```
**********************************************************
   35483 Thu Aug 16 22:11:07 2012
new/usr/src/uts/common/fs/zfs/dmu_tx.c
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 l1
blocks (the loop for this is already present) and assumes a worst-case
distribution of those blocks over the full length given.
**********************************************************
______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, nblks, 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, nl1blks = 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_indblkshift - SPA_BLKPTRSHIFT;
 446         if (dn->dn_maxblkid == 0) {
 447                 if (off == 0 && len >= dn->dn_datablksz) {
 448                         blkid = 0;
 449                         nblks = 1;
 450                 } else {
 451                         rw_exit(&dn->dn_struct_rwlock);
 452                         return;
 453                 }
 454         } else {
 455                 blkid = off >> dn->dn_datablkshift;
 456                 nblks = (len + dn->dn_datablksz - 1) >> dn->dn_datablkshift;

 458                 if (blkid >= dn->dn_maxblkid) {
 459                         rw_exit(&dn->dn_struct_rwlock);
 460                         return;
 461                 }
 462                 if (blkid + nblks > dn->dn_maxblkid)
 463                         nblks = dn->dn_maxblkid - blkid;

 465         }
 466         l0span = nblks;    /* 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 < nblks; i++) {
 471                         blkptr_t *bp = dn->dn_phys->dn_blkptr;
 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                 nl1blks = 1;
 481 #endif /* ! codereview */
 482                 nblks = 0;
 483         }

 432         /*
 433          * Add in memory requirements of higher-level indirects.
 434          * This assumes a worst-possible scenario for dn_nlevels.
 435          */
 436         {
 437                 uint64_t blkcnt = 1 + ((nblks >> epbs) >> epbs);
 438                 int level = (dn->dn_nlevels > 1) ? 2 : 1;

 440                 while (level++ < DN_MAX_LEVELS) {
 441                         txh->txh_memory_tohold += blkcnt << dn->dn_indblkshift;
 442                         blkcnt = 1 + (blkcnt >> epbs);
 443                 }
 444                 ASSERT(blkcnt <= dn->dn_nblkptr);
 445         }

 485         lastblk = blkid + nblks - 1;
 486         while (nblks) {
 487                 dmu_buf_impl_t *dbuf;
 488                 uint64_t ibyte, new_blkid;
 489                 int epb = 1 << epbs;
 490                 int err, i, blkoff, tochk;
 491                 blkptr_t *bp;

 493                 ibyte = blkid << dn->dn_datablkshift;
 494                 err = dnode_next_offset(dn,
 495                     DNODE_FIND_HAVELOCK, &ibyte, 2, 1, 0);
 496                 new_blkid = ibyte >> dn->dn_datablkshift;
 497                 if (err == ESRCH) {
 498                         skipped += (lastblk >> epbs) - (blkid >> epbs) + 1;
 499                         break;
 500                 }
 501                 if (err) {
 502                         txh->txh_tx->tx_err = err;
 503                         break;
 504                 }
 505                 if (new_blkid > lastblk) {
 506                         skipped += (lastblk >> epbs) - (blkid >> epbs) + 1;
 507                         break;
 508                 }

 510                 if (new_blkid > blkid) {
 511                         ASSERT((new_blkid >> epbs) > (blkid >> epbs));
 512                         skipped += (new_blkid >> epbs) - (blkid >> epbs) - 1;
 513                         nblks -= new_blkid - blkid;
 514                         blkid = new_blkid;
 515                 }
 516                 blkoff = P2PHASE(blkid, epb);
 517                 tochk = MIN(epb - blkoff, nblks);

 519                 err = dbuf_hold_impl(dn, 1, blkid >> epbs, FALSE, FTAG, &dbuf);
 520                 if (err) {
 521                         txh->txh_tx->tx_err = err;
 522                         break;
 523                 }

 525                 txh->txh_memory_tohold += dbuf->db.db_size;
```

```
527                 /*
528                  * We don't check memory_tohold against DMU_MAX_ACCESS because
529                  * memory_tohold is an over-estimation (especially the >L1
530                  * indirect blocks), so it could fail.  Callers should have
531                  * already verified that they will not be holding too much
532                  * memory.
533                  */

535                 err = dbuf_read(dbuf, NULL, DB_RF_HAVESTRUCT | DB_RF_CANFAIL);
536                 if (err != 0) {
537                         txh->txh_tx->tx_err = err;
538                         dbuf_rele(dbuf, FTAG);
539                         break;
540                 }

542                 bp = dbuf->db.db_data;
543                 bp += blkoff;

545                 for (i = 0; i < tochk; i++) {
546                         if (dsl_dataset_block_freeable(ds, &bp[i],
547                             bp[i].blk_birth)) {
548                                 dprintf_bp(&bp[i], "can free old%s", "");
549                                 space += bp_get_dsize(spa, &bp[i]);
550                         }
551                         unref += BP_GET_ASIZE(bp);
552                 }
553                 dbuf_rele(dbuf, FTAG);

555                 ++nl1blks;
556 #endif /* ! codereview */
557                 blkid += tochk;
558                 nblks -= tochk;
559         }
560         rw_exit(&dn->dn_struct_rwlock);

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 l1-blocks over the region to free.
566          */
567         {
568                 uint64_t blkcnt = 1 + ((l0span >> epbs) >> epbs);
569                 int level = 2;
570                 /*
571                  * Here we don't use DN_MAX_LEVEL, but calculate it with the
572                  * given datablkshift and indblkshift. This makes the
573                  * difference between 19 and 8 on large files.
574                  */
575                 int maxlevel = 2 + (DN_MAX_OFFSET_SHIFT - dn->dn_datablkshift) /
576                     (dn->dn_indblkshift - SPA_BLKPTRSHIFT);

578                 while (level++ < maxlevel) {
579                         txh->txh_memory_tohold += MIN(blkcnt, (nl1blks >> epbs))
580                             << dn->dn_indblkshift;
581                         blkcnt = 1 + (blkcnt >> epbs);
582                 }
583         }

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 }

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;

605         ASSERT(tx->tx_txg == 0);

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;

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);

620         dmu_tx_count_dnode(txh);

622         if (off >= (dn->dn_maxblkid+1) * dn->dn_datablksz)
623                 return;
624         if (len == DMU_OBJECT_END)
625                 len = (dn->dn_maxblkid+1) * dn->dn_datablksz - off;

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;

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                         i = ibyte >> shift;
644                         if (err == ESRCH)
645                                 break;
646                         if (err) {
647                                 tx->tx_err = err;
648                                 return;
649                         }

651                         err = dmu_tx_check_ioerr(zio, dn, 1, i);
652                         if (err) {
653                                 tx->tx_err = err;
654                                 return;
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;

675         ASSERT(tx->tx_txg == 0);

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;

683         dmu_tx_count_dnode(txh);

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         }

695         ASSERT3P(DMU_OT_BYTESWAP(dn->dn_type), ==, DMU_BSWAP_ZAP);

697         if (dn->dn_maxblkid == 0 && !add) {
698                 blkptr_t *bp;

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                 }

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                 return;
723         }
```



```
725         if (dn->dn_maxblkid > 0 && name) {
726                 /*
727                  * access the name in this fat-zap so that we'll check
728                  * for i/o errors to the leaf blocks, etc.
729                  */
730                 err = zap_lookup(dn->dn_objset, dn->dn_object, name,
731                     8, 0, NULL);
732                 if (err == EIO) {
733                         tx->tx_err = err;
734                         return;
735                 }
736         }

738         err = zap_count_write(dn->dn_objset, dn->dn_object, name, add,
739             &txh->txh_space_towrite, &txh->txh_space_tooverwrite);

741         /*
742          * If the modified blocks are scattered to the four winds,
743          * we'll have to modify an indirect twig for each.
744          */
745         epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
746         for (nblocks = dn->dn_maxblkid >> epbs; nblocks != 0; nblocks >>= epbs)
747                 if (dn->dn_objset->os_dsl_dataset->ds_phys->ds_prev_snap_obj)
748                         txh->txh_space_towrite += 3 << dn->dn_indblkshift;
749                 else
750                         txh->txh_space_tooverwrite += 3 << dn->dn_indblkshift;
751 }

753 void
754 dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object)
755 {
756         dmu_tx_hold_t *txh;

758         ASSERT(tx->tx_txg == 0);

760         txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
761             object, THT_BONUS, 0, 0);
762         if (txh)
763                 dmu_tx_count_dnode(txh);
764 }

766 void
767 dmu_tx_hold_space(dmu_tx_t *tx, uint64_t space)
768 {
769         dmu_tx_hold_t *txh;
770         ASSERT(tx->tx_txg == 0);

772         txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
773             DMU_NEW_OBJECT, THT_SPACE, space, 0);

775         txh->txh_space_towrite += space;
776 }

778 int
779 dmu_tx_holds(dmu_tx_t *tx, uint64_t object)
780 {
781         dmu_tx_hold_t *txh;
782         int holds = 0;

784         /*
785          * By asserting that the tx is assigned, we're counting the
786          * number of dn_tx_holds, which is the same as the number of
787          * dn_holds.  Otherwise, we'd be counting dn_holds, but
788          * dn_tx_holds could be 0.
789          */
```

```
790         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_NEWOBJECT)
833                         match_object = TRUE;
834                 if (txh->txh_dnode == NULL || txh->txh_dnode == dn) {
835                         int datablkshift = dn->dn_datablkshift ?
836                             dn->dn_datablkshift : SPA_MAXBLOCKSHIFT;
837                         int epbs = dn->dn_indblkshift - 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... */
847                         dprintf("found txh type %x beginblk=%llx endblk=%llx\n",
848                             txh->txh_type, beginblk, endblk);
850                         switch (txh->txh_type) {
851                         case THT_WRITE:
852                                 if (blkid >= beginblk && blkid <= endblk)
853                                         match_offset = TRUE;
854                                 /*
855                                  * We will let this hold work for the bonus
```



```
856                                  * or spill buffer so that we don't need to
857                                  * hold it when creating a new object.
858                                  */
859                                 if (blkid == DMU_BONUS_BLKID ||
860                                     blkid == DMU_SPILL_BLKID)
861                                         match_offset = TRUE;
862                                 /*
863                                  * They might have to increase nlevels,
864                                  * thus dirtying the new TLIBs.  Or the
865                                  * might have to change the block size,
866                                  * thus dirying the new lvl=0 blk=0.
867                                  */
868                                 if (blkid == 0)
869                                         match_offset = TRUE;
870                                 break;
871                         case THT_FREE:
872                                 /*
873                                  * We will dirty all the level 1 blocks in
874                                  * the free range and perhaps the first and
875                                  * last level 0 block.
876                                  */
877                                 if (blkid >= beginblk && (blkid <= endblk ||
878                                     txh->txh_arg2 == DMU_OBJECT_END))
879                                         match_offset = TRUE;
880                                 break;
881                         case THT_SPILL:
882                                 if (blkid == DMU_SPILL_BLKID)
883                                         match_offset = TRUE;
884                                 break;
885                         case THT_BONUS:
886                                 if (blkid == DMU_BONUS_BLKID)
887                                         match_offset = TRUE;
888                                 break;
889                         case THT_ZAP:
890                                 match_offset = TRUE;
891                                 break;
892                         case THT_NEWOBJECT:
893                                 match_object = TRUE;
894                                 break;
895                         default:
896                                 ASSERT(!"bad txh_type");
897                         }
898                 }
899                 if (match_object && match_offset) {
900                         DB_DNODE_EXIT(db);
901                         return;
902                 }
903         }
904         DB_DNODE_EXIT(db);
905         panic("dirtying dbuf obj=%llx lvl=%u blkid=%llx but not tx_held\n",
906             (u_longlong_t)db->db.db_object, db->db_level,
907             (u_longlong_t)db->db_blkid);
908 }
909 #endif
911 static int
912 dmu_tx_try_assign(dmu_tx_t *tx, uint64_t txg_how)
913 {
914         dmu_tx_hold_t *txh;
915         spa_t *spa = tx->tx_pool->dp_spa;
916         uint64_t memory, asize, fsize, usize;
917         uint64_t towrite, tofree, tooverwrite, tounref, tohold, fudge;
919         ASSERT3U(tx->tx_txg, ==, 0);
921         if (tx->tx_err)
```

```
922                 return (tx->tx_err);
923
924         if (spa_suspended(spa)) {
925                 /*
926                  * If the user has indicated a blocking failure mode
927                  * then return ERESTART which will block in dmu_tx_wait().
928                  * Otherwise, return EIO so that an error can get
929                  * propagated back to the VOP calls.
930                  *
931                  * Note that we always honor the txg_how flag regardless
932                  * of the failuremode setting.
933                  */
934                 if (spa_get_failmode(spa) == ZIO_FAILURE_MODE_CONTINUE &&
935                     txg_how != TXG_WAIT)
936                         return (EIO);
937
938                 return (ERESTART);
939         }
940
941         tx->tx_txg = txg_hold_open(tx->tx_pool, &tx->tx_txgh);
942         tx->tx_needassign_txh = NULL;
943
944         /*
945          * NB: No error returns are allowed after txg_hold_open, but
946          * before processing the dnode holds, due to the
947          * dmu_tx_unassign() logic.
948          */
949
950         towrite = tofree = tooverwrite = tounref = tohold = fudge = 0;
951         for (txh = list_head(&tx->tx_holds); txh;
952             txh = list_next(&tx->tx_holds, txh)) {
953                 dnode_t *dn = txh->txh_dnode;
954                 if (dn != NULL) {
955                         mutex_enter(&dn->dn_mtx);
956                         if (dn->dn_assigned_txg == tx->tx_txg - 1) {
957                                 mutex_exit(&dn->dn_mtx);
958                                 tx->tx_needassign_txh = txh;
959                                 return (ERESTART);
960                         }
961                         if (dn->dn_assigned_txg == 0)
962                                 dn->dn_assigned_txg = tx->tx_txg;
963                         ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);
964                         (void) refcount_add(&dn->dn_tx_holds, tx);
965                         mutex_exit(&dn->dn_mtx);
966                 }
967                 towrite += txh->txh_space_towrite;
968                 tofree += txh->txh_space_tofree;
969                 tooverwrite += txh->txh_space_tooverwrite;
970                 tounref += txh->txh_space_tounref;
971                 tohold += txh->txh_memory_tohold;
972                 fudge += txh->txh_fudge;
973         }
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         }
1050
1051         txg_rele_to_sync(&tx->tx_txgh);
1052
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;

1078         ASSERT(tx->tx_txg == 0);
1079         ASSERT(txg_how != 0);
1080         ASSERT(!dsl_pool_sync_context(tx->tx_pool));

1082         while ((err = dmu_tx_try_assign(tx, txg_how)) != 0) {
1083                 dmu_tx_unassign(tx);

1085                 if (err != ERESTART || txg_how != TXG_WAIT)
1086                         return (err);

1088                 dmu_tx_wait(tx);
1089         }

1091         txg_rele_to_quiesce(&tx->tx_txgh);

1093         return (0);
1094 }

1096 void
1097 dmu_tx_wait(dmu_tx_t *tx)
1098 {
1099         spa_t *spa = tx->tx_pool->dp_spa;

1101         ASSERT(tx->tx_txg == 0);

1103         /*
1104          * It's possible that the pool has become active after this thread
1105          * has tried to obtain a tx. If that's the case then his
1106          * tx_lasttried_txg would not have been assigned.
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;

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                 tx->tx_needassign_txh = NULL;
1118         } else {
1119                 txg_wait_open(tx->tx_pool, tx->tx_lasttried_txg + 1);
```



```
1120         }
1121 }

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;

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 }

1140 void
1141 dmu_tx_commit(dmu_tx_t *tx)
1142 {
1143         dmu_tx_hold_t *txh;

1145         ASSERT(tx->tx_txg != 0);

1147         while (txh = list_head(&tx->tx_holds)) {
1148                 dnode_t *dn = txh->txh_dnode;

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);

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         }

1165         if (tx->tx_tempreserve_cookie)
1166                 dsl_dir_tempreserve_clear(tx->tx_tempreserve_cookie, tx);

1168         if (!list_is_empty(&tx->tx_callbacks))
1169                 txg_register_callbacks(&tx->tx_txgh, &tx->tx_callbacks);

1171         if (tx->tx_anyobj == FALSE)
1172                 txg_rele_to_sync(&tx->tx_txgh);

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         kmem_free(tx, sizeof (dmu_tx_t));
```

```
1186 }
1188 void
1189 dmu_tx_abort(dmu_tx_t *tx)
1190 {
1191         dmu_tx_hold_t *txh;
1193         ASSERT(tx->tx_txg == 0);
1195         while (txh = list_head(&tx->tx_holds)) {
1196                 dnode_t *dn = txh->txh_dnode;
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         }
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);
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 }
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 }
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;
1233         dcb = kmem_alloc(sizeof (dmu_tx_callback_t), KM_SLEEP);
1235         dcb->dcb_func = func;
1236         dcb->dcb_data = data;
1238         list_insert_tail(&tx->tx_callbacks, dcb);
1239 }
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 }
1256 /*
1257  * Interface to hold a bunch of attributes.
1258  * used for creating new files.
1259  * attrsize is the total size of all attributes
1260  * to be added during object creation
1261  *
1262  * For updating/adding a single attribute dmu_tx_hold_sa() should be used.
1263  */
1265 /*
1266  * hold necessary attribute name for attribute registration.
1267  * should be a very rare case where this is needed.  If it does
1268  * happen it would only happen on the first write to the file system.
1269  */
1270 static void
1271 dmu_tx_sa_registration_hold(sa_os_t *sa, dmu_tx_t *tx)
1272 {
1273         int i;
1275         if (!sa->sa_need_attr_registration)
1276                 return;
1278         for (i = 0; i != sa->sa_num_attrs; i++) {
1279                 if (!sa->sa_attr_table[i].sa_registered) {
1280                         if (sa->sa_reg_attr_obj)
1281                                 dmu_tx_hold_zap(tx, sa->sa_reg_attr_obj,
1282                                     B_TRUE, sa->sa_attr_table[i].sa_name);
1283                         else
1284                                 dmu_tx_hold_zap(tx, DMU_NEW_OBJECT,
1285                                     B_TRUE, sa->sa_attr_table[i].sa_name);
1286                 }
1287         }
1288 }
1291 void
1292 dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object)
1293 {
1294         dnode_t *dn;
1295         dmu_tx_hold_t *txh;
1297         txh = dmu_tx_hold_object_impl(tx, tx->tx_objset, object,
1298             THT_SPILL, 0, 0);
1300         dn = txh->txh_dnode;
1302         if (dn == NULL)
1303                 return;
1305         /* If blkptr doesn't exist then add space to towrite */
1306         if (!(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR)) {
1307                 txh->txh_space_towrite += SPA_MAXBLOCKSIZE;
1308         } else {
1309                 blkptr_t *bp;
1311                 bp = &dn->dn_phys->dn_spill;
1312                 if (dsl_dataset_block_freeable(dn->dn_objset->os_dsl_dataset,
1313                     bp, bp->blk_birth))
1314                         txh->txh_space_tooverwrite += SPA_MAXBLOCKSIZE;
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 attrsize)
1324 {
1325        sa_os_t *sa = tx->tx_objset->os_sa;

1327        dmu_tx_hold_bonus(tx, DMU_NEW_OBJECT);

1329        if (tx->tx_objset->os_sa->sa_master_obj == 0)
1330                return;

1332        if (tx->tx_objset->os_sa->sa_layout_attr_obj)
1333                dmu_tx_hold_zap(tx, sa->sa_layout_attr_obj, B_TRUE, NULL);
1334        else {
1335                dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_LAYOUTS);
1336                dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_REGISTRY);
1337                dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1338                dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1339        }

1341        dmu_tx_sa_registration_hold(sa, tx);

1343        if (attrsize <= DN_MAX_BONUSLEN && !sa->sa_force_spill)
1344                return;

1346        (void) dmu_tx_hold_object_impl(tx, tx->tx_objset, DMU_NEW_OBJECT,
1347            THT_SPILL, 0, 0);
1348 }

1350 /*
1351  * Hold SA attribute
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;

1365        ASSERT(hdl != NULL);

1367        object = sa_handle_object(hdl);

1369        dmu_tx_hold_bonus(tx, object);

1371        if (tx->tx_objset->os_sa->sa_master_obj == 0)
1372                return;

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);
```



```
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);

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                dmu_buf_impl_t *db = (dmu_buf_impl_t *)hdl->sa_bonus;
1392                dnode_t *dn;

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                DB_DNODE_EXIT(db);
1401        }
1402 }
```