataset_rele(recv_ds, FTAG);
1856                 }
1857         }
1858 }

1860 void
1861 dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat)
1862 {
1863         dsl_pool_t *dp = ds->ds_dir->dd_pool;
1864         ASSERT(dsl_pool_config_held(dp));

1866         stat->dds_creation_txg = dsl_dataset_phys(ds)->ds_creation_txg;
1867         stat->dds_inconsistent =
1868             dsl_dataset_phys(ds)->ds_flags & DS_FLAG_INCONSISTENT;
1869         stat->dds_guid = dsl_dataset_phys(ds)->ds_guid;
1870         stat->dds_origin[0] = '\0';
1871         if (ds->ds_is_snapshot) {
1872                 stat->dds_is_snapshot = B_TRUE;
1873                 stat->dds_num_clones =
1874                     dsl_dataset_phys(ds)->ds_num_children - 1;
1875         } else {
1876                 stat->dds_is_snapshot = B_FALSE;
1877                 stat->dds_num_clones = 0;

1879                 if (dsl_dir_is_clone(ds->ds_dir)) {
1880                         dsl_dataset_t *ods;

1882                         VERIFY0(dsl_dataset_hold_obj(dp,
1883                             dsl_dir_phys(ds->ds_dir)->dd_origin_obj,
1884                             FTAG, &ods));
1885                         dsl_dataset_name(ods, stat->dds_origin);
1886                         dsl_dataset_rele(ods, FTAG);
1887                 }
1888         }
1889 }

1891 uint64_t
1892 dsl_dataset_fsid_guid(dsl_dataset_t *ds)
1893 {
1894         return (ds->ds_fsid_guid);
1895 }

1897 void
1898 dsl_dataset_space(dsl_dataset_t *ds,
1899     uint64_t *refdbytesp, uint64_t *availbytesp,
1900     uint64_t *usedobjsp, uint64_t *availobjsp)
1901 {
```



```
1902         *refdbytesp = dsl_dataset_phys(ds)->ds_referenced_bytes;
1903         *availbytesp = dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE);
1904         if (ds->ds_reserved > dsl_dataset_phys(ds)->ds_unique_bytes)
1905                 *availbytesp +=
1906                     ds->ds_reserved - dsl_dataset_phys(ds)->ds_unique_bytes;
1907         if (ds->ds_quota != 0) {
1908                 /*
1909                  * Adjust available bytes according to refquota
1910                  */
1911                 if (*refdbytesp < ds->ds_quota)
1912                         *availbytesp = MIN(*availbytesp,
1913                             ds->ds_quota - *refdbytesp);
1914                 else
1915                         *availbytesp = 0;
1916         }
1917         *usedobjsp = BP_GET_FILL(&dsl_dataset_phys(ds)->ds_bp);
1918         *availobjsp = DN_MAX_OBJECT - *usedobjsp;
1919 }

1921 boolean_t
1922 dsl_dataset_modified_since_snap(dsl_dataset_t *ds, dsl_dataset_t *snap)
1923 {
1924         dsl_pool_t *dp = ds->ds_dir->dd_pool;

1926         ASSERT(dsl_pool_config_held(dp));
1927         if (snap == NULL)
1928                 return (B_FALSE);
1929         if (dsl_dataset_phys(ds)->ds_bp.blk_birth >
1930             dsl_dataset_phys(snap)->ds_creation_txg) {
1931                 objset_t *os, *os_snap;
1932                 /*
1933                  * It may be that only the ZIL differs, because it was
1934                  * reset in the head.  Don't count that as being
1935                  * modified.
1936                  */
1937                 if (dmu_objset_from_ds(ds, &os) != 0)
1938                         return (B_TRUE);
1939                 if (dmu_objset_from_ds(snap, &os_snap) != 0)
1940                         return (B_TRUE);
1941                 return (bcmp(&os->os_phys->os_meta_dnode,
1942                     &os_snap->os_phys->os_meta_dnode,
1943                     sizeof (os->os_phys->os_meta_dnode)) != 0);
1944         }
1945         return (B_FALSE);
1946 }

1948 typedef struct dsl_dataset_rename_snapshot_arg {
1949         const char *ddrsa_fsname;
1950         const char *ddrsa_oldsnapname;
1951         const char *ddrsa_newsnapname;
1952         boolean_t ddrsa_recursive;
1953         dmu_tx_t *ddrsa_tx;
1954 } dsl_dataset_rename_snapshot_arg_t;

1956 /* ARGSUSED */
1957 static int
1958 dsl_dataset_rename_snapshot_check_impl(dsl_pool_t *dp,
1959     dsl_dataset_t *hds, void *arg)
1960 {
1961         dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
1962         int error;
1963         uint64_t val;

1965         error = dsl_dataset_snap_lookup(hds, ddrsa->ddrsa_oldsnapname, &val);
1966         if (error != 0) {
1967                 /* ignore nonexistent snapshots */
```

```
1968                 return (error == ENOENT ? 0 : error);
1969         }

1971         /* new name should not exist */
1972         error = dsl_dataset_snap_lookup(hds, ddrsa->ddrsa_newsnapname, &val);
1973         if (error == 0)
1974                 error = SET_ERROR(EEXIST);
1975         else if (error == ENOENT)
1976                 error = 0;

1978         /* dataset name + 1 for the "@" + the new snapshot name must fit */
1979         if (dsl_dir_namelen(hds->ds_dir) + 1 +
1980             strlen(ddrsa->ddrsa_newsnapname) >= MAXNAMELEN)
1981                 error = SET_ERROR(ENAMETOOLONG);

1983         return (error);
1984 }

1986 static int
1987 dsl_dataset_rename_snapshot_check(void *arg, dmu_tx_t *tx)
1988 {
1989         dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
1990         dsl_pool_t *dp = dmu_tx_pool(tx);
1991         dsl_dataset_t *hds;
1992         int error;

1994         error = dsl_dataset_hold(dp, ddrsa->ddrsa_fsname, FTAG, &hds);
1995         if (error != 0)
1996                 return (error);

1998         if (ddrsa->ddrsa_recursive) {
1999                 error = dmu_objset_find_dp(dp, hds->ds_dir->dd_object,
2000                     dsl_dataset_rename_snapshot_check_impl, ddrsa,
2001                     DS_FIND_CHILDREN);
2002         } else {
2003                 error = dsl_dataset_rename_snapshot_check_impl(dp, hds, ddrsa);
2004         }
2005         dsl_dataset_rele(hds, FTAG);
2006         return (error);
2007 }

2009 static int
2010 dsl_dataset_rename_snapshot_sync_impl(dsl_pool_t *dp,
2011     dsl_dataset_t *hds, void *arg)
2012 {
2013         dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
2014         dsl_dataset_t *ds;
2015         uint64_t val;
2016         dmu_tx_t *tx = ddrsa->ddrsa_tx;
2017         int error;

2019         error = dsl_dataset_snap_lookup(hds, ddrsa->ddrsa_oldsnapname, &val);
2020         ASSERT(error == 0 || error == ENOENT);
2021         if (error == ENOENT) {
2022                 /* ignore nonexistent snapshots */
2023                 return (0);
2024         }

2026         VERIFY0(dsl_dataset_hold_obj(dp, val, FTAG, &ds));

2028         /* log before we change the name */
2029         spa_history_log_internal_ds(ds, "rename", tx,
2030             "-> @%s", ddrsa->ddrsa_newsnapname);

2032         VERIFY0(dsl_dataset_snap_remove(hds, ddrsa->ddrsa_oldsnapname, tx,
2033             B_FALSE));
```



```
2034         mutex_enter(&ds->ds_lock);
2035         (void) strcpy(ds->ds_snapname, ddrsa->ddrsa_newsnapname);
2036         mutex_exit(&ds->ds_lock);
2037         VERIFY0(zap_add(dp->dp_meta_objset,
2038             dsl_dataset_phys(hds)->ds_snapnames_zapobj,
2039             ds->ds_snapname, 8, 1, &ds->ds_object, tx));

2041         dsl_dataset_rele(ds, FTAG);
2042         return (0);
2043 }

2045 static void
2046 dsl_dataset_rename_snapshot_sync(void *arg, dmu_tx_t *tx)
2047 {
2048         dsl_dataset_rename_snapshot_arg_t *ddrsa = arg;
2049         dsl_pool_t *dp = dmu_tx_pool(tx);
2050         dsl_dataset_t *hds;

2052         VERIFY0(dsl_dataset_hold(dp, ddrsa->ddrsa_fsname, FTAG, &hds));
2053         ddrsa->ddrsa_tx = tx;
2054         if (ddrsa->ddrsa_recursive) {
2055                 VERIFY0(dmu_objset_find_dp(dp, hds->ds_dir->dd_object,
2056                     dsl_dataset_rename_snapshot_sync_impl, ddrsa,
2057                     DS_FIND_CHILDREN));
2058         } else {
2059                 VERIFY0(dsl_dataset_rename_snapshot_sync_impl(dp, hds, ddrsa));
2060         }
2061         dsl_dataset_rele(hds, FTAG);
2062 }

2064 int
2065 dsl_dataset_rename_snapshot(const char *fsname,
2066     const char *oldsnapname, const char *newsnapname, boolean_t recursive)
2067 {
2068         dsl_dataset_rename_snapshot_arg_t ddrsa;

2070         ddrsa.ddrsa_fsname = fsname;
2071         ddrsa.ddrsa_oldsnapname = oldsnapname;
2072         ddrsa.ddrsa_newsnapname = newsnapname;
2073         ddrsa.ddrsa_recursive = recursive;

2075         return (dsl_sync_task(fsname, dsl_dataset_rename_snapshot_check,
2076             dsl_dataset_rename_snapshot_sync, &ddrsa,
2077             1, ZFS_SPACE_CHECK_RESERVED));
2078 }

2080 /*
2081  * If we're doing an ownership handoff, we need to make sure that there is
2082  * only one long hold on the dataset.  We're not allowed to change anything here
2083  * so we don't permanently release the long hold or regular hold here.  We want
2084  * to do this only when syncing to avoid the dataset unexpectedly going away
2085  * when we release the long hold.
2086  */
2087 static int
2088 dsl_dataset_handoff_check(dsl_dataset_t *ds, void *owner, dmu_tx_t *tx)
2089 {
2090         boolean_t held;

2092         if (!dmu_tx_is_syncing(tx))
2093                 return (0);

2095         if (owner != NULL) {
2096                 VERIFY3P(ds->ds_owner, ==, owner);
2097                 dsl_dataset_long_rele(ds, owner);
2098         }
```

```
2100         held = dsl_dataset_long_held(ds);
2102         if (owner != NULL)
2103                 dsl_dataset_long_hold(ds, owner);
2105         if (held)
2106                 return (SET_ERROR(EBUSY));
2108         return (0);
2109 }
2111 typedef struct dsl_dataset_rollback_arg {
2112         const char *ddra_fsname;
2113         void *ddra_owner;
2114         nvlist_t *ddra_result;
2115 } dsl_dataset_rollback_arg_t;
2117 static int
2118 dsl_dataset_rollback_check(void *arg, dmu_tx_t *tx)
2119 {
2120         dsl_dataset_rollback_arg_t *ddra = arg;
2121         dsl_pool_t *dp = dmu_tx_pool(tx);
2122         dsl_dataset_t *ds;
2123         int64_t unused_refres_delta;
2124         int error;
2126         error = dsl_dataset_hold(dp, ddra->ddra_fsname, FTAG, &ds);
2127         if (error != 0)
2128                 return (error);
2130         /* must not be a snapshot */
2131         if (ds->ds_is_snapshot) {
2132                 dsl_dataset_rele(ds, FTAG);
2133                 return (SET_ERROR(EINVAL));
2134         }
2136         /* must have a most recent snapshot */
2137         if (dsl_dataset_phys(ds)->ds_prev_snap_txg < TXG_INITIAL) {
2138                 dsl_dataset_rele(ds, FTAG);
2139                 return (SET_ERROR(EINVAL));
2140         }
2142         /* must not have any bookmarks after the most recent snapshot */
2143         nvlist_t *proprequest = fnvlist_alloc();
2144         fnvlist_add_boolean(proprequest, zfs_prop_to_name(ZFS_PROP_CREATETXG));
2145         nvlist_t *bookmarks = fnvlist_alloc();
2146         error = dsl_get_bookmarks_impl(ds, proprequest, bookmarks);
2147         fnvlist_free(proprequest);
2148         if (error != 0)
2149                 return (error);
2150         for (nvpair_t *pair = nvlist_next_nvpair(bookmarks, NULL);
2151             pair != NULL; pair = nvlist_next_nvpair(bookmarks, pair)) {
2152                 nvlist_t *valuenv =
2153                     fnvlist_lookup_nvlist(fnvpair_value_nvlist(pair),
2154                     zfs_prop_to_name(ZFS_PROP_CREATETXG));
2155                 uint64_t createtxg = fnvlist_lookup_uint64(valuenv, "value");
2156                 if (createtxg > dsl_dataset_phys(ds)->ds_prev_snap_txg) {
2157                         fnvlist_free(bookmarks);
2158                         dsl_dataset_rele(ds, FTAG);
2159                         return (SET_ERROR(EEXIST));
2160                 }
2161         }
2162         fnvlist_free(bookmarks);
2164         error = dsl_dataset_handoff_check(ds, ddra->ddra_owner, tx);
2165         if (error != 0) {
```



```
2166                 dsl_dataset_rele(ds, FTAG);
2167                 return (error);
2168         }
2170         /*
2171          * Check if the snap we are rolling back to uses more than
2172          * the refquota.
2173          */
2174         if (ds->ds_quota != 0 &&
2175             dsl_dataset_phys(ds->ds_prev)->ds_referenced_bytes > ds->ds_quota) {
2176                 dsl_dataset_rele(ds, FTAG);
2177                 return (SET_ERROR(EDQUOT));
2178         }
2180         /*
2181          * When we do the clone swap, we will temporarily use more space
2182          * due to the refreservation (the head will no longer have any
2183          * unique space, so the entire amount of the refreservation will need
2184          * to be free).  We will immediately destroy the clone, freeing
2185          * this space, but the freeing happens over many txg's.
2186          */
2187         unused_refres_delta = (int64_t)MIN(ds->ds_reserved,
2188             dsl_dataset_phys(ds)->ds_unique_bytes);
2190         if (unused_refres_delta > 0 &&
2191             unused_refres_delta >
2192             dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE)) {
2193                 dsl_dataset_rele(ds, FTAG);
2194                 return (SET_ERROR(ENOSPC));
2195         }
2197         dsl_dataset_rele(ds, FTAG);
2198         return (0);
2199 }
2201 static void
2202 dsl_dataset_rollback_sync(void *arg, dmu_tx_t *tx)
2203 {
2204         dsl_dataset_rollback_arg_t *ddra = arg;
2205         dsl_pool_t *dp = dmu_tx_pool(tx);
2206         dsl_dataset_t *ds, *clone;
2207         uint64_t cloneobj;
2208         char namebuf[ZFS_MAXNAMELEN];
2210         VERIFY0(dsl_dataset_hold(dp, ddra->ddra_fsname, FTAG, &ds));
2212         dsl_dataset_name(ds->ds_prev, namebuf);
2213         fnvlist_add_string(ddra->ddra_result, "target", namebuf);
2215         cloneobj = dsl_dataset_create_sync(ds->ds_dir, "%rollback",
2216             ds->ds_prev, DS_CREATE_FLAG_NODIRTY, kcred, tx);
2218         VERIFY0(dsl_dataset_hold_obj(dp, cloneobj, FTAG, &clone));
2220         dsl_dataset_clone_swap_sync_impl(clone, ds, tx);
2221         dsl_dataset_zero_zil(ds, tx);
2223         dsl_destroy_head_sync_impl(clone, tx);
2225         dsl_dataset_rele(clone, FTAG);
2226         dsl_dataset_rele(ds, FTAG);
2227 }
2229 /*
2230  * Rolls back the given filesystem or volume to the most recent snapshot.
2231  * The name of the most recent snapshot will be returned under key "target"
```

```
 * in the result nvlist.
 *
 * If owner != NULL:
 * - The existing dataset MUST be owned by the specified owner at entry
 * - Upon return, dataset will still be held by the same owner, whether we
 *   succeed or not.
 *
 * This mode is required any time the existing filesystem is mounted.  See
 * notes above zfs_suspend_fs() for further details.
 */
int
dsl_dataset_rollback(const char *fsname, void *owner, nvlist_t *result)
{
        dsl_dataset_rollback_arg_t ddra;

        ddra.ddra_fsname = fsname;
        ddra.ddra_owner = owner;
        ddra.ddra_result = result;

        return (dsl_sync_task(fsname, dsl_dataset_rollback_check,
            dsl_dataset_rollback_sync, &ddra,
            1, ZFS_SPACE_CHECK_RESERVED));
}

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

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

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

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

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

        hds = ddpa->ddpa_clone;
        max_snap_len = MAXNAMELEN - strlen(ddpa->ddpa_clonename) - 1;

        if (dsl_dataset_phys(hds)->ds_flags & DS_FLAG_NOPROMOTE) {
                promote_rele(ddpa, FTAG);
```



```
                return (SET_ERROR(EXDEV));
        }

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

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

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

        /*
         * Walk the snapshots that we are moving
         *
         * Compute space to transfer.  Consider the incremental changes
         * to used by each snapshot:
         * (my used) = (prev's used) + (blocks born) - (blocks killed)
         * So each snapshot gave birth to:
         * (blocks born) = (my used) - (prev's used) + (blocks killed)
         * So a sequence would look like:
         * (uN - u(N-1) + kN) + ... + (u1 - u0 + k1) + (u0 - 0 + k0)
         * Which simplifies to:
         * uN + kN + kN-1 + ... + k1 + k0
         * Note however, if we stop before we reach the ORIGIN we get:
         * uN + kN + kN-1 + ... + kM - uM-1
         */
        ss_mv_cnt = 0;
        ddpa->used = dsl_dataset_phys(origin_ds)->ds_referenced_bytes;
        ddpa->comp = dsl_dataset_phys(origin_ds)->ds_compressed_bytes;
        ddpa->uncomp = dsl_dataset_phys(origin_ds)->ds_uncompressed_bytes;
        for (snap = list_head(&ddpa->shared_snaps); snap;
            snap = list_next(&ddpa->shared_snaps, snap)) {
                uint64_t val, dlused, dlcomp, dluncomp;
                dsl_dataset_t *ds = snap->ds;

                ss_mv_cnt++;

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

                /* Check that the snapshot name does not conflict */
                VERIFY0(dsl_dataset_get_snapname(ds));
                if (strlen(ds->ds_snapname) >= max_snap_len) {
                        err = SET_ERROR(ENAMETOOLONG);
                        goto out;
                }
                err = dsl_dataset_snap_lookup(hds, ds->ds_snapname, &val);
                if (err == 0) {
```

```
2364                        (void) strcpy(ddpa->err_ds, snap->ds->ds_snapname);
2365                        err = SET_ERROR(EEXIST);
2366                        goto out;
2367                }
2368                if (err != ENOENT)
2369                        goto out;

2371                /* The very first snapshot does not have a deadlist */
2372                if (dsl_dataset_phys(ds)->ds_prev_snap_obj == 0)
2373                        continue;

2375                dsl_deadlist_space(&ds->ds_deadlist,
2376                    &dlused, &dlcomp, &dluncomp);
2377                ddpa->used += dlused;
2378                ddpa->comp += dlcomp;
2379                ddpa->uncomp += dluncomp;
2380        }

2382        /*
2383         * If we are a clone of a clone then we never reached ORIGIN,
2384         * so we need to subtract out the clone origin's used space.
2385         */
2386        if (ddpa->origin_origin) {
2387                ddpa->used -=
2388                    dsl_dataset_phys(ddpa->origin_origin)->ds_referenced_bytes;
2389                ddpa->comp -=
2390                    dsl_dataset_phys(ddpa->origin_origin)->ds_compressed_bytes;
2391                ddpa->uncomp -=
2392                    dsl_dataset_phys(ddpa->origin_origin)->
2393                    ds_uncompressed_bytes;
2394        }

2396        /* Check that there is enough space and limit headroom here */
2397        err = dsl_dir_transfer_possible(origin_ds->ds_dir, hds->ds_dir,
2398            0, ss_mv_cnt, ddpa->used, ddpa->cr);
2399        if (err != 0)
2400                goto out;

2402        /*
2403         * Compute the amounts of space that will be used by snapshots
2404         * after the promotion (for both origin and clone).  For each,
2405         * it is the amount of space that will be on all of their
2406         * deadlists (that was not born before their new origin).
2407         */
2408        if (dsl_dir_phys(hds->ds_dir)->dd_flags & DD_FLAG_USED_BREAKDOWN) {
2409                uint64_t space;

2411                /*
2412                 * Note, typically this will not be a clone of a clone,
2413                 * so dd_origin_txg will be < TXG_INITIAL, so
2414                 * these snaplist_space() -> dsl_deadlist_space_range()
2415                 * calls will be fast because they do not have to
2416                 * iterate over all bps.
2417                 */
2418                snap = list_head(&ddpa->origin_snaps);
2419                err = snaplist_space(&ddpa->shared_snaps,
2420                    snap->ds->ds_dir->dd_origin_txg, &ddpa->cloneusedsnap);
2421                if (err != 0)
2422                        goto out;

2424                err = snaplist_space(&ddpa->clone_snaps,
2425                    snap->ds->ds_dir->dd_origin_txg, &space);
2426                if (err != 0)
2427                        goto out;
2428                ddpa->cloneusedsnap += space;
2429        }
```



```
2430        if (dsl_dir_phys(origin_ds->ds_dir)->dd_flags &
2431            DD_FLAG_USED_BREAKDOWN) {
2432                err = snaplist_space(&ddpa->origin_snaps,
2433                    dsl_dataset_phys(origin_ds)->ds_creation_txg,
2434                    &ddpa->originusedsnap);
2435                if (err != 0)
2436                        goto out;
2437        }

2439 out:
2440        promote_rele(ddpa, FTAG);
2441        return (err);
2442 }

2444 static void
2445 dsl_dataset_promote_sync(void *arg, dmu_tx_t *tx)
2446 {
2447        dsl_dataset_promote_arg_t *ddpa = arg;
2448        dsl_pool_t *dp = dmu_tx_pool(tx);
2449        dsl_dataset_t *hds;
2450        struct promotenode *snap;
2451        dsl_dataset_t *origin_ds;
2452        dsl_dataset_t *origin_head;
2453        dsl_dir_t *dd;
2454        dsl_dir_t *odd = NULL;
2455        uint64_t oldnext_obj;
2456        int64_t delta;

2458        VERIFY0(promote_hold(ddpa, dp, FTAG));
2459        hds = ddpa->ddpa_clone;

2461        ASSERT0(dsl_dataset_phys(hds)->ds_flags & DS_FLAG_NOPROMOTE);

2463        snap = list_head(&ddpa->shared_snaps);
2464        origin_ds = snap->ds;
2465        dd = hds->ds_dir;

2467        snap = list_head(&ddpa->origin_snaps);
2468        origin_head = snap->ds;

2470        /*
2471         * We need to explicitly open odd, since origin_ds's dd will be
2472         * changing.
2473         */
2474        VERIFY0(dsl_dir_hold_obj(dp, origin_ds->ds_dir->dd_object,
2475            NULL, FTAG, &odd));

2477        /* change origin's next snap */
2478        dmu_buf_will_dirty(origin_ds->ds_dbuf, tx);
2479        oldnext_obj = dsl_dataset_phys(origin_ds)->ds_next_snap_obj;
2480        snap = list_tail(&ddpa->clone_snaps);
2481        ASSERT3U(dsl_dataset_phys(snap->ds)->ds_prev_snap_obj, ==,
2482            origin_ds->ds_object);
2483        dsl_dataset_phys(origin_ds)->ds_next_snap_obj = snap->ds->ds_object;

2485        /* change the origin's next clone */
2486        if (dsl_dataset_phys(origin_ds)->ds_next_clones_obj) {
2487                dsl_dataset_remove_from_next_clones(origin_ds,
2488                    snap->ds->ds_object, tx);
2489                VERIFY0(zap_add_int(dp->dp_meta_objset,
2490                    dsl_dataset_phys(origin_ds)->ds_next_clones_obj,
2491                    oldnext_obj, tx));
2492        }

2494        /* change origin */
2495        dmu_buf_will_dirty(dd->dd_dbuf, tx);
```

```
2496         ASSERT3U(dsl_dir_phys(dd)->dd_origin_obj, ==, origin_ds->ds_object);
2497         dsl_dir_phys(dd)->dd_origin_obj = dsl_dir_phys(odd)->dd_origin_obj;
2498         dd->dd_origin_txg = origin_head->ds_dir->dd_origin_txg;
2499         dmu_buf_will_dirty(odd->dd_dbuf, tx);
2500         dsl_dir_phys(odd)->dd_origin_obj = origin_ds->ds_object;
2501         origin_head->ds_dir->dd_origin_txg =
2502             dsl_dataset_phys(origin_ds)->ds_creation_txg;

2504         /* change dd_clone entries */
2505         if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2506                 VERIFY0(zap_remove_int(dp->dp_meta_objset,
2507                     dsl_dir_phys(odd)->dd_clones, hds->ds_object, tx));
2508                 VERIFY0(zap_add_int(dp->dp_meta_objset,
2509                     dsl_dir_phys(ddpa->origin_origin->ds_dir)->dd_clones,
2510                     hds->ds_object, tx));

2512                 VERIFY0(zap_remove_int(dp->dp_meta_objset,
2513                     dsl_dir_phys(ddpa->origin_origin->ds_dir)->dd_clones,
2514                     origin_head->ds_object, tx));
2515                 if (dsl_dir_phys(dd)->dd_clones == 0) {
2516                         dsl_dir_phys(dd)->dd_clones =
2517                             zap_create(dp->dp_meta_objset, DMU_OT_DSL_CLONES,
2518                             DMU_OT_NONE, 0, tx);
2519                 }
2520                 VERIFY0(zap_add_int(dp->dp_meta_objset,
2521                     dsl_dir_phys(dd)->dd_clones, origin_head->ds_object, tx));
2522         }

2524         /* move snapshots to this dir */
2525         for (snap = list_head(&ddpa->shared_snaps); snap;
2526             snap = list_next(&ddpa->shared_snaps, snap)) {
2527                 dsl_dataset_t *ds = snap->ds;

2529                 /*
2530                  * Property callbacks are registered to a particular
2531                  * dsl_dir.  Since ours is changing, evict the objset
2532                  * so that they will be unregistered from the old dsl_dir.
2533                  */
2534                 if (ds->ds_objset) {
2535                         dmu_objset_evict(ds->ds_objset);
2536                         ds->ds_objset = NULL;
2537                 }

2539                 /* move snap name entry */
2540                 VERIFY0(dsl_dataset_get_snapname(ds));
2541                 VERIFY0(dsl_dataset_snap_remove(origin_head,
2542                     ds->ds_snapname, tx, B_TRUE));
2543                 VERIFY0(zap_add(dp->dp_meta_objset,
2544                     dsl_dataset_phys(hds)->ds_snapnames_zapobj, ds->ds_snapname,
2545                     8, 1, &ds->ds_object, tx));
2546                 dsl_fs_ss_count_adjust(hds->ds_dir, 1,
2547                     DD_FIELD_SNAPSHOT_COUNT, tx);

2549                 /* change containing dsl_dir */
2550                 dmu_buf_will_dirty(ds->ds_dbuf, tx);
2551                 ASSERT3U(dsl_dataset_phys(ds)->ds_dir_obj, ==, odd->dd_object);
2552                 dsl_dataset_phys(ds)->ds_dir_obj = dd->dd_object;
2553                 ASSERT3P(ds->ds_dir, ==, odd);
2554                 dsl_dir_rele(ds->ds_dir, ds);
2555                 VERIFY0(dsl_dir_hold_obj(dp, dd->dd_object,
2556                     NULL, ds, &ds->ds_dir));

2558                 /* move any clone references */
2559                 if (dsl_dataset_phys(ds)->ds_next_clones_obj &&
2560                     spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2561                         zap_cursor_t zc;
```



```
2562                         zap_attribute_t za;

2564                         for (zap_cursor_init(&zc, dp->dp_meta_objset,
2565                             dsl_dataset_phys(ds)->ds_next_clones_obj);
2566                             zap_cursor_retrieve(&zc, &za) == 0;
2567                             zap_cursor_advance(&zc)) {
2568                                 dsl_dataset_t *cnds;
2569                                 uint64_t o;

2571                                 if (za.za_first_integer == oldnext_obj) {
2572                                         /*
2573                                          * We've already moved the
2574                                          * origin's reference.
2575                                          */
2576                                         continue;
2577                                 }

2579                                 VERIFY0(dsl_dataset_hold_obj(dp,
2580                                     za.za_first_integer, FTAG, &cnds));
2581                                 o = dsl_dir_phys(cnds->ds_dir)->
2582                                     dd_head_dataset_obj;

2584                                 VERIFY0(zap_remove_int(dp->dp_meta_objset,
2585                                     dsl_dir_phys(odd)->dd_clones, o, tx));
2586                                 VERIFY0(zap_add_int(dp->dp_meta_objset,
2587                                     dsl_dir_phys(dd)->dd_clones, o, tx));
2588                                 dsl_dataset_rele(cnds, FTAG);
2589                         }
2590                         zap_cursor_fini(&zc);
2591                 }

2593                 ASSERT(!dsl_prop_hascb(ds));
2594         }

2596         /*
2597          * Change space accounting.
2598          * Note, pa->*usedsnap and dd_used_breakdown[SNAP] will either
2599          * both be valid, or both be 0 (resulting in delta == 0).  This
2600          * is true for each of {clone,origin} independently.
2601          */

2603         delta = ddpa->cloneusedsnap -
2604             dsl_dir_phys(dd)->dd_used_breakdown[DD_USED_SNAP];
2605         ASSERT3S(delta, >=, 0);
2606         ASSERT3U(ddpa->used, >=, delta);
2607         dsl_dir_diduse_space(dd, DD_USED_SNAP, delta, 0, 0, tx);
2608         dsl_dir_diduse_space(dd, DD_USED_HEAD,
2609             ddpa->used - delta, ddpa->comp, ddpa->uncomp, tx);

2611         delta = ddpa->originusedsnap -
2612             dsl_dir_phys(odd)->dd_used_breakdown[DD_USED_SNAP];
2613         ASSERT3S(delta, <=, 0);
2614         ASSERT3U(ddpa->used, >=, -delta);
2615         dsl_dir_diduse_space(odd, DD_USED_SNAP, delta, 0, 0, tx);
2616         dsl_dir_diduse_space(odd, DD_USED_HEAD,
2617             -ddpa->used - delta, -ddpa->comp, -ddpa->uncomp, tx);

2619         dsl_dataset_phys(origin_ds)->ds_unique_bytes = ddpa->unique;

2621         /* log history record */
2622         spa_history_log_internal_ds(hds, "promote", tx, "");

2624         dsl_dir_rele(odd, FTAG);
2625         promote_rele(ddpa, FTAG);
2626 }
```

```
2628 /*
2629  * Make a list of dsl_dataset_t's for the snapshots between first_obj
2630  * (exclusive) and last_obj (inclusive).  The list will be in reverse
2631  * order (last_obj will be the list_head()).  If first_obj == 0, do all
2632  * snapshots back to this dataset's origin.
2633  */
2634 static int
2635 snaplist_make(dsl_pool_t *dp,
2636     uint64_t first_obj, uint64_t last_obj, list_t *l, void *tag)
2637 {
2638         uint64_t obj = last_obj;
2639
2640         list_create(l, sizeof (struct promotenode),
2641             offsetof(struct promotenode, link));
2642
2643         while (obj != first_obj) {
2644                 dsl_dataset_t *ds;
2645                 struct promotenode *snap;
2646                 int err;
2647
2648                 err = dsl_dataset_hold_obj(dp, obj, tag, &ds);
2649                 ASSERT(err != ENOENT);
2650                 if (err != 0)
2651                         return (err);
2652
2653                 if (first_obj == 0)
2654                         first_obj = dsl_dir_phys(ds->ds_dir)->dd_origin_obj;
2655
2656                 snap = kmem_alloc(sizeof (*snap), KM_SLEEP);
2657                 snap->ds = ds;
2658                 list_insert_tail(l, snap);
2659                 obj = dsl_dataset_phys(ds)->ds_prev_snap_obj;
2660         }
2661
2662         return (0);
2663 }
2664
2665 static int
2666 snaplist_space(list_t *l, uint64_t mintxg, uint64_t *spacep)
2667 {
2668         struct promotenode *snap;
2669
2670         *spacep = 0;
2671         for (snap = list_head(l); snap; snap = list_next(l, snap)) {
2672                 uint64_t used, comp, uncomp;
2673                 dsl_deadlist_space_range(&snap->ds->ds_deadlist,
2674                     mintxg, UINT64_MAX, &used, &comp, &uncomp);
2675                 *spacep += used;
2676         }
2677         return (0);
2678 }
2679
2680 static void
2681 snaplist_destroy(list_t *l, void *tag)
2682 {
2683         struct promotenode *snap;
2684
2685         if (l == NULL || !list_link_active(&l->list_head))
2686                 return;
2687
2688         while ((snap = list_tail(l)) != NULL) {
2689                 list_remove(l, snap);
2690                 dsl_dataset_rele(snap->ds, tag);
2691                 kmem_free(snap, sizeof (*snap));
2692         }
2693         list_destroy(l);
```



```
2694 }
2695
2696 static int
2697 promote_hold(dsl_dataset_promote_arg_t *ddpa, dsl_pool_t *dp, void *tag)
2698 {
2699         int error;
2700         dsl_dir_t *dd;
2701         struct promotenode *snap;
2702
2703         error = dsl_dataset_hold(dp, ddpa->ddpa_clonename, tag,
2704             &ddpa->ddpa_clone);
2705         if (error != 0)
2706                 return (error);
2707         dd = ddpa->ddpa_clone->ds_dir;
2708
2709         if (ddpa->ddpa_clone->ds_is_snapshot ||
2710             !dsl_dir_is_clone(dd)) {
2711                 dsl_dataset_rele(ddpa->ddpa_clone, tag);
2712                 return (SET_ERROR(EINVAL));
2713         }
2714
2715         error = snaplist_make(dp, 0, dsl_dir_phys(dd)->dd_origin_obj,
2716             &ddpa->shared_snaps, tag);
2717         if (error != 0)
2718                 goto out;
2719
2720         error = snaplist_make(dp, 0, ddpa->ddpa_clone->ds_object,
2721             &ddpa->clone_snaps, tag);
2722         if (error != 0)
2723                 goto out;
2724
2725         snap = list_head(&ddpa->shared_snaps);
2726         ASSERT3U(snap->ds->ds_object, ==, dsl_dir_phys(dd)->dd_origin_obj);
2727         error = snaplist_make(dp, dsl_dir_phys(dd)->dd_origin_obj,
2728             dsl_dir_phys(snap->ds->ds_dir)->dd_head_dataset_obj,
2729             &ddpa->origin_snaps, tag);
2730         if (error != 0)
2731                 goto out;
2732
2733         if (dsl_dir_phys(snap->ds->ds_dir)->dd_origin_obj != 0) {
2734                 error = dsl_dataset_hold_obj(dp,
2735                     dsl_dir_phys(snap->ds->ds_dir)->dd_origin_obj,
2736                     tag, &ddpa->origin_origin);
2737                 if (error != 0)
2738                         goto out;
2739         }
2740 out:
2741         if (error != 0)
2742                 promote_rele(ddpa, tag);
2743         return (error);
2744 }
2745
2746 static void
2747 promote_rele(dsl_dataset_promote_arg_t *ddpa, void *tag)
2748 {
2749         snaplist_destroy(&ddpa->shared_snaps, tag);
2750         snaplist_destroy(&ddpa->clone_snaps, tag);
2751         snaplist_destroy(&ddpa->origin_snaps, tag);
2752         if (ddpa->origin_origin != NULL)
2753                 dsl_dataset_rele(ddpa->origin_origin, tag);
2754         dsl_dataset_rele(ddpa->ddpa_clone, tag);
2755 }
2756
2757 /*
2758  * Promote a clone.
2759  *
```

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

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

        ddpa.ddpa_clonename = name;
        ddpa.err_ds = conflsnap;
        ddpa.cr = CRED();

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

int
dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
    dsl_dataset_t *origin_head, boolean_t force, void *owner, dmu_tx_t *tx)
{
        int64_t unused_refres_delta;

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

        /* if we are not forcing, the branch point should be just before them */
        if (!force && clone->ds_prev != origin_head->ds_prev)
                return (SET_ERROR(EINVAL));

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

        /* the clone should be a child of the origin */
        if (clone->ds_dir->dd_parent != origin_head->ds_dir)
                return (SET_ERROR(EINVAL));

        /* origin_head shouldn't be modified unless 'force' */
        if (!force &&
            dsl_dataset_modified_since_snap(origin_head, origin_head->ds_prev))
                return (SET_ERROR(ETXTBSY));

        /* origin_head should have no long holds (e.g. is not mounted) */
        if (dsl_dataset_handoff_check(origin_head, owner, tx))
```



```
                return (SET_ERROR(EBUSY));

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

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

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

        return (0);
}

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

        ASSERT(clone->ds_reserved == 0);
        ASSERT(origin_head->ds_quota == 0 ||
            dsl_dataset_phys(clone)->ds_unique_bytes <= origin_head->ds_quota);
        ASSERT3P(clone->ds_prev, ==, origin_head->ds_prev);

        /*
         * Swap per-dataset feature flags.
         */
        for (spa_feature_t f = 0; f < SPA_FEATURES; f++) {
                if (!(spa_feature_table[f].fi_flags &
                    ZFEATURE_FLAG_PER_DATASET)) {
                        ASSERT(!clone->ds_feature_inuse[f]);
                        ASSERT(!origin_head->ds_feature_inuse[f]);
                        continue;
                }

                boolean_t clone_inuse = clone->ds_feature_inuse[f];
                boolean_t origin_head_inuse = origin_head->ds_feature_inuse[f];

                if (clone_inuse) {
                        dsl_dataset_deactivate_feature(clone->ds_object, f, tx);
                        clone->ds_feature_inuse[f] = B_FALSE;
                }
                if (origin_head_inuse) {
                        dsl_dataset_deactivate_feature(origin_head->ds_object,
                            f, tx);
                        origin_head->ds_feature_inuse[f] = B_FALSE;
                }
                if (clone_inuse) {
                        dsl_dataset_activate_feature(origin_head->ds_object,
                            f, tx);
                        origin_head->ds_feature_inuse[f] = B_TRUE;
                }
                if (origin_head_inuse) {
                        dsl_dataset_activate_feature(clone->ds_object, f, tx);
                        clone->ds_feature_inuse[f] = B_TRUE;
```

```
2892                }
2893        }

2895        dmu_buf_will_dirty(clone->ds_dbuf, tx);
2896        dmu_buf_will_dirty(origin_head->ds_dbuf, tx);

2898        if (clone->ds_objset != NULL) {
2899                dmu_objset_evict(clone->ds_objset);
2900                clone->ds_objset = NULL;
2901        }

2903        if (origin_head->ds_objset != NULL) {
2904                dmu_objset_evict(origin_head->ds_objset);
2905                origin_head->ds_objset = NULL;
2906        }

2908        unused_refres_delta =
2909            (int64_t)MIN(origin_head->ds_reserved,
2910            dsl_dataset_phys(origin_head)->ds_unique_bytes) -
2911            (int64_t)MIN(origin_head->ds_reserved,
2912            dsl_dataset_phys(clone)->ds_unique_bytes);

2914        /*
2915         * Reset origin's unique bytes, if it exists.
2916         */
2917        if (clone->ds_prev) {
2918                dsl_dataset_t *origin = clone->ds_prev;
2919                uint64_t comp, uncomp;

2921                dmu_buf_will_dirty(origin->ds_dbuf, tx);
2922                dsl_deadlist_space_range(&clone->ds_deadlist,
2923                    dsl_dataset_phys(origin)->ds_prev_snap_txg, UINT64_MAX,
2924                    &dsl_dataset_phys(origin)->ds_unique_bytes, &comp, &uncomp);
2925        }

2927        /* swap blkptrs */
2928        {
2929                blkptr_t tmp;
2930                tmp = dsl_dataset_phys(origin_head)->ds_bp;
2931                dsl_dataset_phys(origin_head)->ds_bp =
2932                    dsl_dataset_phys(clone)->ds_bp;
2933                dsl_dataset_phys(clone)->ds_bp = tmp;
2934        }

2936        /* set dd_*_bytes */
2937        {
2938                int64_t dused, dcomp, duncomp;
2939                uint64_t cdl_used, cdl_comp, cdl_uncomp;
2940                uint64_t odl_used, odl_comp, odl_uncomp;

2942                ASSERT3U(dsl_dir_phys(clone->ds_dir)->
2943                    dd_used_breakdown[DD_USED_SNAP], ==, 0);

2945                dsl_deadlist_space(&clone->ds_deadlist,
2946                    &cdl_used, &cdl_comp, &cdl_uncomp);
2947                dsl_deadlist_space(&origin_head->ds_deadlist,
2948                    &odl_used, &odl_comp, &odl_uncomp);

2950                dused = dsl_dataset_phys(clone)->ds_referenced_bytes +
2951                    cdl_used -
2952                    (dsl_dataset_phys(origin_head)->ds_referenced_bytes +
2953                    odl_used);
2954                dcomp = dsl_dataset_phys(clone)->ds_compressed_bytes +
2955                    cdl_comp -
2956                    (dsl_dataset_phys(origin_head)->ds_compressed_bytes +
2957                    odl_comp);
```



```
2958                duncomp = dsl_dataset_phys(clone)->ds_uncompressed_bytes +
2959                    cdl_uncomp -
2960                    (dsl_dataset_phys(origin_head)->ds_uncompressed_bytes +
2961                    odl_uncomp);

2963                dsl_dir_diduse_space(origin_head->ds_dir, DD_USED_HEAD,
2964                    dused, dcomp, duncomp, tx);
2965                dsl_dir_diduse_space(clone->ds_dir, DD_USED_HEAD,
2966                    -dused, -dcomp, -duncomp, tx);

2968                /*
2969                 * The difference in the space used by snapshots is the
2970                 * difference in snapshot space due to the head's
2971                 * deadlist (since that's the only thing that's
2972                 * changing that affects the snapused).
2973                 */
2974                dsl_deadlist_space_range(&clone->ds_deadlist,
2975                    origin_head->ds_dir->dd_origin_txg, UINT64_MAX,
2976                    &cdl_used, &cdl_comp, &cdl_uncomp);
2977                dsl_deadlist_space_range(&origin_head->ds_deadlist,
2978                    origin_head->ds_dir->dd_origin_txg, UINT64_MAX,
2979                    &odl_used, &odl_comp, &odl_uncomp);
2980                dsl_dir_transfer_space(origin_head->ds_dir, cdl_used - odl_used,
2981                    DD_USED_HEAD, DD_USED_SNAP, tx);
2982        }

2984        /* swap ds_*_bytes */
2985        SWITCH64(dsl_dataset_phys(origin_head)->ds_referenced_bytes,
2986            dsl_dataset_phys(clone)->ds_referenced_bytes);
2987        SWITCH64(dsl_dataset_phys(origin_head)->ds_compressed_bytes,
2988            dsl_dataset_phys(clone)->ds_compressed_bytes);
2989        SWITCH64(dsl_dataset_phys(origin_head)->ds_uncompressed_bytes,
2990            dsl_dataset_phys(clone)->ds_uncompressed_bytes);
2991        SWITCH64(dsl_dataset_phys(origin_head)->ds_unique_bytes,
2992            dsl_dataset_phys(clone)->ds_unique_bytes);

2994        /* apply any parent delta for change in unconsumed refreservation */
2995        dsl_dir_diduse_space(origin_head->ds_dir, DD_USED_REFRSRV,
2996            unused_refres_delta, 0, 0, tx);

2998        /*
2999         * Swap deadlists.
3000         */
3001        dsl_deadlist_close(&clone->ds_deadlist);
3002        dsl_deadlist_close(&origin_head->ds_deadlist);
3003        SWITCH64(dsl_dataset_phys(origin_head)->ds_deadlist_obj,
3004            dsl_dataset_phys(clone)->ds_deadlist_obj);
3005        dsl_deadlist_open(&clone->ds_deadlist, dp->dp_meta_objset,
3006            dsl_dataset_phys(clone)->ds_deadlist_obj);
3007        dsl_deadlist_open(&origin_head->ds_deadlist, dp->dp_meta_objset,
3008            dsl_dataset_phys(origin_head)->ds_deadlist_obj);

3010        dsl_scan_ds_clone_swapped(origin_head, clone, tx);

3012        spa_history_log_internal_ds(clone, "clone swap", tx,
3013            "parent=%s", origin_head->ds_dir->dd_myname);
3014 }

3016 /*
3017  * Given a pool name and a dataset object number in that pool,
3018  * return the name of that dataset.
3019  */
3020 int
3021 dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf)
3022 {
3023         dsl_pool_t *dp;
```

```
3024         dsl_dataset_t *ds;
3025         int error;

3027         error = dsl_pool_hold(pname, FTAG, &dp);
3028         if (error != 0)
3029                 return (error);

3031         error = dsl_dataset_hold_obj(dp, obj, FTAG, &ds);
3032         if (error == 0) {
3033                 dsl_dataset_name(ds, buf);
3034                 dsl_dataset_rele(ds, FTAG);
3035         }
3036         dsl_pool_rele(dp, FTAG);

3038         return (error);
3039 }

3041 int
3042 dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
3043     uint64_t asize, uint64_t inflight, uint64_t *used, uint64_t *ref_rsrv)
3044 {
3045         int error = 0;

3047         ASSERT3S(asize, >, 0);

3049         /*
3050          * *ref_rsrv is the portion of asize that will come from any
3051          * unconsumed refreservation space.
3052          */
3053         *ref_rsrv = 0;

3055         mutex_enter(&ds->ds_lock);
3056         /*
3057          * Make a space adjustment for reserved bytes.
3058          */
3059         if (ds->ds_reserved > dsl_dataset_phys(ds)->ds_unique_bytes) {
3060                 ASSERT3U(*used, >=,
3061                     ds->ds_reserved - dsl_dataset_phys(ds)->ds_unique_bytes);
3062                 *used -=
3063                     (ds->ds_reserved - dsl_dataset_phys(ds)->ds_unique_bytes);
3064                 *ref_rsrv =
3065                     asize - MIN(asize, parent_delta(ds, asize + inflight));
3066         }

3068         if (!check_quota || ds->ds_quota == 0) {
3069                 mutex_exit(&ds->ds_lock);
3070                 return (0);
3071         }
3072         /*
3073          * If they are requesting more space, and our current estimate
3074          * is over quota, they get to try again unless the actual
3075          * on-disk is over quota and there are no pending changes (which
3076          * may free up space for us).
3077          */
3078         if (dsl_dataset_phys(ds)->ds_referenced_bytes + inflight >=
3079             ds->ds_quota) {
3080                 if (inflight > 0 ||
3081                     dsl_dataset_phys(ds)->ds_referenced_bytes < ds->ds_quota)
3082                         error = SET_ERROR(ERESTART);
3083                 else
3084                         error = SET_ERROR(EDQUOT);
3085         }
3086         mutex_exit(&ds->ds_lock);

3088         return (error);
3089 }
```



```
3091 typedef struct dsl_dataset_set_qr_arg {
3092         const char *ddsqra_name;
3093         zprop_source_t ddsqra_source;
3094         uint64_t ddsqra_value;
3095 } dsl_dataset_set_qr_arg_t;


3098 /* ARGSUSED */
3099 static int
3100 dsl_dataset_set_refquota_check(void *arg, dmu_tx_t *tx)
3101 {
3102         dsl_dataset_set_qr_arg_t *ddsqra = arg;
3103         dsl_pool_t *dp = dmu_tx_pool(tx);
3104         dsl_dataset_t *ds;
3105         int error;
3106         uint64_t newval;

3108         if (spa_version(dp->dp_spa) < SPA_VERSION_REFQUOTA)
3109                 return (SET_ERROR(ENOTSUP));

3111         error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
3112         if (error != 0)
3113                 return (error);

3115         if (ds->ds_is_snapshot) {
3116                 dsl_dataset_rele(ds, FTAG);
3117                 return (SET_ERROR(EINVAL));
3118         }

3120         error = dsl_prop_predict(ds->ds_dir,
3121             zfs_prop_to_name(ZFS_PROP_REFQUOTA),
3122             ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
3123         if (error != 0) {
3124                 dsl_dataset_rele(ds, FTAG);
3125                 return (error);
3126         }

3128         if (newval == 0) {
3129                 dsl_dataset_rele(ds, FTAG);
3130                 return (0);
3131         }

3133         if (newval < dsl_dataset_phys(ds)->ds_referenced_bytes ||
3134             newval < ds->ds_reserved) {
3135                 dsl_dataset_rele(ds, FTAG);
3136                 return (SET_ERROR(ENOSPC));
3137         }

3139         dsl_dataset_rele(ds, FTAG);
3140         return (0);
3141 }

3143 static void
3144 dsl_dataset_set_refquota_sync(void *arg, dmu_tx_t *tx)
3145 {
3146         dsl_dataset_set_qr_arg_t *ddsqra = arg;
3147         dsl_pool_t *dp = dmu_tx_pool(tx);
3148         dsl_dataset_t *ds;
3149         uint64_t newval;

3151         VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds));

3153         dsl_prop_set_sync_impl(ds,
3154             zfs_prop_to_name(ZFS_PROP_REFQUOTA),
3155             ddsqra->ddsqra_source, sizeof (ddsqra->ddsqra_value), 1,
```

```
3156             &ddsqra->ddsqra_value, tx);

3158         VERIFY0(dsl_prop_get_int_ds(ds,
3159             zfs_prop_to_name(ZFS_PROP_REFQUOTA), &newval));

3161         if (ds->ds_quota != newval) {
3162                 dmu_buf_will_dirty(ds->ds_dbuf, tx);
3163                 ds->ds_quota = newval;
3164         }
3165         dsl_dataset_rele(ds, FTAG);
3166 }

3168 int
3169 dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
3170     uint64_t refquota)
3171 {
3172         dsl_dataset_set_qr_arg_t ddsqra;

3174         ddsqra.ddsqra_name = dsname;
3175         ddsqra.ddsqra_source = source;
3176         ddsqra.ddsqra_value = refquota;

3178         return (dsl_sync_task(dsname, dsl_dataset_set_refquota_check,
3179             dsl_dataset_set_refquota_sync, &ddsqra, 0, ZFS_SPACE_CHECK_NONE));
3180 }

3182 static int
3183 dsl_dataset_set_refreservation_check(void *arg, dmu_tx_t *tx)
3184 {
3185         dsl_dataset_set_qr_arg_t *ddsqra = arg;
3186         dsl_pool_t *dp = dmu_tx_pool(tx);
3187         dsl_dataset_t *ds;
3188         int error;
3189         uint64_t newval, unique;

3191         if (spa_version(dp->dp_spa) < SPA_VERSION_REFRESERVATION)
3192                 return (SET_ERROR(ENOTSUP));

3194         error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
3195         if (error != 0)
3196                 return (error);

3198         if (ds->ds_is_snapshot) {
3199                 dsl_dataset_rele(ds, FTAG);
3200                 return (SET_ERROR(EINVAL));
3201         }

3203         error = dsl_prop_predict(ds->ds_dir,
3204             zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
3205             ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
3206         if (error != 0) {
3207                 dsl_dataset_rele(ds, FTAG);
3208                 return (error);
3209         }

3211         /*
3212          * If we are doing the preliminary check in open context, the
3213          * space estimates may be inaccurate.
3214          */
3215         if (!dmu_tx_is_syncing(tx)) {
3216                 dsl_dataset_rele(ds, FTAG);
3217                 return (0);
3218         }

3220         mutex_enter(&ds->ds_lock);
3221         if (!DS_UNIQUE_IS_ACCURATE(ds))
```



```
3222                 dsl_dataset_recalc_head_uniq(ds);
3223         unique = dsl_dataset_phys(ds)->ds_unique_bytes;
3224         mutex_exit(&ds->ds_lock);

3226         if (MAX(unique, newval) > MAX(unique, ds->ds_reserved)) {
3227                 uint64_t delta = MAX(unique, newval) -
3228                     MAX(unique, ds->ds_reserved);

3230                 if (delta >
3231                     dsl_dir_space_available(ds->ds_dir, NULL, 0, B_TRUE) ||
3232                     (ds->ds_quota > 0 && newval > ds->ds_quota)) {
3233                         dsl_dataset_rele(ds, FTAG);
3234                         return (SET_ERROR(ENOSPC));
3235                 }
3236         }

3238         dsl_dataset_rele(ds, FTAG);
3239         return (0);
3240 }

3242 void
3243 dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
3244     zprop_source_t source, uint64_t value, dmu_tx_t *tx)
3245 {
3246         uint64_t newval;
3247         uint64_t unique;
3248         int64_t delta;

3250         dsl_prop_set_sync_impl(ds, zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
3251             source, sizeof (value), 1, &value, tx);

3253         VERIFY0(dsl_prop_get_int_ds(ds,
3254             zfs_prop_to_name(ZFS_PROP_REFRESERVATION), &newval));

3256         dmu_buf_will_dirty(ds->ds_dbuf, tx);
3257         mutex_enter(&ds->ds_dir->dd_lock);
3258         mutex_enter(&ds->ds_lock);
3259         ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
3260         unique = dsl_dataset_phys(ds)->ds_unique_bytes;
3261         delta = MAX(0, (int64_t)(newval - unique)) -
3262             MAX(0, (int64_t)(ds->ds_reserved - unique));
3263         ds->ds_reserved = newval;
3264         mutex_exit(&ds->ds_lock);

3266         dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV, delta, 0, 0, tx);
3267         mutex_exit(&ds->ds_dir->dd_lock);
3268 }

3270 static void
3271 dsl_dataset_set_refreservation_sync(void *arg, dmu_tx_t *tx)
3272 {
3273         dsl_dataset_set_qr_arg_t *ddsqra = arg;
3274         dsl_pool_t *dp = dmu_tx_pool(tx);
3275         dsl_dataset_t *ds;

3277         VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds));
3278         dsl_dataset_set_refreservation_sync_impl(ds,
3279             ddsqra->ddsqra_source, ddsqra->ddsqra_value, tx);
3280         dsl_dataset_rele(ds, FTAG);
3281 }

3283 int
3284 dsl_dataset_set_refreservation(const char *dsname, zprop_source_t source,
3285     uint64_t refreservation)
3286 {
3287         dsl_dataset_set_qr_arg_t ddsqra;
```

```
3289         ddsqra.ddsqra_name = dsname;
3290         ddsqra.ddsqra_source = source;
3291         ddsqra.ddsqra_value = refreservation;

3293         return (dsl_sync_task(dsname, dsl_dataset_set_refreservation_check,
3294             dsl_dataset_set_refreservation_sync, &ddsqra,
3295             0, ZFS_SPACE_CHECK_NONE));
3296 }

3298 /*
3299  * Return (in *usedp) the amount of space written in new that is not
3300  * present in oldsnap.  New may be a snapshot or the head.  Old must be
3301  * a snapshot before new, in new's filesystem (or its origin).  If not then
3302  * fail and return EINVAL.
3303  *
3304  * The written space is calculated by considering two components:  First, we
3305  * ignore any freed space, and calculate the written as new's used space
3306  * minus old's used space.  Next, we add in the amount of space that was freed
3307  * between the two snapshots, thus reducing new's used space relative to old's.
3308  * Specifically, this is the space that was born before old->ds_creation_txg,
3309  * and freed before new (ie. on new's deadlist or a previous deadlist).
3310  *
3311  * space freed                         [--------------------]
3312  * snapshots                       ---O-------O--------O-------O------
3313  *                                         oldsnap            new
3314  */
3315 int
3316 dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
3317     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
3318 {
3319         int err = 0;
3320         uint64_t snapobj;
3321         dsl_pool_t *dp = new->ds_dir->dd_pool;

3323         ASSERT(dsl_pool_config_held(dp));

3325         *usedp = 0;
3326         *usedp += dsl_dataset_phys(new)->ds_referenced_bytes;
3327         *usedp -= dsl_dataset_phys(oldsnap)->ds_referenced_bytes;

3329         *compp = 0;
3330         *compp += dsl_dataset_phys(new)->ds_compressed_bytes;
3331         *compp -= dsl_dataset_phys(oldsnap)->ds_compressed_bytes;

3333         *uncompp = 0;
3334         *uncompp += dsl_dataset_phys(new)->ds_uncompressed_bytes;
3335         *uncompp -= dsl_dataset_phys(oldsnap)->ds_uncompressed_bytes;

3337         snapobj = new->ds_object;
3338         while (snapobj != oldsnap->ds_object) {
3339                 dsl_dataset_t *snap;
3340                 uint64_t used, comp, uncomp;

3342                 if (snapobj == new->ds_object) {
3343                         snap = new;
3344                 } else {
3345                         err = dsl_dataset_hold_obj(dp, snapobj, FTAG, &snap);
3346                         if (err != 0)
3347                                 break;
3348                 }

3350                 if (dsl_dataset_phys(snap)->ds_prev_snap_txg ==
3351                     dsl_dataset_phys(oldsnap)->ds_creation_txg) {
3352                         /*
3353                          * The blocks in the deadlist can not be born after
```



```
3354                          * ds_prev_snap_txg, so get the whole deadlist space,
3355                          * which is more efficient (especially for old-format
3356                          * deadlists).  Unfortunately the deadlist code
3357                          * doesn't have enough information to make this
3358                          * optimization itself.
3359                          */
3360                         dsl_deadlist_space(&snap->ds_deadlist,
3361                             &used, &comp, &uncomp);
3362                 } else {
3363                         dsl_deadlist_space_range(&snap->ds_deadlist,
3364                             0, dsl_dataset_phys(oldsnap)->ds_creation_txg,
3365                             &used, &comp, &uncomp);
3366                 }
3367                 *usedp += used;
3368                 *compp += comp;
3369                 *uncompp += uncomp;

3371                 /*
3372                  * If we get to the beginning of the chain of snapshots
3373                  * (ds_prev_snap_obj == 0) before oldsnap, then oldsnap
3374                  * was not a snapshot of/before new.
3375                  */
3376                 snapobj = dsl_dataset_phys(snap)->ds_prev_snap_obj;
3377                 if (snap != new)
3378                         dsl_dataset_rele(snap, FTAG);
3379                 if (snapobj == 0) {
3380                         err = SET_ERROR(EINVAL);
3381                         break;
3382                 }

3384         }
3385         return (err);
3386 }

3388 /*
3389  * Return (in *usedp) the amount of space that will be reclaimed if firstsnap,
3390  * lastsnap, and all snapshots in between are deleted.
3391  *
3392  * blocks that would be freed            [---------------------------]
3393  * snapshots                       ---O-------O--------O-------O--------O
3394  *                                           firstsnap        lastsnap
3395  *
3396  * This is the set of blocks that were born after the snap before firstsnap,
3397  * (birth > firstsnap->prev_snap_txg) and died before the snap after the
3398  * last snap (ie, is on lastsnap->ds_next->ds_deadlist or an earlier deadlist).
3399  * We calculate this by iterating over the relevant deadlists (from the snap
3400  * after lastsnap, backward to the snap after firstsnap), summing up the
3401  * space on the deadlist that was born after the snap before firstsnap.
3402  */
3403 int
3404 dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap,
3405     dsl_dataset_t *lastsnap,
3406     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
3407 {
3408         int err = 0;
3409         uint64_t snapobj;
3410         dsl_pool_t *dp = firstsnap->ds_dir->dd_pool;

3412         ASSERT(firstsnap->ds_is_snapshot);
3413         ASSERT(lastsnap->ds_is_snapshot);

3415         /*
3416          * Check that the snapshots are in the same dsl_dir, and firstsnap
3417          * is before lastsnap.
3418          */
3419         if (firstsnap->ds_dir != lastsnap->ds_dir ||
```

```
3420             dsl_dataset_phys(firstsnap)->ds_creation_txg >
3421             dsl_dataset_phys(lastsnap)->ds_creation_txg)
3422                 return (SET_ERROR(EINVAL));

3424         *usedp = *compp = *uncompp = 0;

3426         snapobj = dsl_dataset_phys(lastsnap)->ds_next_snap_obj;
3427         while (snapobj != firstsnap->ds_object) {
3428                 dsl_dataset_t *ds;
3429                 uint64_t used, comp, uncomp;

3431                 err = dsl_dataset_hold_obj(dp, snapobj, FTAG, &ds);
3432                 if (err != 0)
3433                         break;

3435                 dsl_deadlist_space_range(&ds->ds_deadlist,
3436                     dsl_dataset_phys(firstsnap)->ds_prev_snap_txg, UINT64_MAX,
3437                     &used, &comp, &uncomp);
3438                 *usedp += used;
3439                 *compp += comp;
3440                 *uncompp += uncomp;

3442                 snapobj = dsl_dataset_phys(ds)->ds_prev_snap_obj;
3443                 ASSERT3U(snapobj, !=, 0);
3444                 dsl_dataset_rele(ds, FTAG);
3445         }
3446         return (err);
3447 }

3449 /*
3450  * Return TRUE if 'earlier' is an earlier snapshot in 'later's timeline.
3451  * For example, they could both be snapshots of the same filesystem, and
3452  * 'earlier' is before 'later'.  Or 'earlier' could be the origin of
3453  * 'later's filesystem.  Or 'earlier' could be an older snapshot in the origin's
3454  * filesystem.  Or 'earlier' could be the origin's origin.
3455  *
3456  * If non-zero, earlier_txg is used instead of earlier's ds_creation_txg.
3457  */
3458 boolean_t
3459 dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier,
3460         uint64_t earlier_txg)
3461 {
3462         dsl_pool_t *dp = later->ds_dir->dd_pool;
3463         int error;
3464         boolean_t ret;

3466         ASSERT(dsl_pool_config_held(dp));
3467         ASSERT(earlier->ds_is_snapshot || earlier_txg != 0);

3469         if (earlier_txg == 0)
3470                 earlier_txg = dsl_dataset_phys(earlier)->ds_creation_txg;

3472         if (later->ds_is_snapshot &&
3473             earlier_txg >= dsl_dataset_phys(later)->ds_creation_txg)
3474                 return (B_FALSE);

3476         if (later->ds_dir == earlier->ds_dir)
3477                 return (B_TRUE);
3478         if (!dsl_dir_is_clone(later->ds_dir))
3479                 return (B_FALSE);

3481         if (dsl_dir_phys(later->ds_dir)->dd_origin_obj == earlier->ds_object)
3482                 return (B_TRUE);
3483         dsl_dataset_t *origin;
3484         error = dsl_dataset_hold_obj(dp,
3485             dsl_dir_phys(later->ds_dir)->dd_origin_obj, FTAG, &origin);
```



```
3486         if (error != 0)
3487                 return (B_FALSE);
3488         ret = dsl_dataset_is_before(origin, earlier, earlier_txg);
3489         dsl_dataset_rele(origin, FTAG);
3490         return (ret);
3491 }

3493 void
3494 dsl_dataset_zapify(dsl_dataset_t *ds, dmu_tx_t *tx)
3495 {
3496         objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
3497         dmu_object_zapify(mos, ds->ds_object, DMU_OT_DSL_DATASET, tx);
3498 }

3500 boolean_t
3501 dsl_dataset_is_zapified(dsl_dataset_t *ds)
3502 {
3503         dmu_object_info_t doi;

3505         dmu_object_info_from_db(ds->ds_dbuf, &doi);
3506         return (doi.doi_type == DMU_OTN_ZAP_METADATA);
3507 }

3509 boolean_t
3510 dsl_dataset_has_resume_receive_state(dsl_dataset_t *ds)
3511 {
3512         return (dsl_dataset_is_zapified(ds) &&
3513             zap_contains(ds->ds_dir->dd_pool->dp_meta_objset,
3514             ds->ds_object, DS_FIELD_RESUME_TOGUID) == 0);
3515 }
```