

new/usr/src/uts/common/fs/zfs/dmu_tx.c

1

```
*****
35483 Fri Aug 17 09:23:31 2012
new/usr/src/uts/common/fs/zfs/dmu_tx.c
1862 incremental zfs receive fails for sparse file > 8PB
dmu_tx_count_free is doing a horrible over-estimation of used memory. It
assumes that the file is fully non-sparse and calculates a worst-case estimate
of how much memory is needed to hold all metadata for the file. If a large
hole needs to be freed, the estimation goes into the TB-range, which obviously
fails later on.
This patch tries to calculate a more realistic estimate by counting the 11
blocks (the loop for this is already present) and assumes a worst-case
distribution of those blocks over the full length given.
Reviewed by: Matt Ahrens <matthew.ahrens@delphix.com>
Reviewed by: Simon Klinkert <klinkert@webgods.de>
*****
_____unchanged_portion_omitted_____
423 static void
424 dmu_tx_count_free(dmu_tx_hold_t *txh, uint64_t off, uint64_t len)
425 {
426     uint64_t blkid, nblk, lastblk;
427     uint64_t space = 0, unref = 0, skipped = 0;
428     dnode_t *dn = txh->txh_dnode;
429     dsl_dataset_t *ds = dn->dn_objset->os_dsl_dataset;
430     spa_t *spa = txh->txh_tx->tx_pool->dp_spa;
431     int epbs;
432     uint64_t l0span = 0, n11blk = 0;
433 #endif /* ! codereview */
435     if (dn->dn_nlevels == 0)
436         return;
438     /*
439      * The struct_rwlock protects us against dn_nlevels
440      * changing, in case (against all odds) we manage to dirty &
441      * sync out the changes after we check for being dirty.
442      * Also, dbuf_hold_impl() wants us to have the struct_rwlock.
443     */
444     rw_enter(&dn->dn_struct_rwlock, RW_READER);
445     epbs = dn->dn_inblkshift - SPA_BLKPTRSHIFT;
446     if (dn->dn_maxblkid == 0) {
447         if (off == 0 && len >= dn->dn_datblksz) {
448             blkid = 0;
449             nblk = 1;
450         } else {
451             rw_exit(&dn->dn_struct_rwlock);
452             return;
453         }
454     } else {
455         blkid = off >> dn->dn_datblkshift;
456         nblk = (len + dn->dn_datblksz - 1) >> dn->dn_datblkshift;
458         if (blkid >= dn->dn_maxblkid) {
459             rw_exit(&dn->dn_struct_rwlock);
460             return;
461         }
462         if (blkid + nblk > dn->dn_maxblkid)
463             nblk = dn->dn_maxblkid - blkid;
465     }
466     l0span = nblk; /* save for later use to calc level > 1 overhead */
467 #endif /* ! codereview */
468     if (dn->dn_nlevels == 1) {
469         int i;
470         for (i = 0; i < nblk; i++) {
471             blkptr_t *bp = dn->dn_phys->dn_blkptr;
```

new/usr/src/uts/common/fs/zfs/dmu_tx.c

2

```
472             ASSERT3U(blkid + i, <, dn->dn_nblkptr);
473             bp += blkid + i;
474             if (dsl_dataset_block_freeable(ds, bp, bp->blk_birth)) {
475                 dprintf_bp(bp, "can free old%s", "");
476                 space += bp_get_dsize(spa, bp);
477             }
478             unref += BP_GET_ASIZE(bp);
479         }
480         n11blk = 1;
481 #endif /* ! codereview */
482         nblk = 0;
483     }
484
485     /*
486      * Add in memory requirements of higher-level indirects.
487      * This assumes a worst-possible scenario for dn_nlevels.
488     */
489     {
490         uint64_t blkcnt = 1 + ((nblk >> epbs) >> epbs);
491         int level = (dn->dn_nlevels > 1) ? 2 : 1;
492
493         while (level++ < DN_MAX_LEVELS) {
494             txh->txh_memory_tohold += blkcnt << dn->dn_inblkshift;
495             blkcnt = 1 + (blkcnt >> epbs);
496         }
497         ASSERT(blkcnt <= dn->dn_nblkptr);
498     }
499
500     lastblk = blkid + nblk - 1;
501     while (nblk) {
502         dmu_buf_impl_t *dbuf;
503         uint64_t ibyte, new_blkid;
504         int epb = 1 << epbs;
505         int err, i, blkoff, tochk;
506         blkptr_t *bp;
507
508         ibyte = blkid << dn->dn_datblkshift;
509         err = dnode_next_offset(dn,
510             DNODE_FIND_HAVELOCK, &ibyte, 2, 1, 0);
511         new_blkid = ibyte >> dn->dn_datblkshift;
512         if (err == ESRCH) {
513             skipped += (lastblk >> epbs) - (blkid >> epbs) + 1;
514             break;
515         }
516         if (err) {
517             txh->txh_tx->tx_err = err;
518             break;
519         }
520         if (new_blkid > lastblk) {
521             skipped += (lastblk >> epbs) - (blkid >> epbs) + 1;
522             break;
523         }
524         if (new_blkid > blkid) {
525             ASSERT((new_blkid >> epbs) > (blkid >> epbs));
526             skipped += (new_blkid >> epbs) - (blkid >> epbs) - 1;
527             nblk -= new_blkid - blkid;
528             blkid = new_blkid;
529         }
530         blkoff = P2PHASE(blkid, epb);
531         tochk = MIN(epb - blkoff, nblk);
532
533         err = dbuf_hold_impl(dn, 1, blkid >> epbs, FALSE, FTAG, &dbuf);
534         if (err) {
535             txh->txh_tx->tx_err = err;
536             break;
537         }
538     }
```

```

523         }
525     txh->txh_memory_tohold += dbuf->db.db_size;
527
528     /*
529      * We don't check memory_tohold against DMU_MAX_ACCESS because
530      * memory_tohold is an over-estimation (especially the >Ll
531      * indirect blocks), so it could fail. Callers should have
532      * already verified that they will not be holding too much
533      * memory.
534     */
535
536     err = dbuf_read(dbuf, NULL, DB_RF_HAVESTRUCT | DB_RF_CANFAIL);
537     if (err != 0) {
538         txh->txh_tx->tx_err = err;
539         dbuf_rele(dbuf, FTAG);
540         break;
541     }
542
543     bp = dbuf->db.db_data;
544     bp += blkoff;
545
546     for (i = 0; i < tochk; i++) {
547         if (ds1_dataset_block_freeable(ds, &bp[i],
548                                       bp[i].blk_birth)) {
549             dprintf_bp(&bp[i], "can free old%u", "");
550             space += bp_get_dsize(spa, &bp[i]);
551         }
552         unref += BP_GET_ASIZE(bp);
553     }
554     dbuf_rele(dbuf, FTAG);
555
556 #endif /* ! codereview */
557     blkid += tochk;
558     nblkks -= tochk;
559 }
560 rw_exit(&dn->dn_struct_rwlock);
561
562 /*
563  * Add in memory requirements of higher-level indirects.
564  * This assumes a worst-possible scenario for dn_nlevels and a
565  * worst-possible distribution of ll-blocks over the region to free.
566  */
567 {
568     uint64_t blkcnt = 1 + ((10span >> epbs) >> epbs);
569     int level = 2;
570
571     /*
572      * Here we don't use DN_MAX_LEVEL, but calculate it with the
573      * given datablkshift and indblkshift. This makes the
574      * difference between 19 and 8 on large files.
575     */
576     int maxlevel = 2 + (DN_MAX_OFFSET_SHIFT - dn->dn_datablkshift) /
577                     (dn->dn_indblkshift - SPA_BLKPTRSHIFT);
578
579     while (level++ < maxlevel) {
580         txh->txh_memory_tohold += MIN(blkcnt, (nblkks >> epbs))
581             << dn->dn_indblkshift;
582         blkcnt = 1 + (blkcnt >> epbs);
583     }
584
585 #endif /* ! codereview */
586     /* account for new level 1 indirect blocks that might show up */
587     if (skipped > 0) {
588         txh->txh_fudge += skipped << dn->dn_indblkshift;

```

```

589     skipped = MIN(skipped, DMU_MAX_DELETEBLKCNT >> epbs);
590     txh->txh_memory_tohold += skipped << dn->dn_indblkshift;
591
592     txh->txh_space_tofree += space;
593     txh->txh_space_tounref += unref;
594 }
595
596 void
597 dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off, uint64_t len)
598 {
599     dmu_tx_hold_t *txh;
600     dnode_t *dn;
601     uint64_t start, end, i;
602     int err, shift;
603     zio_t *zio;
604
605     ASSERT(tx->tx_txg == 0);
606
607     txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
608                                   object, THT_FREE, off, len);
609     if (txh == NULL)
610         return;
611     dn = txh->txh_dnode;
612
613     /* first block */
614     if (off != 0)
615         dmu_tx_count_write(txh, off, 1);
616     /* last block */
617     if (len != DMU_OBJECT_END)
618         dmu_tx_count_write(txh, off+len, 1);
619
620     dmu_tx_count_dnode(txh);
621
622     if (off >= (dn->dn_maxblkid+1) * dn->dn_datablkksz)
623         return;
624     if (len == DMU_OBJECT_END)
625         len = (dn->dn_maxblkid+1) * dn->dn_datablkksz - off;
626
627     /*
628      * For i/o error checking, read the first and last level-0
629      * blocks, and all the level-1 blocks. The above count_write's
630      * have already taken care of the level-0 blocks.
631     */
632     if (dn->dn_nlevels > 1) {
633         shift = dn->dn_datablkshift + dn->dn_indblkshift -
634                 SPA_BLKPTRSHIFT;
635         start = off >> shift;
636         end = dn->dn_datablkshift ? ((off+len) >> shift) : 0;
637
638         zio = zio_root(tx->tx_pool->dp_spa,
639                        NULL, NULL, ZIO_FLAG_CANFAIL);
640         for (i = start; i <= end; i++) {
641             uint64_t ibyte = i << shift;
642             err = dnode_next_offset(dn, 0, &ibyte, 2, 1, 0);
643             if (err == ESRCH)
644                 break;
645             if (err) {
646                 tx->tx_err = err;
647                 return;
648             }
649
650             err = dmu_tx_check_iocerr(zio, dn, 1, i);
651             if (err) {
652                 tx->tx_err = err;
653                 return;
654             }

```

```

655         }
656     }
657     err = zio_wait(zio);
658     if (err) {
659         tx->tx_err = err;
660         return;
661     }
662 }
664 dmu_tx_count_free(txh, off, len);
665 }

667 void
668 dmu_tx_hold_zap(dmu_tx_t *tx, uint64_t object, int add, const char *name)
669 {
670     dmu_tx_hold_t *txh;
671     dnode_t *dn;
672     uint64_t nblocks;
673     int epbs, err;
674
675     ASSERT(tx->tx_txg == 0);
676
677     txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
678                                   object, THT_ZAP, add, (uintptr_t)name);
679     if (txh == NULL)
680         return;
681     dn = txh->txh_dnode;
682
683     dmu_tx_count_dnode(txh);
684
685     if (dn == NULL) {
686         /*
687          * We will be able to fit a new object's entries into one leaf
688          * block. So there will be at most 2 blocks total,
689          * including the header block.
690         */
691         dmu_tx_count_write(txh, 0, 2 << fzap_default_block_shift);
692         return;
693     }
694
695     ASSERT3P(DMU_OT_BYTESWAP(dn->dn_type), ==, DMU_BSWAP_ZAP);
696
697     if (dn->dn_maxblkid == 0 && !add) {
698         blkptr_t *bp;
699
700         /*
701          * If there is only one block (i.e. this is a micro-zap)
702          * and we are not adding anything, the accounting is simple.
703         */
704         err = dmu_tx_check_ioerr(NULL, dn, 0, 0);
705         if (err) {
706             tx->tx_err = err;
707             return;
708         }
709
710         /*
711          * Use max block size here, since we don't know how much
712          * the size will change between now and the dbuf dirty call.
713         */
714         bp = &dn->dn_phys->dn_blkptr[0];
715         if (dsl_dataset_block_freeable(dn->dn_objset->os_dsl_dataset,
716                                       bp, bp->blk_birth))
717             txh->txh_space_tooverwrite += SPA_MAXBLOCKSIZE;
718         else
719             txh->txh_space_towrite += SPA_MAXBLOCKSIZE;
720         if (!BP_IS_HOLE(bp))
721             txh->txh_space_tounref += SPA_MAXBLOCKSIZE;
722     }
723 }
724
725     txh->txh_space_tounref += SPA_MAXBLOCKSIZE;
726     return;
727 }
728
729 if (dn->dn_maxblkid > 0 && name) {
730     /*
731      * access the name in this fat-zap so that we'll check
732      * for i/o errors to the leaf blocks, etc.
733      */
734     err = zap_lookup(dn->dn_objset, dn->dn_object, name,
735                      8, 0, NULL);
736     if (err == EIO) {
737         tx->tx_err = err;
738         return;
739     }
740
741     err = zap_count_write(dn->dn_objset, dn->dn_object, name, add,
742                           &txh->txh_space_towrite, &txh->txh_space_tooverwrite);
743
744     /*
745      * If the modified blocks are scattered to the four winds,
746      * we'll have to modify an indirect twig for each.
747      */
748     epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
749     for (nblocks = dn->dn_maxblkid >> epbs; nblocks != 0; nblocks >>= epbs)
750         if (dn->dn_objset->os_dsl_dataset->ds_phys->ds_prev_snap_obj)
751             txh->txh_space_towrite += 3 << dn->dn_indblkshift;
752         else
753             txh->txh_space_tooverwrite += 3 << dn->dn_indblkshift;
754 }
755 }
756
757 void
758 dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object)
759 {
760     dmu_tx_hold_t *txh;
761
762     ASSERT(tx->tx_txg == 0);
763
764     txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
765                                   object, THT_BONUS, 0, 0);
766     if (txh)
767         dmu_tx_count_dnode(txh);
768
769     txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
770                                   DMU_NEW_OBJECT, THT_SPACE, space, 0);
771
772     txh->txh_space_towrite += space;
773
774     int
775     dmu_tx_holds(dmu_tx_t *tx, uint64_t object)
776     {
777         dmu_tx_hold_t *txh;
778         int holds = 0;
779
780         /*
781          * By asserting that the tx is assigned, we're counting the
782          * number of dn_tx_holds, which is the same as the number of
783         */
784     }
785 }
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
277
```

new/usr/src/uts/common/fs/zfs/dmu_tx.c

7

```

878     * dn_holds. Otherwise, we'd be counting dn_holds, but
879     * dn_tx_holds could be 0.
880     */
881     ASSERT(tx->tx_txg != 0);

792     /* if (tx->tx_anyobj == TRUE) */
793     /* return (0); */

795     for (txh = list_head(&tx->tx_holds); txh;
796         txh = list_next(&tx->tx_holds, txh)) {
797         if (txh->txh_dnode && txh->txh_dnode->dn_object == object)
798             holds++;
799     }

801     return (holds);
802 }

804 #ifdef ZFS_DEBUG
805 void
806 dmu_tx_dirty_buf(dmu_tx_t *tx, dmu_buf_impl_t *db)
807 {
808     dmu_tx_hold_t *txh;
809     int match_object = FALSE, match_offset = FALSE;
810     dnode_t *dn;

812     DB_DNODE_ENTER(db);
813     dn = DB_DNODE(db);
814     ASSERT(tx->tx_txg != 0);
815     ASSERT(tx->tx_objset == NULL || dn->dn_objset == tx->tx_objset);
816     ASSERT3U(dn->dn_object, ==, db->db.db_object);

818     if (tx->tx_anyobj) {
819         DB_DNODE_EXIT(db);
820         return;
821     }

823     /* XXX No checking on the meta dnode for now */
824     if (db->db.db_object == DMU_META_DNODE_OBJECT) {
825         DB_DNODE_EXIT(db);
826         return;
827     }

829     for (txh = list_head(&tx->tx_holds); txh;
830         txh = list_next(&tx->tx_holds, txh)) {
831         ASSERT(dn == NULL || dn->dn_assigned_txg == tx->tx_txg);
832         if (txh->txh_dnode == dn && txh->txh_type != THT_NEWORDJECT)
833             match_object = TRUE;
834         if (txh->txh_dnode == NULL || txh->txh_dnode == dn) {
835             int datablkshift = dn->dn_datblkshift ?
836                 dn->dn_datblkshift : SPA_MAXBLOCKSHIFT;
837             int epbs = dn->dn_inblkshift - SPA_BLKPTRSHIFT;
838             int shift = datablkshift + epbs * db->db_level;
839             uint64_t beginblk = shift >= 64 ? 0 :
840                 (txh->txh_arg1 >> shift);
841             uint64_t endblk = shift >= 64 ? 0 :
842                 ((txh->txh_arg1 + txh->txh_arg2 - 1) >> shift);
843             uint64_t blkid = db->db_blkid;

845             /* XXX txh_arg2 better not be zero... */
846             dprintf("found txh type %x beginblk=%llx endblk=%llx\n",
847                   txh->txh_type, beginblk, endblk);

850             switch (txh->txh_type) {
851             case THT_WRITE:
852                 if (blkid >= beginblk && blkid <= endblk)

```

new/usr/src/uts/common/fs/zfs/dmu_tx.c

```

        match_offset = TRUE;
    /*
     * We will let this hold work for the bonus
     * or spill buffer so that we don't need to
     * hold it when creating a new object.
     */
    if (blkid == DMU_BONUS_BLKID ||
        blkid == DMU_SPILL_BLKID)
        match_offset = TRUE;
    /*
     * They might have to increase nlevels,
     * thus dirtying the new TLIBs. Or the
     * might have to change the block size,
     * thus dirying the new lvl=0 blk=0.
     */
    if (blkid == 0)
        match_offset = TRUE;
    break;
case THT_FREE:
/*
 * We will dirty all the level 1 blocks in
 * the free range and perhaps the first and
 * last level 0 block.
 */
if (blkid >= beginblk && (blkid <= endblk ||
    txh->txh_arg2 == DMU_OBJECT_END))
    match_offset = TRUE;
break;
case THT_SPILL:
    if (blkid == DMU_SPILL_BLKID)
        match_offset = TRUE;
    break;
case THT_BONUS:
    if (blkid == DMU_BONUS_BLKID)
        match_offset = TRUE;
    break;
case THT_ZAP:
    match_offset = TRUE;
    break;
case THT_NEWSOBJECT:
    match_object = TRUE;
    break;
default:
    ASSERT(!"bad txh_type");
}

match_object && match_offset) {
    DB_DNODE_EXIT(db);
    return;
}

db);
long dbuf obj=%llx lvl=%u blkid=%llx but not tx_held\n",
ng_t)db->db.db_object, db->db_level,
ng_t)db->db_blkid);

tx_t *tx, uint64_t txg_how)

*txh;
tx->tx_pool->dp_spa;
try, asize, fsize, usize;
ite, tofree, tooverwrite, tounref, tohold, fudge;

```

```

919     ASSERT3U(tx->tx_txg, ==, 0);
921
922     if (tx->tx_err)
923         return (tx->tx_err);
924
925     if (spa_suspended(spa)) {
926         /*
927         * If the user has indicated a blocking failure mode
928         * then return ERESTART which will block in dmu_tx_wait().
929         * Otherwise, return EIO so that an error can get
930         * propagated back to the VOP calls.
931         *
932         * Note that we always honor the txg_how flag regardless
933         * of the failuremode setting.
934         */
935     if (spa_get_failmode(spa) == ZIO_FAILURE_MODE_CONTINUE &&
936         txg_how != TXG_WAIT)
937         return (EIO);
938
939     return (ERESTART);
940 }
941
942 tx->tx_txg = txg_hold_open(tx->tx_pool, &tx->tx_txgh);
943 tx->tx_needassign_txh = NULL;
944
945 /*
946 * NB: No error returns are allowed after txg_hold_open, but
947 * before processing the dnode holds, due to the
948 * dmu_tx_unassign() logic.
949 */
950
951 towrite = tofree = tooverwrite = tounref = tohold = fudge = 0;
952 for (txh = list_head(&tx->tx_holds); txh;
953     txh = list_next(&tx->tx_holds, txh)) {
954     dnode_t *dn = txh->txh_dnode;
955     if (dn != NULL) {
956         mutex_enter(&dn->dn_mtx);
957         if (dn->dn_assigned_txg == tx->tx_txg - 1) {
958             mutex_exit(&dn->dn_mtx);
959             tx->tx_needassign_txh = txh;
960             return (ERESTART);
961         }
962         if (dn->dn_assigned_txg == 0)
963             dn->dn_assigned_txg = tx->tx_txg;
964         ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);
965         (void) refcount_add(&dn->dn_tx_holds, tx);
966         mutex_exit(&dn->dn_mtx);
967     }
968     towrite += txh->txh_space_towrite;
969     tofree += txh->txh_space_tofree;
970     tooverwrite += txh->txh_space_tooverwrite;
971     tounref += txh->txh_space_tounref;
972     tohold += txh->txh_memory_tohold;
973     fudge += txh->txh_fudge;
974 }
975 /*
976 * NB: This check must be after we've held the dnodes, so that
977 * the dmu_tx_unassign() logic will work properly
978 */
979 if (txg_how >= TXG_INITIAL && txg_how != tx->tx_txg)
980     return (ERESTART);
981
982 /*
983 * If a snapshot has been taken since we made our estimates,
984 * assume that we won't be able to free or overwrite anything.

```

```

985     /*
986     if (tx->tx_objset &&
987         dsl_dataset_prev_snap_txg(tx->tx_objset->os_dsl_dataset) >
988         tx->tx_lastsnap_txg) {
989         towrite += tooverwrite;
990         tooverwrite = tofree = 0;
991     }
992
993     /* needed allocation: worst-case estimate of write space */
994     asize = spa_get_asize(tx->tx_pool->dp_spa, towrite + tooverwrite);
995     /* freed space estimate: worst-case overwrite + free estimate */
996     fsize = spa_get_asize(tx->tx_pool->dp_spa, tooverwrite) + tofree;
997     /* convert unrefd space to worst-case estimate */
998     usize = spa_get_asize(tx->tx_pool->dp_spa, tounref);
999     /* calculate memory footprint estimate */
1000    memory = towrite + tooverwrite + tohold;
1001
1002 #ifdef ZFS_DEBUG
1003 /*
1004  * Add in 'tohold' to account for our dirty holds on this memory
1005  * XXX - the "fudge" factor is to account for skipped blocks that
1006  * we missed because dnode_next_offset() misses in-core-only blocks.
1007  */
1008 tx->tx_space_towrite = asize +
1009     spa_get_asize(tx->tx_pool->dp_spa, tohold + fudge);
1010 tx->tx_space_tofree = tofree;
1011 tx->tx_space_tooverwrite = tooverwrite;
1012 tx->tx_space_tounref = tounref;
1013#endif
1014
1015 if (tx->tx_dir && asize != 0) {
1016     int err = dsl_dir_tempreserve_space(tx->tx_dir, memory,
1017                                         asize, fsize, usize, &tx->tx_tempreserve_cookie, tx);
1018     if (err)
1019         return (err);
1020 }
1021
1022 return (0);
1023}
1024
1025 static void
1026 dmu_tx_unassign(dmu_tx_t *tx)
1027 {
1028     dmu_tx_hold_t *txh;
1029
1030     if (tx->tx_txg == 0)
1031         return;
1032
1033     txg_rele_to_quiesce(&tx->tx_txgh);
1034
1035     for (txh = list_head(&tx->tx_holds); txh != tx->tx_needassign_txh;
1036         txh = list_next(&tx->tx_holds, txh)) {
1037         dnode_t *dn = txh->txh_dnode;
1038
1039         if (dn == NULL)
1040             continue;
1041         mutex_enter(&dn->dn_mtx);
1042         ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);
1043
1044         if (refcount_remove(&dn->dn_tx_holds, tx) == 0) {
1045             dn->dn_assigned_txg = 0;
1046             cv_broadcast(&dn->dn_notxholds);
1047         }
1048         mutex_exit(&dn->dn_mtx);
1049     }

```

```

1051     txg_rele_to_sync(&tx->tx_txgh);
1053     tx->tx_lasttried_txg = tx->tx_txg;
1054     tx->tx_txg = 0;
1055 }
1057 /*
1058 * Assign tx to a transaction group. txg_how can be one of:
1059 *
1060 * (1) TXG_WAIT. If the current open txg is full, waits until there's
1061 * a new one. This should be used when you're not holding locks.
1062 * If will only fail if we're truly out of space (or over quota).
1063 *
1064 * (2) TXG_NOWAIT. If we can't assign into the current open txg without
1065 * blocking, returns immediately with ERESTART. This should be used
1066 * whenever you're holding locks. On an ERESTART error, the caller
1067 * should drop locks, do a dmu_tx_wait(tx), and try again.
1068 *
1069 * (3) A specific txg. Use this if you need to ensure that multiple
1070 * transactions all sync in the same txg. Like TXG_NOWAIT, it
1071 * returns ERESTART if it can't assign you into the requested txg.
1072 */
1073 int
1074 dmu_tx_assign(dmu_tx_t *tx, uint64_t txg_how)
1075 {
1076     int err;
1077
1078     ASSERT(tx->tx_txg == 0);
1079     ASSERT(txg_how != 0);
1080     ASSERT(!dsl_pool_sync_context(tx->tx_pool));
1081
1082     while ((err = dmu_tx_try_assign(tx, txg_how)) != 0) {
1083         dmu_tx_unassign(tx);
1084
1085         if (err != ERESTART || txg_how != TXG_WAIT)
1086             return (err);
1087
1088         dmu_tx_wait(tx);
1089     }
1090
1091     txg_rele_to_quiesce(&tx->tx_txgh);
1092
1093     return (0);
1094 }
1095 void
1096 dmu_tx_wait(dmu_tx_t *tx)
1097 {
1098     spa_t *spa = tx->tx_pool->dp_spa;
1099
1100     ASSERT(tx->tx_txg == 0);
1101
1102     /*
1103      * It's possible that the pool has become active after this thread
1104      * has tried to obtain a tx. If that's the case then his
1105      * tx.lasttried_txg would not have been assigned.
1106     */
1107
1108     if (spa_suspended(spa) || tx->tx_lasttried_txg == 0) {
1109         txg_wait_synced(tx->tx_pool, spa_last_synced_txg(spa) + 1);
1110     } else if (tx->tx_needassign_txh) {
1111         dnode_t *dn = tx->tx_needassign_txh->txh_dnode;
1112
1113         mutex_enter(&dn->dn_mtx);
1114         while (dn->dn_assigned_txg == tx->tx_lasttried_txg - 1)
1115             cv_wait(&dn->dn_notxholds, &dn->dn_mtx);
1116         mutex_exit(&dn->dn_mtx);
1117     }
1118 }

```

```

1119     tx->tx_needassign_txh = NULL;
1120 }
1121 }
1122 }
1123 void
1124 dmu_tx_willuse_space(dmu_tx_t *tx, int64_t delta)
1125 {
1126 #ifdef ZFS_DEBUG
1127     if (tx->tx_dir == NULL || delta == 0)
1128         return;
1129
1130     if (delta > 0) {
1131         ASSERT3U(refcount_count(&tx->tx_space_written) + delta, <=,
1132                  tx->tx_space_towrite);
1133         (void) refcount_add_many(&tx->tx_space_written, delta, NULL);
1134     } else {
1135         (void) refcount_add_many(&tx->tx_space_freed, -delta, NULL);
1136     }
1137 #endif
1138 }
1139
1140 void
1141 dmu_tx_commit(dmu_tx_t *tx)
1142 {
1143     dmu_tx_hold_t *txh;
1144
1145     ASSERT(tx->tx_txg != 0);
1146
1147     while (txh = list_head(&tx->tx_holds)) {
1148         dnode_t *dn = txh->txh_dnode;
1149
1150         list_remove(&tx->tx_holds, txh);
1151         kmem_free(txh, sizeof (dmu_tx_hold_t));
1152         if (dn == NULL)
1153             continue;
1154         mutex_enter(&dn->dn_mtx);
1155         ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);
1156
1157         if (refcount_remove(&dn->dn_tx_holds, tx) == 0) {
1158             dn->dn_assigned_txg = 0;
1159             cv_broadcast(&dn->dn_notxholds);
1160         }
1161         mutex_exit(&dn->dn_mtx);
1162         dnode_rele(dn, tx);
1163     }
1164
1165     if (tx->tx_tempreserve_cookie)
1166         dsl_dir_tempreserve_clear(tx->tx_tempreserve_cookie, tx);
1167
1168     if (!list_is_empty(&tx->tx_callbacks))
1169         txg_register_callbacks(&tx->tx_txgh, &tx->tx_callbacks);
1170
1171     if (tx->tx_anyobj == FALSE)
1172         txg_rele_to_sync(&tx->tx_txgh);
1173
1174     list_destroy(&tx->tx_callbacks);
1175     list_destroy(&tx->tx_holds);
1176 #ifdef ZFS_DEBUG
1177     dprintf("towrite=%llu written=%llu tofree=%llu freed=%llu\n",
1178            tx->tx_space_towrite, refcount_count(&tx->tx_space_written),
1179            tx->tx_space_tofree, refcount_count(&tx->tx_space_freed));
1180     refcount_destroy_many(&tx->tx_space_written,
1181                           refcount_count(&tx->tx_space_written));
1182     refcount_destroy_many(&tx->tx_space_freed,
1183                           refcount_count(&tx->tx_space_freed));
1184 #endif
1185 }

```

```

1183     refcount_count(&tx->tx_space_freed));
1184 #endif
1185     kmem_free(tx, sizeof (dmu_tx_t));
1186 }
1187
1188 void
1189 dmu_tx_abort(dmu_tx_t *tx)
1190 {
1191     dmu_tx_hold_t *txh;
1192
1193     ASSERT(tx->tx_txg == 0);
1194
1195     while (txh = list_head(&tx->tx_holds)) {
1196         dnode_t *dn = txh->txh_dnode;
1197
1198         list_remove(&tx->tx_holds, txh);
1199         kmem_free(txh, sizeof (dmu_tx_hold_t));
1200         if (dn != NULL)
1201             dnode_rele(dn, tx);
1202     }
1203
1204     /*
1205      * Call any registered callbacks with an error code.
1206      */
1207     if (!list_is_empty(&tx->tx_callbacks))
1208         dmu_tx_do_callbacks(&tx->tx_callbacks, ECANCELED);
1209
1210     list_destroy(&tx->tx_callbacks);
1211     list_destroy(&tx->tx_holds);
1212 #ifdef ZFS_DEBUG
1213     refcount_destroy_many(&tx->tx_space_written,
1214                           refcount_count(&tx->tx_space_written));
1215     refcount_destroy_many(&tx->tx_space_freed,
1216                           refcount_count(&tx->tx_space_freed));
1217 #endif
1218     kmem_free(tx, sizeof (dmu_tx_t));
1219 }
1220
1221 uint64_t
1222 dmu_tx_get_txg(dmu_tx_t *tx)
1223 {
1224     ASSERT(tx->tx_txg != 0);
1225     return (tx->tx_txg);
1226 }
1227
1228 void
1229 dmu_tx_callback_register(dmu_tx_t *tx, dmu_tx_callback_func_t *func, void *data)
1230 {
1231     dmu_tx_callback_t *dcb;
1232
1233     dcb = kmem_alloc(sizeof (dmu_tx_callback_t), KM_SLEEP);
1234
1235     dcb->dcb_func = func;
1236     dcb->dcb_data = data;
1237
1238     list_insert_tail(&tx->tx_callbacks, dcb);
1239 }
1240
1241 /*
1242  * Call all the commit callbacks on a list, with a given error code.
1243  */
1244 void
1245 dmu_tx_do_callbacks(list_t *cb_list, int error)
1246 {
1247     dmu_tx_callback_t *dcb;

```

```

1249     while (dcb = list_head(cb_list)) {
1250         list_remove(cb_list, dcb);
1251         dcb->dcb_func(dcb->dcb_data, error);
1252         kmem_free(dcb, sizeof (dmu_tx_callback_t));
1253     }
1254
1255     /*
1256      * Interface to hold a bunch of attributes.
1257      * used for creating new files.
1258      * attrsize is the total size of all attributes
1259      * to be added during object creation
1260      * For updating/adding a single attribute dmu_tx_hold_sa() should be used.
1261      */
1262
1263     /*
1264      * hold necessary attribute name for attribute registration.
1265      * should be a very rare case where this is needed. If it does
1266      * happen it would only happen on the first write to the file system.
1267      */
1268
1269 static void
1270 dmu_tx_sa_registration_hold(sa_os_t *sa, dmu_tx_t *tx)
1271 {
1272     int i;
1273
1274     if (!sa->sa_need_attr_registration)
1275         return;
1276
1277     for (i = 0; i != sa->sa_num_attrs; i++) {
1278         if (!sa->sa_attr_table[i].sa_registered) {
1279             if (sa->sa_reg_attr_obj)
1280                 dmu_tx_hold_zap(tx, sa->sa_reg_attr_obj,
1281                                 B_TRUE, sa->sa_attr_table[i].sa_name);
1282             else
1283                 dmu_tx_hold_zap(tx, DMU_NEW_OBJECT,
1284                                 B_TRUE, sa->sa_attr_table[i].sa_name);
1285         }
1286     }
1287 }
1288
1289 void
1290 dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object)
1291 {
1292     dnode_t *dn;
1293     dmu_tx_hold_t *txh;
1294
1295     txh = dmu_tx_hold_object_impl(tx, tx->tx_objset, object,
1296                                  THT_SPILL, 0, 0);
1297
1298     dn = txh->txh_dnode;
1299
1300     if (dn == NULL)
1301         return;
1302
1303     /* If blkptr doesn't exist then add space to towrite */
1304     if (!(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR)) {
1305         txh->txh_space_towrite += SPA_MAXBLOCKSIZE;
1306     } else {
1307         blkptr_t *bp;
1308
1309         bp = &dn->dn_phys->dn_spill;
1310         if (ds1_dataset_block_freeable(dn->dn_objset->os_dsl_dataset,
1311                                       bp, bp->blk_birth))
1312             txh->txh_space_tooverwrite += SPA_MAXBLOCKSIZE;
1313
1314

```

```

1315         else
1316             txh->txh_space_towrite += SPA_MAXBLOCKSIZE;
1317             if (!BP_IS_HOLE(bp))
1318                 txh->txh_space_tounref += SPA_MAXBLOCKSIZE;
1319     }
1320 }

1322 void
1323 dmu_tx_hold_sa_create(dmu_tx_t *tx, int attrszie)
1324 {
1325     sa_os_t *sa = tx->tx_objset->os_sa;
1326     dmu_tx_hold_bonus(tx, DMU_NEW_OBJECT);
1327
1328     if (tx->tx_objset->os_sa->sa_master_obj == 0)
1329         return;
1330
1331     if (tx->tx_objset->os_sa->sa_layout_attr_obj)
1332         dmu_tx_hold_zap(tx, sa->sa_layout_attr_obj, B_TRUE, NULL);
1333     else {
1334         dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_LAYOUTS);
1335         dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_REGISTRY);
1336         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1337         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1338     }
1339
1340     dmu_tx_sa_registration_hold(sa, tx);
1341
1342     if (attrszie <= DN_MAX_BONUSLEN && !sa->sa_force_spill)
1343         return;
1344
1345     (void) dmu_tx_hold_object_impl(tx, tx->tx_objset, DMU_NEW_OBJECT,
1346         THT_SPILL, 0, 0);
1347 }
1348

1349 /*
1350 * Hold SA attribute
1351 *
1352 */
1353 * dmu_tx_hold_sa(dmu_tx_t *tx, sa_handle_t *, attribute, add, size)
1354 *
1355 * variable_size is the total size of all variable sized attributes
1356 * passed to this function. It is not the total size of all
1357 * variable size attributes that *may* exist on this object.
1358 */
1359 void
1360 dmu_tx_hold_sa(dmu_tx_t *tx, sa_handle_t *hdl, boolean_t may_grow)
1361 {
1362     uint64_t object;
1363     sa_os_t *sa = tx->tx_objset->os_sa;
1364
1365     ASSERT(hdl != NULL);
1366
1367     object = sa_handle_object(hdl);
1368
1369     dmu_tx_hold_bonus(tx, object);
1370
1371     if (tx->tx_objset->os_sa->sa_master_obj == 0)
1372         return;
1373
1374     if (tx->tx_objset->os_sa->sa_reg_attr_obj == 0 ||
1375         tx->tx_objset->os_sa->sa_layout_attr_obj == 0) {
1376         dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_LAYOUTS);
1377         dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_REGISTRY);
1378         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1379         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1380     }

```

```

1382     dmu_tx_sa_registration_hold(sa, tx);
1383
1384     if (may_grow && tx->tx_objset->os_sa->sa_layout_attr_obj)
1385         dmu_tx_hold_zap(tx, sa->sa_layout_attr_obj, B_TRUE, NULL);
1386
1387     if (sa->sa_force_spill || may_grow || hdl->sa_spill) {
1388         ASSERT(tx->tx_txg == 0);
1389         dmu_tx_hold_spill(tx, object);
1390     } else {
1391         dm_buf_impl_t *db = (dm_buf_impl_t *)hdl->sa_bonus;
1392         dnode_t *dn;
1393
1394         DB_DNODE_ENTER(db);
1395         dn = DB_DNODE(db);
1396         if (dn->dn_have_spill) {
1397             ASSERT(tx->tx_txg == 0);
1398             dmu_tx_hold_spill(tx, object);
1399         }
1400     }
1401     DB_DNODE_EXIT(db);
1402 }

```