```
**********************************************************
   89077 Wed Apr  6 14:26:56 2016
new/usr/src/uts/common/fs/zfs/dbuf.c
patch first-pass
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
  24  * Copyright (c) 2012, 2015 by Delphix. All rights reserved.
  25  * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
  26  * Copyright (c) 2013, Joyent, Inc. All rights reserved.
  27  * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
  28  * Copyright (c) 2014 Integros [integros.com]
  29  */

  31 #include <sys/zfs_context.h>
  32 #include <sys/dmu.h>
  33 #include <sys/dmu_send.h>
  34 #include <sys/dmu_impl.h>
  35 #include <sys/dbuf.h>
  36 #include <sys/dmu_objset.h>
  37 #include <sys/dsl_dataset.h>
  38 #include <sys/dsl_dir.h>
  39 #include <sys/dmu_tx.h>
  40 #include <sys/spa.h>
  41 #include <sys/zio.h>
  42 #include <sys/dmu_zfetch.h>
  43 #include <sys/sa.h>
  44 #include <sys/sa_impl.h>
  45 #include <sys/zfeature.h>
  46 #include <sys/blkptr.h>
  47 #include <sys/range_tree.h>

  49 /*
  50  * Number of times that zfs_free_range() took the slow path while doing
  51  * a zfs receive.  A nonzero value indicates a potential performance problem.
  52  */
  53 uint64_t zfs_free_range_recv_miss;

  55 static void dbuf_destroy(dmu_buf_impl_t *db);
  56 static boolean_t dbuf_undirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
  57 static void dbuf_write(dbuf_dirty_record_t *dr, arc_buf_t *data, dmu_tx_t *tx);

  59 #ifndef __lint
  60 extern inline void dmu_buf_init_user(dmu_buf_user_t *dbu,
  61     dmu_buf_evict_func_t *evict_func_prep, dmu_buf_evict_func_t *evict_func,
```



```
  62     dmu_buf_t **clear_on_evict_dbufp);
  61     dmu_buf_evict_func_t *evict_func, dmu_buf_t **clear_on_evict_dbufp);
  63 #endif /* ! __lint */

  65 /*
  66  * Global data structures and functions for the dbuf cache.
  67  */
  68 static kmem_cache_t *dbuf_cache;
  69 static taskq_t *dbu_evict_taskq;

  71 /* ARGSUSED */
  72 static int
  73 dbuf_cons(void *vdb, void *unused, int kmflag)
  74 {
  75         dmu_buf_impl_t *db = vdb;
  76         bzero(db, sizeof (dmu_buf_impl_t));

  78         mutex_init(&db->db_mtx, NULL, MUTEX_DEFAULT, NULL);
  79         cv_init(&db->db_changed, NULL, CV_DEFAULT, NULL);
  80         refcount_create(&db->db_holds);

  82         return (0);
  83 }
______unchanged_portion_omitted_

 285 static void
 286 dbuf_evict_user(dmu_buf_impl_t *db)
 287 {
 288         dmu_buf_user_t *dbu = db->db_user;

 290         ASSERT(MUTEX_HELD(&db->db_mtx));

 292         if (dbu == NULL)
 293                 return;

 295         dbuf_verify_user(db, DBVU_EVICTING);
 296         db->db_user = NULL;

 298 #ifdef ZFS_DEBUG
 299         if (dbu->dbu_clear_on_evict_dbufp != NULL)
 300                 *dbu->dbu_clear_on_evict_dbufp = NULL;
 301 #endif

 303         if (dbu->dbu_evict_func_prep != NULL)
 304                 dbu->dbu_evict_func_prep(dbu);
 305 #endif /* ! codereview */

 307         /*
 308          * Invoke the callback from a taskq to avoid lock order reversals
 309          * and limit stack depth.
 310          */
 311         taskq_dispatch_ent(dbu_evict_taskq, dbu->dbu_evict_func, dbu, 0,
 312             &dbu->dbu_tqent);
 313 }

 315 boolean_t
 316 dbuf_is_metadata(dmu_buf_impl_t *db)
 317 {
 318         if (db->db_level > 0) {
 319                 return (B_TRUE);
 320         } else {
 321                 boolean_t is_metadata;

 323                 DB_DNODE_ENTER(db);
 324                 is_metadata = DMU_OT_IS_METADATA(DB_DNODE(db)->dn_type);
 325                 DB_DNODE_EXIT(db);
```

```
327                 return (is_metadata);
328         }
329 }

331 void
332 dbuf_evict(dmu_buf_impl_t *db)
333 {
334         ASSERT(MUTEX_HELD(&db->db_mtx));
335         ASSERT(db->db_buf == NULL);
336         ASSERT(db->db_data_pending == NULL);

338         dbuf_clear(db);
339         dbuf_destroy(db);
340 }

342 void
343 dbuf_init(void)
344 {
345         uint64_t hsize = 1ULL << 16;
346         dbuf_hash_table_t *h = &dbuf_hash_table;
347         int i;

349         /*
350          * The hash table is big enough to fill all of physical memory
351          * with an average 4K block size.  The table will take up
352          * totalmem*sizeof(void*)/4K (i.e. 2MB/GB with 8-byte pointers).
353          */
354         while (hsize * 4096 < physmem * PAGESIZE)
355                 hsize <<= 1;

357 retry:
358         h->hash_table_mask = hsize - 1;
359         h->hash_table = kmem_zalloc(hsize * sizeof (void *), KM_NOSLEEP);
360         if (h->hash_table == NULL) {
361                 /* XXX - we should really return an error instead of assert */
362                 ASSERT(hsize > (1ULL << 10));
363                 hsize >>= 1;
364                 goto retry;
365         }

367         dbuf_cache = kmem_cache_create("dmu_buf_impl_t",
368             sizeof (dmu_buf_impl_t),
369             0, dbuf_cons, dbuf_dest, NULL, NULL, NULL, 0);

371         for (i = 0; i < DBUF_MUTEXES; i++)
372                 mutex_init(&h->hash_mutexes[i], NULL, MUTEX_DEFAULT, NULL);

374         /*
375          * All entries are queued via taskq_dispatch_ent(), so min/maxalloc
376          * configuration is not required.
377          */
378         dbu_evict_taskq = taskq_create("dbu_evict", 1, minclsyspri, 0, 0, 0);
379 }

381 void
382 dbuf_fini(void)
383 {
384         dbuf_hash_table_t *h = &dbuf_hash_table;
385         int i;

387         for (i = 0; i < DBUF_MUTEXES; i++)
388                 mutex_destroy(&h->hash_mutexes[i]);
389         kmem_free(h->hash_table, (h->hash_table_mask + 1) * sizeof (void *));
390         kmem_cache_destroy(dbuf_cache);
391         taskq_destroy(dbu_evict_taskq);
```



```
392 }

394 /*
395  * Other stuff.
396  */

398 #ifdef ZFS_DEBUG
399 static void
400 dbuf_verify(dmu_buf_impl_t *db)
401 {
402         dnode_t *dn;
403         dbuf_dirty_record_t *dr;

405         ASSERT(MUTEX_HELD(&db->db_mtx));

407         if (!(zfs_flags & ZFS_DEBUG_DBUF_VERIFY))
408                 return;

410         ASSERT(db->db_objset != NULL);
411         DB_DNODE_ENTER(db);
412         dn = DB_DNODE(db);
413         if (dn == NULL) {
414                 ASSERT(db->db_parent == NULL);
415                 ASSERT(db->db_blkptr == NULL);
416         } else {
417                 ASSERT3U(db->db.db_object, ==, dn->dn_object);
418                 ASSERT3P(db->db_objset, ==, dn->dn_objset);
419                 ASSERT3U(db->db_level, <, dn->dn_nlevels);
420                 ASSERT(db->db_blkid == DMU_BONUS_BLKID ||
421                     db->db_blkid == DMU_SPILL_BLKID ||
422                     !avl_is_empty(&dn->dn_dbufs));
423         }
424         if (db->db_blkid == DMU_BONUS_BLKID) {
425                 ASSERT(dn != NULL);
426                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
427                 ASSERT3U(db->db.db_offset, ==, DMU_BONUS_BLKID);
428         } else if (db->db_blkid == DMU_SPILL_BLKID) {
429                 ASSERT(dn != NULL);
430                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
431                 ASSERT0(db->db.db_offset);
432         } else {
433                 ASSERT3U(db->db.db_offset, ==, db->db_blkid * db->db.db_size);
434         }

436         for (dr = db->db_data_pending; dr != NULL; dr = dr->dr_next)
437                 ASSERT(dr->dr_dbuf == db);

439         for (dr = db->db_last_dirty; dr != NULL; dr = dr->dr_next)
440                 ASSERT(dr->dr_dbuf == db);

442         /*
443          * We can't assert that db_size matches dn_datablksz because it
444          * can be momentarily different when another thread is doing
445          * dnode_set_blksz().
446          */
447         if (db->db_level == 0 && db->db.db_object == DMU_META_DNODE_OBJECT) {
448                 dr = db->db_data_pending;
449                 /*
450                  * It should only be modified in syncing context, so
451                  * make sure we only have one copy of the data.
452                  */
453                 ASSERT(dr == NULL || dr->dt.dl.dr_data == db->db_buf);
454         }

456         /* verify db->db_blkptr */
457         if (db->db_blkptr) {
```

```
458                 if (db->db_parent == dn->dn_dbuf) {
459                         /* db is pointed to by the dnode */
460                         /* ASSERT3U(db->db_blkid, <, dn->dn_nblkptr); */
461                         if (DMU_OBJECT_IS_SPECIAL(db->db.db_object))
462                                 ASSERT(db->db_parent == NULL);
463                         else
464                                 ASSERT(db->db_parent != NULL);
465                         if (db->db_blkid != DMU_SPILL_BLKID)
466                                 ASSERT3P(db->db_blkptr, ==,
467                                     &dn->dn_phys->dn_blkptr[db->db_blkid]);
468                 } else {
469                         /* db is pointed to by an indirect block */
470                         int epb = db->db_parent->db.db_size >> SPA_BLKPTRSHIFT;
471                         ASSERT3U(db->db_parent->db_level, ==, db->db_level+1);
472                         ASSERT3U(db->db_parent->db.db_object, ==,
473                             db->db.db_object);
474                         /*
475                          * dnode_grow_indblksz() can make this fail if we don't
476                          * have the struct_rwlock.  XXX indblksz no longer
477                          * grows.  safe to do this now?
478                          */
479                         if (RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
480                                 ASSERT3P(db->db_blkptr, ==,
481                                     ((blkptr_t *)db->db_parent->db.db_data +
482                                     db->db_blkid % epb));
483                         }
484                 }
485         }
486         if ((db->db_blkptr == NULL || BP_IS_HOLE(db->db_blkptr)) &&
487             (db->db_buf == NULL || db->db_buf->b_data) &&
488             db->db.db_data && db->db_blkid != DMU_BONUS_BLKID &&
489             db->db_state != DB_FILL && !dn->dn_free_txg) {
490                 /*
491                  * If the blkptr isn't set but they have nonzero data,
492                  * it had better be dirty, otherwise we'll lose that
493                  * data when we evict this buffer.
494                  */
495                 if (db->db_dirtycnt == 0) {
496                         uint64_t *buf = db->db.db_data;
497                         int i;
498
499                         for (i = 0; i < db->db.db_size >> 3; i++) {
500                                 ASSERT(buf[i] == 0);
501                         }
502                 }
503         }
504         DB_DNODE_EXIT(db);
505 }
506 #endif
507
508 static void
509 dbuf_clear_data(dmu_buf_impl_t *db)
510 {
511         ASSERT(MUTEX_HELD(&db->db_mtx));
512         dbuf_evict_user(db);
513         db->db_buf = NULL;
514         db->db.db_data = NULL;
515         if (db->db_state != DB_NOFILL)
516                 db->db_state = DB_UNCACHED;
517 }
518
519 static void
520 dbuf_set_data(dmu_buf_impl_t *db, arc_buf_t *buf)
521 {
522         ASSERT(MUTEX_HELD(&db->db_mtx));
523         ASSERT(buf != NULL);
```



```
524
525         db->db_buf = buf;
526         ASSERT(buf->b_data != NULL);
527         db->db.db_data = buf->b_data;
528         if (!arc_released(buf))
529                 arc_set_callback(buf, dbuf_do_evict, db);
530 }
531
532 /*
533  * Loan out an arc_buf for read.  Return the loaned arc_buf.
534  */
535 arc_buf_t *
536 dbuf_loan_arcbuf(dmu_buf_impl_t *db)
537 {
538         arc_buf_t *abuf;
539
540         mutex_enter(&db->db_mtx);
541         if (arc_released(db->db_buf) || refcount_count(&db->db_holds) > 1) {
542                 int blksz = db->db.db_size;
543                 spa_t *spa = db->db_objset->os_spa;
544
545                 mutex_exit(&db->db_mtx);
546                 abuf = arc_loan_buf(spa, blksz);
547                 bcopy(db->db.db_data, abuf->b_data, blksz);
548         } else {
549                 abuf = db->db_buf;
550                 arc_loan_inuse_buf(abuf, db);
551                 dbuf_clear_data(db);
552                 mutex_exit(&db->db_mtx);
553         }
554         return (abuf);
555 }
556
557 /*
558  * Calculate which level n block references the data at the level 0 offset
559  * provided.
560  */
561 uint64_t
562 dbuf_whichblock(dnode_t *dn, int64_t level, uint64_t offset)
563 {
564         if (dn->dn_datablkshift != 0 && dn->dn_indblkshift != 0) {
565                 /*
566                  * The level n blkid is equal to the level 0 blkid divided by
567                  * the number of level 0s in a level n block.
568                  *
569                  * The level 0 blkid is offset >> datablkshift =
570                  * offset / 2^datablkshift.
571                  *
572                  * The number of level 0s in a level n is the number of block
573                  * pointers in an indirect block, raised to the power of level.
574                  * This is 2^(indblkshift - SPA_BLKPTRSHIFT)^level =
575                  * 2^(level*(indblkshift - SPA_BLKPTRSHIFT)).
576                  *
577                  * Thus, the level n blkid is: offset /
578                  * ((2^datablkshift)*(2^(level*(indblkshift - SPA_BLKPTRSHIFT)))
579                  * = offset / 2^(datablkshift + level *
580                  *   (indblkshift - SPA_BLKPTRSHIFT))
581                  * = offset >> (datablkshift + level *
582                  *   (indblkshift - SPA_BLKPTRSHIFT))
583                  */
584                 return (offset >> (dn->dn_datablkshift + level *
585                     (dn->dn_indblkshift - SPA_BLKPTRSHIFT)));
586         } else {
587                 ASSERT3U(offset, <, dn->dn_datablksz);
588                 return (0);
589         }
```

```
590 }

592 static void
593 dbuf_read_done(zio_t *zio, arc_buf_t *buf, void *vdb)
594 {
595         dmu_buf_impl_t *db = vdb;

597         mutex_enter(&db->db_mtx);
598         ASSERT3U(db->db_state, ==, DB_READ);
599         /*
600          * All reads are synchronous, so we must have a hold on the dbuf
601          */
602         ASSERT(refcount_count(&db->db_holds) > 0);
603         ASSERT(db->db_buf == NULL);
604         ASSERT(db->db.db_data == NULL);
605         if (db->db_level == 0 && db->db_freed_in_flight) {
606                 /* we were freed in flight; disregard any error */
607                 arc_release(buf, db);
608                 bzero(buf->b_data, db->db.db_size);
609                 arc_buf_freeze(buf);
610                 db->db_freed_in_flight = FALSE;
611                 dbuf_set_data(db, buf);
612                 db->db_state = DB_CACHED;
613         } else if (zio == NULL || zio->io_error == 0) {
614                 dbuf_set_data(db, buf);
615                 db->db_state = DB_CACHED;
616         } else {
617                 ASSERT(db->db_blkid != DMU_BONUS_BLKID);
618                 ASSERT3P(db->db_buf, ==, NULL);
619                 VERIFY(arc_buf_remove_ref(buf, db));
620                 db->db_state = DB_UNCACHED;
621         }
622         cv_broadcast(&db->db_changed);
623         dbuf_rele_and_unlock(db, NULL);
624 }

626 static void
627 dbuf_read_impl(dmu_buf_impl_t *db, zio_t *zio, uint32_t flags)
628 {
629         dnode_t *dn;
630         zbookmark_phys_t zb;
631         arc_flags_t aflags = ARC_FLAG_NOWAIT;

633         DB_DNODE_ENTER(db);
634         dn = DB_DNODE(db);
635         ASSERT(!refcount_is_zero(&db->db_holds));
636         /* We need the struct_rwlock to prevent db_blkptr from changing. */
637         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
638         ASSERT(MUTEX_HELD(&db->db_mtx));
639         ASSERT(db->db_state == DB_UNCACHED);
640         ASSERT(db->db_buf == NULL);

642         if (db->db_blkid == DMU_BONUS_BLKID) {
643                 int bonuslen = MIN(dn->dn_bonuslen, dn->dn_phys->dn_bonuslen);

645                 ASSERT3U(bonuslen, <=, db->db.db_size);
646                 db->db.db_data = zio_buf_alloc(DN_MAX_BONUSLEN);
647                 arc_space_consume(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
648                 if (bonuslen < DN_MAX_BONUSLEN)
649                         bzero(db->db.db_data, DN_MAX_BONUSLEN);
650                 if (bonuslen)
651                         bcopy(DN_BONUS(dn->dn_phys), db->db.db_data, bonuslen);
652                 DB_DNODE_EXIT(db);
653                 db->db_state = DB_CACHED;
654                 mutex_exit(&db->db_mtx);
655                 return;
```



```
656         }

658         /*
659          * Recheck BP_IS_HOLE() after dnode_block_freed() in case dnode_sync()
660          * processes the delete record and clears the bp while we are waiting
661          * for the dn_mtx (resulting in a "no" from block_freed).
662          */
663         if (db->db_blkptr == NULL || BP_IS_HOLE(db->db_blkptr) ||
664             (db->db_level == 0 && (dnode_block_freed(dn, db->db_blkid) ||
665             BP_IS_HOLE(db->db_blkptr)))) {
666                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);

668                 DB_DNODE_EXIT(db);
669                 dbuf_set_data(db, arc_buf_alloc(db->db_objset->os_spa,
670                     db->db.db_size, db, type));
671                 bzero(db->db.db_data, db->db.db_size);
672                 db->db_state = DB_CACHED;
673                 mutex_exit(&db->db_mtx);
674                 return;
675         }

677         DB_DNODE_EXIT(db);

679         db->db_state = DB_READ;
680         mutex_exit(&db->db_mtx);

682         if (DBUF_IS_L2CACHEABLE(db))
683                 aflags |= ARC_FLAG_L2CACHE;
684         if (DBUF_IS_L2COMPRESSIBLE(db))
685                 aflags |= ARC_FLAG_L2COMPRESS;

687         SET_BOOKMARK(&zb, db->db_objset->os_dsl_dataset ?
688             db->db_objset->os_dsl_dataset->ds_object : DMU_META_OBJSET,
689             db->db.db_object, db->db_level, db->db_blkid);

691         dbuf_add_ref(db, NULL);

693         (void) arc_read(zio, db->db_objset->os_spa, db->db_blkptr,
694             dbuf_read_done, db, ZIO_PRIORITY_SYNC_READ,
695             (flags & DB_RF_CANFAIL) ? ZIO_FLAG_CANFAIL : ZIO_FLAG_MUSTSUCCEED,
696             &aflags, &zb);
697 }

699 int
700 dbuf_read(dmu_buf_impl_t *db, zio_t *zio, uint32_t flags)
701 {
702         int err = 0;
703         boolean_t havepzio = (zio != NULL);
704         boolean_t prefetch;
705         dnode_t *dn;

707         /*
708          * We don't have to hold the mutex to check db_state because it
709          * can't be freed while we have a hold on the buffer.
710          */
711         ASSERT(!refcount_is_zero(&db->db_holds));

713         if (db->db_state == DB_NOFILL)
714                 return (SET_ERROR(EIO));

716         DB_DNODE_ENTER(db);
717         dn = DB_DNODE(db);
718         if ((flags & DB_RF_HAVESTRUCT) == 0)
719                 rw_enter(&dn->dn_struct_rwlock, RW_READER);

721         prefetch = db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
```

```
722             (flags & DB_RF_NOPREFETCH) == 0 && dn != NULL &&
723             DBUF_IS_CACHEABLE(db);

725         mutex_enter(&db->db_mtx);
726         if (db->db_state == DB_CACHED) {
727                 mutex_exit(&db->db_mtx);
728                 if (prefetch)
729                         dmu_zfetch(&dn->dn_zfetch, db->db_blkid, 1);
730                 if ((flags & DB_RF_HAVESTRUCT) == 0)
731                         rw_exit(&dn->dn_struct_rwlock);
732                 DB_DNODE_EXIT(db);
733         } else if (db->db_state == DB_UNCACHED) {
734                 spa_t *spa = dn->dn_objset->os_spa;

736                 if (zio == NULL)
737                         zio = zio_root(spa, NULL, NULL, ZIO_FLAG_CANFAIL);
738                 dbuf_read_impl(db, zio, flags);

740                 /* dbuf_read_impl has dropped db_mtx for us */

742                 if (prefetch)
743                         dmu_zfetch(&dn->dn_zfetch, db->db_blkid, 1);

745                 if ((flags & DB_RF_HAVESTRUCT) == 0)
746                         rw_exit(&dn->dn_struct_rwlock);
747                 DB_DNODE_EXIT(db);

749                 if (!havepzio)
750                         err = zio_wait(zio);
751         } else {
752                 /*
753                  * Another reader came in while the dbuf was in flight
754                  * between UNCACHED and CACHED.  Either a writer will finish
755                  * writing the buffer (sending the dbuf to CACHED) or the
756                  * first reader's request will reach the read_done callback
757                  * and send the dbuf to CACHED.  Otherwise, a failure
758                  * occurred and the dbuf went to UNCACHED.
759                  */
760                 mutex_exit(&db->db_mtx);
761                 if (prefetch)
762                         dmu_zfetch(&dn->dn_zfetch, db->db_blkid, 1);
763                 if ((flags & DB_RF_HAVESTRUCT) == 0)
764                         rw_exit(&dn->dn_struct_rwlock);
765                 DB_DNODE_EXIT(db);

767                 /* Skip the wait per the caller's request. */
768                 mutex_enter(&db->db_mtx);
769                 if ((flags & DB_RF_NEVERWAIT) == 0) {
770                         while (db->db_state == DB_READ ||
771                             db->db_state == DB_FILL) {
772                                 ASSERT(db->db_state == DB_READ ||
773                                     (flags & DB_RF_HAVESTRUCT) == 0);
774                                 DTRACE_PROBE2(blocked__read, dmu_buf_impl_t *,
775                                     db, zio_t *, zio);
776                                 cv_wait(&db->db_changed, &db->db_mtx);
777                         }
778                         if (db->db_state == DB_UNCACHED)
779                                 err = SET_ERROR(EIO);
780                 }
781                 mutex_exit(&db->db_mtx);
782         }

784         ASSERT(err || havepzio || db->db_state == DB_CACHED);
785         return (err);
786 }
```



```
788 static void
789 dbuf_noread(dmu_buf_impl_t *db)
790 {
791         ASSERT(!refcount_is_zero(&db->db_holds));
792         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
793         mutex_enter(&db->db_mtx);
794         while (db->db_state == DB_READ || db->db_state == DB_FILL)
795                 cv_wait(&db->db_changed, &db->db_mtx);
796         if (db->db_state == DB_UNCACHED) {
797                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
798                 spa_t *spa = db->db_objset->os_spa;

800                 ASSERT(db->db_buf == NULL);
801                 ASSERT(db->db.db_data == NULL);
802                 dbuf_set_data(db, arc_buf_alloc(spa, db->db.db_size, db, type));
803                 db->db_state = DB_FILL;
804         } else if (db->db_state == DB_NOFILL) {
805                 dbuf_clear_data(db);
806         } else {
807                 ASSERT3U(db->db_state, ==, DB_CACHED);
808         }
809         mutex_exit(&db->db_mtx);
810 }

812 /*
813  * This is our just-in-time copy function.  It makes a copy of
814  * buffers, that have been modified in a previous transaction
815  * group, before we modify them in the current active group.
816  *
817  * This function is used in two places: when we are dirtying a
818  * buffer for the first time in a txg, and when we are freeing
819  * a range in a dnode that includes this buffer.
820  *
821  * Note that when we are called from dbuf_free_range() we do
822  * not put a hold on the buffer, we just traverse the active
823  * dbuf list for the dnode.
824  */
825 static void
826 dbuf_fix_old_data(dmu_buf_impl_t *db, uint64_t txg)
827 {
828         dbuf_dirty_record_t *dr = db->db_last_dirty;

830         ASSERT(MUTEX_HELD(&db->db_mtx));
831         ASSERT(db->db.db_data != NULL);
832         ASSERT(db->db_level == 0);
833         ASSERT(db->db.db_object != DMU_META_DNODE_OBJECT);

835         if (dr == NULL ||
836             (dr->dt.dl.dr_data !=
837             ((db->db_blkid  == DMU_BONUS_BLKID) ? db->db.db_data : db->db_buf)))
838                 return;

840         /*
841          * If the last dirty record for this dbuf has not yet synced
842          * and its referencing the dbuf data, either:
843          *      reset the reference to point to a new copy,
844          * or (if there a no active holders)
845          *      just null out the current db_data pointer.
846          */
847         ASSERT(dr->dr_txg >= txg - 2);
848         if (db->db_blkid == DMU_BONUS_BLKID) {
849                 /* Note that the data bufs here are zio_bufs */
850                 dr->dt.dl.dr_data = zio_buf_alloc(DN_MAX_BONUSLEN);
851                 arc_space_consume(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
852                 bcopy(db->db.db_data, dr->dt.dl.dr_data, DN_MAX_BONUSLEN);
853         } else if (refcount_count(&db->db_holds) > db->db_dirtycnt) {
```

```
854                 int size = db->db.db_size;
855                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
856                 spa_t *spa = db->db_objset->os_spa;

858                 dr->dt.dl.dr_data = arc_buf_alloc(spa, size, db, type);
859                 bcopy(db->db.db_data, dr->dt.dl.dr_data->b_data, size);
860         } else {
861                 dbuf_clear_data(db);
862         }
863 }

865 void
866 dbuf_unoverride(dbuf_dirty_record_t *dr)
867 {
868         dmu_buf_impl_t *db = dr->dr_dbuf;
869         blkptr_t *bp = &dr->dt.dl.dr_overridden_by;
870         uint64_t txg = dr->dr_txg;

872         ASSERT(MUTEX_HELD(&db->db_mtx));
873         ASSERT(dr->dt.dl.dr_override_state != DR_IN_DMU_SYNC);
874         ASSERT(db->db_level == 0);

876         if (db->db_blkid == DMU_BONUS_BLKID ||
877             dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN)
878                 return;

880         ASSERT(db->db_data_pending != dr);

882         /* free this block */
883         if (!BP_IS_HOLE(bp) && !dr->dt.dl.dr_nopwrite)
884                 zio_free(db->db_objset->os_spa, txg, bp);

886         dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
887         dr->dt.dl.dr_nopwrite = B_FALSE;

889         /*
890          * Release the already-written buffer, so we leave it in
891          * a consistent dirty state.  Note that all callers are
892          * modifying the buffer, so they will immediately do
893          * another (redundant) arc_release().  Therefore, leave
894          * the buf thawed to save the effort of freezing &
895          * immediately re-thawing it.
896          */
897         arc_release(dr->dt.dl.dr_data, db);
898 }

900 /*
901  * Evict (if its unreferenced) or clear (if its referenced) any level-0
902  * data blocks in the free range, so that any future readers will find
903  * empty blocks.
904  *
905  * This is a no-op if the dataset is in the middle of an incremental
906  * receive; see comment below for details.
907  */
908 void
909 dbuf_free_range(dnode_t *dn, uint64_t start_blkid, uint64_t end_blkid,
910     dmu_tx_t *tx)
911 {
912         dmu_buf_impl_t db_search;
913         dmu_buf_impl_t *db, *db_next;
914         uint64_t txg = tx->tx_txg;
915         avl_index_t where;

917         if (end_blkid > dn->dn_maxblkid && (end_blkid != DMU_SPILL_BLKID))
918                 end_blkid = dn->dn_maxblkid;
919         dprintf_dnode(dn, "start=%llu end=%llu\n", start_blkid, end_blkid);
```



```
921         db_search.db_level = 0;
922         db_search.db_blkid = start_blkid;
923         db_search.db_state = DB_SEARCH;

925         mutex_enter(&dn->dn_dbufs_mtx);
926         if (start_blkid >= dn->dn_unlisted_l0_blkid) {
927                 /* There can't be any dbufs in this range; no need to search. */
928 #ifdef DEBUG
929                 db = avl_find(&dn->dn_dbufs, &db_search, &where);
930                 ASSERT3P(db, ==, NULL);
931                 db = avl_nearest(&dn->dn_dbufs, where, AVL_AFTER);
932                 ASSERT(db == NULL || db->db_level > 0);
933 #endif
934                 mutex_exit(&dn->dn_dbufs_mtx);
935                 return;
936         } else if (dmu_objset_is_receiving(dn->dn_objset)) {
937                 /*
938                  * If we are receiving, we expect there to be no dbufs in
939                  * the range to be freed, because receive modifies each
940                  * block at most once, and in offset order.  If this is
941                  * not the case, it can lead to performance problems,
942                  * so note that we unexpectedly took the slow path.
943                  */
944                 atomic_inc_64(&zfs_free_range_recv_miss);
945         }

947         db = avl_find(&dn->dn_dbufs, &db_search, &where);
948         ASSERT3P(db, ==, NULL);
949         db = avl_nearest(&dn->dn_dbufs, where, AVL_AFTER);

951         for (; db != NULL; db = db_next) {
952                 db_next = AVL_NEXT(&dn->dn_dbufs, db);
953                 ASSERT(db->db_blkid != DMU_BONUS_BLKID);

955                 if (db->db_level != 0 || db->db_blkid > end_blkid) {
956                         break;
957                 }
958                 ASSERT3U(db->db_blkid, >=, start_blkid);

960                 /* found a level 0 buffer in the range */
961                 mutex_enter(&db->db_mtx);
962                 if (dbuf_undirty(db, tx)) {
963                         /* mutex has been dropped and dbuf destroyed */
964                         continue;
965                 }

967                 if (db->db_state == DB_UNCACHED ||
968                     db->db_state == DB_NOFILL ||
969                     db->db_state == DB_EVICTING) {
970                         ASSERT(db->db.db_data == NULL);
971                         mutex_exit(&db->db_mtx);
972                         continue;
973                 }
974                 if (db->db_state == DB_READ || db->db_state == DB_FILL) {
975                         /* will be handled in dbuf_read_done or dbuf_rele */
976                         db->db_freed_in_flight = TRUE;
977                         mutex_exit(&db->db_mtx);
978                         continue;
979                 }
980                 if (refcount_count(&db->db_holds) == 0) {
981                         ASSERT(db->db_buf);
982                         dbuf_clear(db);
983                         continue;
984                 }
985                 /* The dbuf is referenced */
```

```
 987                 if (db->db_last_dirty != NULL) {
 988                         dbuf_dirty_record_t *dr = db->db_last_dirty;

 990                         if (dr->dr_txg == txg) {
 991                                 /*
 992                                  * This buffer is "in-use", re-adjust the file
 993                                  * size to reflect that this buffer may
 994                                  * contain new data when we sync.
 995                                  */
 996                                 if (db->db_blkid != DMU_SPILL_BLKID &&
 997                                     db->db_blkid > dn->dn_maxblkid)
 998                                         dn->dn_maxblkid = db->db_blkid;
 999                                 dbuf_unoverride(dr);
1000                         } else {
1001                                 /*
1002                                  * This dbuf is not dirty in the open context.
1003                                  * Either uncache it (if its not referenced in
1004                                  * the open context) or reset its contents to
1005                                  * empty.
1006                                  */
1007                                 dbuf_fix_old_data(db, txg);
1008                         }
1009                 }
1010                 /* clear the contents if its cached */
1011                 if (db->db_state == DB_CACHED) {
1012                         ASSERT(db->db.db_data != NULL);
1013                         arc_release(db->db_buf, db);
1014                         bzero(db->db.db_data, db->db.db_size);
1015                         arc_buf_freeze(db->db_buf);
1016                 }

1018                 mutex_exit(&db->db_mtx);
1019         }
1020         mutex_exit(&dn->dn_dbufs_mtx);
1021 }

1023 static int
1024 dbuf_block_freeable(dmu_buf_impl_t *db)
1025 {
1026         dsl_dataset_t *ds = db->db_objset->os_dsl_dataset;
1027         uint64_t birth_txg = 0;

1029         /*
1030          * We don't need any locking to protect db_blkptr:
1031          * If it's syncing, then db_last_dirty will be set
1032          * so we'll ignore db_blkptr.
1033          *
1034          * This logic ensures that only block births for
1035          * filled blocks are considered.
1036          */
1037         ASSERT(MUTEX_HELD(&db->db_mtx));
1038         if (db->db_last_dirty && (db->db_blkptr == NULL ||
1039             !BP_IS_HOLE(db->db_blkptr))) {
1040                 birth_txg = db->db_last_dirty->dr_txg;
1041         } else if (db->db_blkptr != NULL && !BP_IS_HOLE(db->db_blkptr)) {
1042                 birth_txg = db->db_blkptr->blk_birth;
1043         }

1045         /*
1046          * If this block don't exist or is in a snapshot, it can't be freed.
1047          * Don't pass the bp to dsl_dataset_block_freeable() since we
1048          * are holding the db_mtx lock and might deadlock if we are
1049          * prefetching a dedup-ed block.
1050          */
1051         if (birth_txg != 0)
```



```
1052                 return (ds == NULL ||
1053                     dsl_dataset_block_freeable(ds, NULL, birth_txg));
1054         else
1055                 return (B_FALSE);
1056 }

1058 void
1059 dbuf_new_size(dmu_buf_impl_t *db, int size, dmu_tx_t *tx)
1060 {
1061         arc_buf_t *buf, *obuf;
1062         int osize = db->db.db_size;
1063         arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
1064         dnode_t *dn;

1066         ASSERT(db->db_blkid != DMU_BONUS_BLKID);

1068         DB_DNODE_ENTER(db);
1069         dn = DB_DNODE(db);

1071         /* XXX does *this* func really need the lock? */
1072         ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));

1074         /*
1075          * This call to dmu_buf_will_dirty() with the dn_struct_rwlock held
1076          * is OK, because there can be no other references to the db
1077          * when we are changing its size, so no concurrent DB_FILL can
1078          * be happening.
1079          */
1080         /*
1081          * XXX we should be doing a dbuf_read, checking the return
1082          * value and returning that up to our callers
1083          */
1084         dmu_buf_will_dirty(&db->db, tx);

1086         /* create the data buffer for the new block */
1087         buf = arc_buf_alloc(dn->dn_objset->os_spa, size, db, type);

1089         /* copy old block data to the new block */
1090         obuf = db->db_buf;
1091         bcopy(obuf->b_data, buf->b_data, MIN(osize, size));
1092         /* zero the remainder */
1093         if (size > osize)
1094                 bzero((uint8_t *)buf->b_data + osize, size - osize);

1096         mutex_enter(&db->db_mtx);
1097         dbuf_set_data(db, buf);
1098         VERIFY(arc_buf_remove_ref(obuf, db));
1099         db->db.db_size = size;

1101         if (db->db_level == 0) {
1102                 ASSERT3U(db->db_last_dirty->dr_txg, ==, tx->tx_txg);
1103                 db->db_last_dirty->dt.dl.dr_data = buf;
1104         }
1105         mutex_exit(&db->db_mtx);

1107         dnode_willuse_space(dn, size-osize, tx);
1108         DB_DNODE_EXIT(db);
1109 }

1111 void
1112 dbuf_release_bp(dmu_buf_impl_t *db)
1113 {
1114         objset_t *os = db->db_objset;

1116         ASSERT(dsl_pool_sync_context(dmu_objset_pool(os)));
1117         ASSERT(arc_released(os->os_phys_buf) ||
```

```
1118             list_link_active(&os->os_dsl_dataset->ds_synced_link));
1119         ASSERT(db->db_parent == NULL || arc_released(db->db_parent->db_buf));
1120
1121         (void) arc_release(db->db_buf, db);
1122 }
1123
1124 /*
1125  * We already have a dirty record for this TXG, and we are being
1126  * dirtied again.
1127  */
1128 static void
1129 dbuf_redirty(dbuf_dirty_record_t *dr)
1130 {
1131         dmu_buf_impl_t *db = dr->dr_dbuf;
1132
1133         ASSERT(MUTEX_HELD(&db->db_mtx));
1134
1135         if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID) {
1136                 /*
1137                  * If this buffer has already been written out,
1138                  * we now need to reset its state.
1139                  */
1140                 dbuf_unoverride(dr);
1141                 if (db->db.db_object != DMU_META_DNODE_OBJECT &&
1142                     db->db_state != DB_NOFILL) {
1143                         /* Already released on initial dirty, so just thaw. */
1144                         ASSERT(arc_released(db->db_buf));
1145                         arc_buf_thaw(db->db_buf);
1146                 }
1147         }
1148 }
1149
1150 dbuf_dirty_record_t *
1151 dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1152 {
1153         dnode_t *dn;
1154         objset_t *os;
1155         dbuf_dirty_record_t **drp, *dr;
1156         int drop_struct_lock = FALSE;
1157         boolean_t do_free_accounting = B_FALSE;
1158         int txgoff = tx->tx_txg & TXG_MASK;
1159
1160         ASSERT(tx->tx_txg != 0);
1161         ASSERT(!refcount_is_zero(&db->db_holds));
1162         DMU_TX_DIRTY_BUF(tx, db);
1163
1164         DB_DNODE_ENTER(db);
1165         dn = DB_DNODE(db);
1166         /*
1167          * Shouldn't dirty a regular buffer in syncing context.  Private
1168          * objects may be dirtied in syncing context, but only if they
1169          * were already pre-dirtied in open context.
1170          */
1171         ASSERT(!dmu_tx_is_syncing(tx) ||
1172             BP_IS_HOLE(dn->dn_objset->os_rootbp) ||
1173             DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1174             dn->dn_objset->os_dsl_dataset == NULL);
1175         /*
1176          * We make this assert for private objects as well, but after we
1177          * check if we're already dirty.  They are allowed to re-dirty
1178          * in syncing context.
1179          */
1180         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1181             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1182             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));
```



```
1183
1184         mutex_enter(&db->db_mtx);
1185         /*
1186          * XXX make this true for indirects too?  The problem is that
1187          * transactions created with dmu_tx_create_assigned() from
1188          * syncing context don't bother holding ahead.
1189          */
1190         ASSERT(db->db_level != 0 ||
1191             db->db_state == DB_CACHED || db->db_state == DB_FILL ||
1192             db->db_state == DB_NOFILL);
1193
1194         mutex_enter(&dn->dn_mtx);
1195         /*
1196          * Don't set dirtyctx to SYNC if we're just modifying this as we
1197          * initialize the objset.
1198          */
1199         if (dn->dn_dirtyctx == DN_UNDIRTIED &&
1200             !BP_IS_HOLE(dn->dn_objset->os_rootbp)) {
1201                 dn->dn_dirtyctx =
1202                     (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN);
1203                 ASSERT(dn->dn_dirtyctx_firstset == NULL);
1204                 dn->dn_dirtyctx_firstset = kmem_alloc(1, KM_SLEEP);
1205         }
1206         mutex_exit(&dn->dn_mtx);
1207
1208         if (db->db_blkid == DMU_SPILL_BLKID)
1209                 dn->dn_have_spill = B_TRUE;
1210
1211         /*
1212          * If this buffer is already dirty, we're done.
1213          */
1214         drp = &db->db_last_dirty;
1215         ASSERT(*drp == NULL || (*drp)->dr_txg <= tx->tx_txg ||
1216             db->db.db_object == DMU_META_DNODE_OBJECT);
1217         while ((dr = *drp) != NULL && dr->dr_txg > tx->tx_txg)
1218                 drp = &dr->dr_next;
1219         if (dr && dr->dr_txg == tx->tx_txg) {
1220                 DB_DNODE_EXIT(db);
1221
1222                 dbuf_redirty(dr);
1223                 mutex_exit(&db->db_mtx);
1224                 return (dr);
1225         }
1226
1227         /*
1228          * Only valid if not already dirty.
1229          */
1230         ASSERT(dn->dn_object == 0 ||
1231             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1232             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));
1233
1234         ASSERT3U(dn->dn_nlevels, >, db->db_level);
1235         ASSERT((dn->dn_phys->dn_nlevels == 0 && db->db_level == 0) ||
1236             dn->dn_phys->dn_nlevels > db->db_level ||
1237             dn->dn_next_nlevels[txgoff] > db->db_level ||
1238             dn->dn_next_nlevels[(tx->tx_txg-1) & TXG_MASK] > db->db_level ||
1239             dn->dn_next_nlevels[(tx->tx_txg-2) & TXG_MASK] > db->db_level);
1240
1241         /*
1242          * We should only be dirtying in syncing context if it's the
1243          * mos or we're initializing the os or it's a special object.
1244          * However, we are allowed to dirty in syncing context provided
1245          * we already dirtied it in open context.  Hence we must make
1246          * this assertion only if we're not already dirty.
1247          */
1248         os = dn->dn_objset;
1249         ASSERT(!dmu_tx_is_syncing(tx) || DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
```

```
1250             os->os_dsl_dataset == NULL || BP_IS_HOLE(os->os_rootbp));
1251         ASSERT(db->db.db_size != 0);
1252
1253         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);
1254
1255         if (db->db_blkid != DMU_BONUS_BLKID) {
1256                 /*
1257                  * Update the accounting.
1258                  * Note: we delay "free accounting" until after we drop
1259                  * the db_mtx.  This keeps us from grabbing other locks
1260                  * (and possibly deadlocking) in bp_get_dsize() while
1261                  * also holding the db_mtx.
1262                  */
1263                 dnode_willuse_space(dn, db->db.db_size, tx);
1264                 do_free_accounting = dbuf_block_freeable(db);
1265         }
1266
1267         /*
1268          * If this buffer is dirty in an old transaction group we need
1269          * to make a copy of it so that the changes we make in this
1270          * transaction group won't leak out when we sync the older txg.
1271          */
1272         dr = kmem_zalloc(sizeof (dbuf_dirty_record_t), KM_SLEEP);
1273         if (db->db_level == 0) {
1274                 void *data_old = db->db_buf;
1275
1276                 if (db->db_state != DB_NOFILL) {
1277                         if (db->db_blkid == DMU_BONUS_BLKID) {
1278                                 dbuf_fix_old_data(db, tx->tx_txg);
1279                                 data_old = db->db.db_data;
1280                         } else if (db->db.db_object != DMU_META_DNODE_OBJECT) {
1281                                 /*
1282                                  * Release the data buffer from the cache so
1283                                  * that we can modify it without impacting
1284                                  * possible other users of this cached data
1285                                  * block.  Note that indirect blocks and
1286                                  * private objects are not released until the
1287                                  * syncing state (since they are only modified
1288                                  * then).
1289                                  */
1290                                 arc_release(db->db_buf, db);
1291                                 dbuf_fix_old_data(db, tx->tx_txg);
1292                                 data_old = db->db_buf;
1293                         }
1294                         ASSERT(data_old != NULL);
1295                 }
1296                 dr->dt.dl.dr_data = data_old;
1297         } else {
1298                 mutex_init(&dr->dt.di.dr_mtx, NULL, MUTEX_DEFAULT, NULL);
1299                 list_create(&dr->dt.di.dr_children,
1300                     sizeof (dbuf_dirty_record_t),
1301                     offsetof(dbuf_dirty_record_t, dr_dirty_node));
1302         }
1303         if (db->db_blkid != DMU_BONUS_BLKID && os->os_dsl_dataset != NULL)
1304                 dr->dr_accounted = db->db.db_size;
1305         dr->dr_dbuf = db;
1306         dr->dr_txg = tx->tx_txg;
1307         dr->dr_next = *drp;
1308         *drp = dr;
1309
1310         /*
1311          * We could have been freed_in_flight between the dbuf_noread
1312          * and dbuf_dirty.  We win, as though the dbuf_noread() had
1313          * happened after the free.
1314          */
1315         if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
```



```
1316             db->db_blkid != DMU_SPILL_BLKID) {
1317                 mutex_enter(&dn->dn_mtx);
1318                 if (dn->dn_free_ranges[txgoff] != NULL) {
1319                         range_tree_clear(dn->dn_free_ranges[txgoff],
1320                             db->db_blkid, 1);
1321                 }
1322                 mutex_exit(&dn->dn_mtx);
1323                 db->db_freed_in_flight = FALSE;
1324         }
1325
1326         /*
1327          * This buffer is now part of this txg
1328          */
1329         dbuf_add_ref(db, (void *)(uintptr_t)tx->tx_txg);
1330         db->db_dirtycnt += 1;
1331         ASSERT3U(db->db_dirtycnt, <=, 3);
1332
1333         mutex_exit(&db->db_mtx);
1334
1335         if (db->db_blkid == DMU_BONUS_BLKID ||
1336             db->db_blkid == DMU_SPILL_BLKID) {
1337                 mutex_enter(&dn->dn_mtx);
1338                 ASSERT(!list_link_active(&dr->dr_dirty_node));
1339                 list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1340                 mutex_exit(&dn->dn_mtx);
1341                 dnode_setdirty(dn, tx);
1342                 DB_DNODE_EXIT(db);
1343                 return (dr);
1344         } else if (do_free_accounting) {
1345                 blkptr_t *bp = db->db_blkptr;
1346                 int64_t willfree = (bp && !BP_IS_HOLE(bp)) ?
1347                     bp_get_dsize(os->os_spa, bp) : db->db.db_size;
1348                 /*
1349                  * This is only a guess -- if the dbuf is dirty
1350                  * in a previous txg, we don't know how much
1351                  * space it will use on disk yet.  We should
1352                  * really have the struct_rwlock to access
1353                  * db_blkptr, but since this is just a guess,
1354                  * it's OK if we get an odd answer.
1355                  */
1356                 ddt_prefetch(os->os_spa, bp);
1357                 dnode_willuse_space(dn, -willfree, tx);
1358         }
1359
1360         if (!RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
1361                 rw_enter(&dn->dn_struct_rwlock, RW_READER);
1362                 drop_struct_lock = TRUE;
1363         }
1364
1365         if (db->db_level == 0) {
1366                 dnode_new_blkid(dn, db->db_blkid, tx, drop_struct_lock);
1367                 ASSERT(dn->dn_maxblkid >= db->db_blkid);
1368         }
1369
1370         if (db->db_level+1 < dn->dn_nlevels) {
1371                 dmu_buf_impl_t *parent = db->db_parent;
1372                 dbuf_dirty_record_t *di;
1373                 int parent_held = FALSE;
1374
1375                 if (db->db_parent == NULL || db->db_parent == dn->dn_dbuf) {
1376                         int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
1377
1378                         parent = dbuf_hold_level(dn, db->db_level+1,
1379                             db->db_blkid >> epbs, FTAG);
1380                         ASSERT(parent != NULL);
1381                         parent_held = TRUE;
```

```
1382                 }
1383                 if (drop_struct_lock)
1384                         rw_exit(&dn->dn_struct_rwlock);
1385                 ASSERT3U(db->db_level+1, ==, parent->db_level);
1386                 di = dbuf_dirty(parent, tx);
1387                 if (parent_held)
1388                         dbuf_rele(parent, FTAG);

1390                 mutex_enter(&db->db_mtx);
1391                 /*
1392                  * Since we've dropped the mutex, it's possible that
1393                  * dbuf_undirty() might have changed this out from under us.
1394                  */
1395                 if (db->db_last_dirty == dr ||
1396                     dn->dn_object == DMU_META_DNODE_OBJECT) {
1397                         mutex_enter(&di->dt.di.dr_mtx);
1398                         ASSERT3U(di->dr_txg, ==, tx->tx_txg);
1399                         ASSERT(!list_link_active(&dr->dr_dirty_node));
1400                         list_insert_tail(&di->dt.di.dr_children, dr);
1401                         mutex_exit(&di->dt.di.dr_mtx);
1402                         dr->dr_parent = di;
1403                 }
1404                 mutex_exit(&db->db_mtx);
1405         } else {
1406                 ASSERT(db->db_level+1 == dn->dn_nlevels);
1407                 ASSERT(db->db_blkid < dn->dn_nblkptr);
1408                 ASSERT(db->db_parent == NULL || db->db_parent == dn->dn_dbuf);
1409                 mutex_enter(&dn->dn_mtx);
1410                 ASSERT(!list_link_active(&dr->dr_dirty_node));
1411                 list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1412                 mutex_exit(&dn->dn_mtx);
1413                 if (drop_struct_lock)
1414                         rw_exit(&dn->dn_struct_rwlock);
1415         }

1417         dnode_setdirty(dn, tx);
1418         DB_DNODE_EXIT(db);
1419         return (dr);
1420 }

1422 /*
1423  * Undirty a buffer in the transaction group referenced by the given
1424  * transaction.  Return whether this evicted the dbuf.
1425  */
1426 static boolean_t
1427 dbuf_undirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1428 {
1429         dnode_t *dn;
1430         uint64_t txg = tx->tx_txg;
1431         dbuf_dirty_record_t *dr, **drp;

1433         ASSERT(txg != 0);

1435         /*
1436          * Due to our use of dn_nlevels below, this can only be called
1437          * in open context, unless we are operating on the MOS.
1438          * From syncing context, dn_nlevels may be different from the
1439          * dn_nlevels used when dbuf was dirtied.
1440          */
1441         ASSERT(db->db_objset ==
1442             dmu_objset_pool(db->db_objset)->dp_meta_objset ||
1443             txg != spa_syncing_txg(dmu_objset_spa(db->db_objset)));
1444         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1445         ASSERT0(db->db_level);
1446         ASSERT(MUTEX_HELD(&db->db_mtx));
```



```
1448         /*
1449          * If this buffer is not dirty, we're done.
1450          */
1451         for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1452                 if (dr->dr_txg <= txg)
1453                         break;
1454         if (dr == NULL || dr->dr_txg < txg)
1455                 return (B_FALSE);
1456         ASSERT(dr->dr_txg == txg);
1457         ASSERT(dr->dr_dbuf == db);

1459         DB_DNODE_ENTER(db);
1460         dn = DB_DNODE(db);

1462         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1464         ASSERT(db->db.db_size != 0);

1466         dsl_pool_undirty_space(dmu_objset_pool(dn->dn_objset),
1467             dr->dr_accounted, txg);

1469         *drp = dr->dr_next;

1471         /*
1472          * Note that there are three places in dbuf_dirty()
1473          * where this dirty record may be put on a list.
1474          * Make sure to do a list_remove corresponding to
1475          * every one of those list_insert calls.
1476          */
1477         if (dr->dr_parent) {
1478                 mutex_enter(&dr->dr_parent->dt.di.dr_mtx);
1479                 list_remove(&dr->dr_parent->dt.di.dr_children, dr);
1480                 mutex_exit(&dr->dr_parent->dt.di.dr_mtx);
1481         } else if (db->db_blkid == DMU_SPILL_BLKID ||
1482             db->db_level + 1 == dn->dn_nlevels) {
1483                 ASSERT(db->db_blkptr == NULL || db->db_parent == dn->dn_dbuf);
1484                 mutex_enter(&dn->dn_mtx);
1485                 list_remove(&dn->dn_dirty_records[txg & TXG_MASK], dr);
1486                 mutex_exit(&dn->dn_mtx);
1487         }
1488         DB_DNODE_EXIT(db);

1490         if (db->db_state != DB_NOFILL) {
1491                 dbuf_unoverride(dr);

1493                 ASSERT(db->db_buf != NULL);
1494                 ASSERT(dr->dt.dl.dr_data != NULL);
1495                 if (dr->dt.dl.dr_data != db->db_buf)
1496                         VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data, db));
1497         }

1499         kmem_free(dr, sizeof (dbuf_dirty_record_t));

1501         ASSERT(db->db_dirtycnt > 0);
1502         db->db_dirtycnt -= 1;

1504         if (refcount_remove(&db->db_holds, (void *)(uintptr_t)txg) == 0) {
1505                 arc_buf_t *buf = db->db_buf;

1507                 ASSERT(db->db_state == DB_NOFILL || arc_released(buf));
1508                 dbuf_clear_data(db);
1509                 VERIFY(arc_buf_remove_ref(buf, db));
1510                 dbuf_evict(db);
1511                 return (B_TRUE);
1512         }
```

```
1514         return (B_FALSE);
1515 }

1517 void
1518 dmu_buf_will_dirty(dmu_buf_t *db_fake, dmu_tx_t *tx)
1519 {
1520         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
1521         int rf = DB_RF_MUST_SUCCEED | DB_RF_NOPREFETCH;

1523         ASSERT(tx->tx_txg != 0);
1524         ASSERT(!refcount_is_zero(&db->db_holds));

1526         /*
1527          * Quick check for dirtyness.  For already dirty blocks, this
1528          * reduces runtime of this function by >90%, and overall performance
1529          * by 50% for some workloads (e.g. file deletion with indirect blocks
1530          * cached).
1531          */
1532         mutex_enter(&db->db_mtx);
1533         dbuf_dirty_record_t *dr;
1534         for (dr = db->db_last_dirty;
1535             dr != NULL && dr->dr_txg >= tx->tx_txg; dr = dr->dr_next) {
1536                 /*
1537                  * It's possible that it is already dirty but not cached,
1538                  * because there are some calls to dbuf_dirty() that don't
1539                  * go through dmu_buf_will_dirty().
1540                  */
1541                 if (dr->dr_txg == tx->tx_txg && db->db_state == DB_CACHED) {
1542                         /* This dbuf is already dirty and cached. */
1543                         dbuf_redirty(dr);
1544                         mutex_exit(&db->db_mtx);
1545                         return;
1546                 }
1547         }
1548         mutex_exit(&db->db_mtx);

1550         DB_DNODE_ENTER(db);
1551         if (RW_WRITE_HELD(&DB_DNODE(db)->dn_struct_rwlock))
1552                 rf |= DB_RF_HAVESTRUCT;
1553         DB_DNODE_EXIT(db);
1554         (void) dbuf_read(db, NULL, rf);
1555         (void) dbuf_dirty(db, tx);
1556 }

1558 void
1559 dmu_buf_will_not_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
1560 {
1561         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1563         db->db_state = DB_NOFILL;

1565         dmu_buf_will_fill(db_fake, tx);
1566 }

1568 void
1569 dmu_buf_will_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
1570 {
1571         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1573         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1574         ASSERT(tx->tx_txg != 0);
1575         ASSERT(db->db_level == 0);
1576         ASSERT(!refcount_is_zero(&db->db_holds));

1578         ASSERT(db->db.db_object != DMU_META_DNODE_OBJECT ||
1579             dmu_tx_private_ok(tx));
```



```
1581         dbuf_noread(db);
1582         (void) dbuf_dirty(db, tx);
1583 }

1585 #pragma weak dmu_buf_fill_done = dbuf_fill_done
1586 /* ARGSUSED */
1587 void
1588 dbuf_fill_done(dmu_buf_impl_t *db, dmu_tx_t *tx)
1589 {
1590         mutex_enter(&db->db_mtx);
1591         DBUF_VERIFY(db);

1593         if (db->db_state == DB_FILL) {
1594                 if (db->db_level == 0 && db->db_freed_in_flight) {
1595                         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1596                         /* we were freed while filling */
1597                         /* XXX dbuf_undirty? */
1598                         bzero(db->db.db_data, db->db.db_size);
1599                         db->db_freed_in_flight = FALSE;
1600                 }
1601                 db->db_state = DB_CACHED;
1602                 cv_broadcast(&db->db_changed);
1603         }
1604         mutex_exit(&db->db_mtx);
1605 }

1607 void
1608 dmu_buf_write_embedded(dmu_buf_t *dbuf, void *data,
1609     bp_embedded_type_t etype, enum zio_compress comp,
1610     int uncompressed_size, int compressed_size, int byteorder,
1611     dmu_tx_t *tx)
1612 {
1613         dmu_buf_impl_t *db = (dmu_buf_impl_t *)dbuf;
1614         struct dirty_leaf *dl;
1615         dmu_object_type_t type;

1617         if (etype == BP_EMBEDDED_TYPE_DATA) {
1618                 ASSERT(spa_feature_is_active(dmu_objset_spa(db->db_objset),
1619                     SPA_FEATURE_EMBEDDED_DATA));
1620         }

1622         DB_DNODE_ENTER(db);
1623         type = DB_DNODE(db)->dn_type;
1624         DB_DNODE_EXIT(db);

1626         ASSERT0(db->db_level);
1627         ASSERT(db->db_blkid != DMU_BONUS_BLKID);

1629         dmu_buf_will_not_fill(dbuf, tx);

1631         ASSERT3U(db->db_last_dirty->dr_txg, ==, tx->tx_txg);
1632         dl = &db->db_last_dirty->dt.dl;
1633         encode_embedded_bp_compressed(&dl->dr_overridden_by,
1634             data, comp, uncompressed_size, compressed_size);
1635         BPE_SET_ETYPE(&dl->dr_overridden_by, etype);
1636         BP_SET_TYPE(&dl->dr_overridden_by, type);
1637         BP_SET_LEVEL(&dl->dr_overridden_by, 0);
1638         BP_SET_BYTEORDER(&dl->dr_overridden_by, byteorder);

1640         dl->dr_override_state = DR_OVERRIDDEN;
1641         dl->dr_overridden_by.blk_birth = db->db_last_dirty->dr_txg;
1642 }

1644 /*
1645  * Directly assign a provided arc buf to a given dbuf if it's not referenced
```

```
 * by anybody except our caller. Otherwise copy arcbuf's contents to dbuf.
 */
void
dbuf_assign_arcbuf(dmu_buf_impl_t *db, arc_buf_t *buf, dmu_tx_t *tx)
{
        ASSERT(!refcount_is_zero(&db->db_holds));
        ASSERT(db->db_blkid != DMU_BONUS_BLKID);
        ASSERT(db->db_level == 0);
        ASSERT(DBUF_GET_BUFC_TYPE(db) == ARC_BUFC_DATA);
        ASSERT(buf != NULL);
        ASSERT(arc_buf_size(buf) == db->db.db_size);
        ASSERT(tx->tx_txg != 0);

        arc_return_buf(buf, db);
        ASSERT(arc_released(buf));

        mutex_enter(&db->db_mtx);

        while (db->db_state == DB_READ || db->db_state == DB_FILL)
                cv_wait(&db->db_changed, &db->db_mtx);

        ASSERT(db->db_state == DB_CACHED || db->db_state == DB_UNCACHED);

        if (db->db_state == DB_CACHED &&
            refcount_count(&db->db_holds) - 1 > db->db_dirtycnt) {
                mutex_exit(&db->db_mtx);
                (void) dbuf_dirty(db, tx);
                bcopy(buf->b_data, db->db.db_data, db->db.db_size);
                VERIFY(arc_buf_remove_ref(buf, db));
                xuio_stat_wbuf_copied();
                return;
        }

        xuio_stat_wbuf_nocopy();
        if (db->db_state == DB_CACHED) {
                dbuf_dirty_record_t *dr = db->db_last_dirty;

                ASSERT(db->db_buf != NULL);
                if (dr != NULL && dr->dr_txg == tx->tx_txg) {
                        ASSERT(dr->dt.dl.dr_data == db->db_buf);
                        if (!arc_released(db->db_buf)) {
                                ASSERT(dr->dt.dl.dr_override_state ==
                                    DR_OVERRIDDEN);
                                arc_release(db->db_buf, db);
                        }
                        dr->dt.dl.dr_data = buf;
                        VERIFY(arc_buf_remove_ref(db->db_buf, db));
                } else if (dr == NULL || dr->dt.dl.dr_data != db->db_buf) {
                        arc_release(db->db_buf, db);
                        VERIFY(arc_buf_remove_ref(db->db_buf, db));
                }
                db->db_buf = NULL;
        }
        ASSERT(db->db_buf == NULL);
        dbuf_set_data(db, buf);
        db->db_state = DB_FILL;
        mutex_exit(&db->db_mtx);
        (void) dbuf_dirty(db, tx);
        dmu_buf_fill_done(&db->db, tx);
}

/*
 * "Clear" the contents of this dbuf.  This will mark the dbuf
 * EVICTING and clear *most* of its references.  Unfortunately,
 * when we are not holding the dn_dbufs_mtx, we can't clear the
 * entry in the dn_dbufs list.  We have to wait until dbuf_destroy()
```



```
 * in this case.  For callers from the DMU we will usually see:
 *      dbuf_clear()->arc_clear_callback()->dbuf_do_evict()->dbuf_destroy()
 * For the arc callback, we will usually see:
 *      dbuf_do_evict()->dbuf_clear();dbuf_destroy()
 * Sometimes, though, we will get a mix of these two:
 *      DMU: dbuf_clear()->arc_clear_callback()
 *      ARC: dbuf_do_evict()->dbuf_destroy()
 *
 * This routine will dissociate the dbuf from the arc, by calling
 * arc_clear_callback(), but will not evict the data from the ARC.
 */
void
dbuf_clear(dmu_buf_impl_t *db)
{
        dnode_t *dn;
        dmu_buf_impl_t *parent = db->db_parent;
        dmu_buf_impl_t *dndb;
        boolean_t dbuf_gone = B_FALSE;

        ASSERT(MUTEX_HELD(&db->db_mtx));
        ASSERT(refcount_is_zero(&db->db_holds));

        dbuf_evict_user(db);

        if (db->db_state == DB_CACHED) {
                ASSERT(db->db.db_data != NULL);
                if (db->db_blkid == DMU_BONUS_BLKID) {
                        zio_buf_free(db->db.db_data, DN_MAX_BONUSLEN);
                        arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
                }
                db->db.db_data = NULL;
                db->db_state = DB_UNCACHED;
        }

        ASSERT(db->db_state == DB_UNCACHED || db->db_state == DB_NOFILL);
        ASSERT(db->db_data_pending == NULL);

        db->db_state = DB_EVICTING;
        db->db_blkptr = NULL;

        DB_DNODE_ENTER(db);
        dn = DB_DNODE(db);
        dndb = dn->dn_dbuf;
        if (db->db_blkid != DMU_BONUS_BLKID && MUTEX_HELD(&dn->dn_dbufs_mtx)) {
                avl_remove(&dn->dn_dbufs, db);
                atomic_dec_32(&dn->dn_dbufs_count);
                membar_producer();
                DB_DNODE_EXIT(db);
                /*
                 * Decrementing the dbuf count means that the hold corresponding
                 * to the removed dbuf is no longer discounted in dnode_move(),
                 * so the dnode cannot be moved until after we release the hold.
                 * The membar_producer() ensures visibility of the decremented
                 * value in dnode_move(), since DB_DNODE_EXIT doesn't actually
                 * release any lock.
                 */
                dnode_rele(dn, db);
                db->db_dnode_handle = NULL;
        } else {
                DB_DNODE_EXIT(db);
        }

        if (db->db_buf)
                dbuf_gone = arc_clear_callback(db->db_buf);

        if (!dbuf_gone)
```

```
1778                 mutex_exit(&db->db_mtx);

1780         /*
1781          * If this dbuf is referenced from an indirect dbuf,
1782          * decrement the ref count on the indirect dbuf.
1783          */
1784         if (parent && parent != dndb)
1785                 dbuf_rele(parent, db);
1786 }

1788 /*
1789  * Note: While bpp will always be updated if the function returns success,
1790  * parentp will not be updated if the dnode does not have dn_dbuf filled in;
1791  * this happens when the dnode is the meta-dnode, or a userused or groupused
1792  * object.
1793  */
1794 static int
1795 dbuf_findbp(dnode_t *dn, int level, uint64_t blkid, int fail_sparse,
1796     dmu_buf_impl_t **parentp, blkptr_t **bpp)
1797 {
1798         int nlevels, epbs;

1800         *parentp = NULL;
1801         *bpp = NULL;

1803         ASSERT(blkid != DMU_BONUS_BLKID);

1805         if (blkid == DMU_SPILL_BLKID) {
1806                 mutex_enter(&dn->dn_mtx);
1807                 if (dn->dn_have_spill &&
1808                     (dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR))
1809                         *bpp = &dn->dn_phys->dn_spill;
1810                 else
1811                         *bpp = NULL;
1812                 dbuf_add_ref(dn->dn_dbuf, NULL);
1813                 *parentp = dn->dn_dbuf;
1814                 mutex_exit(&dn->dn_mtx);
1815                 return (0);
1816         }

1818         if (dn->dn_phys->dn_nlevels == 0)
1819                 nlevels = 1;
1820         else
1821                 nlevels = dn->dn_phys->dn_nlevels;

1823         epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;

1825         ASSERT3U(level * epbs, <, 64);
1826         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1827         if (level >= nlevels ||
1828             (blkid > (dn->dn_phys->dn_maxblkid >> (level * epbs)))) {
1829                 /* the buffer has no parent yet */
1830                 return (SET_ERROR(ENOENT));
1831         } else if (level < nlevels-1) {
1832                 /* this block is referenced from an indirect block */
1833                 int err = dbuf_hold_impl(dn, level+1,
1834                     blkid >> epbs, fail_sparse, FALSE, NULL, parentp);
1835                 if (err)
1836                         return (err);
1837                 err = dbuf_read(*parentp, NULL,
1838                     (DB_RF_HAVESTRUCT | DB_RF_NOPREFETCH | DB_RF_CANFAIL));
1839                 if (err) {
1840                         dbuf_rele(*parentp, NULL);
1841                         *parentp = NULL;
1842                         return (err);
1843                 }
```



```
1844                 *bpp = ((blkptr_t *)(*parentp)->db.db_data) +
1845                     (blkid & ((1ULL << epbs) - 1));
1846                 return (0);
1847         } else {
1848                 /* the block is referenced from the dnode */
1849                 ASSERT3U(level, ==, nlevels-1);
1850                 ASSERT(dn->dn_phys->dn_nblkptr == 0 ||
1851                     blkid < dn->dn_phys->dn_nblkptr);
1852                 if (dn->dn_dbuf) {
1853                         dbuf_add_ref(dn->dn_dbuf, NULL);
1854                         *parentp = dn->dn_dbuf;
1855                 }
1856                 *bpp = &dn->dn_phys->dn_blkptr[blkid];
1857                 return (0);
1858         }
1859 }

1861 static dmu_buf_impl_t *
1862 dbuf_create(dnode_t *dn, uint8_t level, uint64_t blkid,
1863     dmu_buf_impl_t *parent, blkptr_t *blkptr)
1864 {
1865         objset_t *os = dn->dn_objset;
1866         dmu_buf_impl_t *db, *odb;

1868         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1869         ASSERT(dn->dn_type != DMU_OT_NONE);

1871         db = kmem_cache_alloc(dbuf_cache, KM_SLEEP);

1873         db->db_objset = os;
1874         db->db.db_object = dn->dn_object;
1875         db->db_level = level;
1876         db->db_blkid = blkid;
1877         db->db_last_dirty = NULL;
1878         db->db_dirtycnt = 0;
1879         db->db_dnode_handle = dn->dn_handle;
1880         db->db_parent = parent;
1881         db->db_blkptr = blkptr;

1883         db->db_user = NULL;
1884         db->db_user_immediate_evict = FALSE;
1885         db->db_freed_in_flight = FALSE;
1886         db->db_pending_evict = FALSE;

1888         if (blkid == DMU_BONUS_BLKID) {
1889                 ASSERT3P(parent, ==, dn->dn_dbuf);
1890                 db->db.db_size = DN_MAX_BONUSLEN -
1891                     (dn->dn_nblkptr-1) * sizeof (blkptr_t);
1892                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
1893                 db->db.db_offset = DMU_BONUS_BLKID;
1894                 db->db_state = DB_UNCACHED;
1895                 /* the bonus dbuf is not placed in the hash table */
1896                 arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1897                 return (db);
1898         } else if (blkid == DMU_SPILL_BLKID) {
1899                 db->db.db_size = (blkptr != NULL) ?
1900                     BP_GET_LSIZE(blkptr) : SPA_MINBLOCKSIZE;
1901                 db->db.db_offset = 0;
1902         } else {
1903                 int blocksize =
1904                     db->db_level ? 1 << dn->dn_indblkshift : dn->dn_datablksz;
1905                 db->db.db_size = blocksize;
1906                 db->db.db_offset = db->db_blkid * blocksize;
1907         }

1909         /*
```

```
1910          * Hold the dn_dbufs_mtx while we get the new dbuf
1911          * in the hash table *and* added to the dbufs list.
1912          * This prevents a possible deadlock with someone
1913          * trying to look up this dbuf before its added to the
1914          * dn_dbufs list.
1915          */
1916         mutex_enter(&dn->dn_dbufs_mtx);
1917         db->db_state = DB_EVICTING;
1918         if ((odb = dbuf_hash_insert(db)) != NULL) {
1919                 /* someone else inserted it first */
1920                 kmem_cache_free(dbuf_cache, db);
1921                 mutex_exit(&dn->dn_dbufs_mtx);
1922                 return (odb);
1923         }
1924         avl_add(&dn->dn_dbufs, db);
1925         if (db->db_level == 0 && db->db_blkid >=
1926             dn->dn_unlisted_l0_blkid)
1927                 dn->dn_unlisted_l0_blkid = db->db_blkid + 1;
1928         db->db_state = DB_UNCACHED;
1929         mutex_exit(&dn->dn_dbufs_mtx);
1930         arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);

1932         if (parent && parent != dn->dn_dbuf)
1933                 dbuf_add_ref(parent, db);

1935         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1936             refcount_count(&dn->dn_holds) > 0);
1937         (void) refcount_add(&dn->dn_holds, db);
1938         atomic_inc_32(&dn->dn_dbufs_count);

1940         dprintf_dbuf(db, "db=%p\n", db);

1942         return (db);
1943 }

1945 static int
1946 dbuf_do_evict(void *private)
1947 {
1948         dmu_buf_impl_t *db = private;

1950         if (!MUTEX_HELD(&db->db_mtx))
1951                 mutex_enter(&db->db_mtx);

1953         ASSERT(refcount_is_zero(&db->db_holds));

1955         if (db->db_state != DB_EVICTING) {
1956                 ASSERT(db->db_state == DB_CACHED);
1957                 DBUF_VERIFY(db);
1958                 db->db_buf = NULL;
1959                 dbuf_evict(db);
1960         } else {
1961                 mutex_exit(&db->db_mtx);
1962                 dbuf_destroy(db);
1963         }
1964         return (0);
1965 }

1967 static void
1968 dbuf_destroy(dmu_buf_impl_t *db)
1969 {
1970         ASSERT(refcount_is_zero(&db->db_holds));

1972         if (db->db_blkid != DMU_BONUS_BLKID) {
1973                 /*
1974                  * If this dbuf is still on the dn_dbufs list,
1975                  * remove it from that list.
```



```
1976                  */
1977                 if (db->db_dnode_handle != NULL) {
1978                         dnode_t *dn;

1980                         DB_DNODE_ENTER(db);
1981                         dn = DB_DNODE(db);
1982                         mutex_enter(&dn->dn_dbufs_mtx);
1983                         avl_remove(&dn->dn_dbufs, db);
1984                         atomic_dec_32(&dn->dn_dbufs_count);
1985                         mutex_exit(&dn->dn_dbufs_mtx);
1986                         DB_DNODE_EXIT(db);
1987                         /*
1988                          * Decrementing the dbuf count means that the hold
1989                          * corresponding to the removed dbuf is no longer
1990                          * discounted in dnode_move(), so the dnode cannot be
1991                          * moved until after we release the hold.
1992                          */
1993                         dnode_rele(dn, db);
1994                         db->db_dnode_handle = NULL;
1995                 }
1996                 dbuf_hash_remove(db);
1997         }
1998         db->db_parent = NULL;
1999         db->db_buf = NULL;

2001         ASSERT(db->db.db_data == NULL);
2002         ASSERT(db->db_hash_next == NULL);
2003         ASSERT(db->db_blkptr == NULL);
2004         ASSERT(db->db_data_pending == NULL);

2006         kmem_cache_free(dbuf_cache, db);
2007         arc_space_return(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
2008 }

2010 typedef struct dbuf_prefetch_arg {
2011         spa_t *dpa_spa; /* The spa to issue the prefetch in. */
2012         zbookmark_phys_t dpa_zb; /* The target block to prefetch. */
2013         int dpa_epbs; /* Entries (blkptr_t's) Per Block Shift. */
2014         int dpa_curlevel; /* The current level that we're reading */
2015         zio_priority_t dpa_prio; /* The priority I/Os should be issued at. */
2016         zio_t *dpa_zio; /* The parent zio_t for all prefetches. */
2017         arc_flags_t dpa_aflags; /* Flags to pass to the final prefetch. */
2018 } dbuf_prefetch_arg_t;

2020 /*
2021  * Actually issue the prefetch read for the block given.
2022  */
2023 static void
2024 dbuf_issue_final_prefetch(dbuf_prefetch_arg_t *dpa, blkptr_t *bp)
2025 {
2026         if (BP_IS_HOLE(bp) || BP_IS_EMBEDDED(bp))
2027                 return;

2029         arc_flags_t aflags =
2030             dpa->dpa_aflags | ARC_FLAG_NOWAIT | ARC_FLAG_PREFETCH;

2032         ASSERT3U(dpa->dpa_curlevel, ==, BP_GET_LEVEL(bp));
2033         ASSERT3U(dpa->dpa_curlevel, ==, dpa->dpa_zb.zb_level);
2034         ASSERT(dpa->dpa_zio != NULL);
2035         (void) arc_read(dpa->dpa_zio, dpa->dpa_spa, bp, NULL, NULL,
2036             dpa->dpa_prio, ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE,
2037             &aflags, &dpa->dpa_zb);
2038 }

2040 /*
2041  * Called when an indirect block above our prefetch target is read in.  This
```

```
 * will either read in the next indirect block down the tree or issue the actual
 * prefetch if the next block down is our target.
 */
static void
dbuf_prefetch_indirect_done(zio_t *zio, arc_buf_t *abuf, void *private)
{
        dbuf_prefetch_arg_t *dpa = private;

        ASSERT3S(dpa->dpa_zb.zb_level, <, dpa->dpa_curlevel);
        ASSERT3S(dpa->dpa_curlevel, >, 0);
        if (zio != NULL) {
                ASSERT3S(BP_GET_LEVEL(zio->io_bp), ==, dpa->dpa_curlevel);
                ASSERT3U(BP_GET_LSIZE(zio->io_bp), ==, zio->io_size);
                ASSERT3P(zio->io_spa, ==, dpa->dpa_spa);
        }

        dpa->dpa_curlevel--;

        uint64_t nextblkid = dpa->dpa_zb.zb_blkid >>
            (dpa->dpa_epbs * (dpa->dpa_curlevel - dpa->dpa_zb.zb_level));
        blkptr_t *bp = ((blkptr_t *)abuf->b_data) +
            P2PHASE(nextblkid, 1ULL << dpa->dpa_epbs);
        if (BP_IS_HOLE(bp) || (zio != NULL && zio->io_error != 0)) {
                kmem_free(dpa, sizeof (*dpa));
        } else if (dpa->dpa_curlevel == dpa->dpa_zb.zb_level) {
                ASSERT3U(nextblkid, ==, dpa->dpa_zb.zb_blkid);
                dbuf_issue_final_prefetch(dpa, bp);
                kmem_free(dpa, sizeof (*dpa));
        } else {
                arc_flags_t iter_aflags = ARC_FLAG_NOWAIT;
                zbookmark_phys_t zb;

                ASSERT3U(dpa->dpa_curlevel, ==, BP_GET_LEVEL(bp));

                SET_BOOKMARK(&zb, dpa->dpa_zb.zb_objset,
                    dpa->dpa_zb.zb_object, dpa->dpa_curlevel, nextblkid);

                (void) arc_read(dpa->dpa_zio, dpa->dpa_spa,
                    bp, dbuf_prefetch_indirect_done, dpa, dpa->dpa_prio,
                    ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE,
                    &iter_aflags, &zb);
        }
        (void) arc_buf_remove_ref(abuf, private);
}

/*
 * Issue prefetch reads for the given block on the given level.  If the indirect
 * blocks above that block are not in memory, we will read them in
 * asynchronously.  As a result, this call never blocks waiting for a read to
 * complete.
 */
void
dbuf_prefetch(dnode_t *dn, int64_t level, uint64_t blkid, zio_priority_t prio,
    arc_flags_t aflags)
{
        blkptr_t bp;
        int epbs, nlevels, curlevel;
        uint64_t curblkid;

        ASSERT(blkid != DMU_BONUS_BLKID);
        ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));

        if (blkid > dn->dn_maxblkid)
                return;

        if (dnode_block_freed(dn, blkid))
```



```
                return;

        /*
         * This dnode hasn't been written to disk yet, so there's nothing to
         * prefetch.
         */
        nlevels = dn->dn_phys->dn_nlevels;
        if (level >= nlevels || dn->dn_phys->dn_nblkptr == 0)
                return;

        epbs = dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;
        if (dn->dn_phys->dn_maxblkid < blkid << (epbs * level))
                return;

        dmu_buf_impl_t *db = dbuf_find(dn->dn_objset, dn->dn_object,
            level, blkid);
        if (db != NULL) {
                mutex_exit(&db->db_mtx);
                /*
                 * This dbuf already exists.  It is either CACHED, or
                 * (we assume) about to be read or filled.
                 */
                return;
        }

        /*
         * Find the closest ancestor (indirect block) of the target block
         * that is present in the cache.  In this indirect block, we will
         * find the bp that is at curlevel, curblkid.
         */
        curlevel = level;
        curblkid = blkid;
        while (curlevel < nlevels - 1) {
                int parent_level = curlevel + 1;
                uint64_t parent_blkid = curblkid >> epbs;
                dmu_buf_impl_t *db;

                if (dbuf_hold_impl(dn, parent_level, parent_blkid,
                    FALSE, TRUE, FTAG, &db) == 0) {
                        blkptr_t *bpp = db->db_buf->b_data;
                        bp = bpp[P2PHASE(curblkid, 1 << epbs)];
                        dbuf_rele(db, FTAG);
                        break;
                }

                curlevel = parent_level;
                curblkid = parent_blkid;
        }

        if (curlevel == nlevels - 1) {
                /* No cached indirect blocks found. */
                ASSERT3U(curblkid, <, dn->dn_phys->dn_nblkptr);
                bp = dn->dn_phys->dn_blkptr[curblkid];
        }
        if (BP_IS_HOLE(&bp))
                return;

        ASSERT3U(curlevel, ==, BP_GET_LEVEL(&bp));

        zio_t *pio = zio_root(dmu_objset_spa(dn->dn_objset), NULL, NULL,
            ZIO_FLAG_CANFAIL);

        dbuf_prefetch_arg_t *dpa = kmem_zalloc(sizeof (*dpa), KM_SLEEP);
        dsl_dataset_t *ds = dn->dn_objset->os_dsl_dataset;
        SET_BOOKMARK(&dpa->dpa_zb, ds != NULL ? ds->ds_object : DMU_META_OBJSET,
            dn->dn_object, level, blkid);
```

```
2174         dpa->dpa_curlevel = curlevel;
2175         dpa->dpa_prio = prio;
2176         dpa->dpa_aflags = aflags;
2177         dpa->dpa_spa = dn->dn_objset->os_spa;
2178         dpa->dpa_epbs = epbs;
2179         dpa->dpa_zio = pio;

2181         /*
2182          * If we have the indirect just above us, no need to do the asynchronous
2183          * prefetch chain; we'll just run the last step ourselves.  If we're at
2184          * a higher level, though, we want to issue the prefetches for all the
2185          * indirect blocks asynchronously, so we can go on with whatever we were
2186          * doing.
2187          */
2188         if (curlevel == level) {
2189                 ASSERT3U(curblkid, ==, blkid);
2190                 dbuf_issue_final_prefetch(dpa, &bp);
2191                 kmem_free(dpa, sizeof (*dpa));
2192         } else {
2193                 arc_flags_t iter_aflags = ARC_FLAG_NOWAIT;
2194                 zbookmark_phys_t zb;

2196                 SET_BOOKMARK(&zb, ds != NULL ? ds->ds_object : DMU_META_OBJSET,
2197                     dn->dn_object, curlevel, curblkid);
2198                 (void) arc_read(dpa->dpa_zio, dpa->dpa_spa,
2199                     &bp, dbuf_prefetch_indirect_done, dpa, prio,
2200                     ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE,
2201                     &iter_aflags, &zb);
2202         }
2203         /*
2204          * We use pio here instead of dpa_zio since it's possible that
2205          * dpa may have already been freed.
2206          */
2207         zio_nowait(pio);
2208 }

2210 /*
2211  * Returns with db_holds incremented, and db_mtx not held.
2212  * Note: dn_struct_rwlock must be held.
2213  */
2214 int
2215 dbuf_hold_impl(dnode_t *dn, uint8_t level, uint64_t blkid,
2216     boolean_t fail_sparse, boolean_t fail_uncached,
2217     void *tag, dmu_buf_impl_t **dbp)
2218 {
2219         dmu_buf_impl_t *db, *parent = NULL;

2221         ASSERT(blkid != DMU_BONUS_BLKID);
2222         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
2223         ASSERT3U(dn->dn_nlevels, >, level);

2225         *dbp = NULL;
2226 top:
2227         /* dbuf_find() returns with db_mtx held */
2228         db = dbuf_find(dn->dn_objset, dn->dn_object, level, blkid);

2230         if (db == NULL) {
2231                 blkptr_t *bp = NULL;
2232                 int err;

2234                 if (fail_uncached)
2235                         return (SET_ERROR(ENOENT));

2237                 ASSERT3P(parent, ==, NULL);
2238                 err = dbuf_findbp(dn, level, blkid, fail_sparse, &parent, &bp);
2239                 if (fail_sparse) {
```



```
2240                         if (err == 0 && bp && BP_IS_HOLE(bp))
2241                                 err = SET_ERROR(ENOENT);
2242                         if (err) {
2243                                 if (parent)
2244                                         dbuf_rele(parent, NULL);
2245                                 return (err);
2246                         }
2247                 }
2248                 if (err && err != ENOENT)
2249                         return (err);
2250                 db = dbuf_create(dn, level, blkid, parent, bp);
2251         }

2253         if (fail_uncached && db->db_state != DB_CACHED) {
2254                 mutex_exit(&db->db_mtx);
2255                 return (SET_ERROR(ENOENT));
2256         }

2258         if (db->db_buf && refcount_is_zero(&db->db_holds)) {
2259                 arc_buf_add_ref(db->db_buf, db);
2260                 if (db->db_buf->b_data == NULL) {
2261                         dbuf_clear(db);
2262                         if (parent) {
2263                                 dbuf_rele(parent, NULL);
2264                                 parent = NULL;
2265                         }
2266                         goto top;
2267                 }
2268                 ASSERT3P(db->db.db_data, ==, db->db_buf->b_data);
2269         }

2271         ASSERT(db->db_buf == NULL || arc_referenced(db->db_buf));

2273         /*
2274          * If this buffer is currently syncing out, and we are are
2275          * still referencing it from db_data, we need to make a copy
2276          * of it in case we decide we want to dirty it again in this txg.
2277          */
2278         if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
2279             dn->dn_object != DMU_META_DNODE_OBJECT &&
2280             db->db_state == DB_CACHED && db->db_data_pending) {
2281                 dbuf_dirty_record_t *dr = db->db_data_pending;

2283                 if (dr->dt.dl.dr_data == db->db_buf) {
2284                         arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);

2286                         dbuf_set_data(db,
2287                             arc_buf_alloc(dn->dn_objset->os_spa,
2288                             db->db.db_size, db, type));
2289                         bcopy(dr->dt.dl.dr_data->b_data, db->db.db_data,
2290                             db->db.db_size);
2291                 }
2292         }

2294         (void) refcount_add(&db->db_holds, tag);
2295         DBUF_VERIFY(db);
2296         mutex_exit(&db->db_mtx);

2298         /* NOTE: we can't rele the parent until after we drop the db_mtx */
2299         if (parent)
2300                 dbuf_rele(parent, NULL);

2302         ASSERT3P(DB_DNODE(db), ==, dn);
2303         ASSERT3U(db->db_blkid, ==, blkid);
2304         ASSERT3U(db->db_level, ==, level);
2305         *dbp = db;
```

```
2307         return (0);
2308 }
2309
2310 dmu_buf_impl_t *
2311 dbuf_hold(dnode_t *dn, uint64_t blkid, void *tag)
2312 {
2313         return (dbuf_hold_level(dn, 0, blkid, tag));
2314 }
2315
2316 dmu_buf_impl_t *
2317 dbuf_hold_level(dnode_t *dn, int level, uint64_t blkid, void *tag)
2318 {
2319         dmu_buf_impl_t *db;
2320         int err = dbuf_hold_impl(dn, level, blkid, FALSE, FALSE, tag, &db);
2321         return (err ? NULL : db);
2322 }
2323
2324 void
2325 dbuf_create_bonus(dnode_t *dn)
2326 {
2327         ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));
2328
2329         ASSERT(dn->dn_bonus == NULL);
2330         dn->dn_bonus = dbuf_create(dn, 0, DMU_BONUS_BLKID, dn->dn_dbuf, NULL);
2331 }
2332
2333 int
2334 dbuf_spill_set_blksz(dmu_buf_t *db_fake, uint64_t blksz, dmu_tx_t *tx)
2335 {
2336         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
2337         dnode_t *dn;
2338
2339         if (db->db_blkid != DMU_SPILL_BLKID)
2340                 return (SET_ERROR(ENOTSUP));
2341         if (blksz == 0)
2342                 blksz = SPA_MINBLOCKSIZE;
2343         ASSERT3U(blksz, <=, spa_maxblocksize(dmu_objset_spa(db->db_objset)));
2344         blksz = P2ROUNDUP(blksz, SPA_MINBLOCKSIZE);
2345
2346         DB_DNODE_ENTER(db);
2347         dn = DB_DNODE(db);
2348         rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
2349         dbuf_new_size(db, blksz, tx);
2350         rw_exit(&dn->dn_struct_rwlock);
2351         DB_DNODE_EXIT(db);
2352
2353         return (0);
2354 }
2355
2356 void
2357 dbuf_rm_spill(dnode_t *dn, dmu_tx_t *tx)
2358 {
2359         dbuf_free_range(dn, DMU_SPILL_BLKID, DMU_SPILL_BLKID, tx);
2360 }
2361
2362 #pragma weak dmu_buf_add_ref = dbuf_add_ref
2363 void
2364 dbuf_add_ref(dmu_buf_impl_t *db, void *tag)
2365 {
2366         int64_t holds = refcount_add(&db->db_holds, tag);
2367         ASSERT(holds > 1);
2368 }
2369
2370 #pragma weak dmu_buf_try_add_ref = dbuf_try_add_ref
2371 boolean_t
```



```
2372 dbuf_try_add_ref(dmu_buf_t *db_fake, objset_t *os, uint64_t obj, uint64_t blkid,
2373     void *tag)
2374 {
2375         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
2376         dmu_buf_impl_t *found_db;
2377         boolean_t result = B_FALSE;
2378
2379         if (db->db_blkid == DMU_BONUS_BLKID)
2380                 found_db = dbuf_find_bonus(os, obj);
2381         else
2382                 found_db = dbuf_find(os, obj, 0, blkid);
2383
2384         if (found_db != NULL) {
2385                 if (db == found_db && dbuf_refcount(db) > db->db_dirtycnt) {
2386                         (void) refcount_add(&db->db_holds, tag);
2387                         result = B_TRUE;
2388                 }
2389                 mutex_exit(&db->db_mtx);
2390         }
2391         return (result);
2392 }
2393
2394 /*
2395  * If you call dbuf_rele() you had better not be referencing the dnode handle
2396  * unless you have some other direct or indirect hold on the dnode. (An indirect
2397  * hold is a hold on one of the dnode's dbufs, including the bonus buffer.)
2398  * Without that, the dbuf_rele() could lead to a dnode_rele() followed by the
2399  * dnode's parent dbuf evicting its dnode handles.
2400  */
2401 void
2402 dbuf_rele(dmu_buf_impl_t *db, void *tag)
2403 {
2404         mutex_enter(&db->db_mtx);
2405         dbuf_rele_and_unlock(db, tag);
2406 }
2407
2408 void
2409 dmu_buf_rele(dmu_buf_t *db, void *tag)
2410 {
2411         dbuf_rele((dmu_buf_impl_t *)db, tag);
2412 }
2413
2414 /*
2415  * dbuf_rele() for an already-locked dbuf.  This is necessary to allow
2416  * db_dirtycnt and db_holds to be updated atomically.
2417  */
2418 void
2419 dbuf_rele_and_unlock(dmu_buf_impl_t *db, void *tag)
2420 {
2421         int64_t holds;
2422
2423         ASSERT(MUTEX_HELD(&db->db_mtx));
2424         DBUF_VERIFY(db);
2425
2426         /*
2427          * Remove the reference to the dbuf before removing its hold on the
2428          * dnode so we can guarantee in dnode_move() that a referenced bonus
2429          * buffer has a corresponding dnode hold.
2430          */
2431         holds = refcount_remove(&db->db_holds, tag);
2432         ASSERT(holds >= 0);
2433
2434         /*
2435          * We can't freeze indirects if there is a possibility that they
2436          * may be modified in the current syncing context.
2437          */
```

```
2438         if (db->db_buf && holds == (db->db_level == 0 ? db->db_dirtycnt : 0))
2439                 arc_buf_freeze(db->db_buf);
2440
2441         if (holds == db->db_dirtycnt &&
2442             db->db_level == 0 && db->db_user_immediate_evict)
2443                 dbuf_evict_user(db);
2444
2445         if (holds == 0) {
2446                 if (db->db_blkid == DMU_BONUS_BLKID) {
2447                         dnode_t *dn;
2448                         boolean_t evict_dbuf = db->db_pending_evict;
2449
2450                         /*
2451                          * If the dnode moves here, we cannot cross this
2452                          * barrier until the move completes.
2453                          */
2454                         DB_DNODE_ENTER(db);
2455
2456                         dn = DB_DNODE(db);
2457                         atomic_dec_32(&dn->dn_dbufs_count);
2458
2459                         /*
2460                          * Decrementing the dbuf count means that the bonus
2461                          * buffer's dnode hold is no longer discounted in
2462                          * dnode_move(). The dnode cannot move until after
2463                          * the dnode_rele() below.
2464                          */
2465                         DB_DNODE_EXIT(db);
2466
2467                         /*
2468                          * Do not reference db after its lock is dropped.
2469                          * Another thread may evict it.
2470                          */
2471                         mutex_exit(&db->db_mtx);
2472
2473                         if (evict_dbuf)
2474                                 dnode_evict_bonus(dn);
2475
2476                         dnode_rele(dn, db);
2477                 } else if (db->db_buf == NULL) {
2478                         /*
2479                          * This is a special case: we never associated this
2480                          * dbuf with any data allocated from the ARC.
2481                          */
2482                         ASSERT(db->db_state == DB_UNCACHED ||
2483                             db->db_state == DB_NOFILL);
2484                         dbuf_evict(db);
2485                 } else if (arc_released(db->db_buf)) {
2486                         arc_buf_t *buf = db->db_buf;
2487                         /*
2488                          * This dbuf has anonymous data associated with it.
2489                          */
2490                         dbuf_clear_data(db);
2491                         VERIFY(arc_buf_remove_ref(buf, db));
2492                         dbuf_evict(db);
2493                 } else {
2494                         VERIFY(!arc_buf_remove_ref(db->db_buf, db));
2495
2496                         /*
2497                          * A dbuf will be eligible for eviction if either the
2498                          * 'primarycache' property is set or a duplicate
2499                          * copy of this buffer is already cached in the arc.
2500                          *
2501                          * In the case of the 'primarycache' a buffer
2502                          * is considered for eviction if it matches the
2503                          * criteria set in the property.
```



```
2504                          *
2505                          * To decide if our buffer is considered a
2506                          * duplicate, we must call into the arc to determine
2507                          * if multiple buffers are referencing the same
2508                          * block on-disk. If so, then we simply evict
2509                          * ourselves.
2510                          */
2511                         if (!DBUF_IS_CACHEABLE(db)) {
2512                                 if (db->db_blkptr != NULL &&
2513                                     !BP_IS_HOLE(db->db_blkptr) &&
2514                                     !BP_IS_EMBEDDED(db->db_blkptr)) {
2515                                         spa_t *spa =
2516                                             dmu_objset_spa(db->db_objset);
2517                                         blkptr_t bp = *db->db_blkptr;
2518                                         dbuf_clear(db);
2519                                         arc_freed(spa, &bp);
2520                                 } else {
2521                                         dbuf_clear(db);
2522                                 }
2523                         } else if (db->db_pending_evict ||
2524                             arc_buf_eviction_needed(db->db_buf)) {
2525                                 dbuf_clear(db);
2526                         } else {
2527                                 mutex_exit(&db->db_mtx);
2528                         }
2529                 }
2530         } else {
2531                 mutex_exit(&db->db_mtx);
2532         }
2533 }
2534
2535 #pragma weak dmu_buf_refcount = dbuf_refcount
2536 uint64_t
2537 dbuf_refcount(dmu_buf_impl_t *db)
2538 {
2539         return (refcount_count(&db->db_holds));
2540 }
2541
2542 void *
2543 dmu_buf_replace_user(dmu_buf_t *db_fake, dmu_buf_user_t *old_user,
2544     dmu_buf_user_t *new_user)
2545 {
2546         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
2547
2548         mutex_enter(&db->db_mtx);
2549         dbuf_verify_user(db, DBVU_NOT_EVICTING);
2550         if (db->db_user == old_user)
2551                 db->db_user = new_user;
2552         else
2553                 old_user = db->db_user;
2554         dbuf_verify_user(db, DBVU_NOT_EVICTING);
2555         mutex_exit(&db->db_mtx);
2556
2557         return (old_user);
2558 }
2559
2560 void *
2561 dmu_buf_set_user(dmu_buf_t *db_fake, dmu_buf_user_t *user)
2562 {
2563         return (dmu_buf_replace_user(db_fake, NULL, user));
2564 }
2565
2566 void *
2567 dmu_buf_set_user_ie(dmu_buf_t *db_fake, dmu_buf_user_t *user)
2568 {
2569         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
```

```
2571         db->db_user_immediate_evict = TRUE;
2572         return (dmu_buf_set_user(db_fake, user));
2573 }

2575 void *
2576 dmu_buf_remove_user(dmu_buf_t *db_fake, dmu_buf_user_t *user)
2577 {
2578         return (dmu_buf_replace_user(db_fake, user, NULL));
2579 }

2581 void *
2582 dmu_buf_get_user(dmu_buf_t *db_fake)
2583 {
2584         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

2586         dbuf_verify_user(db, DBVU_NOT_EVICTING);
2587         return (db->db_user);
2588 }

2590 void
2591 dmu_buf_user_evict_wait()
2592 {
2593         taskq_wait(dbu_evict_taskq);
2594 }

2596 boolean_t
2597 dmu_buf_freeable(dmu_buf_t *dbuf)
2598 {
2599         boolean_t res = B_FALSE;
2600         dmu_buf_impl_t *db = (dmu_buf_impl_t *)dbuf;

2602         if (db->db_blkptr)
2603                 res = dsl_dataset_block_freeable(db->db_objset->os_dsl_dataset,
2604                     db->db_blkptr, db->db_blkptr->blk_birth);

2606         return (res);
2607 }

2609 blkptr_t *
2610 dmu_buf_get_blkptr(dmu_buf_t *db)
2611 {
2612         dmu_buf_impl_t *dbi = (dmu_buf_impl_t *)db;
2613         return (dbi->db_blkptr);
2614 }

2616 static void
2617 dbuf_check_blkptr(dnode_t *dn, dmu_buf_impl_t *db)
2618 {
2619         /* ASSERT(dmu_tx_is_syncing(tx) */
2620         ASSERT(MUTEX_HELD(&db->db_mtx));

2622         if (db->db_blkptr != NULL)
2623                 return;

2625         if (db->db_blkid == DMU_SPILL_BLKID) {
2626                 db->db_blkptr = &dn->dn_phys->dn_spill;
2627                 BP_ZERO(db->db_blkptr);
2628                 return;
2629         }
2630         if (db->db_level == dn->dn_phys->dn_nlevels-1) {
2631                 /*
2632                  * This buffer was allocated at a time when there was
2633                  * no available blkptrs from the dnode, or it was
2634                  * inappropriate to hook it in (i.e., nlevels mis-match).
2635                  */
```



```
2636                 ASSERT(db->db_blkid < dn->dn_phys->dn_nblkptr);
2637                 ASSERT(db->db_parent == NULL);
2638                 db->db_parent = dn->dn_dbuf;
2639                 db->db_blkptr = &dn->dn_phys->dn_blkptr[db->db_blkid];
2640                 DBUF_VERIFY(db);
2641         } else {
2642                 dmu_buf_impl_t *parent = db->db_parent;
2643                 int epbs = dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;

2645                 ASSERT(dn->dn_phys->dn_nlevels > 1);
2646                 if (parent == NULL) {
2647                         mutex_exit(&db->db_mtx);
2648                         rw_enter(&dn->dn_struct_rwlock, RW_READER);
2649                         parent = dbuf_hold_level(dn, db->db_level + 1,
2650                             db->db_blkid >> epbs, db);
2651                         rw_exit(&dn->dn_struct_rwlock);
2652                         mutex_enter(&db->db_mtx);
2653                         db->db_parent = parent;
2654                 }
2655                 db->db_blkptr = (blkptr_t *)parent->db.db_data +
2656                     (db->db_blkid & ((1ULL << epbs) - 1));
2657                 DBUF_VERIFY(db);
2658         }
2659 }

2661 static void
2662 dbuf_sync_indirect(dbuf_dirty_record_t *dr, dmu_tx_t *tx)
2663 {
2664         dmu_buf_impl_t *db = dr->dr_dbuf;
2665         dnode_t *dn;
2666         zio_t *zio;

2668         ASSERT(dmu_tx_is_syncing(tx));

2670         dprintf_dbuf_bp(db, db->db_blkptr, "blkptr=%p", db->db_blkptr);

2672         mutex_enter(&db->db_mtx);

2674         ASSERT(db->db_level > 0);
2675         DBUF_VERIFY(db);

2677         /* Read the block if it hasn't been read yet. */
2678         if (db->db_buf == NULL) {
2679                 mutex_exit(&db->db_mtx);
2680                 (void) dbuf_read(db, NULL, DB_RF_MUST_SUCCEED);
2681                 mutex_enter(&db->db_mtx);
2682         }
2683         ASSERT3U(db->db_state, ==, DB_CACHED);
2684         ASSERT(db->db_buf != NULL);

2686         DB_DNODE_ENTER(db);
2687         dn = DB_DNODE(db);
2688         /* Indirect block size must match what the dnode thinks it is. */
2689         ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2690         dbuf_check_blkptr(dn, db);
2691         DB_DNODE_EXIT(db);

2693         /* Provide the pending dirty record to child dbufs */
2694         db->db_data_pending = dr;

2696         mutex_exit(&db->db_mtx);
2697         dbuf_write(dr, db->db_buf, tx);

2699         zio = dr->dr_zio;
2700         mutex_enter(&dr->dt.di.dr_mtx);
2701         dbuf_sync_list(&dr->dt.di.dr_children, db->db_level - 1, tx);
```

```
2702         ASSERT(list_head(&dr->dt.di.dr_children) == NULL);
2703         mutex_exit(&dr->dt.di.dr_mtx);
2704         zio_nowait(zio);
2705 }

2707 static void
2708 dbuf_sync_leaf(dbuf_dirty_record_t *dr, dmu_tx_t *tx)
2709 {
2710         arc_buf_t **datap = &dr->dt.dl.dr_data;
2711         dmu_buf_impl_t *db = dr->dr_dbuf;
2712         dnode_t *dn;
2713         objset_t *os;
2714         uint64_t txg = tx->tx_txg;

2716         ASSERT(dmu_tx_is_syncing(tx));

2718         dprintf_dbuf_bp(db, db->db_blkptr, "blkptr=%p", db->db_blkptr);

2720         mutex_enter(&db->db_mtx);
2721         /*
2722          * To be synced, we must be dirtied.  But we
2723          * might have been freed after the dirty.
2724          */
2725         if (db->db_state == DB_UNCACHED) {
2726                 /* This buffer has been freed since it was dirtied */
2727                 ASSERT(db->db.db_data == NULL);
2728         } else if (db->db_state == DB_FILL) {
2729                 /* This buffer was freed and is now being re-filled */
2730                 ASSERT(db->db.db_data != dr->dt.dl.dr_data);
2731         } else {
2732                 ASSERT(db->db_state == DB_CACHED || db->db_state == DB_NOFILL);
2733         }
2734         DBUF_VERIFY(db);

2736         DB_DNODE_ENTER(db);
2737         dn = DB_DNODE(db);

2739         if (db->db_blkid == DMU_SPILL_BLKID) {
2740                 mutex_enter(&dn->dn_mtx);
2741                 dn->dn_phys->dn_flags |= DNODE_FLAG_SPILL_BLKPTR;
2742                 mutex_exit(&dn->dn_mtx);
2743         }

2745         /*
2746          * If this is a bonus buffer, simply copy the bonus data into the
2747          * dnode.  It will be written out when the dnode is synced (and it
2748          * will be synced, since it must have been dirty for dbuf_sync to
2749          * be called).
2750          */
2751         if (db->db_blkid == DMU_BONUS_BLKID) {
2752                 dbuf_dirty_record_t **drp;

2754                 ASSERT(*datap != NULL);
2755                 ASSERT0(db->db_level);
2756                 ASSERT3U(dn->dn_phys->dn_bonuslen, <=, DN_MAX_BONUSLEN);
2757                 bcopy(*datap, DN_BONUS(dn->dn_phys), dn->dn_phys->dn_bonuslen);
2758                 DB_DNODE_EXIT(db);

2760                 if (*datap != db->db.db_data) {
2761                         zio_buf_free(*datap, DN_MAX_BONUSLEN);
2762                         arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
2763                 }
2764                 db->db_data_pending = NULL;
2765                 drp = &db->db_last_dirty;
2766                 while (*drp != dr)
2767                         drp = &(*drp)->dr_next;
```



```
2768                 ASSERT(dr->dr_next == NULL);
2769                 ASSERT(dr->dr_dbuf == db);
2770                 *drp = dr->dr_next;
2771                 kmem_free(dr, sizeof (dbuf_dirty_record_t));
2772                 ASSERT(db->db_dirtycnt > 0);
2773                 db->db_dirtycnt -= 1;
2774                 dbuf_rele_and_unlock(db, (void *)(uintptr_t)txg);
2775                 return;
2776         }

2778         os = dn->dn_objset;

2780         /*
2781          * This function may have dropped the db_mtx lock allowing a dmu_sync
2782          * operation to sneak in. As a result, we need to ensure that we
2783          * don't check the dr_override_state until we have returned from
2784          * dbuf_check_blkptr.
2785          */
2786         dbuf_check_blkptr(dn, db);

2788         /*
2789          * If this buffer is in the middle of an immediate write,
2790          * wait for the synchronous IO to complete.
2791          */
2792         while (dr->dt.dl.dr_override_state == DR_IN_DMU_SYNC) {
2793                 ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
2794                 cv_wait(&db->db_changed, &db->db_mtx);
2795                 ASSERT(dr->dt.dl.dr_override_state != DR_NOT_OVERRIDDEN);
2796         }

2798         if (db->db_state != DB_NOFILL &&
2799             dn->dn_object != DMU_META_DNODE_OBJECT &&
2800             refcount_count(&db->db_holds) > 1 &&
2801             dr->dt.dl.dr_override_state != DR_OVERRIDDEN &&
2802             *datap == db->db_buf) {
2803                 /*
2804                  * If this buffer is currently "in use" (i.e., there
2805                  * are active holds and db_data still references it),
2806                  * then make a copy before we start the write so that
2807                  * any modifications from the open txg will not leak
2808                  * into this write.
2809                  *
2810                  * NOTE: this copy does not need to be made for
2811                  * objects only modified in the syncing context (e.g.
2812                  * DNONE_DNODE blocks).
2813                  */
2814                 int blksz = arc_buf_size(*datap);
2815                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
2816                 *datap = arc_buf_alloc(os->os_spa, blksz, db, type);
2817                 bcopy(db->db.db_data, (*datap)->b_data, blksz);
2818         }
2819         db->db_data_pending = dr;

2821         mutex_exit(&db->db_mtx);

2823         dbuf_write(dr, *datap, tx);

2825         ASSERT(!list_link_active(&dr->dr_dirty_node));
2826         if (dn->dn_object == DMU_META_DNODE_OBJECT) {
2827                 list_insert_tail(&dn->dn_dirty_records[txg&TXG_MASK], dr);
2828                 DB_DNODE_EXIT(db);
2829         } else {
2830                 /*
2831                  * Although zio_nowait() does not "wait for an IO", it does
2832                  * initiate the IO. If this is an empty write it seems plausible
2833                  * that the IO could actually be completed before the nowait
```

```
2834                 * returns. We need to DB_DNODE_EXIT() first in case
2835                 * zio_nowait() invalidates the dbuf.
2836                 */
2837                DB_DNODE_EXIT(db);
2838                zio_nowait(dr->dr_zio);
2839        }
2840 }

2842 void
2843 dbuf_sync_list(list_t *list, int level, dmu_tx_t *tx)
2844 {
2845        dbuf_dirty_record_t *dr;

2847        while (dr = list_head(list)) {
2848                if (dr->dr_zio != NULL) {
2849                        /*
2850                         * If we find an already initialized zio then we
2851                         * are processing the meta-dnode, and we have finished.
2852                         * The dbufs for all dnodes are put back on the list
2853                         * during processing, so that we can zio_wait()
2854                         * these IOs after initiating all child IOs.
2855                         */
2856                        ASSERT3U(dr->dr_dbuf->db.db_object, ==,
2857                            DMU_META_DNODE_OBJECT);
2858                        break;
2859                }
2860                if (dr->dr_dbuf->db_blkid != DMU_BONUS_BLKID &&
2861                    dr->dr_dbuf->db_blkid != DMU_SPILL_BLKID) {
2862                        VERIFY3U(dr->dr_dbuf->db_level, ==, level);
2863                }
2864                list_remove(list, dr);
2865                if (dr->dr_dbuf->db_level > 0)
2866                        dbuf_sync_indirect(dr, tx);
2867                else
2868                        dbuf_sync_leaf(dr, tx);
2869        }
2870 }

2872 /* ARGSUSED */
2873 static void
2874 dbuf_write_ready(zio_t *zio, arc_buf_t *buf, void *vdb)
2875 {
2876        dmu_buf_impl_t *db = vdb;
2877        dnode_t *dn;
2878        blkptr_t *bp = zio->io_bp;
2879        blkptr_t *bp_orig = &zio->io_bp_orig;
2880        spa_t *spa = zio->io_spa;
2881        int64_t delta;
2882        uint64_t fill = 0;
2883        int i;

2885        ASSERT3P(db->db_blkptr, ==, bp);

2887        DB_DNODE_ENTER(db);
2888        dn = DB_DNODE(db);
2889        delta = bp_get_dsize_sync(spa, bp) - bp_get_dsize_sync(spa, bp_orig);
2890        dnode_diduse_space(dn, delta - zio->io_prev_space_delta);
2891        zio->io_prev_space_delta = delta;

2893        if (bp->blk_birth != 0) {
2894                ASSERT((db->db_blkid != DMU_SPILL_BLKID &&
2895                    BP_GET_TYPE(bp) == dn->dn_type) ||
2896                    (db->db_blkid == DMU_SPILL_BLKID &&
2897                    BP_GET_TYPE(bp) == dn->dn_bonustype) ||
2898                    BP_IS_EMBEDDED(bp));
2899                ASSERT(BP_GET_LEVEL(bp) == db->db_level);
```



```
2900        }

2902        mutex_enter(&db->db_mtx);

2904 #ifdef ZFS_DEBUG
2905        if (db->db_blkid == DMU_SPILL_BLKID) {
2906                ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
2907                ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
2908                    db->db_blkptr == &dn->dn_phys->dn_spill);
2909        }
2910 #endif

2912        if (db->db_level == 0) {
2913                mutex_enter(&dn->dn_mtx);
2914                if (db->db_blkid > dn->dn_phys->dn_maxblkid &&
2915                    db->db_blkid != DMU_SPILL_BLKID)
2916                        dn->dn_phys->dn_maxblkid = db->db_blkid;
2917                mutex_exit(&dn->dn_mtx);

2919                if (dn->dn_type == DMU_OT_DNODE) {
2920                        dnode_phys_t *dnp = db->db.db_data;
2921                        for (i = db->db.db_size >> DNODE_SHIFT; i > 0;
2922                            i--, dnp++) {
2923                                if (dnp->dn_type != DMU_OT_NONE)
2924                                        fill++;
2925                        }
2926                } else {
2927                        if (BP_IS_HOLE(bp)) {
2928                                fill = 0;
2929                        } else {
2930                                fill = 1;
2931                        }
2932                }
2933        } else {
2934                blkptr_t *ibp = db->db.db_data;
2935                ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2936                for (i = db->db.db_size >> SPA_BLKPTRSHIFT; i > 0; i--, ibp++) {
2937                        if (BP_IS_HOLE(ibp))
2938                                continue;
2939                        fill += BP_GET_FILL(ibp);
2940                }
2941        }
2942        DB_DNODE_EXIT(db);

2944        if (!BP_IS_EMBEDDED(bp))
2945                bp->blk_fill = fill;

2947        mutex_exit(&db->db_mtx);
2948 }

2950 /*
2951  * The SPA will call this callback several times for each zio - once
2952  * for every physical child i/o (zio->io_phys_children times).  This
2953  * allows the DMU to monitor the progress of each logical i/o.  For example,
2954  * there may be 2 copies of an indirect block, or many fragments of a RAID-Z
2955  * block.  There may be a long delay before all copies/fragments are completed,
2956  * so this callback allows us to retire dirty space gradually, as the physical
2957  * i/os complete.
2958  */
2959 /* ARGSUSED */
2960 static void
2961 dbuf_write_physdone(zio_t *zio, arc_buf_t *buf, void *arg)
2962 {
2963        dmu_buf_impl_t *db = arg;
2964        objset_t *os = db->db_objset;
2965        dsl_pool_t *dp = dmu_objset_pool(os);
```

```
2966         dbuf_dirty_record_t *dr;
2967         int delta = 0;

2969         dr = db->db_data_pending;
2970         ASSERT3U(dr->dr_txg, ==, zio->io_txg);

2972         /*
2973          * The callback will be called io_phys_children times.  Retire one
2974          * portion of our dirty space each time we are called.  Any rounding
2975          * error will be cleaned up by dsl_pool_sync()'s call to
2976          * dsl_pool_undirty_space().
2977          */
2978         delta = dr->dr_accounted / zio->io_phys_children;
2979         dsl_pool_undirty_space(dp, delta, zio->io_txg);
2980 }

2982 /* ARGSUSED */
2983 static void
2984 dbuf_write_done(zio_t *zio, arc_buf_t *buf, void *vdb)
2985 {
2986         dmu_buf_impl_t *db = vdb;
2987         blkptr_t *bp_orig = &zio->io_bp_orig;
2988         blkptr_t *bp = db->db_blkptr;
2989         objset_t *os = db->db_objset;
2990         dmu_tx_t *tx = os->os_synctx;
2991         dbuf_dirty_record_t **drp, *dr;

2993         ASSERT0(zio->io_error);
2994         ASSERT(db->db_blkptr == bp);

2996         /*
2997          * For nopwrites and rewrites we ensure that the bp matches our
2998          * original and bypass all the accounting.
2999          */
3000         if (zio->io_flags & (ZIO_FLAG_IO_REWRITE | ZIO_FLAG_NOPWRITE)) {
3001                 ASSERT(BP_EQUAL(bp, bp_orig));
3002         } else {
3003                 dsl_dataset_t *ds = os->os_dsl_dataset;
3004                 (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
3005                 dsl_dataset_block_born(ds, bp, tx);
3006         }

3008         mutex_enter(&db->db_mtx);

3010         DBUF_VERIFY(db);

3012         drp = &db->db_last_dirty;
3013         while ((dr = *drp) != db->db_data_pending)
3014                 drp = &dr->dr_next;
3015         ASSERT(!list_link_active(&dr->dr_dirty_node));
3016         ASSERT(dr->dr_dbuf == db);
3017         ASSERT(dr->dr_next == NULL);
3018         *drp = dr->dr_next;

3020 #ifdef ZFS_DEBUG
3021         if (db->db_blkid == DMU_SPILL_BLKID) {
3022                 dnode_t *dn;

3024                 DB_DNODE_ENTER(db);
3025                 dn = DB_DNODE(db);
3026                 ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
3027                 ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
3028                     db->db_blkptr == &dn->dn_phys->dn_spill);
3029                 DB_DNODE_EXIT(db);
3030         }
3031 #endif
```



```
3033         if (db->db_level == 0) {
3034                 ASSERT(db->db_blkid != DMU_BONUS_BLKID);
3035                 ASSERT(dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN);
3036                 if (db->db_state != DB_NOFILL) {
3037                         if (dr->dt.dl.dr_data != db->db_buf)
3038                                 VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data,
3039                                     db));
3040                         else if (!arc_released(db->db_buf))
3041                                 arc_set_callback(db->db_buf, dbuf_do_evict, db);
3042                 }
3043         } else {
3044                 dnode_t *dn;

3046                 DB_DNODE_ENTER(db);
3047                 dn = DB_DNODE(db);
3048                 ASSERT(list_head(&dr->dt.di.dr_children) == NULL);
3049                 ASSERT3U(db->db.db_size, ==, 1 << dn->dn_phys->dn_indblkshift);
3050                 if (!BP_IS_HOLE(db->db_blkptr)) {
3051                         int epbs =
3052                             dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;
3053                         ASSERT3U(db->db_blkid, <=,
3054                             dn->dn_phys->dn_maxblkid >> (db->db_level * epbs));
3055                         ASSERT3U(BP_GET_LSIZE(db->db_blkptr), ==,
3056                             db->db.db_size);
3057                         if (!arc_released(db->db_buf))
3058                                 arc_set_callback(db->db_buf, dbuf_do_evict, db);
3059                 }
3060                 DB_DNODE_EXIT(db);
3061                 mutex_destroy(&dr->dt.di.dr_mtx);
3062                 list_destroy(&dr->dt.di.dr_children);
3063         }
3064         kmem_free(dr, sizeof (dbuf_dirty_record_t));

3066         cv_broadcast(&db->db_changed);
3067         ASSERT(db->db_dirtycnt > 0);
3068         db->db_dirtycnt -= 1;
3069         db->db_data_pending = NULL;
3070         dbuf_rele_and_unlock(db, (void *)(uintptr_t)tx->tx_txg);
3071 }

3073 static void
3074 dbuf_write_nofill_ready(zio_t *zio)
3075 {
3076         dbuf_write_ready(zio, NULL, zio->io_private);
3077 }

3079 static void
3080 dbuf_write_nofill_done(zio_t *zio)
3081 {
3082         dbuf_write_done(zio, NULL, zio->io_private);
3083 }

3085 static void
3086 dbuf_write_override_ready(zio_t *zio)
3087 {
3088         dbuf_dirty_record_t *dr = zio->io_private;
3089         dmu_buf_impl_t *db = dr->dr_dbuf;

3091         dbuf_write_ready(zio, NULL, db);
3092 }

3094 static void
3095 dbuf_write_override_done(zio_t *zio)
3096 {
3097         dbuf_dirty_record_t *dr = zio->io_private;
```

```
3098         dmu_buf_impl_t *db = dr->dr_dbuf;
3099         blkptr_t *obp = &dr->dt.dl.dr_overridden_by;

3101         mutex_enter(&db->db_mtx);
3102         if (!BP_EQUAL(zio->io_bp, obp)) {
3103                 if (!BP_IS_HOLE(obp))
3104                         dsl_free(spa_get_dsl(zio->io_spa), zio->io_txg, obp);
3105                 arc_release(dr->dt.dl.dr_data, db);
3106         }
3107         mutex_exit(&db->db_mtx);

3109         dbuf_write_done(zio, NULL, db);
3110 }

3112 /* Issue I/O to commit a dirty buffer to disk. */
3113 static void
3114 dbuf_write(dbuf_dirty_record_t *dr, arc_buf_t *data, dmu_tx_t *tx)
3115 {
3116         dmu_buf_impl_t *db = dr->dr_dbuf;
3117         dnode_t *dn;
3118         objset_t *os;
3119         dmu_buf_impl_t *parent = db->db_parent;
3120         uint64_t txg = tx->tx_txg;
3121         zbookmark_phys_t zb;
3122         zio_prop_t zp;
3123         zio_t *zio;
3124         int wp_flag = 0;

3126         DB_DNODE_ENTER(db);
3127         dn = DB_DNODE(db);
3128         os = dn->dn_objset;

3130         if (db->db_state != DB_NOFILL) {
3131                 if (db->db_level > 0 || dn->dn_type == DMU_OT_DNODE) {
3132                         /*
3133                          * Private object buffers are released here rather
3134                          * than in dbuf_dirty() since they are only modified
3135                          * in the syncing context and we don't want the
3136                          * overhead of making multiple copies of the data.
3137                          */
3138                         if (BP_IS_HOLE(db->db_blkptr)) {
3139                                 arc_buf_thaw(data);
3140                         } else {
3141                                 dbuf_release_bp(db);
3142                         }
3143                 }
3144         }

3146         if (parent != dn->dn_dbuf) {
3147                 /* Our parent is an indirect block. */
3148                 /* We have a dirty parent that has been scheduled for write. */
3149                 ASSERT(parent && parent->db_data_pending);
3150                 /* Our parent's buffer is one level closer to the dnode. */
3151                 ASSERT(db->db_level == parent->db_level-1);
3152                 /*
3153                  * We're about to modify our parent's db_data by modifying
3154                  * our block pointer, so the parent must be released.
3155                  */
3156                 ASSERT(arc_released(parent->db_buf));
3157                 zio = parent->db_data_pending->dr_zio;
3158         } else {
3159                 /* Our parent is the dnode itself. */
3160                 ASSERT((db->db_level == dn->dn_phys->dn_nlevels-1 &&
3161                     db->db_blkid != DMU_SPILL_BLKID) ||
3162                     (db->db_blkid == DMU_SPILL_BLKID && db->db_level == 0));
3163                 if (db->db_blkid != DMU_SPILL_BLKID)
```



```
3164                         ASSERT3P(db->db_blkptr, ==,
3165                             &dn->dn_phys->dn_blkptr[db->db_blkid]);
3166                 zio = dn->dn_zio;
3167         }

3169         ASSERT(db->db_level == 0 || data == db->db_buf);
3170         ASSERT3U(db->db_blkptr->blk_birth, <=, txg);
3171         ASSERT(zio);

3173         SET_BOOKMARK(&zb, os->os_dsl_dataset ?
3174             os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
3175             db->db.db_object, db->db_level, db->db_blkid);

3177         if (db->db_blkid == DMU_SPILL_BLKID)
3178                 wp_flag = WP_SPILL;
3179         wp_flag |= (db->db_state == DB_NOFILL) ? WP_NOFILL : 0;

3181         dmu_write_policy(os, dn, db->db_level, wp_flag, &zp);
3182         DB_DNODE_EXIT(db);

3184         if (db->db_level == 0 &&
3185             dr->dt.dl.dr_override_state == DR_OVERRIDDEN) {
3186                 /*
3187                  * The BP for this block has been provided by open context
3188                  * (by dmu_sync() or dmu_buf_write_embedded()).
3189                  */
3190                 void *contents = (data != NULL) ? data->b_data : NULL;

3192                 dr->dr_zio = zio_write(zio, os->os_spa, txg,
3193                     db->db_blkptr, contents, db->db.db_size, &zp,
3194                     dbuf_write_override_ready, NULL, dbuf_write_override_done,
3195                     dr, ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
3196                 mutex_enter(&db->db_mtx);
3197                 dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
3198                 zio_write_override(dr->dr_zio, &dr->dt.dl.dr_overridden_by,
3199                     dr->dt.dl.dr_copies, dr->dt.dl.dr_nopwrite);
3200                 mutex_exit(&db->db_mtx);
3201         } else if (db->db_state == DB_NOFILL) {
3202                 ASSERT(zp.zp_checksum == ZIO_CHECKSUM_OFF ||
3203                     zp.zp_checksum == ZIO_CHECKSUM_NOPARITY);
3204                 dr->dr_zio = zio_write(zio, os->os_spa, txg,
3205                     db->db_blkptr, NULL, db->db.db_size, &zp,
3206                     dbuf_write_nofill_ready, NULL, dbuf_write_nofill_done, db,
3207                     ZIO_PRIORITY_ASYNC_WRITE,
3208                     ZIO_FLAG_MUSTSUCCEED | ZIO_FLAG_NODATA, &zb);
3209         } else {
3210                 ASSERT(arc_released(data));
3211                 dr->dr_zio = arc_write(zio, os->os_spa, txg,
3212                     db->db_blkptr, data, DBUF_IS_L2CACHEABLE(db),
3213                     DBUF_IS_L2COMPRESSIBLE(db), &zp, dbuf_write_ready,
3214                     dbuf_write_physdone, dbuf_write_done, db,
3215                     ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
3216         }
3217 }
```

```
**********************************************************
   57616 Wed Apr  6 14:26:56 2016
new/usr/src/uts/common/fs/zfs/dnode.c
patch first-pass
**********************************************************
______unchanged_portion_omitted_

1055 /*
1056  * errors:
1057  * EINVAL - invalid object number.
1058  * EIO - i/o error.
1059  * succeeds even for free dnodes.
1060  */
1061 int
1062 dnode_hold_impl(objset_t *os, uint64_t object, int flag,
1063     void *tag, dnode_t **dnp)
1064 {
1065         int epb, idx, err;
1066         int drop_struct_lock = FALSE;
1067         int type;
1068         uint64_t blk;
1069         dnode_t *mdn, *dn;
1070         dmu_buf_impl_t *db;
1071         dnode_children_t *children_dnodes;
1072         dnode_handle_t *dnh;

1074         /*
1075          * If you are holding the spa config lock as writer, you shouldn't
1076          * be asking the DMU to do *anything* unless it's the root pool
1077          * which may require us to read from the root filesystem while
1078          * holding some (not all) of the locks as writer.
1079          */
1080         ASSERT(spa_config_held(os->os_spa, SCL_ALL, RW_WRITER) == 0 ||
1081             (spa_is_root(os->os_spa) &&
1082             spa_config_held(os->os_spa, SCL_STATE, RW_WRITER)));

1084         if (object == DMU_USERUSED_OBJECT || object == DMU_GROUPUSED_OBJECT) {
1085                 dn = (object == DMU_USERUSED_OBJECT) ?
1086                     DMU_USERUSED_DNODE(os) : DMU_GROUPUSED_DNODE(os);
1087                 if (dn == NULL)
1088                         return (SET_ERROR(ENOENT));
1089                 type = dn->dn_type;
1090                 if ((flag & DNODE_MUST_BE_ALLOCATED) && type == DMU_OT_NONE)
1091                         return (SET_ERROR(ENOENT));
1092                 if ((flag & DNODE_MUST_BE_FREE) && type != DMU_OT_NONE)
1093                         return (SET_ERROR(EEXIST));
1094                 DNODE_VERIFY(dn);
1095                 (void) refcount_add(&dn->dn_holds, tag);
1096                 *dnp = dn;
1097                 return (0);
1098         }

1100         if (object == 0 || object >= DN_MAX_OBJECT)
1101                 return (SET_ERROR(EINVAL));

1103         mdn = DMU_META_DNODE(os);
1104         ASSERT(mdn->dn_object == DMU_META_DNODE_OBJECT);

1106         DNODE_VERIFY(mdn);

1108         if (!RW_WRITE_HELD(&mdn->dn_struct_rwlock)) {
1109                 rw_enter(&mdn->dn_struct_rwlock, RW_READER);
1110                 drop_struct_lock = TRUE;
1111         }

1113         blk = dbuf_whichblock(mdn, 0, object * sizeof (dnode_phys_t));
```



```
1115         db = dbuf_hold(mdn, blk, FTAG);
1116         if (drop_struct_lock)
1117                 rw_exit(&mdn->dn_struct_rwlock);
1118         if (db == NULL)
1119                 return (SET_ERROR(EIO));
1120         err = dbuf_read(db, NULL, DB_RF_CANFAIL);
1121         if (err) {
1122                 dbuf_rele(db, FTAG);
1123                 return (err);
1124         }

1126         ASSERT3U(db->db.db_size, >=, 1<<DNODE_SHIFT);
1127         epb = db->db.db_size >> DNODE_SHIFT;

1129         idx = object & (epb-1);

1131         ASSERT(DB_DNODE(db)->dn_type == DMU_OT_DNODE);
1132         children_dnodes = dmu_buf_get_user(&db->db);
1133         if (children_dnodes == NULL) {
1134                 int i;
1135                 dnode_children_t *winner;
1136                 children_dnodes = kmem_zalloc(sizeof (dnode_children_t) +
1137                     epb * sizeof (dnode_handle_t), KM_SLEEP);
1138                 children_dnodes->dnc_count = epb;
1139                 dnh = &children_dnodes->dnc_children[0];
1140                 for (i = 0; i < epb; i++) {
1141                         zrl_init(&dnh[i].dnh_zrlock);
1142                 }
1143                 dmu_buf_init_user(&children_dnodes->dnc_dbu, NULL,
1143                 dmu_buf_init_user(&children_dnodes->dnc_dbu,
1144                     dnode_buf_pageout, NULL);
1145                 winner = dmu_buf_set_user(&db->db, &children_dnodes->dnc_dbu);
1146                 if (winner != NULL) {

1148                         for (i = 0; i < epb; i++) {
1149                                 zrl_destroy(&dnh[i].dnh_zrlock);
1150                         }

1152                         kmem_free(children_dnodes, sizeof (dnode_children_t) +
1153                             epb * sizeof (dnode_handle_t));
1154                         children_dnodes = winner;
1155                 }
1156         }
1157         ASSERT(children_dnodes->dnc_count == epb);

1159         dnh = &children_dnodes->dnc_children[idx];
1160         zrl_add(&dnh->dnh_zrlock);
1161         dn = dnh->dnh_dnode;
1162         if (dn == NULL) {
1163                 dnode_phys_t *phys = (dnode_phys_t *)db->db.db_data+idx;

1165                 dn = dnode_create(os, phys, db, object, dnh);
1166         }

1168         mutex_enter(&dn->dn_mtx);
1169         type = dn->dn_type;
1170         if (dn->dn_free_txg ||
1171             ((flag & DNODE_MUST_BE_ALLOCATED) && type == DMU_OT_NONE) ||
1172             ((flag & DNODE_MUST_BE_FREE) &&
1173             (type != DMU_OT_NONE || !refcount_is_zero(&dn->dn_holds)))) {
1174                 mutex_exit(&dn->dn_mtx);
1175                 zrl_remove(&dnh->dnh_zrlock);
1176                 dbuf_rele(db, FTAG);
1177                 return (type == DMU_OT_NONE ? ENOENT : EEXIST);
1178         }
```

```
1179         if (refcount_add(&dn->dn_holds, tag) == 1)
1180                 dbuf_add_ref(db, dnh);
1181         mutex_exit(&dn->dn_mtx);

1183         /* Now we can rely on the hold to prevent the dnode from moving. */
1184         zrl_remove(&dnh->dnh_zrlock);

1186         DNODE_VERIFY(dn);
1187         ASSERT3P(dn->dn_dbuf, ==, db);
1188         ASSERT3U(dn->dn_object, ==, object);
1189         dbuf_rele(db, FTAG);

1191         *dnp = dn;
1192         return (0);
1193 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  101251 Wed Apr  6 14:26:56 2016
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
patch first-pass
**********************************************************
______unchanged_portion_omitted_

 272 static void
 273 dsl_dataset_evict_prep(void *dbu)
 274 {
 275         dsl_dataset_t *ds = dbu;

 277         ASSERT(ds->ds_owner == NULL);

 279         unique_remove(ds->ds_fsid_guid);
 280 }

 282 static void
 283 #endif /* ! codereview */
 284 dsl_dataset_evict(void *dbu)
 285 {
 286         dsl_dataset_t *ds = dbu;

 288         ASSERT(ds->ds_owner == NULL);

 290         ds->ds_dbuf = NULL;

 273         unique_remove(ds->ds_fsid_guid);

 292         if (ds->ds_objset != NULL)
 293                 dmu_objset_evict(ds->ds_objset);

 295         if (ds->ds_prev) {
 296                 dsl_dataset_rele(ds->ds_prev, ds);
 297                 ds->ds_prev = NULL;
 298         }

 300         bplist_destroy(&ds->ds_pending_deadlist);
 301         if (ds->ds_deadlist.dl_os != NULL)
 302                 dsl_deadlist_close(&ds->ds_deadlist);
 303         if (ds->ds_dir)
 304                 dsl_dir_async_rele(ds->ds_dir, ds);

 306         ASSERT(!list_link_active(&ds->ds_synced_link));

 308         list_destroy(&ds->ds_prop_cbs);
 309         mutex_destroy(&ds->ds_lock);
 310         mutex_destroy(&ds->ds_opening_lock);
 311         mutex_destroy(&ds->ds_sendstream_lock);
 312         refcount_destroy(&ds->ds_longholds);

 314         kmem_free(ds, sizeof (dsl_dataset_t));
 315 }
______unchanged_portion_omitted_

 407 int
 408 dsl_dataset_hold_obj(dsl_pool_t *dp, uint64_t dsobj, void *tag,
 409     dsl_dataset_t **dsp)
 410 {
 411         objset_t *mos = dp->dp_meta_objset;
 412         dmu_buf_t *dbuf;
 413         dsl_dataset_t *ds;
 414         int err;
 415         dmu_object_info_t doi;

 417         ASSERT(dsl_pool_config_held(dp));
```



```
 419         err = dmu_bonus_hold(mos, dsobj, tag, &dbuf);
 420         if (err != 0)
 421                 return (err);

 423         /* Make sure dsobj has the correct object type. */
 424         dmu_object_info_from_db(dbuf, &doi);
 425         if (doi.doi_bonus_type != DMU_OT_DSL_DATASET) {
 426                 dmu_buf_rele(dbuf, tag);
 427                 return (SET_ERROR(EINVAL));
 428         }

 430         ds = dmu_buf_get_user(dbuf);
 431         if (ds == NULL) {
 432                 dsl_dataset_t *winner = NULL;

 434                 ds = kmem_zalloc(sizeof (dsl_dataset_t), KM_SLEEP);
 435                 ds->ds_dbuf = dbuf;
 436                 ds->ds_object = dsobj;
 437                 ds->ds_is_snapshot = dsl_dataset_phys(ds)->ds_num_children != 0;

 439                 mutex_init(&ds->ds_lock, NULL, MUTEX_DEFAULT, NULL);
 440                 mutex_init(&ds->ds_opening_lock, NULL, MUTEX_DEFAULT, NULL);
 441                 mutex_init(&ds->ds_sendstream_lock, NULL, MUTEX_DEFAULT, NULL);
 442                 refcount_create(&ds->ds_longholds);

 444                 bplist_create(&ds->ds_pending_deadlist);
 445                 dsl_deadlist_open(&ds->ds_deadlist,
 446                     mos, dsl_dataset_phys(ds)->ds_deadlist_obj);

 448                 list_create(&ds->ds_sendstreams, sizeof (dmu_sendarg_t),
 449                     offsetof(dmu_sendarg_t, dsa_link));

 451                 list_create(&ds->ds_prop_cbs, sizeof (dsl_prop_cb_record_t),
 452                     offsetof(dsl_prop_cb_record_t, cbr_ds_node));

 454                 if (doi.doi_type == DMU_OTN_ZAP_METADATA) {
 455                         for (spa_feature_t f = 0; f < SPA_FEATURES; f++) {
 456                                 if (!(spa_feature_table[f].fi_flags &
 457                                     ZFEATURE_FLAG_PER_DATASET))
 458                                         continue;
 459                                 err = zap_contains(mos, dsobj,
 460                                     spa_feature_table[f].fi_guid);
 461                                 if (err == 0) {
 462                                         ds->ds_feature_inuse[f] = B_TRUE;
 463                                 } else {
 464                                         ASSERT3U(err, ==, ENOENT);
 465                                         err = 0;
 466                                 }
 467                         }
 468                 }

 470                 err = dsl_dir_hold_obj(dp,
 471                     dsl_dataset_phys(ds)->ds_dir_obj, NULL, ds, &ds->ds_dir);
 472                 if (err != 0) {
 473                         mutex_destroy(&ds->ds_lock);
 474                         mutex_destroy(&ds->ds_opening_lock);
 475                         mutex_destroy(&ds->ds_sendstream_lock);
 476                         refcount_destroy(&ds->ds_longholds);
 477                         bplist_destroy(&ds->ds_pending_deadlist);
 478                         dsl_deadlist_close(&ds->ds_deadlist);
 479                         kmem_free(ds, sizeof (dsl_dataset_t));
 480                         dmu_buf_rele(dbuf, tag);
 481                         return (err);
 482                 }
```

```
484                 if (!ds->ds_is_snapshot) {
485                         ds->ds_snapname[0] = '\0';
486                         if (dsl_dataset_phys(ds)->ds_prev_snap_obj != 0) {
487                                 err = dsl_dataset_hold_obj(dp,
488                                     dsl_dataset_phys(ds)->ds_prev_snap_obj,
489                                     ds, &ds->ds_prev);
490                         }
491                         if (doi.doi_type == DMU_OTN_ZAP_METADATA) {
492                                 int zaperr = zap_lookup(mos, ds->ds_object,
493                                     DS_FIELD_BOOKMARK_NAMES,
494                                     sizeof (ds->ds_bookmarks), 1,
495                                     &ds->ds_bookmarks);
496                                 if (zaperr != ENOENT)
497                                         VERIFY0(zaperr);
498                         }
499                 } else {
500                         if (zfs_flags & ZFS_DEBUG_SNAPNAMES)
501                                 err = dsl_dataset_get_snapname(ds);
502                         if (err == 0 &&
503                             dsl_dataset_phys(ds)->ds_userrefs_obj != 0) {
504                                 err = zap_count(
505                                     ds->ds_dir->dd_pool->dp_meta_objset,
506                                     dsl_dataset_phys(ds)->ds_userrefs_obj,
507                                     &ds->ds_userrefs);
508                         }
509                 }

511                 if (err == 0 && !ds->ds_is_snapshot) {
512                         err = dsl_prop_get_int_ds(ds,
513                             zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
514                             &ds->ds_reserved);
515                         if (err == 0) {
516                                 err = dsl_prop_get_int_ds(ds,
517                                     zfs_prop_to_name(ZFS_PROP_REFQUOTA),
518                                     &ds->ds_quota);
519                         }
520                 } else {
521                         ds->ds_reserved = ds->ds_quota = 0;
522                 }

524                 dmu_buf_init_user(&ds->ds_dbu, dsl_dataset_evict_prep,
525                     dsl_dataset_evict, &ds->ds_dbuf);
507                 dmu_buf_init_user(&ds->ds_dbu, dsl_dataset_evict, &ds->ds_dbuf);
526                 if (err == 0)
527                         winner = dmu_buf_set_user_ie(dbuf, &ds->ds_dbu);

529                 if (err != 0 || winner != NULL) {
530                         bplist_destroy(&ds->ds_pending_deadlist);
531                         dsl_deadlist_close(&ds->ds_deadlist);
532                         if (ds->ds_prev)
533                                 dsl_dataset_rele(ds->ds_prev, ds);
534                         dsl_dir_rele(ds->ds_dir, ds);
535                         mutex_destroy(&ds->ds_lock);
536                         mutex_destroy(&ds->ds_opening_lock);
537                         mutex_destroy(&ds->ds_sendstream_lock);
538                         refcount_destroy(&ds->ds_longholds);
539                         kmem_free(ds, sizeof (dsl_dataset_t));
540                         if (err != 0) {
541                                 dmu_buf_rele(dbuf, tag);
542                                 return (err);
543                         }
544                         ds = winner;
545                 } else {
546                         ds->ds_fsid_guid =
547                             unique_insert(dsl_dataset_phys(ds)->ds_fsid_guid);
548                 }
```



```
549         }
550         ASSERT3P(ds->ds_dbuf, ==, dbuf);
551         ASSERT3P(dsl_dataset_phys(ds), ==, dbuf->db_data);
552         ASSERT(dsl_dataset_phys(ds)->ds_prev_snap_obj != 0 ||
553             spa_version(dp->dp_spa) < SPA_VERSION_ORIGIN ||
554             dp->dp_origin_snap == NULL || ds == dp->dp_origin_snap);
555         *dsp = ds;
556         return (0);
557 }
______unchanged_portion_omitted_
```

```
**********************************************************
   56559 Wed Apr  6 14:26:57 2016
new/usr/src/uts/common/fs/zfs/dsl_dir.c
patch first-pass
**********************************************************
______unchanged_portion_omitted_

 155 int
 156 dsl_dir_hold_obj(dsl_pool_t *dp, uint64_t ddobj,
 157     const char *tail, void *tag, dsl_dir_t **ddp)
 158 {
 159         dmu_buf_t *dbuf;
 160         dsl_dir_t *dd;
 161         int err;

 163         ASSERT(dsl_pool_config_held(dp));

 165         err = dmu_bonus_hold(dp->dp_meta_objset, ddobj, tag, &dbuf);
 166         if (err != 0)
 167                 return (err);
 168         dd = dmu_buf_get_user(dbuf);
 169 #ifdef ZFS_DEBUG
 170         {
 171                 dmu_object_info_t doi;
 172                 dmu_object_info_from_db(dbuf, &doi);
 173                 ASSERT3U(doi.doi_bonus_type, ==, DMU_OT_DSL_DIR);
 174                 ASSERT3U(doi.doi_bonus_size, >=, sizeof (dsl_dir_phys_t));
 175         }
 176 #endif
 177         if (dd == NULL) {
 178                 dsl_dir_t *winner;

 180                 dd = kmem_zalloc(sizeof (dsl_dir_t), KM_SLEEP);
 181                 dd->dd_object = ddobj;
 182                 dd->dd_dbuf = dbuf;
 183                 dd->dd_pool = dp;
 184                 mutex_init(&dd->dd_lock, NULL, MUTEX_DEFAULT, NULL);
 185                 dsl_prop_init(dd);

 187                 dsl_dir_snap_cmtime_update(dd);

 189                 if (dsl_dir_phys(dd)->dd_parent_obj) {
 190                         err = dsl_dir_hold_obj(dp,
 191                             dsl_dir_phys(dd)->dd_parent_obj, NULL, dd,
 192                             &dd->dd_parent);
 193                         if (err != 0)
 194                                 goto errout;
 195                         if (tail) {
 196 #ifdef ZFS_DEBUG
 197                                 uint64_t foundobj;

 199                                 err = zap_lookup(dp->dp_meta_objset,
 200                                     dsl_dir_phys(dd->dd_parent)->
 201                                     dd_child_dir_zapobj, tail,
 202                                     sizeof (foundobj), 1, &foundobj);
 203                                 ASSERT(err || foundobj == ddobj);
 204 #endif
 205                                 (void) strcpy(dd->dd_myname, tail);
 206                         } else {
 207                                 err = zap_value_search(dp->dp_meta_objset,
 208                                     dsl_dir_phys(dd->dd_parent)->
 209                                     dd_child_dir_zapobj,
 210                                     ddobj, 0, dd->dd_myname);
 211                         }
 212                         if (err != 0)
 213                                 goto errout;
```



```
 214                 } else {
 215                         (void) strcpy(dd->dd_myname, spa_name(dp->dp_spa));
 216                 }

 218                 if (dsl_dir_is_clone(dd)) {
 219                         dmu_buf_t *origin_bonus;
 220                         dsl_dataset_phys_t *origin_phys;

 222                         /*
 223                          * We can't open the origin dataset, because
 224                          * that would require opening this dsl_dir.
 225                          * Just look at its phys directly instead.
 226                          */
 227                         err = dmu_bonus_hold(dp->dp_meta_objset,
 228                             dsl_dir_phys(dd)->dd_origin_obj, FTAG,
 229                             &origin_bonus);
 230                         if (err != 0)
 231                                 goto errout;
 232                         origin_phys = origin_bonus->db_data;
 233                         dd->dd_origin_txg =
 234                             origin_phys->ds_creation_txg;
 235                         dmu_buf_rele(origin_bonus, FTAG);
 236                 }

 238                 dmu_buf_init_user(&dd->dd_dbu, NULL, dsl_dir_evict,
 239                     &dd->dd_dbuf);
 238                 dmu_buf_init_user(&dd->dd_dbu, dsl_dir_evict, &dd->dd_dbuf);
 240                 winner = dmu_buf_set_user_ie(dbuf, &dd->dd_dbu);
 241                 if (winner != NULL) {
 242                         if (dd->dd_parent)
 243                                 dsl_dir_rele(dd->dd_parent, dd);
 244                         dsl_prop_fini(dd);
 245                         mutex_destroy(&dd->dd_lock);
 246                         kmem_free(dd, sizeof (dsl_dir_t));
 247                         dd = winner;
 248                 } else {
 249                         spa_open_ref(dp->dp_spa, dd);
 250                 }
 251         }

 253         /*
 254          * The dsl_dir_t has both open-to-close and instantiate-to-evict
 255          * holds on the spa.  We need the open-to-close holds because
 256          * otherwise the spa_refcnt wouldn't change when we open a
 257          * dir which the spa also has open, so we could incorrectly
 258          * think it was OK to unload/export/destroy the pool.  We need
 259          * the instantiate-to-evict hold because the dsl_dir_t has a
 260          * pointer to the dd_pool, which has a pointer to the spa_t.
 261          */
 262         spa_open_ref(dp->dp_spa, tag);
 263         ASSERT3P(dd->dd_pool, ==, dp);
 264         ASSERT3U(dd->dd_object, ==, ddobj);
 265         ASSERT3P(dd->dd_dbuf, ==, dbuf);
 266         *ddp = dd;
 267         return (0);

 269 errout:
 270         if (dd->dd_parent)
 271                 dsl_dir_rele(dd->dd_parent, dd);
 272         dsl_prop_fini(dd);
 273         mutex_destroy(&dd->dd_lock);
 274         kmem_free(dd, sizeof (dsl_dir_t));
 275         dmu_buf_rele(dbuf, tag);
 276         return (err);
 277 }
______unchanged_portion_omitted_
```

```
**********************************************************
   52327 Wed Apr  6 14:26:57 2016
new/usr/src/uts/common/fs/zfs/sa.c
patch first-pass
**********************************************************
______unchanged_portion_omitted_

1360 int
1361 sa_handle_get_from_db(objset_t *os, dmu_buf_t *db, void *userp,
1362     sa_handle_type_t hdl_type, sa_handle_t **handlepp)
1363 {
1364         int error = 0;
1365         dmu_object_info_t doi;
1366         sa_handle_t *handle = NULL;

1368 #ifdef ZFS_DEBUG
1369         dmu_object_info_from_db(db, &doi);
1370         ASSERT(doi.doi_bonus_type == DMU_OT_SA ||
1371             doi.doi_bonus_type == DMU_OT_ZNODE);
1372 #endif
1373         /* find handle, if it exists */
1374         /* if one doesn't exist then create a new one, and initialize it */

1376         if (hdl_type == SA_HDL_SHARED)
1377                 handle = dmu_buf_get_user(db);

1379         if (handle == NULL) {
1380                 sa_handle_t *winner = NULL;

1382                 handle = kmem_cache_alloc(sa_cache, KM_SLEEP);
1383                 handle->sa_dbu.dbu_evict_func_prep = NULL;
1384 #endif /* ! codereview */
1385                 handle->sa_dbu.dbu_evict_func = NULL;
1386                 handle->sa_userp = userp;
1387                 handle->sa_bonus = db;
1388                 handle->sa_os = os;
1389                 handle->sa_spill = NULL;
1390                 handle->sa_bonus_tab = NULL;
1391                 handle->sa_spill_tab = NULL;

1393                 error = sa_build_index(handle, SA_BONUS);

1395                 if (hdl_type == SA_HDL_SHARED) {
1396                         dmu_buf_init_user(&handle->sa_dbu, NULL, sa_evict,
1397                             NULL);
1383                         dmu_buf_init_user(&handle->sa_dbu, sa_evict, NULL);
1398                         winner = dmu_buf_set_user_ie(db, &handle->sa_dbu);
1399                 }

1401                 if (winner != NULL) {
1402                         kmem_cache_free(sa_cache, handle);
1403                         handle = winner;
1404                 }
1405         }
1406         *handlepp = handle;

1408         return (error);
1409 }
______unchanged_portion_omitted_
```

```
**********************************************************
   34120 Wed Apr  6 14:26:57 2016
new/usr/src/uts/common/fs/zfs/sys/dmu.h
patch first-pass
**********************************************************
______unchanged_portion_omitted_

 297 /*
 298  * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
 299  */
 300 #define DMU_POOL_DIRECTORY_OBJECT       1
 301 #define DMU_POOL_CONFIG                 "config"
 302 #define DMU_POOL_FEATURES_FOR_WRITE     "features_for_write"
 303 #define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
 304 #define DMU_POOL_FEATURE_DESCRIPTIONS   "feature_descriptions"
 305 #define DMU_POOL_FEATURE_ENABLED_TXG    "feature_enabled_txg"
 306 #define DMU_POOL_ROOT_DATASET           "root_dataset"
 307 #define DMU_POOL_SYNC_BPOBJ             "sync_bplist"
 308 #define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
 309 #define DMU_POOL_ERRLOG_LAST            "errlog_last"
 310 #define DMU_POOL_SPARES                 "spares"
 311 #define DMU_POOL_DEFLATE                "deflate"
 312 #define DMU_POOL_HISTORY                "history"
 313 #define DMU_POOL_PROPS                  "pool_props"
 314 #define DMU_POOL_L2CACHE                "l2cache"
 315 #define DMU_POOL_TMP_USERREFS           "tmp_userrefs"
 316 #define DMU_POOL_DDT                    "DDT-%s-%s-%s"
 317 #define DMU_POOL_DDT_STATS              "DDT-statistics"
 318 #define DMU_POOL_CREATION_VERSION       "creation_version"
 319 #define DMU_POOL_SCAN                   "scan"
 320 #define DMU_POOL_FREE_BPOBJ             "free_bpobj"
 321 #define DMU_POOL_BPTREE_OBJ             "bptree_obj"
 322 #define DMU_POOL_EMPTY_BPOBJ            "empty_bpobj"
 323 #define DMU_POOL_CHECKSUM_SALT          "org.illumos:checksum_salt"

 325 /*
 326  * Allocate an object from this objset.  The range of object numbers
 327  * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
 328  *
 329  * The transaction must be assigned to a txg.  The newly allocated
 330  * object will be "held" in the transaction (ie. you can modify the
 331  * newly allocated object in this transaction).
 332  *
 333  * dmu_object_alloc() chooses an object and returns it in *objectp.
 334  *
 335  * dmu_object_claim() allocates a specific object number.  If that
 336  * number is already allocated, it fails and returns EEXIST.
 337  *
 338  * Return 0 on success, or ENOSPC or EEXIST as specified above.
 339  */
 340 uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
 341     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 342 int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 343     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 344 int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 345     int blocksize, dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *txp);

 347 /*
 348  * Free an object from this objset.
 349  *
 350  * The object's data will be freed as well (ie. you don't need to call
 351  * dmu_free(object, 0, -1, tx)).
 352  *
 353  * The object need not be held in the transaction.
 354  *
 355  * If there are any holds on this object's buffers (via dmu_buf_hold()),
```



```
 356  * or tx holds on the object (via dmu_tx_hold_object()), you can not
 357  * free it; it fails and returns EBUSY.
 358  *
 359  * If the object is not allocated, it fails and returns ENOENT.
 360  *
 361  * Return 0 on success, or EBUSY or ENOENT as specified above.
 362  */
 363 int dmu_object_free(objset_t *os, uint64_t object, dmu_tx_t *tx);

 365 /*
 366  * Find the next allocated or free object.
 367  *
 368  * The objectp parameter is in-out.  It will be updated to be the next
 369  * object which is allocated.  Ignore objects which have not been
 370  * modified since txg.
 371  *
 372  * XXX Can only be called on a objset with no dirty data.
 373  *
 374  * Returns 0 on success, or ENOENT if there are no more objects.
 375  */
 376 int dmu_object_next(objset_t *os, uint64_t *objectp,
 377     boolean_t hole, uint64_t txg);

 379 /*
 380  * Set the data blocksize for an object.
 381  *
 382  * The object cannot have any blocks allcated beyond the first.  If
 383  * the first block is allocated already, the new size must be greater
 384  * than the current block size.  If these conditions are not met,
 385  * ENOTSUP will be returned.
 386  *
 387  * Returns 0 on success, or EBUSY if there are any holds on the object
 388  * contents, or ENOTSUP as described above.
 389  */
 390 int dmu_object_set_blocksize(objset_t *os, uint64_t object, uint64_t size,
 391     int ibs, dmu_tx_t *tx);

 393 /*
 394  * Set the checksum property on a dnode.  The new checksum algorithm will
 395  * apply to all newly written blocks; existing blocks will not be affected.
 396  */
 397 void dmu_object_set_checksum(objset_t *os, uint64_t object, uint8_t checksum,
 398     dmu_tx_t *tx);

 400 /*
 401  * Set the compress property on a dnode.  The new compression algorithm will
 402  * apply to all newly written blocks; existing blocks will not be affected.
 403  */
 404 void dmu_object_set_compress(objset_t *os, uint64_t object, uint8_t compress,
 405     dmu_tx_t *tx);

 407 void
 408 dmu_write_embedded(objset_t *os, uint64_t object, uint64_t offset,
 409     void *data, uint8_t etype, uint8_t comp, int uncompressed_size,
 410     int compressed_size, int byteorder, dmu_tx_t *tx);

 412 /*
 413  * Decide how to write a block: checksum, compression, number of copies, etc.
 414  */
 415 #define WP_NOFILL       0x1
 416 #define WP_DMU_SYNC     0x2
 417 #define WP_SPILL        0x4

 419 void dmu_write_policy(objset_t *os, struct dnode *dn, int level, int wp,
 420     struct zio_prop *zp);
 421 /*
```

```
 422  * The bonus data is accessed more or less like a regular buffer.
 423  * You must dmu_bonus_hold() to get the buffer, which will give you a
 424  * dmu_buf_t with db_offset==-1ULL, and db_size = the size of the bonus
 425  * data.  As with any normal buffer, you must call dmu_buf_read() to
 426  * read db_data, dmu_buf_will_dirty() before modifying it, and the
 427  * object must be held in an assigned transaction before calling
 428  * dmu_buf_will_dirty.  You may use dmu_buf_set_user() on the bonus
 429  * buffer as well.  You must release your hold with dmu_buf_rele().
 430  *
 431  * Returns ENOENT, EIO, or 0.
 432  */
 433 int dmu_bonus_hold(objset_t *os, uint64_t object, void *tag, dmu_buf_t **);
 434 int dmu_bonus_max(void);
 435 int dmu_set_bonus(dmu_buf_t *, int, dmu_tx_t *);
 436 int dmu_set_bonustype(dmu_buf_t *, dmu_object_type_t, dmu_tx_t *);
 437 dmu_object_type_t dmu_get_bonustype(dmu_buf_t *);
 438 int dmu_rm_spill(objset_t *, uint64_t, dmu_tx_t *);

 440 /*
 441  * Special spill buffer support used by "SA" framework
 442  */

 444 int dmu_spill_hold_by_bonus(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);
 445 int dmu_spill_hold_by_dnode(struct dnode *dn, uint32_t flags,
 446     void *tag, dmu_buf_t **dbp);
 447 int dmu_spill_hold_existing(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);

 449 /*
 450  * Obtain the DMU buffer from the specified object which contains the
 451  * specified offset.  dmu_buf_hold() puts a "hold" on the buffer, so
 452  * that it will remain in memory.  You must release the hold with
 453  * dmu_buf_rele().  You musn't access the dmu_buf_t after releasing your
 454  * hold.  You must have a hold on any dmu_buf_t* you pass to the DMU.
 455  *
 456  * You must call dmu_buf_read, dmu_buf_will_dirty, or dmu_buf_will_fill
 457  * on the returned buffer before reading or writing the buffer's
 458  * db_data.  The comments for those routines describe what particular
 459  * operations are valid after calling them.
 460  *
 461  * The object number must be a valid, allocated object number.
 462  */
 463 int dmu_buf_hold(objset_t *os, uint64_t object, uint64_t offset,
 464     void *tag, dmu_buf_t **, int flags);

 466 /*
 467  * Add a reference to a dmu buffer that has already been held via
 468  * dmu_buf_hold() in the current context.
 469  */
 470 void dmu_buf_add_ref(dmu_buf_t *db, void* tag);

 472 /*
 473  * Attempt to add a reference to a dmu buffer that is in an unknown state,
 474  * using a pointer that may have been invalidated by eviction processing.
 475  * The request will succeed if the passed in dbuf still represents the
 476  * same os/object/blkid, is ineligible for eviction, and has at least
 477  * one hold by a user other than the syncer.
 478  */
 479 boolean_t dmu_buf_try_add_ref(dmu_buf_t *, objset_t *os, uint64_t object,
 480     uint64_t blkid, void *tag);

 482 void dmu_buf_rele(dmu_buf_t *db, void *tag);
 483 uint64_t dmu_buf_refcount(dmu_buf_t *db);

 485 /*
 486  * dmu_buf_hold_array holds the DMU buffers which contain all bytes in a
 487  * range of an object.  A pointer to an array of dmu_buf_t*'s is
```



```
 488  * returned (in *dbpp).
 489  *
 490  * dmu_buf_rele_array releases the hold on an array of dmu_buf_t*'s, and
 491  * frees the array.  The hold on the array of buffers MUST be released
 492  * with dmu_buf_rele_array.  You can NOT release the hold on each buffer
 493  * individually with dmu_buf_rele.
 494  */
 495 int dmu_buf_hold_array_by_bonus(dmu_buf_t *db, uint64_t offset,
 496     uint64_t length, boolean_t read, void *tag,
 497     int *numbufsp, dmu_buf_t ***dbpp);
 498 void dmu_buf_rele_array(dmu_buf_t **, int numbufs, void *tag);

 500 typedef void dmu_buf_evict_func_t(void *user_ptr);

 502 /*
 503  * A DMU buffer user object may be associated with a dbuf for the
 504  * duration of its lifetime.  This allows the user of a dbuf (client)
 505  * to attach private data to a dbuf (e.g. in-core only data such as a
 506  * dnode_children_t, zap_t, or zap_leaf_t) and be optionally notified
 507  * when that dbuf has been evicted.  Clients typically respond to the
 508  * eviction notification by freeing their private data, thus ensuring
 509  * the same lifetime for both dbuf and private data.
 510  *
 511  * The mapping from a dmu_buf_user_t to any client private data is the
 512  * client's responsibility.  All current consumers of the API with private
 513  * data embed a dmu_buf_user_t as the first member of the structure for
 514  * their private data.  This allows conversions between the two types
 515  * with a simple cast.  Since the DMU buf user API never needs access
 516  * to the private data, other strategies can be employed if necessary
 517  * or convenient for the client (e.g. using container_of() to do the
 518  * conversion for private data that cannot have the dmu_buf_user_t as
 519  * its first member).
 520  *
 521  * Eviction callbacks are executed without the dbuf mutex held or any
 522  * other type of mechanism to guarantee that the dbuf is still available.
 523  * For this reason, users must assume the dbuf has already been freed
 524  * and not reference the dbuf from the callback context.
 525  *
 526  * Users requesting "immediate eviction" are notified as soon as the dbuf
 527  * is only referenced by dirty records (dirties == holds).  Otherwise the
 528  * notification occurs after eviction processing for the dbuf begins.
 529  */
 530 typedef struct dmu_buf_user {
 531         /*
 532          * Asynchronous user eviction callback state.
 533          */
 534         taskq_ent_t     dbu_tqent;

 536         /*
 537          * This instance's eviction function pointers.
 538          *
 539          * dbu_evict_func_prep is called synchronously while dbu_evict_func
 540          * is executed asynchronously on a taskq.
 541          */
 542         dmu_buf_evict_func_t *dbu_evict_func_prep;
 536         /* This instance's eviction function pointer. */
 543         dmu_buf_evict_func_t *dbu_evict_func;
 544 #ifdef ZFS_DEBUG
 545         /*
 546          * Pointer to user's dbuf pointer.  NULL for clients that do
 547          * not associate a dbuf with their user data.
 548          *
 549          * The dbuf pointer is cleared upon eviction so as to catch
 550          * use-after-evict bugs in clients.
 551          */
 552         dmu_buf_t **dbu_clear_on_evict_dbufp;
```

```
#endif
} dmu_buf_user_t;

/*
 * Initialize the given dmu_buf_user_t instance with the eviction function
 * evict_func, to be called when the user is evicted.
 *
 * NOTE: This function should only be called once on a given dmu_buf_user_t.
 *       To allow enforcement of this, dbu must already be zeroed on entry.
 */
#ifdef __lint
/* Very ugly, but it beats issuing suppression directives in many Makefiles. */
extern void
dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func_prep,
    dmu_buf_evict_func_t *evict_func, dmu_buf_t **clear_on_evict_dbufp);
dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func,
    dmu_buf_t **clear_on_evict_dbufp);
#else /* __lint */
inline void
dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func_prep,
    dmu_buf_evict_func_t *evict_func, dmu_buf_t **clear_on_evict_dbufp)
dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func,
    dmu_buf_t **clear_on_evict_dbufp)
{
        ASSERT(dbu->dbu_evict_func_prep == NULL);
#endif /* ! codereview */
        ASSERT(dbu->dbu_evict_func == NULL);
        ASSERT(evict_func != NULL);
        dbu->dbu_evict_func_prep = evict_func_prep;
#endif /* ! codereview */
        dbu->dbu_evict_func = evict_func;
#ifdef ZFS_DEBUG
        dbu->dbu_clear_on_evict_dbufp = clear_on_evict_dbufp;
#endif
}
#endif /* __lint */

/*
 * Attach user data to a dbuf and mark it for normal (when the dbuf's
 * data is cleared or its reference count goes to zero) eviction processing.
 *
 * Returns NULL on success, or the existing user if another user currently
 * owns the buffer.
 */
void *dmu_buf_set_user(dmu_buf_t *db, dmu_buf_user_t *user);

/*
 * Attach user data to a dbuf and mark it for immediate (its dirty and
 * reference counts are equal) eviction processing.
 *
 * Returns NULL on success, or the existing user if another user currently
 * owns the buffer.
 */
void *dmu_buf_set_user_ie(dmu_buf_t *db, dmu_buf_user_t *user);

/*
 * Replace the current user of a dbuf.
 *
 * If given the current user of a dbuf, replaces the dbuf's user with
 * "new_user" and returns the user data pointer that was replaced.
 * Otherwise returns the current, and unmodified, dbuf user pointer.
 */
void *dmu_buf_replace_user(dmu_buf_t *db,
    dmu_buf_user_t *old_user, dmu_buf_user_t *new_user);

/*
```



```
 * Remove the specified user data for a DMU buffer.
 *
 * Returns the user that was removed on success, or the current user if
 * another user currently owns the buffer.
 */
void *dmu_buf_remove_user(dmu_buf_t *db, dmu_buf_user_t *user);

/*
 * Returns the user data (dmu_buf_user_t *) associated with this dbuf.
 */
void *dmu_buf_get_user(dmu_buf_t *db);

/* Block until any in-progress dmu buf user evictions complete. */
void dmu_buf_user_evict_wait(void);

/*
 * Returns the blkptr associated with this dbuf, or NULL if not set.
 */
struct blkptr *dmu_buf_get_blkptr(dmu_buf_t *db);

/*
 * Indicate that you are going to modify the buffer's data (db_data).
 *
 * The transaction (tx) must be assigned to a txg (ie. you've called
 * dmu_tx_assign()).  The buffer's object must be held in the tx
 * (ie. you've called dmu_tx_hold_object(tx, db->db_object)).
 */
void dmu_buf_will_dirty(dmu_buf_t *db, dmu_tx_t *tx);

/*
 * Tells if the given dbuf is freeable.
 */
boolean_t dmu_buf_freeable(dmu_buf_t *);

/*
 * You must create a transaction, then hold the objects which you will
 * (or might) modify as part of this transaction.  Then you must assign
 * the transaction to a transaction group.  Once the transaction has
 * been assigned, you can modify buffers which belong to held objects as
 * part of this transaction.  You can't modify buffers before the
 * transaction has been assigned; you can't modify buffers which don't
 * belong to objects which this transaction holds; you can't hold
 * objects once the transaction has been assigned.  You may hold an
 * object which you are going to free (with dmu_object_free()), but you
 * don't have to.
 *
 * You can abort the transaction before it has been assigned.
 *
 * Note that you may hold buffers (with dmu_buf_hold) at any time,
 * regardless of transaction state.
 */

#define DMU_NEW_OBJECT  (-1ULL)
#define DMU_OBJECT_END  (-1ULL)

dmu_tx_t *dmu_tx_create(objset_t *os);
void dmu_tx_hold_write(dmu_tx_t *tx, uint64_t object, uint64_t off, int len);
void dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off,
    uint64_t len);
void dmu_tx_hold_zap(dmu_tx_t *tx, uint64_t object, int add, const char *name);
void dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object);
void dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object);
void dmu_tx_hold_sa(dmu_tx_t *tx, struct sa_handle *hdl, boolean_t may_grow);
void dmu_tx_hold_sa_create(dmu_tx_t *tx, int total_size);
void dmu_tx_abort(dmu_tx_t *tx);
int dmu_tx_assign(dmu_tx_t *tx, enum txg_how txg_how);
```

```
void dmu_tx_wait(dmu_tx_t *tx);
void dmu_tx_commit(dmu_tx_t *tx);
void dmu_tx_mark_netfree(dmu_tx_t *tx);

/*
 * To register a commit callback, dmu_tx_callback_register() must be called.
 *
 * dcb_data is a pointer to caller private data that is passed on as a
 * callback parameter. The caller is responsible for properly allocating and
 * freeing it.
 *
 * When registering a callback, the transaction must be already created, but
 * it cannot be committed or aborted. It can be assigned to a txg or not.
 *
 * The callback will be called after the transaction has been safely written
 * to stable storage and will also be called if the dmu_tx is aborted.
 * If there is any error which prevents the transaction from being committed to
 * disk, the callback will be called with a value of error != 0.
 */
typedef void dmu_tx_callback_func_t(void *dcb_data, int error);

void dmu_tx_callback_register(dmu_tx_t *tx, dmu_tx_callback_func_t *dcb_func,
    void *dcb_data);

/*
 * Free up the data blocks for a defined range of a file.  If size is
 * -1, the range from offset to end-of-file is freed.
 */
int dmu_free_range(objset_t *os, uint64_t object, uint64_t offset,
        uint64_t size, dmu_tx_t *tx);
int dmu_free_long_range(objset_t *os, uint64_t object, uint64_t offset,
        uint64_t size);
int dmu_free_long_object(objset_t *os, uint64_t object);

/*
 * Convenience functions.
 *
 * Canfail routines will return 0 on success, or an errno if there is a
 * nonrecoverable I/O error.
 */
#define DMU_READ_PREFETCH       0 /* prefetch */
#define DMU_READ_NO_PREFETCH    1 /* don't prefetch */
int dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        void *buf, uint32_t flags);
void dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        const void *buf, dmu_tx_t *tx);
void dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        dmu_tx_t *tx);
int dmu_read_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size);
int dmu_read_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size);
int dmu_write_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size,
    dmu_tx_t *tx);
int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
    dmu_tx_t *tx);
int dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset,
    uint64_t size, struct page *pp, dmu_tx_t *tx);
struct arc_buf *dmu_request_arcbuf(dmu_buf_t *handle, int size);
void dmu_return_arcbuf(struct arc_buf *buf);
void dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, struct arc_buf *buf,
    dmu_tx_t *tx);
int dmu_xuio_init(struct xuio *uio, int niov);
void dmu_xuio_fini(struct xuio *uio);
int dmu_xuio_add(struct xuio *uio, struct arc_buf *abuf, offset_t off,
    size_t n);
int dmu_xuio_cnt(struct xuio *uio);
struct arc_buf *dmu_xuio_arcbuf(struct xuio *uio, int i);
```



```
void dmu_xuio_clear(struct xuio *uio, int i);
void xuio_stat_wbuf_copied();
void xuio_stat_wbuf_nocopy();

extern boolean_t zfs_prefetch_disable;
extern int zfs_max_recordsize;

/*
 * Asynchronously try to read in the data.
 */
void dmu_prefetch(objset_t *os, uint64_t object, int64_t level, uint64_t offset,
    uint64_t len, enum zio_priority pri);

typedef struct dmu_object_info {
        /* All sizes are in bytes unless otherwise indicated. */
        uint32_t doi_data_block_size;
        uint32_t doi_metadata_block_size;
        dmu_object_type_t doi_type;
        dmu_object_type_t doi_bonus_type;
        uint64_t doi_bonus_size;
        uint8_t doi_indirection;                /* 2 = dnode->indirect->data */
        uint8_t doi_checksum;
        uint8_t doi_compress;
        uint8_t doi_nblkptr;
        uint8_t doi_pad[4];
        uint64_t doi_physical_blocks_512;       /* data + metadata, 512b blks */
        uint64_t doi_max_offset;
        uint64_t doi_fill_count;                /* number of non-empty blocks */
} dmu_object_info_t;

typedef void arc_byteswap_func_t(void *buf, size_t size);

typedef struct dmu_object_type_info {
        dmu_object_byteswap_t   ot_byteswap;
        boolean_t               ot_metadata;
        char                    *ot_name;
} dmu_object_type_info_t;

typedef struct dmu_object_byteswap_info {
        arc_byteswap_func_t     *ob_func;
        char                    *ob_name;
} dmu_object_byteswap_info_t;

extern const dmu_object_type_info_t dmu_ot[DMU_OT_NUMTYPES];
extern const dmu_object_byteswap_info_t dmu_ot_byteswap[DMU_BSWAP_NUMFUNCS];

/*
 * Get information on a DMU object.
 *
 * Return 0 on success or ENOENT if object is not allocated.
 *
 * If doi is NULL, just indicates whether the object exists.
 */
int dmu_object_info(objset_t *os, uint64_t object, dmu_object_info_t *doi);
/* Like dmu_object_info, but faster if you have a held dnode in hand. */
void dmu_object_info_from_dnode(struct dnode *dn, dmu_object_info_t *doi);
/* Like dmu_object_info, but faster if you have a held dbuf in hand. */
void dmu_object_info_from_db(dmu_buf_t *db, dmu_object_info_t *doi);
/*
 * Like dmu_object_info_from_db, but faster still when you only care about
 * the size.  This is specifically optimized for zfs_getattr().
 */
void dmu_object_size_from_db(dmu_buf_t *db, uint32_t *blksize,
    u_longlong_t *nblk512);

typedef struct dmu_objset_stats {
```

```
813         uint64_t dds_num_clones; /* number of clones of this */
814         uint64_t dds_creation_txg;
815         uint64_t dds_guid;
816         dmu_objset_type_t dds_type;
817         uint8_t dds_is_snapshot;
818         uint8_t dds_inconsistent;
819         char dds_origin[MAXNAMELEN];
820 } dmu_objset_stats_t;

822 /*
823  * Get stats on a dataset.
824  */
825 void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);

827 /*
828  * Add entries to the nvlist for all the objset's properties.  See
829  * zfs_prop_table[] and zfs(1m) for details on the properties.
830  */
831 void dmu_objset_stats(objset_t *os, struct nvlist *nv);

833 /*
834  * Get the space usage statistics for statvfs().
835  *
836  * refdbytes is the amount of space "referenced" by this objset.
837  * availbytes is the amount of space available to this objset, taking
838  * into account quotas & reservations, assuming that no other objsets
839  * use the space first.  These values correspond to the 'referenced' and
840  * 'available' properties, described in the zfs(1m) manpage.
841  *
842  * usedobjs and availobjs are the number of objects currently allocated,
843  * and available.
844  */
845 void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
846     uint64_t *usedobjsp, uint64_t *availobjsp);

848 /*
849  * The fsid_guid is a 56-bit ID that can change to avoid collisions.
850  * (Contrast with the ds_guid which is a 64-bit ID that will never
851  * change, so there is a small probability that it will collide.)
852  */
853 uint64_t dmu_objset_fsid_guid(objset_t *os);

855 /*
856  * Get the [cm]time for an objset's snapshot dir
857  */
858 timestruc_t dmu_objset_snap_cmtime(objset_t *os);

860 int dmu_objset_is_snapshot(objset_t *os);

862 extern struct spa *dmu_objset_spa(objset_t *os);
863 extern struct zilog *dmu_objset_zil(objset_t *os);
864 extern struct dsl_pool *dmu_objset_pool(objset_t *os);
865 extern struct dsl_dataset *dmu_objset_ds(objset_t *os);
866 extern void dmu_objset_name(objset_t *os, char *buf);
867 extern dmu_objset_type_t dmu_objset_type(objset_t *os);
868 extern uint64_t dmu_objset_id(objset_t *os);
869 extern zfs_sync_type_t dmu_objset_syncprop(objset_t *os);
870 extern zfs_logbias_op_t dmu_objset_logbias(objset_t *os);
871 extern int dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
872     uint64_t *id, uint64_t *offp, boolean_t *case_conflict);
873 extern int dmu_snapshot_realname(objset_t *os, char *name, char *real,
874     int maxlen, boolean_t *conflict);
875 extern int dmu_dir_list_next(objset_t *os, int namelen, char *name,
876     uint64_t *idp, uint64_t *offp);

878 typedef int objset_used_cb_t(dmu_object_type_t bonustype,
```



```
879     void *bonus, uint64_t *userp, uint64_t *groupp);
880 extern void dmu_objset_register_type(dmu_objset_type_t ost,
881     objset_used_cb_t *cb);
882 extern void dmu_objset_set_user(objset_t *os, void *user_ptr);
883 extern void *dmu_objset_get_user(objset_t *os);

885 /*
886  * Return the txg number for the given assigned transaction.
887  */
888 uint64_t dmu_tx_get_txg(dmu_tx_t *tx);

890 /*
891  * Synchronous write.
892  * If a parent zio is provided this function initiates a write on the
893  * provided buffer as a child of the parent zio.
894  * In the absence of a parent zio, the write is completed synchronously.
895  * At write completion, blk is filled with the bp of the written block.
896  * Note that while the data covered by this function will be on stable
897  * storage when the write completes this new data does not become a
898  * permanent part of the file until the associated transaction commits.
899  */

901 /*
902  * {zfs,zvol,ztest}_get_done() args
903  */
904 typedef struct zgd {
905         struct zilog    *zgd_zilog;
906         struct blkptr   *zgd_bp;
907         dmu_buf_t       *zgd_db;
908         struct rl       *zgd_rl;
909         void            *zgd_private;
910 } zgd_t;

912 typedef void dmu_sync_cb_t(zgd_t *arg, int error);
913 int dmu_sync(struct zio *zio, uint64_t txg, dmu_sync_cb_t *done, zgd_t *zgd);

915 /*
916  * Find the next hole or data block in file starting at *off
917  * Return found offset in *off. Return ESRCH for end of file.
918  */
919 int dmu_offset_next(objset_t *os, uint64_t object, boolean_t hole,
920     uint64_t *off);

922 /*
923  * Check if a DMU object has any dirty blocks. If so, sync out
924  * all pending transaction groups. Otherwise, this function
925  * does not alter DMU state. This could be improved to only sync
926  * out the necessary transaction groups for this particular
927  * object.
928  */
929 int dmu_object_wait_synced(objset_t *os, uint64_t object);

931 /*
932  * Initial setup and final teardown.
933  */
934 extern void dmu_init(void);
935 extern void dmu_fini(void);

937 typedef void (*dmu_traverse_cb_t)(objset_t *os, void *arg, struct blkptr *bp,
938     uint64_t object, uint64_t offset, int len);
939 void dmu_traverse_objset(objset_t *os, uint64_t txg_start,
940     dmu_traverse_cb_t cb, void *arg);

942 int dmu_diff(const char *tosnap_name, const char *fromsnap_name,
943     struct vnode *vp, offset_t *offp);
```

```
945 /* CRC64 table */
946 #define ZFS_CRC64_POLY  0xC96C5795D7870F42ULL   /* ECMA-182, reflected form */
947 extern uint64_t zfs_crc64_table[256];

949 extern int zfs_mdcomp_disable;

951 #ifdef  __cplusplus
952 }
953 #endif

955 #endif  /* _SYS_DMU_H */
```

```
**********************************************************
   33565 Wed Apr  6 14:26:57 2016
new/usr/src/uts/common/fs/zfs/zap.c
patch first-pass
**********************************************************
______unchanged_portion_omitted_

  73 void
  74 fzap_upgrade(zap_t *zap, dmu_tx_t *tx, zap_flags_t flags)
  75 {
  76         dmu_buf_t *db;
  77         zap_leaf_t *l;
  78         int i;
  79         zap_phys_t *zp;

  81         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
  82         zap->zap_ismicro = FALSE;

  84         zap->zap_dbu.dbu_evict_func_prep = NULL;
  85 #endif /* ! codereview */
  86         zap->zap_dbu.dbu_evict_func = zap_evict;

  88         mutex_init(&zap->zap_f.zap_num_entries_mtx, 0, 0, 0);
  89         zap->zap_f.zap_block_shift = highbit64(zap->zap_dbuf->db_size) - 1;

  91         zp = zap_f_phys(zap);
  92         /*
  93          * explicitly zero it since it might be coming from an
  94          * initialized microzap
  95          */
  96         bzero(zap->zap_dbuf->db_data, zap->zap_dbuf->db_size);
  97         zp->zap_block_type = ZBT_HEADER;
  98         zp->zap_magic = ZAP_MAGIC;

 100         zp->zap_ptrtbl.zt_shift = ZAP_EMBEDDED_PTRTBL_SHIFT(zap);

 102         zp->zap_freeblk = 2;            /* block 1 will be the first leaf */
 103         zp->zap_num_leafs = 1;
 104         zp->zap_num_entries = 0;
 105         zp->zap_salt = zap->zap_salt;
 106         zp->zap_normflags = zap->zap_normflags;
 107         zp->zap_flags = flags;

 109         /* block 1 will be the first leaf */
 110         for (i = 0; i < (1<<zp->zap_ptrtbl.zt_shift); i++)
 111                 ZAP_EMBEDDED_PTRTBL_ENT(zap, i) = 1;

 113         /*
 114          * set up block 1 - the first leaf
 115          */
 116         VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
 117             1<<FZAP_BLOCK_SHIFT(zap), FTAG, &db, DMU_READ_NO_PREFETCH));
 118         dmu_buf_will_dirty(db, tx);

 120         l = kmem_zalloc(sizeof (zap_leaf_t), KM_SLEEP);
 121         l->l_dbuf = db;

 123         zap_leaf_init(l, zp->zap_normflags != 0);

 125         kmem_free(l, sizeof (zap_leaf_t));
 126         dmu_buf_rele(db, FTAG);
 127 }

 129 static int
 130 zap_tryupgradedir(zap_t *zap, dmu_tx_t *tx)
 131 {
```



```
 132         if (RW_WRITE_HELD(&zap->zap_rwlock))
 133                 return (1);
 134         if (rw_tryupgrade(&zap->zap_rwlock)) {
 135                 dmu_buf_will_dirty(zap->zap_dbuf, tx);
 136                 return (1);
 137         }
 138         return (0);
 139 }

 141 /*
 142  * Generic routines for dealing with the pointer & cookie tables.
 143  */

 145 static int
 146 zap_table_grow(zap_t *zap, zap_table_phys_t *tbl,
 147     void (*transfer_func)(const uint64_t *src, uint64_t *dst, int n),
 148     dmu_tx_t *tx)
 149 {
 150         uint64_t b, newblk;
 151         dmu_buf_t *db_old, *db_new;
 152         int err;
 153         int bs = FZAP_BLOCK_SHIFT(zap);
 154         int hepb = 1<<(bs-4);
 155         /* hepb = half the number of entries in a block */

 157         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
 158         ASSERT(tbl->zt_blk != 0);
 159         ASSERT(tbl->zt_numblks > 0);

 161         if (tbl->zt_nextblk != 0) {
 162                 newblk = tbl->zt_nextblk;
 163         } else {
 164                 newblk = zap_allocate_blocks(zap, tbl->zt_numblks * 2);
 165                 tbl->zt_nextblk = newblk;
 166                 ASSERT0(tbl->zt_blks_copied);
 167                 dmu_prefetch(zap->zap_objset, zap->zap_object, 0,
 168                     tbl->zt_blk << bs, tbl->zt_numblks << bs,
 169                     ZIO_PRIORITY_SYNC_READ);
 170         }

 172         /*
 173          * Copy the ptrtbl from the old to new location.
 174          */

 176         b = tbl->zt_blks_copied;
 177         err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
 178             (tbl->zt_blk + b) << bs, FTAG, &db_old, DMU_READ_NO_PREFETCH);
 179         if (err)
 180                 return (err);

 182         /* first half of entries in old[b] go to new[2*b+0] */
 183         VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
 184             (newblk + 2*b+0) << bs, FTAG, &db_new, DMU_READ_NO_PREFETCH));
 185         dmu_buf_will_dirty(db_new, tx);
 186         transfer_func(db_old->db_data, db_new->db_data, hepb);
 187         dmu_buf_rele(db_new, FTAG);

 189         /* second half of entries in old[b] go to new[2*b+1] */
 190         VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
 191             (newblk + 2*b+1) << bs, FTAG, &db_new, DMU_READ_NO_PREFETCH));
 192         dmu_buf_will_dirty(db_new, tx);
 193         transfer_func((uint64_t *)db_old->db_data + hepb,
 194             db_new->db_data, hepb);
 195         dmu_buf_rele(db_new, FTAG);

 197         dmu_buf_rele(db_old, FTAG);
```

```
199         tbl->zt_blks_copied++;
201         dprintf("copied block %llu of %llu\n",
202             tbl->zt_blks_copied, tbl->zt_numblks);
204         if (tbl->zt_blks_copied == tbl->zt_numblks) {
205                 (void) dmu_free_range(zap->zap_objset, zap->zap_object,
206                     tbl->zt_blk << bs, tbl->zt_numblks << bs, tx);
208                 tbl->zt_blk = newblk;
209                 tbl->zt_numblks *= 2;
210                 tbl->zt_shift++;
211                 tbl->zt_nextblk = 0;
212                 tbl->zt_blks_copied = 0;
214                 dprintf("finished; numblocks now %llu (%lluk entries)\n",
215                     tbl->zt_numblks, 1<<(tbl->zt_shift-10));
216         }
218         return (0);
219 }
221 static int
222 zap_table_store(zap_t *zap, zap_table_phys_t *tbl, uint64_t idx, uint64_t val,
223     dmu_tx_t *tx)
224 {
225         int err;
226         uint64_t blk, off;
227         int bs = FZAP_BLOCK_SHIFT(zap);
228         dmu_buf_t *db;
230         ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));
231         ASSERT(tbl->zt_blk != 0);
233         dprintf("storing %llx at index %llx\n", val, idx);
235         blk = idx >> (bs-3);
236         off = idx & ((1<<(bs-3))-1);
238         err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
239             (tbl->zt_blk + blk) << bs, FTAG, &db, DMU_READ_NO_PREFETCH);
240         if (err)
241                 return (err);
242         dmu_buf_will_dirty(db, tx);
244         if (tbl->zt_nextblk != 0) {
245                 uint64_t idx2 = idx * 2;
246                 uint64_t blk2 = idx2 >> (bs-3);
247                 uint64_t off2 = idx2 & ((1<<(bs-3))-1);
248                 dmu_buf_t *db2;
250                 err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
251                     (tbl->zt_nextblk + blk2) << bs, FTAG, &db2,
252                     DMU_READ_NO_PREFETCH);
253                 if (err) {
254                         dmu_buf_rele(db, FTAG);
255                         return (err);
256                 }
257                 dmu_buf_will_dirty(db2, tx);
258                 ((uint64_t *)db2->db_data)[off2] = val;
259                 ((uint64_t *)db2->db_data)[off2+1] = val;
260                 dmu_buf_rele(db2, FTAG);
261         }
263         ((uint64_t *)db->db_data)[off] = val;
```



```
264         dmu_buf_rele(db, FTAG);
266         return (0);
267 }
269 static int
270 zap_table_load(zap_t *zap, zap_table_phys_t *tbl, uint64_t idx, uint64_t *valp)
271 {
272         uint64_t blk, off;
273         int err;
274         dmu_buf_t *db;
275         int bs = FZAP_BLOCK_SHIFT(zap);
277         ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));
279         blk = idx >> (bs-3);
280         off = idx & ((1<<(bs-3))-1);
282         err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
283             (tbl->zt_blk + blk) << bs, FTAG, &db, DMU_READ_NO_PREFETCH);
284         if (err)
285                 return (err);
286         *valp = ((uint64_t *)db->db_data)[off];
287         dmu_buf_rele(db, FTAG);
289         if (tbl->zt_nextblk != 0) {
290                 /*
291                  * read the nextblk for the sake of i/o error checking,
292                  * so that zap_table_load() will catch errors for
293                  * zap_table_store.
294                  */
295                 blk = (idx*2) >> (bs-3);
297                 err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
298                     (tbl->zt_nextblk + blk) << bs, FTAG, &db,
299                     DMU_READ_NO_PREFETCH);
300                 if (err == 0)
301                         dmu_buf_rele(db, FTAG);
302         }
303         return (err);
304 }
306 /*
307  * Routines for growing the ptrtbl.
308  */
310 static void
311 zap_ptrtbl_transfer(const uint64_t *src, uint64_t *dst, int n)
312 {
313         int i;
314         for (i = 0; i < n; i++) {
315                 uint64_t lb = src[i];
316                 dst[2*i+0] = lb;
317                 dst[2*i+1] = lb;
318         }
319 }
321 static int
322 zap_grow_ptrtbl(zap_t *zap, dmu_tx_t *tx)
323 {
324         /*
325          * The pointer table should never use more hash bits than we
326          * have (otherwise we'd be using useless zero bits to index it).
327          * If we are within 2 bits of running out, stop growing, since
328          * this is already an aberrant condition.
329          */
```

```
330         if (zap_f_phys(zap)->zap_ptrtbl.zt_shift >= zap_hashbits(zap) - 2)
331                 return (SET_ERROR(ENOSPC));
332
333         if (zap_f_phys(zap)->zap_ptrtbl.zt_numblks == 0) {
334                 /*
335                  * We are outgrowing the "embedded" ptrtbl (the one
336                  * stored in the header block).  Give it its own entire
337                  * block, which will double the size of the ptrtbl.
338                  */
339                 uint64_t newblk;
340                 dmu_buf_t *db_new;
341                 int err;
342
343                 ASSERT3U(zap_f_phys(zap)->zap_ptrtbl.zt_shift, ==,
344                     ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
345                 ASSERT0(zap_f_phys(zap)->zap_ptrtbl.zt_blk);
346
347                 newblk = zap_allocate_blocks(zap, 1);
348                 err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
349                     newblk << FZAP_BLOCK_SHIFT(zap), FTAG, &db_new,
350                     DMU_READ_NO_PREFETCH);
351                 if (err)
352                         return (err);
353                 dmu_buf_will_dirty(db_new, tx);
354                 zap_ptrtbl_transfer(&ZAP_EMBEDDED_PTRTBL_ENT(zap, 0),
355                     db_new->db_data, 1 << ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
356                 dmu_buf_rele(db_new, FTAG);
357
358                 zap_f_phys(zap)->zap_ptrtbl.zt_blk = newblk;
359                 zap_f_phys(zap)->zap_ptrtbl.zt_numblks = 1;
360                 zap_f_phys(zap)->zap_ptrtbl.zt_shift++;
361
362                 ASSERT3U(1ULL << zap_f_phys(zap)->zap_ptrtbl.zt_shift, ==,
363                     zap_f_phys(zap)->zap_ptrtbl.zt_numblks <<
364                     (FZAP_BLOCK_SHIFT(zap)-3));
365
366                 return (0);
367         } else {
368                 return (zap_table_grow(zap, &zap_f_phys(zap)->zap_ptrtbl,
369                     zap_ptrtbl_transfer, tx));
370         }
371 }
372
373 static void
374 zap_increment_num_entries(zap_t *zap, int delta, dmu_tx_t *tx)
375 {
376         dmu_buf_will_dirty(zap->zap_dbuf, tx);
377         mutex_enter(&zap->zap_f.zap_num_entries_mtx);
378         ASSERT(delta > 0 || zap_f_phys(zap)->zap_num_entries >= -delta);
379         zap_f_phys(zap)->zap_num_entries += delta;
380         mutex_exit(&zap->zap_f.zap_num_entries_mtx);
381 }
382
383 static uint64_t
384 zap_allocate_blocks(zap_t *zap, int nblocks)
385 {
386         uint64_t newblk;
387         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
388         newblk = zap_f_phys(zap)->zap_freeblk;
389         zap_f_phys(zap)->zap_freeblk += nblocks;
390         return (newblk);
391 }
392
393 static void
394 zap_leaf_pageout(void *dbu)
395 {
```



```
396         zap_leaf_t *l = dbu;
397
398         rw_destroy(&l->l_rwlock);
399         kmem_free(l, sizeof (zap_leaf_t));
400 }
401
402 static zap_leaf_t *
403 zap_create_leaf(zap_t *zap, dmu_tx_t *tx)
404 {
405         void *winner;
406         zap_leaf_t *l = kmem_zalloc(sizeof (zap_leaf_t), KM_SLEEP);
407
408         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
409
410         rw_init(&l->l_rwlock, 0, 0, 0);
411         rw_enter(&l->l_rwlock, RW_WRITER);
412         l->l_blkid = zap_allocate_blocks(zap, 1);
413         l->l_dbuf = NULL;
414
415         VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
416             l->l_blkid << FZAP_BLOCK_SHIFT(zap), NULL, &l->l_dbuf,
417             DMU_READ_NO_PREFETCH));
418         dmu_buf_init_user(&l->l_dbu, NULL, zap_leaf_pageout, &l->l_dbuf);
 84         dmu_buf_init_user(&l->l_dbu, zap_leaf_pageout, &l->l_dbuf);
419         winner = dmu_buf_set_user(l->l_dbuf, &l->l_dbu);
420         ASSERT(winner == NULL);
421         dmu_buf_will_dirty(l->l_dbuf, tx);
422
423         zap_leaf_init(l, zap->zap_normflags != 0);
424
425         zap_f_phys(zap)->zap_num_leafs++;
426
427         return (l);
428 }
_____unchanged_portion_omitted_

451 static zap_leaf_t *
452 zap_open_leaf(uint64_t blkid, dmu_buf_t *db)
453 {
454         zap_leaf_t *l, *winner;
455
456         ASSERT(blkid != 0);
457
458         l = kmem_zalloc(sizeof (zap_leaf_t), KM_SLEEP);
459         rw_init(&l->l_rwlock, 0, 0, 0);
460         rw_enter(&l->l_rwlock, RW_WRITER);
461         l->l_blkid = blkid;
462         l->l_bs = highbit64(db->db_size) - 1;
463         l->l_dbuf = db;
464
465         dmu_buf_init_user(&l->l_dbu, NULL, zap_leaf_pageout, &l->l_dbuf);
131         dmu_buf_init_user(&l->l_dbu, zap_leaf_pageout, &l->l_dbuf);
466         winner = dmu_buf_set_user(db, &l->l_dbu);
467
468         rw_exit(&l->l_rwlock);
469         if (winner != NULL) {
470                 /* someone else set it first */
471                 zap_leaf_pageout(&l->l_dbu);
472                 l = winner;
473         }
474
475         /*
476          * lhr_pad was previously used for the next leaf in the leaf
477          * chain.  There should be no chained leafs (as we have removed
478          * support for them).
479          */
```

```
480         ASSERT0(zap_leaf_phys(l)->l_hdr.lh_pad1);

482         /*
483          * There should be more hash entries than there can be
484          * chunks to put in the hash table
485          */
486         ASSERT3U(ZAP_LEAF_HASH_NUMENTRIES(l), >, ZAP_LEAF_NUMCHUNKS(l) / 3);

488         /* The chunks should begin at the end of the hash table */
489         ASSERT3P(&ZAP_LEAF_CHUNK(l, 0), ==,
490             &zap_leaf_phys(l)->l_hash[ZAP_LEAF_HASH_NUMENTRIES(l)]);

492         /* The chunks should end at the end of the block */
493         ASSERT3U((uintptr_t)&ZAP_LEAF_CHUNK(l, ZAP_LEAF_NUMCHUNKS(l)) -
494             (uintptr_t)zap_leaf_phys(l), ==, l->l_dbuf->db_size);

496         return (l);
497 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   34364 Wed Apr  6 14:26:57 2016
new/usr/src/uts/common/fs/zfs/zap_micro.c
patch first-pass
**********************************************************
______unchanged_portion_omitted_

 364 static zap_t *
 365 mzap_open(objset_t *os, uint64_t obj, dmu_buf_t *db)
 366 {
 367         zap_t *winner;
 368         zap_t *zap;
 369         int i;

 371         ASSERT3U(MZAP_ENT_LEN, ==, sizeof (mzap_ent_phys_t));

 373         zap = kmem_zalloc(sizeof (zap_t), KM_SLEEP);
 374         rw_init(&zap->zap_rwlock, 0, 0, 0);
 375         rw_enter(&zap->zap_rwlock, RW_WRITER);
 376         zap->zap_objset = os;
 377         zap->zap_object = obj;
 378         zap->zap_dbuf = db;

 380         if (*(uint64_t *)db->db_data != ZBT_MICRO) {
 381                 mutex_init(&zap->zap_f.zap_num_entries_mtx, 0, 0, 0);
 382                 zap->zap_f.zap_block_shift = highbit64(db->db_size) - 1;
 383         } else {
 384                 zap->zap_ismicro = TRUE;
 385         }

 387         /*
 388          * Make sure that zap_ismicro is set before we let others see
 389          * it, because zap_lockdir() checks zap_ismicro without the lock
 390          * held.
 391          */
 392         dmu_buf_init_user(&zap->zap_dbu, NULL, zap_evict, &zap->zap_dbuf);
 392         dmu_buf_init_user(&zap->zap_dbu, zap_evict, &zap->zap_dbuf);
 393         winner = dmu_buf_set_user(db, &zap->zap_dbu);

 395         if (winner != NULL) {
 396                 rw_exit(&zap->zap_rwlock);
 397                 rw_destroy(&zap->zap_rwlock);
 398                 if (!zap->zap_ismicro)
 399                         mutex_destroy(&zap->zap_f.zap_num_entries_mtx);
 400                 kmem_free(zap, sizeof (zap_t));
 401                 return (winner);
 402         }

 404         if (zap->zap_ismicro) {
 405                 zap->zap_salt = zap_m_phys(zap)->mz_salt;
 406                 zap->zap_normflags = zap_m_phys(zap)->mz_normflags;
 407                 zap->zap_m.zap_num_chunks = db->db_size / MZAP_ENT_LEN - 1;
 408                 avl_create(&zap->zap_m.zap_avl, mze_compare,
 409                     sizeof (mzap_ent_t), offsetof(mzap_ent_t, mze_node));

 411                 for (i = 0; i < zap->zap_m.zap_num_chunks; i++) {
 412                         mzap_ent_phys_t *mze =
 413                             &zap_m_phys(zap)->mz_chunk[i];
 414                         if (mze->mze_name[0]) {
 415                                 zap_name_t *zn;

 417                                 zap->zap_m.zap_num_entries++;
 418                                 zn = zap_name_alloc(zap, mze->mze_name,
 419                                     MT_EXACT);
 420                                 mze_insert(zap, i, zn->zn_hash);
 421                                 zap_name_free(zn);
```



```
 422                         }
 423                 }
 424         } else {
 425                 zap->zap_salt = zap_f_phys(zap)->zap_salt;
 426                 zap->zap_normflags = zap_f_phys(zap)->zap_normflags;

 428                 ASSERT3U(sizeof (struct zap_leaf_header), ==,
 429                     2*ZAP_LEAF_CHUNKSIZE);

 431                 /*
 432                  * The embedded pointer table should not overlap the
 433                  * other members.
 434                  */
 435                 ASSERT3P(&ZAP_EMBEDDED_PTRTBL_ENT(zap, 0), >,
 436                     &zap_f_phys(zap)->zap_salt);

 438                 /*
 439                  * The embedded pointer table should end at the end of
 440                  * the block
 441                  */
 442                 ASSERT3U((uintptr_t)&ZAP_EMBEDDED_PTRTBL_ENT(zap,
 443                     1<<ZAP_EMBEDDED_PTRTBL_SHIFT(zap)) -
 444                     (uintptr_t)zap_f_phys(zap), ==,
 445                     zap->zap_dbuf->db_size);
 446         }
 447         rw_exit(&zap->zap_rwlock);
 448         return (zap);
 449 }
______unchanged_portion_omitted_
```