```
**********************************************************
  160377 Fri Aug  2 13:20:50 2013
new/usr/src/cmd/ztest/ztest.c
3955 ztest failure: assertion refcount_count(&tx->tx_space_written) + delta <= t
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Dan Kimmel <dan.kimmel@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

3498 /*
3499  * Verify that dmu_{read,write} work as expected.
3500  */
3501 void
3502 ztest_dmu_read_write(ztest_ds_t *zd, uint64_t id)
3503 {
3504         objset_t *os = zd->zd_os;
3505         ztest_od_t od[2];
3506         dmu_tx_t *tx;
3507         int i, freeit, error;
3508         uint64_t n, s, txg;
3509         bufwad_t *packbuf, *bigbuf, *pack, *bigH, *bigT;
3510         uint64_t packobj, packoff, packsize, bigobj, bigoff, bigsize;
3511         uint64_t chunksize = (1000 + ztest_random(1000)) * sizeof (uint64_t);
3512         uint64_t regions = 997;
3513         uint64_t stride = 123456789ULL;
3514         uint64_t width = 40;
3515         int free_percent = 5;

3517         /*
3518          * This test uses two objects, packobj and bigobj, that are always
3519          * updated together (i.e. in the same tx) so that their contents are
3520          * in sync and can be compared.  Their contents relate to each other
3521          * in a simple way: packobj is a dense array of 'bufwad' structures,
3522          * while bigobj is a sparse array of the same bufwads.  Specifically,
3523          * for any index n, there are three bufwads that should be identical:
3524          *
3525          *      packobj, at offset n * sizeof (bufwad_t)
3526          *      bigobj, at the head of the nth chunk
3527          *      bigobj, at the tail of the nth chunk
3528          *
3529          * The chunk size is arbitrary. It doesn't have to be a power of two,
3530          * and it doesn't have any relation to the object blocksize.
3531          * The only requirement is that it can hold at least two bufwads.
3532          *
3533          * Normally, we write the bufwad to each of these locations.
3534          * However, free_percent of the time we instead write zeroes to
3535          * packobj and perform a dmu_free_range() on bigobj.  By comparing
3536          * bigobj to packobj, we can verify that the DMU is correctly
3537          * tracking which parts of an object are allocated and free,
3538          * and that the contents of the allocated blocks are correct.
3539          */

3541         /*
3542          * Read the directory info.  If it's the first time, set things up.
3543          */
3544         ztest_od_init(&od[0], id, FTAG, 0, DMU_OT_UINT64_OTHER, 0, chunksize);
3545         ztest_od_init(&od[1], id, FTAG, 1, DMU_OT_UINT64_OTHER, 0, chunksize);

3547         if (ztest_object_init(zd, od, sizeof (od), B_FALSE) != 0)
3548                 return;

3550         bigobj = od[0].od_object;
3551         packobj = od[1].od_object;
3552         chunksize = od[0].od_gen;
3553         ASSERT(chunksize == od[1].od_gen);
```



```
3555         /*
3556          * Prefetch a random chunk of the big object.
3557          * Our aim here is to get some async reads in flight
3558          * for blocks that we may free below; the DMU should
3559          * handle this race correctly.
3560          */
3561         n = ztest_random(regions) * stride + ztest_random(width);
3562         s = 1 + ztest_random(2 * width - 1);
3563         dmu_prefetch(os, bigobj, n * chunksize, s * chunksize);

3565         /*
3566          * Pick a random index and compute the offsets into packobj and bigobj.
3567          */
3568         n = ztest_random(regions) * stride + ztest_random(width);
3569         s = 1 + ztest_random(width - 1);

3571         packoff = n * sizeof (bufwad_t);
3572         packsize = s * sizeof (bufwad_t);

3574         bigoff = n * chunksize;
3575         bigsize = s * chunksize;

3577         packbuf = umem_alloc(packsize, UMEM_NOFAIL);
3578         bigbuf = umem_alloc(bigsize, UMEM_NOFAIL);

3580         /*
3581          * free_percent of the time, free a range of bigobj rather than
3582          * overwriting it.
3583          */
3584         freeit = (ztest_random(100) < free_percent);

3586         /*
3587          * Read the current contents of our objects.
3588          */
3589         error = dmu_read(os, packobj, packoff, packsize, packbuf,
3590             DMU_READ_PREFETCH);
3591         ASSERT0(error);
3592         error = dmu_read(os, bigobj, bigoff, bigsize, bigbuf,
3593             DMU_READ_PREFETCH);
3594         ASSERT0(error);

3596         /*
3597          * Get a tx for the mods to both packobj and bigobj.
3598          */
3599         tx = dmu_tx_create(os);

3601         dmu_tx_hold_write(tx, packobj, packoff, packsize);

3603         if (freeit)
3604                 dmu_tx_hold_free(tx, bigobj, bigoff, bigsize);
3605         else
3606                 dmu_tx_hold_write(tx, bigobj, bigoff, bigsize);

3608         /* This accounts for setting the checksum/compression. */
3609         dmu_tx_hold_bonus(tx, bigobj);

3611         txg = ztest_tx_assign(tx, TXG_MIGHTWAIT, FTAG);
3612         if (txg == 0) {
3613                 umem_free(packbuf, packsize);
3614                 umem_free(bigbuf, bigsize);
3615                 return;
3616         }

3618         dmu_object_set_checksum(os, bigobj,
3619             (enum zio_checksum)ztest_random_dsl_prop(ZFS_PROP_CHECKSUM), tx);
```

```
3621         dmu_object_set_compress(os, bigobj,
3622             (enum zio_compress)ztest_random_dsl_prop(ZFS_PROP_COMPRESSION), tx);

3624         /*
3625          * For each index from n to n + s, verify that the existing bufwad
3626          * in packobj matches the bufwads at the head and tail of the
3627          * corresponding chunk in bigobj.  Then update all three bufwads
3628          * with the new values we want to write out.
3629          */
3630         for (i = 0; i < s; i++) {
3631                 /* LINTED */
3632                 pack = (bufwad_t *)((char *)packbuf + i * sizeof (bufwad_t));
3633                 /* LINTED */
3634                 bigH = (bufwad_t *)((char *)bigbuf + i * chunksize);
3635                 /* LINTED */
3636                 bigT = (bufwad_t *)((char *)bigH + chunksize) - 1;

3638                 ASSERT((uintptr_t)bigH - (uintptr_t)bigbuf < bigsize);
3639                 ASSERT((uintptr_t)bigT - (uintptr_t)bigbuf < bigsize);

3641                 if (pack->bw_txg > txg)
3642                         fatal(0, "future leak: got %llx, open txg is %llx",
3643                             pack->bw_txg, txg);

3645                 if (pack->bw_data != 0 && pack->bw_index != n + i)
3646                         fatal(0, "wrong index: got %llx, wanted %llx+%llx",
3647                             pack->bw_index, n, i);

3649                 if (bcmp(pack, bigH, sizeof (bufwad_t)) != 0)
3650                         fatal(0, "pack/bigH mismatch in %p/%p", pack, bigH);

3652                 if (bcmp(pack, bigT, sizeof (bufwad_t)) != 0)
3653                         fatal(0, "pack/bigT mismatch in %p/%p", pack, bigT);

3655                 if (freeit) {
3656                         bzero(pack, sizeof (bufwad_t));
3657                 } else {
3658                         pack->bw_index = n + i;
3659                         pack->bw_txg = txg;
3660                         pack->bw_data = 1 + ztest_random(-2ULL);
3661                 }
3662                 *bigH = *pack;
3663                 *bigT = *pack;
3664         }

3666         /*
3667          * We've verified all the old bufwads, and made new ones.
3668          * Now write them out.
3669          */
3670         dmu_write(os, packobj, packoff, packsize, packbuf, tx);

3672         if (freeit) {
3673                 if (ztest_opts.zo_verbose >= 7) {
3674                         (void) printf("freeing offset %llx size %llx"
3675                             " txg %llx\n",
3676                             (u_longlong_t)bigoff,
3677                             (u_longlong_t)bigsize,
3678                             (u_longlong_t)txg);
3679                 }
3680                 VERIFY(0 == dmu_free_range(os, bigobj, bigoff, bigsize, tx));
3681         } else {
3682                 if (ztest_opts.zo_verbose >= 7) {
3683                         (void) printf("writing offset %llx size %llx"
3684                             " txg %llx\n",
3685                             (u_longlong_t)bigoff,
```



```
3686                             (u_longlong_t)bigsize,
3687                             (u_longlong_t)txg);
3688                 }
3689                 dmu_write(os, bigobj, bigoff, bigsize, bigbuf, tx);
3690         }

3692         dmu_tx_commit(tx);

3694         /*
3695          * Sanity check the stuff we just wrote.
3696          */
3697         {
3698                 void *packcheck = umem_alloc(packsize, UMEM_NOFAIL);
3699                 void *bigcheck = umem_alloc(bigsize, UMEM_NOFAIL);

3701                 VERIFY(0 == dmu_read(os, packobj, packoff,
3702                     packsize, packcheck, DMU_READ_PREFETCH));
3703                 VERIFY(0 == dmu_read(os, bigobj, bigoff,
3704                     bigsize, bigcheck, DMU_READ_PREFETCH));

3706                 ASSERT(bcmp(packbuf, packcheck, packsize) == 0);
3707                 ASSERT(bcmp(bigbuf, bigcheck, bigsize) == 0);

3709                 umem_free(packcheck, packsize);
3710                 umem_free(bigcheck, bigsize);
3711         }

3713         umem_free(packbuf, packsize);
3714         umem_free(bigbuf, bigsize);
3715 }
```

unchanged_portion_omitted_

```
**********************************************************
   35999 Fri Aug  2 13:20:52 2013
new/usr/src/uts/common/fs/zfs/dmu_tx.c
3955 ztest failure: assertion refcount_count(&tx->tx_space_written) + delta <= t
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Dan Kimmel <dan.kimmel@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 417 static void
 418 dmu_tx_count_free(dmu_tx_hold_t *txh, uint64_t off, uint64_t len)
 419 {
 420         uint64_t blkid, nblks, lastblk;
 421         uint64_t space = 0, unref = 0, skipped = 0;
 422         dnode_t *dn = txh->txh_dnode;
 423         dsl_dataset_t *ds = dn->dn_objset->os_dsl_dataset;
 424         spa_t *spa = txh->txh_tx->tx_pool->dp_spa;
 425         int epbs;
 426         uint64_t l0span = 0, nl1blks = 0;

 428         if (dn->dn_nlevels == 0)
 429                 return;

 431         /*
 432          * The struct_rwlock protects us against dn_nlevels
 433          * changing, in case (against all odds) we manage to dirty &
 434          * sync out the changes after we check for being dirty.
 435          * Also, dbuf_hold_impl() wants us to have the struct_rwlock.
 436          */
 437         rw_enter(&dn->dn_struct_rwlock, RW_READER);
 438         epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
 439         if (dn->dn_maxblkid == 0) {
 440                 if (off == 0 && len >= dn->dn_datablksz) {
 441                         blkid = 0;
 442                         nblks = 1;
 443                 } else {
 444                         rw_exit(&dn->dn_struct_rwlock);
 445                         return;
 446                 }
 447         } else {
 448                 blkid = off >> dn->dn_datablkshift;
 449                 nblks = (len + dn->dn_datablksz - 1) >> dn->dn_datablkshift;

 451                 if (blkid > dn->dn_maxblkid) {
 451                 if (blkid >= dn->dn_maxblkid) {
 452                         rw_exit(&dn->dn_struct_rwlock);
 453                         return;
 454                 }
 455                 if (blkid + nblks > dn->dn_maxblkid)
 456                         nblks = dn->dn_maxblkid - blkid + 1;
 456                         nblks = dn->dn_maxblkid - blkid;

 458         }
 459         l0span = nblks;    /* save for later use to calc level > 1 overhead */
 460         if (dn->dn_nlevels == 1) {
 461                 int i;
 462                 for (i = 0; i < nblks; i++) {
 463                         blkptr_t *bp = dn->dn_phys->dn_blkptr;
 464                         ASSERT3U(blkid + i, <, dn->dn_nblkptr);
 465                         bp += blkid + i;
 466                         if (dsl_dataset_block_freeable(ds, bp, bp->blk_birth)) {
 467                                 dprintf_bp(bp, "can free old%s", "");
 468                                 space += bp_get_dsize(spa, bp);
 469                         }
 470                         unref += BP_GET_ASIZE(bp);
```



```
 471                 }
 472                 nl1blks = 1;
 473                 nblks = 0;
 474         }

 476         lastblk = blkid + nblks - 1;
 477         while (nblks) {
 478                 dmu_buf_impl_t *dbuf;
 479                 uint64_t ibyte, new_blkid;
 480                 int epb = 1 << epbs;
 481                 int err, i, blkoff, tochk;
 482                 blkptr_t *bp;

 484                 ibyte = blkid << dn->dn_datablkshift;
 485                 err = dnode_next_offset(dn,
 486                     DNODE_FIND_HAVELOCK, &ibyte, 2, 1, 0);
 487                 new_blkid = ibyte >> dn->dn_datablkshift;
 488                 if (err == ESRCH) {
 489                         skipped += (lastblk >> epbs) - (blkid >> epbs) + 1;
 490                         break;
 491                 }
 492                 if (err) {
 493                         txh->txh_tx->tx_err = err;
 494                         break;
 495                 }
 496                 if (new_blkid > lastblk) {
 497                         skipped += (lastblk >> epbs) - (blkid >> epbs) + 1;
 498                         break;
 499                 }

 501                 if (new_blkid > blkid) {
 502                         ASSERT((new_blkid >> epbs) > (blkid >> epbs));
 503                         skipped += (new_blkid >> epbs) - (blkid >> epbs) - 1;
 504                         nblks -= new_blkid - blkid;
 505                         blkid = new_blkid;
 506                 }
 507                 blkoff = P2PHASE(blkid, epb);
 508                 tochk = MIN(epb - blkoff, nblks);

 510                 err = dbuf_hold_impl(dn, 1, blkid >> epbs, FALSE, FTAG, &dbuf);
 511                 if (err) {
 512                         txh->txh_tx->tx_err = err;
 513                         break;
 514                 }

 516                 txh->txh_memory_tohold += dbuf->db.db_size;

 518                 /*
 519                  * We don't check memory_tohold against DMU_MAX_ACCESS because
 520                  * memory_tohold is an over-estimation (especially the >L1
 521                  * indirect blocks), so it could fail.  Callers should have
 522                  * already verified that they will not be holding too much
 523                  * memory.
 524                  */

 526                 err = dbuf_read(dbuf, NULL, DB_RF_HAVESTRUCT | DB_RF_CANFAIL);
 527                 if (err != 0) {
 528                         txh->txh_tx->tx_err = err;
 529                         dbuf_rele(dbuf, FTAG);
 530                         break;
 531                 }

 533                 bp = dbuf->db.db_data;
 534                 bp += blkoff;

 536                 for (i = 0; i < tochk; i++) {
```

```
537                         if (dsl_dataset_block_freeable(ds, &bp[i],
538                             bp[i].blk_birth)) {
539                                 dprintf_bp(&bp[i], "can free old%s", "");
540                                 space += bp_get_dsize(spa, &bp[i]);
541                         }
542                         unref += BP_GET_ASIZE(bp);
543                 }
544                 dbuf_rele(dbuf, FTAG);

546                 ++nl1blks;
547                 blkid += tochk;
548                 nblks -= tochk;
549         }
550         rw_exit(&dn->dn_struct_rwlock);

552         /*
553          * Add in memory requirements of higher-level indirects.
554          * This assumes a worst-possible scenario for dn_nlevels and a
555          * worst-possible distribution of l1-blocks over the region to free.
556          */
557         {
558                 uint64_t blkcnt = 1 + ((l0span >> epbs) >> epbs);
559                 int level = 2;
560                 /*
561                  * Here we don't use DN_MAX_LEVEL, but calculate it with the
562                  * given datablkshift and indblkshift. This makes the
563                  * difference between 19 and 8 on large files.
564                  */
565                 int maxlevel = 2 + (DN_MAX_OFFSET_SHIFT - dn->dn_datablkshift) /
566                     (dn->dn_indblkshift - SPA_BLKPTRSHIFT);

568                 while (level++ < maxlevel) {
569                         txh->txh_memory_tohold += MAX(MIN(blkcnt, nl1blks), 1)
570                             << dn->dn_indblkshift;
571                         blkcnt = 1 + (blkcnt >> epbs);
572                 }
573         }

575         /* account for new level 1 indirect blocks that might show up */
576         if (skipped > 0) {
577                 txh->txh_fudge += skipped << dn->dn_indblkshift;
578                 skipped = MIN(skipped, DMU_MAX_DELETEBLKCNT >> epbs);
579                 txh->txh_memory_tohold += skipped << dn->dn_indblkshift;
580         }
581         txh->txh_space_tofree += space;
582         txh->txh_space_tounref += unref;
583 }
```

unchanged_portion_omitted_