```
**********************************************************
   89077 Wed Apr  6 14:26:56 2016
new/usr/src/uts/common/fs/zfs/dbuf.c
patch first-pass
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2012, 2015 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 * Copyright (c) 2014 Integros [integros.com]
 */

#include <sys/zfs_context.h>
#include <sys/dmu.h>
#include <sys/dmu_send.h>
#include <sys/dmu_impl.h>
#include <sys/dbuf.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dmu_tx.h>
#include <sys/spa.h>
#include <sys/zio.h>
#include <sys/dmu_zfetch.h>
#include <sys/sa.h>
#include <sys/sa_impl.h>
#include <sys/zfeature.h>
#include <sys/blkptr.h>
#include <sys/range_tree.h>

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

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

#ifndef __lint
extern 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_evict_func_t *evict_func, dmu_buf_t **clear_on_evict_dbufp);
#endif /* ! __lint */

/*
 * Global data structures and functions for the dbuf cache.
 */
static kmem_cache_t *dbuf_cache;
static taskq_t *dbu_evict_taskq;

/* ARGSUSED */
static int
dbuf_cons(void *vdb, void *unused, int kmflag)
{
        dmu_buf_impl_t *db = vdb;
        bzero(db, sizeof (dmu_buf_impl_t));

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

        return (0);
}
______unchanged_portion_omitted_

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

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

        if (dbu == NULL)
                return;

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

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

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

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

boolean_t
dbuf_is_metadata(dmu_buf_impl_t *db)
{
        if (db->db_level > 0) {
                return (B_TRUE);
        } else {
                boolean_t is_metadata;

                DB_DNODE_ENTER(db);
                is_metadata = DMU_OT_IS_METADATA(DB_DNODE(db)->dn_type);
                DB_DNODE_EXIT(db);
                return (is_metadata);
        }
}

void
dbuf_evict(dmu_buf_impl_t *db)
{
        ASSERT(MUTEX_HELD(&db->db_mtx));
        ASSERT(db->db_buf == NULL);
        ASSERT(db->db_data_pending == NULL);

        dbuf_clear(db);
        dbuf_destroy(db);
}

void
dbuf_init(void)
{
        uint64_t hsize = 1ULL << 16;
        dbuf_hash_table_t *h = &dbuf_hash_table;
        int i;

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

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

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

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

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

void
dbuf_fini(void)
{
        dbuf_hash_table_t *h = &dbuf_hash_table;
        int i;

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

```
}

/*
 * Other stuff.
 */

#ifdef ZFS_DEBUG
static void
dbuf_verify(dmu_buf_impl_t *db)
{
        dnode_t *dn;
        dbuf_dirty_record_t *dr;

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

        if (!(zfs_flags & ZFS_DEBUG_DBUF_VERIFY))
                return;

        ASSERT(db->db_objset != NULL);
        DB_DNODE_ENTER(db);
        dn = DB_DNODE(db);
        if (dn == NULL) {
                ASSERT(db->db_parent == NULL);
                ASSERT(db->db_blkptr == NULL);
        } else {
                ASSERT3U(db->db.db_object, ==, dn->dn_object);
                ASSERT3P(db->db_objset, ==, dn->dn_objset);
                ASSERT3U(db->db_level, <, dn->dn_nlevels);
                ASSERT(db->db_blkid == DMU_BONUS_BLKID ||
                    db->db_blkid == DMU_SPILL_BLKID ||
                    !avl_is_empty(&dn->dn_dbufs));
        }
        if (db->db_blkid == DMU_BONUS_BLKID) {
                ASSERT(dn != NULL);
                ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
                ASSERT3U(db->db.db_offset, ==, DMU_BONUS_BLKID);
        } else if (db->db_blkid == DMU_SPILL_BLKID) {
                ASSERT(dn != NULL);
                ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
                ASSERT0(db->db.db_offset);
        } else {
                ASSERT3U(db->db.db_offset, ==, db->db_blkid * db->db.db_size);
        }

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

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

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

        /* verify db->db_blkptr */
        if (db->db_blkptr) {
                if (db->db_parent == dn->dn_dbuf) {
                        /* db is pointed to by the dnode */
                        /* ASSERT3U(db->db_blkid, <, dn->dn_nblkptr); */
                        if (DMU_OBJECT_IS_SPECIAL(db->db.db_object))
                                ASSERT(db->db_parent == NULL);
                        else
                                ASSERT(db->db_parent != NULL);
                        if (db->db_blkid != DMU_SPILL_BLKID)
                                ASSERT3P(db->db_blkptr, ==,
                                    &dn->dn_phys->dn_blkptr[db->db_blkid]);
                } else {
                        /* db is pointed to by an indirect block */
                        int epb = db->db_parent->db.db_size >> SPA_BLKPTRSHIFT;
                        ASSERT3U(db->db_parent->db_level, ==, db->db_level+1);
                        ASSERT3U(db->db_parent->db.db_object, ==,
                            db->db.db_object);
                        /*
                         * dnode_grow_indblksz() can make this fail if we don’t
                         * have the struct_rwlock.  XXX indblksz no longer
                         * grows.  safe to do this now?
                         */
                        if (RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
                                ASSERT3P(db->db_blkptr, ==,
                                    ((blkptr_t *)db->db_parent->db.db_data +
                                    db->db_blkid % epb));
                        }
                }
        }
        if ((db->db_blkptr == NULL || BP_IS_HOLE(db->db_blkptr)) &&
            (db->db_buf == NULL || db->db_buf->b_data) &&
            db->db.db_data && db->db_blkid != DMU_BONUS_BLKID &&
            db->db_state != DB_FILL && !dn->dn_free_txg) {
                /*
                 * If the blkptr isn’t set but they have nonzero data,
                 * it had better be dirty, otherwise we’ll lose that
                 * data when we evict this buffer.
                 */
                if (db->db_dirtycnt == 0) {
                        uint64_t *buf = db->db.db_data;
                        int i;

                        for (i = 0; i < db->db.db_size >> 3; i++) {
                                ASSERT(buf[i] == 0);
                        }
                }
        }
        DB_DNODE_EXIT(db);
}
#endif

static void
dbuf_clear_data(dmu_buf_impl_t *db)
{
        ASSERT(MUTEX_HELD(&db->db_mtx));
        dbuf_evict_user(db);
        db->db_buf = NULL;
        db->db.db_data = NULL;
        if (db->db_state != DB_NOFILL)
                db->db_state = DB_UNCACHED;
}

static void
dbuf_set_data(dmu_buf_impl_t *db, arc_buf_t *buf)
{
        ASSERT(MUTEX_HELD(&db->db_mtx));
        ASSERT(buf != NULL);
```

```

        db->db_buf = buf;
        ASSERT(buf->b_data != NULL);
        db->db.db_data = buf->b_data;
        if (!arc_released(buf))
                arc_set_callback(buf, dbuf_do_evict, db);
}

/*
 * Loan out an arc_buf for read.  Return the loaned arc_buf.
 */
arc_buf_t *
dbuf_loan_arcbuf(dmu_buf_impl_t *db)
{
        arc_buf_t *abuf;

        mutex_enter(&db->db_mtx);
        if (arc_released(db->db_buf) || refcount_count(&db->db_holds) > 1) {
                int blksz = db->db.db_size;
                spa_t *spa = db->db_objset->os_spa;

                mutex_exit(&db->db_mtx);
                abuf = arc_loan_buf(spa, blksz);
                bcopy(db->db.db_data, abuf->b_data, blksz);
        } else {
                abuf = db->db_buf;
                arc_loan_inuse_buf(abuf, db);
                dbuf_clear_data(db);
                mutex_exit(&db->db_mtx);
        }
        return (abuf);
}

/*
 * Calculate which level n block references the data at the level 0 offset
 * provided.
 */
uint64_t
dbuf_whichblock(dnode_t *dn, int64_t level, uint64_t offset)
{
        if (dn->dn_datablkshift != 0 && dn->dn_indblkshift != 0) {
                /*
                 * The level n blkid is equal to the level 0 blkid divided by
                 * the number of level 0s in a level n block.
                 *
                 * The level 0 blkid is offset >> datablkshift =
                 * offset / 2^datablkshift.
                 *
                 * The number of level 0s in a level n is the number of block
                 * pointers in an indirect block, raised to the power of level.
                 * This is 2^(indblkshift - SPA_BLKPTRSHIFT)^level =
                 * 2^(level*(indblkshift - SPA_BLKPTRSHIFT)).
                 *
                 * Thus, the level n blkid is: offset /
                 * ((2^datablkshift)*(2^(level*(indblkshift - SPA_BLKPTRSHIFT)))
                 * = offset / 2^(datablkshift + level *
                 *   (indblkshift - SPA_BLKPTRSHIFT))
                 * = offset >> (datablkshift + level *
                 *   (indblkshift - SPA_BLKPTRSHIFT))
                 */
                return (offset >> (dn->dn_datablkshift + level *
                    (dn->dn_indblkshift - SPA_BLKPTRSHIFT)));
        } else {
                ASSERT3U(offset, <, dn->dn_datablksz);
                return (0);
        }
```

```
}

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

        mutex_enter(&db->db_mtx);
        ASSERT3U(db->db_state, ==, DB_READ);
        /*
         * All reads are synchronous, so we must have a hold on the dbuf
         */
        ASSERT(refcount_count(&db->db_holds) > 0);
        ASSERT(db->db_buf == NULL);
        ASSERT(db->db.db_data == NULL);
        if (db->db_level == 0 && db->db_freed_in_flight) {
                /* we were freed in flight; disregard any error */
                arc_release(buf, db);
                bzero(buf->b_data, db->db.db_size);
                arc_buf_freeze(buf);
                db->db_freed_in_flight = FALSE;
                dbuf_set_data(db, buf);
                db->db_state = DB_CACHED;
        } else if (zio == NULL || zio->io_error == 0) {
                dbuf_set_data(db, buf);
                db->db_state = DB_CACHED;
        } else {
                ASSERT(db->db_blkid != DMU_BONUS_BLKID);
                ASSERT3P(db->db_buf, ==, NULL);
                VERIFY(arc_buf_remove_ref(buf, db));
                db->db_state = DB_UNCACHED;
        }
        cv_broadcast(&db->db_changed);
        dbuf_rele_and_unlock(db, NULL);
}

static void
dbuf_read_impl(dmu_buf_impl_t *db, zio_t *zio, uint32_t flags)
{
        dnode_t *dn;
        zbookmark_phys_t zb;
        arc_flags_t aflags = ARC_FLAG_NOWAIT;

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

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

                ASSERT3U(bonuslen, <=, db->db.db_size);
                db->db.db_data = zio_buf_alloc(DN_MAX_BONUSLEN);
                arc_space_consume(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
                if (bonuslen < DN_MAX_BONUSLEN)
                        bzero(db->db.db_data, DN_MAX_BONUSLEN);
                if (bonuslen)
                        bcopy(DN_BONUS(dn->dn_phys), db->db.db_data, bonuslen);
                DB_DNODE_EXIT(db);
                db->db_state = DB_CACHED;
                mutex_exit(&db->db_mtx);
                return;
```

```
        }

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

                DB_DNODE_EXIT(db);
                dbuf_set_data(db, arc_buf_alloc(db->db_objset->os_spa,
                    db->db.db_size, db, type));
                bzero(db->db.db_data, db->db.db_size);
                db->db_state = DB_CACHED;
                mutex_exit(&db->db_mtx);
                return;
        }

        DB_DNODE_EXIT(db);

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

        if (DBUF_IS_L2CACHEABLE(db))
                aflags |= ARC_FLAG_L2CACHE;
        if (DBUF_IS_L2COMPRESSIBLE(db))
                aflags |= ARC_FLAG_L2COMPRESS;

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

        dbuf_add_ref(db, NULL);

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

int
dbuf_read(dmu_buf_impl_t *db, zio_t *zio, uint32_t flags)
{
        int err = 0;
        boolean_t havepzio = (zio != NULL);
        boolean_t prefetch;
        dnode_t *dn;

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

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

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

        prefetch = db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
            (flags & DB_RF_NOPREFETCH) == 0 && dn != NULL &&
            DBUF_IS_CACHEABLE(db);

        mutex_enter(&db->db_mtx);
        if (db->db_state == DB_CACHED) {
                mutex_exit(&db->db_mtx);
                if (prefetch)
                        dmu_zfetch(&dn->dn_zfetch, db->db_blkid, 1);
                if ((flags & DB_RF_HAVESTRUCT) == 0)
                        rw_exit(&dn->dn_struct_rwlock);
                DB_DNODE_EXIT(db);
        } else if (db->db_state == DB_UNCACHED) {
                spa_t *spa = dn->dn_objset->os_spa;

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

                /* dbuf_read_impl has dropped db_mtx for us */

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

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

                if (!havepzio)
                        err = zio_wait(zio);
        } else {
                /*
                 * Another reader came in while the dbuf was in flight
                 * between UNCACHED and CACHED.  Either a writer will finish
                 * writing the buffer (sending the dbuf to CACHED) or the
                 * first reader’s request will reach the read_done callback
                 * and send the dbuf to CACHED.  Otherwise, a failure
                 * occurred and the dbuf went to UNCACHED.
                 */
                mutex_exit(&db->db_mtx);
                if (prefetch)
                        dmu_zfetch(&dn->dn_zfetch, db->db_blkid, 1);
                if ((flags & DB_RF_HAVESTRUCT) == 0)
                        rw_exit(&dn->dn_struct_rwlock);
                DB_DNODE_EXIT(db);

                /* Skip the wait per the caller’s request. */
                mutex_enter(&db->db_mtx);
                if ((flags & DB_RF_NEVERWAIT) == 0) {
                        while (db->db_state == DB_READ ||
                            db->db_state == DB_FILL) {
                                ASSERT(db->db_state == DB_READ ||
                                    (flags & DB_RF_HAVESTRUCT) == 0);
                                DTRACE_PROBE2(blocked__read, dmu_buf_impl_t *,
                                    db, zio_t *, zio);
                                cv_wait(&db->db_changed, &db->db_mtx);
                        }
                        if (db->db_state == DB_UNCACHED)
                                err = SET_ERROR(EIO);
                }
                mutex_exit(&db->db_mtx);
        }

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

```
static void
dbuf_noread(dmu_buf_impl_t *db)
{
        ASSERT(!refcount_is_zero(&db->db_holds));
        ASSERT(db->db_blkid != DMU_BONUS_BLKID);
        mutex_enter(&db->db_mtx);
        while (db->db_state == DB_READ || db->db_state == DB_FILL)
                cv_wait(&db->db_changed, &db->db_mtx);
        if (db->db_state == DB_UNCACHED) {
                arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
                spa_t *spa = db->db_objset->os_spa;

                ASSERT(db->db_buf == NULL);
                ASSERT(db->db.db_data == NULL);
                dbuf_set_data(db, arc_buf_alloc(spa, db->db.db_size, db, type));
                db->db_state = DB_FILL;
        } else if (db->db_state == DB_NOFILL) {
                dbuf_clear_data(db);
        } else {
                ASSERT3U(db->db_state, ==, DB_CACHED);
        }
        mutex_exit(&db->db_mtx);
}

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

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

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

        /*
         * If the last dirty record for this dbuf has not yet synced
         * and its referencing the dbuf data, either:
         *      reset the reference to point to a new copy,
         * or (if there a no active holders)
         *      just null out the current db_data pointer.
         */
        ASSERT(dr->dr_txg >= txg - 2);
        if (db->db_blkid == DMU_BONUS_BLKID) {
                /* Note that the data bufs here are zio_bufs */
                dr->dt.dl.dr_data = zio_buf_alloc(DN_MAX_BONUSLEN);
                arc_space_consume(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
                bcopy(db->db.db_data, dr->dt.dl.dr_data, DN_MAX_BONUSLEN);
        } else if (refcount_count(&db->db_holds) > db->db_dirtycnt) {
                int size = db->db.db_size;
                arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
                spa_t *spa = db->db_objset->os_spa;

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

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

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

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

        ASSERT(db->db_data_pending != dr);

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

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

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

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

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

```
        db_search.db_level = 0;
        db_search.db_blkid = start_blkid;
        db_search.db_state = DB_SEARCH;

        mutex_enter(&dn->dn_dbufs_mtx);
        if (start_blkid >= dn->dn_unlisted_l0_blkid) {
                /* There can’t be any dbufs in this range; no need to search. */
#ifdef DEBUG
                db = avl_find(&dn->dn_dbufs, &db_search, &where);
                ASSERT3P(db, ==, NULL);
                db = avl_nearest(&dn->dn_dbufs, where, AVL_AFTER);
                ASSERT(db == NULL || db->db_level > 0);
#endif
                mutex_exit(&dn->dn_dbufs_mtx);
                return;
        } else if (dmu_objset_is_receiving(dn->dn_objset)) {
                /*
                 * If we are receiving, we expect there to be no dbufs in
                 * the range to be freed, because receive modifies each
                 * block at most once, and in offset order.  If this is
                 * not the case, it can lead to performance problems,
                 * so note that we unexpectedly took the slow path.
                 */
                atomic_inc_64(&zfs_free_range_recv_miss);
        }

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

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

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

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

                if (db->db_state == DB_UNCACHED ||
                    db->db_state == DB_NOFILL ||
                    db->db_state == DB_EVICTING) {
                        ASSERT(db->db.db_data == NULL);
                        mutex_exit(&db->db_mtx);
                        continue;
                }
                if (db->db_state == DB_READ || db->db_state == DB_FILL) {
                        /* will be handled in dbuf_read_done or dbuf_rele */
                        db->db_freed_in_flight = TRUE;
                        mutex_exit(&db->db_mtx);
                        continue;
                }
                if (refcount_count(&db->db_holds) == 0) {
                        ASSERT(db->db_buf);
                        dbuf_clear(db);
                        continue;
                }
                /* The dbuf is referenced */
                if (db->db_last_dirty != NULL) {
                        dbuf_dirty_record_t *dr = db->db_last_dirty;

                        if (dr->dr_txg == txg) {
                                /*
                                 * This buffer is "in-use", re-adjust the file
                                 * size to reflect that this buffer may
                                 * contain new data when we sync.
                                 */
                                if (db->db_blkid != DMU_SPILL_BLKID &&
                                    db->db_blkid > dn->dn_maxblkid)
                                        dn->dn_maxblkid = db->db_blkid;
                                dbuf_unoverride(dr);
                        } else {
                                /*
                                 * This dbuf is not dirty in the open context.
                                 * Either uncache it (if its not referenced in
                                 * the open context) or reset its contents to
                                 * empty.
                                 */
                                dbuf_fix_old_data(db, txg);
                        }
                }
                /* clear the contents if its cached */
                if (db->db_state == DB_CACHED) {
                        ASSERT(db->db.db_data != NULL);
                        arc_release(db->db_buf, db);
                        bzero(db->db.db_data, db->db.db_size);
                        arc_buf_freeze(db->db_buf);
                }

                mutex_exit(&db->db_mtx);
        }
        mutex_exit(&dn->dn_dbufs_mtx);
}

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

        /*
         * We don’t need any locking to protect db_blkptr:
         * If it’s syncing, then db_last_dirty will be set
         * so we’ll ignore db_blkptr.
         *
         * This logic ensures that only block births for
         * filled blocks are considered.
         */
        ASSERT(MUTEX_HELD(&db->db_mtx));
        if (db->db_last_dirty && (db->db_blkptr == NULL ||
            !BP_IS_HOLE(db->db_blkptr))) {
                birth_txg = db->db_last_dirty->dr_txg;
        } else if (db->db_blkptr != NULL && !BP_IS_HOLE(db->db_blkptr)) {
                birth_txg = db->db_blkptr->blk_birth;
        }

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

```
                return (ds == NULL ||
                    dsl_dataset_block_freeable(ds, NULL, birth_txg));
        else
                return (B_FALSE);
}

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

        ASSERT(db->db_blkid != DMU_BONUS_BLKID);

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

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

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

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

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

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

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

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

void
dbuf_release_bp(dmu_buf_impl_t *db)
{
        objset_t *os = db->db_objset;

        ASSERT(dsl_pool_sync_context(dmu_objset_pool(os)));
        ASSERT(arc_released(os->os_phys_buf) ||
            list_link_active(&os->os_dsl_dataset->ds_synced_link));
        ASSERT(db->db_parent == NULL || arc_released(db->db_parent->db_buf));

        (void) arc_release(db->db_buf, db);
}

/*
 * We already have a dirty record for this TXG, and we are being
 * dirtied again.
 */
static void
dbuf_redirty(dbuf_dirty_record_t *dr)
{
        dmu_buf_impl_t *db = dr->dr_dbuf;

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

        if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID) {
                /*
                 * If this buffer has already been written out,
                 * we now need to reset its state.
                 */
                dbuf_unoverride(dr);
                if (db->db.db_object != DMU_META_DNODE_OBJECT &&
                    db->db_state != DB_NOFILL) {
                        /* Already released on initial dirty, so just thaw. */
                        ASSERT(arc_released(db->db_buf));
                        arc_buf_thaw(db->db_buf);
                }
        }
}

dbuf_dirty_record_t *
dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
{
        dnode_t *dn;
        objset_t *os;
        dbuf_dirty_record_t **drp, *dr;
        int drop_struct_lock = FALSE;
        boolean_t do_free_accounting = B_FALSE;
        int txgoff = tx->tx_txg & TXG_MASK;

        ASSERT(tx->tx_txg != 0);
        ASSERT(!refcount_is_zero(&db->db_holds));
        DMU_TX_DIRTY_BUF(tx, db);

        DB_DNODE_ENTER(db);
        dn = DB_DNODE(db);
        /*
         * Shouldn't dirty a regular buffer in syncing context.  Private
         * objects may be dirtied in syncing context, but only if they
         * were already pre-dirtied in open context.
         */
        ASSERT(!dmu_tx_is_syncing(tx) ||
            BP_IS_HOLE(dn->dn_objset->os_rootbp) ||
            DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
            dn->dn_objset->os_dsl_dataset == NULL);
        /*
         * We make this assert for private objects as well, but after we
         * check if we're already dirty.  They are allowed to re-dirty
         * in syncing context.
         */
        ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
            dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
            (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));
```

```
        mutex_enter(&db->db_mtx);
        /*
         * XXX make this true for indirects too?  The problem is that
         * transactions created with dmu_tx_create_assigned() from
         * syncing context don't bother holding ahead.
         */
        ASSERT(db->db_level != 0 ||
            db->db_state == DB_CACHED || db->db_state == DB_FILL ||
            db->db_state == DB_NOFILL);

        mutex_enter(&dn->dn_mtx);
        /*
         * Don't set dirtyctx to SYNC if we're just modifying this as we
         * initialize the objset.
         */
        if (dn->dn_dirtyctx == DN_UNDIRTIED &&
            !BP_IS_HOLE(dn->dn_objset->os_rootbp)) {
                dn->dn_dirtyctx =
                    (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN);
                ASSERT(dn->dn_dirtyctx_firstset == NULL);
                dn->dn_dirtyctx_firstset = kmem_alloc(1, KM_SLEEP);
        }
        mutex_exit(&dn->dn_mtx);

        if (db->db_blkid == DMU_SPILL_BLKID)
                dn->dn_have_spill = B_TRUE;

        /*
         * If this buffer is already dirty, we're done.
         */
        drp = &db->db_last_dirty;
        ASSERT(*drp == NULL || (*drp)->dr_txg <= tx->tx_txg ||
            db->db.db_object == DMU_META_DNODE_OBJECT);
        while ((dr = *drp) != NULL && dr->dr_txg > tx->tx_txg)
                drp = &dr->dr_next;
        if (dr && dr->dr_txg == tx->tx_txg) {
                DB_DNODE_EXIT(db);

                dbuf_redirty(dr);
                mutex_exit(&db->db_mtx);
                return (dr);
        }

        /*
         * Only valid if not already dirty.
         */
        ASSERT(dn->dn_object == 0 ||
            dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
            (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));

        ASSERT3U(dn->dn_nlevels, >, db->db_level);
        ASSERT((dn->dn_phys->dn_nlevels == 0 && db->db_level == 0) ||
            dn->dn_phys->dn_nlevels > db->db_level ||
            dn->dn_next_nlevels[txgoff] > db->db_level ||
            dn->dn_next_nlevels[(tx->tx_txg-1) & TXG_MASK] > db->db_level ||
            dn->dn_next_nlevels[(tx->tx_txg-2) & TXG_MASK] > db->db_level);

        /*
         * We should only be dirtying in syncing context if it's the
         * mos or we're initializing the os or it's a special object.
         * However, we are allowed to dirty in syncing context provided
         * we already dirtied it in open context.  Hence we must make
         * this assertion only if we're not already dirty.
         */
        os = dn->dn_objset;
        ASSERT(!dmu_tx_is_syncing(tx) || DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
            os->os_dsl_dataset == NULL || BP_IS_HOLE(os->os_rootbp));
        ASSERT(db->db.db_size != 0);

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

        if (db->db_blkid != DMU_BONUS_BLKID) {
                /*
                 * Update the accounting.
                 * Note: we delay "free accounting" until after we drop
                 * the db_mtx.  This keeps us from grabbing other locks
                 * (and possibly deadlocking) in bp_get_dsize() while
                 * also holding the db_mtx.
                 */
                dnode_willuse_space(dn, db->db.db_size, tx);
                do_free_accounting = dbuf_block_freeable(db);
        }

        /*
         * If this buffer is dirty in an old transaction group we need
         * to make a copy of it so that the changes we make in this
         * transaction group won’t leak out when we sync the older txg.
         */
        dr = kmem_zalloc(sizeof (dbuf_dirty_record_t), KM_SLEEP);
        if (db->db_level == 0) {
                void *data_old = db->db_buf;

                if (db->db_state != DB_NOFILL) {
                        if (db->db_blkid == DMU_BONUS_BLKID) {
                                dbuf_fix_old_data(db, tx->tx_txg);
                                data_old = db->db.db_data;
                        } else if (db->db.db_object != DMU_META_DNODE_OBJECT) {
                                /*
                                 * Release the data buffer from the cache so
                                 * that we can modify it without impacting
                                 * possible other users of this cached data
                                 * block.  Note that indirect blocks and
                                 * private objects are not released until the
                                 * syncing state (since they are only modified
                                 * then).
                                 */
                                arc_release(db->db_buf, db);
                                dbuf_fix_old_data(db, tx->tx_txg);
                                data_old = db->db_buf;
                        }
                        ASSERT(data_old != NULL);
                }
                dr->dt.dl.dr_data = data_old;
        } else {
                mutex_init(&dr->dt.di.dr_mtx, NULL, MUTEX_DEFAULT, NULL);
                list_create(&dr->dt.di.dr_children,
                    sizeof (dbuf_dirty_record_t),
                    offsetof(dbuf_dirty_record_t, dr_dirty_node));
        }
        if (db->db_blkid != DMU_BONUS_BLKID && os->os_dsl_dataset != NULL)
                dr->dr_accounted = db->db.db_size;
        dr->dr_dbuf = db;
        dr->dr_txg = tx->tx_txg;
        dr->dr_next = *drp;
        *drp = dr;

        /*
         * We could have been freed_in_flight between the dbuf_noread
         * and dbuf_dirty.  We win, as though the dbuf_noread() had
         * happened after the free.
         */
        if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
```

```
            db->db_blkid != DMU_SPILL_BLKID) {
                mutex_enter(&dn->dn_mtx);
                if (dn->dn_free_ranges[txgoff] != NULL) {
                        range_tree_clear(dn->dn_free_ranges[txgoff],
                            db->db_blkid, 1);
                }
                mutex_exit(&dn->dn_mtx);
                db->db_freed_in_flight = FALSE;
        }

        /*
         * This buffer is now part of this txg
         */
        dbuf_add_ref(db, (void *)(uintptr_t)tx->tx_txg);
        db->db_dirtycnt += 1;
        ASSERT3U(db->db_dirtycnt, <=, 3);

        mutex_exit(&db->db_mtx);

        if (db->db_blkid == DMU_BONUS_BLKID ||
            db->db_blkid == DMU_SPILL_BLKID) {
                mutex_enter(&dn->dn_mtx);
                ASSERT(!list_link_active(&dr->dr_dirty_node));
                list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
                mutex_exit(&dn->dn_mtx);
                dnode_setdirty(dn, tx);
                DB_DNODE_EXIT(db);
                return (dr);
        } else if (do_free_accounting) {
                blkptr_t *bp = db->db_blkptr;
                int64_t willfree = (bp && !BP_IS_HOLE(bp)) ?
                    bp_get_dsize(os->os_spa, bp) : db->db.db_size;
                /*
                 * This is only a guess -- if the dbuf is dirty
                 * in a previous txg, we don’t know how much
                 * space it will use on disk yet.  We should
                 * really have the struct_rwlock to access
                 * db_blkptr, but since this is just a guess,
                 * it’s OK if we get an odd answer.
                 */
                ddt_prefetch(os->os_spa, bp);
                dnode_willuse_space(dn, -willfree, tx);
        }

        if (!RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
                rw_enter(&dn->dn_struct_rwlock, RW_READER);
                drop_struct_lock = TRUE;
        }

        if (db->db_level == 0) {
                dnode_new_blkid(dn, db->db_blkid, tx, drop_struct_lock);
                ASSERT(dn->dn_maxblkid >= db->db_blkid);
        }

        if (db->db_level+1 < dn->dn_nlevels) {
                dmu_buf_impl_t *parent = db->db_parent;
                dbuf_dirty_record_t *di;
                int parent_held = FALSE;

                if (db->db_parent == NULL || db->db_parent == dn->dn_dbuf) {
                        int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;

                        parent = dbuf_hold_level(dn, db->db_level+1,
                            db->db_blkid >> epbs, FTAG);
                        ASSERT(parent != NULL);
                        parent_held = TRUE;
                }
                if (drop_struct_lock)
                        rw_exit(&dn->dn_struct_rwlock);
                ASSERT3U(db->db_level+1, ==, parent->db_level);
                di = dbuf_dirty(parent, tx);
                if (parent_held)
                        dbuf_rele(parent, FTAG);

                mutex_enter(&db->db_mtx);
                /*
                 * Since we’ve dropped the mutex, it’s possible that
                 * dbuf_undirty() might have changed this out from under us.
                 */
                if (db->db_last_dirty == dr ||
                    dn->dn_object == DMU_META_DNODE_OBJECT) {
                        mutex_enter(&di->dt.di.dr_mtx);
                        ASSERT3U(di->dr_txg, ==, tx->tx_txg);
                        ASSERT(!list_link_active(&dr->dr_dirty_node));
                        list_insert_tail(&di->dt.di.dr_children, dr);
                        mutex_exit(&di->dt.di.dr_mtx);
                        dr->dr_parent = di;
                }
                mutex_exit(&db->db_mtx);
        } else {
                ASSERT(db->db_level+1 == dn->dn_nlevels);
                ASSERT(db->db_blkid < dn->dn_nblkptr);
                ASSERT(db->db_parent == NULL || db->db_parent == dn->dn_dbuf);
                mutex_enter(&dn->dn_mtx);
                ASSERT(!list_link_active(&dr->dr_dirty_node));
                list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
                mutex_exit(&dn->dn_mtx);
                if (drop_struct_lock)
                        rw_exit(&dn->dn_struct_rwlock);
        }

        dnode_setdirty(dn, tx);
        DB_DNODE_EXIT(db);
        return (dr);
}

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

        ASSERT(txg != 0);

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

```
        /*
         * If this buffer is not dirty, we’re done.
         */
        for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
                if (dr->dr_txg <= txg)
                        break;
        if (dr == NULL || dr->dr_txg < txg)
                return (B_FALSE);
        ASSERT(dr->dr_txg == txg);
        ASSERT(dr->dr_dbuf == db);

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

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

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

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

        *drp = dr->dr_next;

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

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

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

        kmem_free(dr, sizeof (dbuf_dirty_record_t));

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

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

                ASSERT(db->db_state == DB_NOFILL || arc_released(buf));
                dbuf_clear_data(db);
                VERIFY(arc_buf_remove_ref(buf, db));
                dbuf_evict(db);
                return (B_TRUE);
        }
        return (B_FALSE);
}

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

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

        /*
         * Quick check for dirtyness.  For already dirty blocks, this
         * reduces runtime of this function by >90%, and overall performance
         * by 50% for some workloads (e.g. file deletion with indirect blocks
         * cached).
         */
        mutex_enter(&db->db_mtx);
        dbuf_dirty_record_t *dr;
        for (dr = db->db_last_dirty;
            dr != NULL && dr->dr_txg >= tx->tx_txg; dr = dr->dr_next) {
                /*
                 * It’s possible that it is already dirty but not cached,
                 * because there are some calls to dbuf_dirty() that don’t
                 * go through dmu_buf_will_dirty().
                 */
                if (dr->dr_txg == tx->tx_txg && db->db_state == DB_CACHED) {
                        /* This dbuf is already dirty and cached. */
                        dbuf_redirty(dr);
                        mutex_exit(&db->db_mtx);
                        return;
                }
        }
        mutex_exit(&db->db_mtx);

        DB_DNODE_ENTER(db);
        if (RW_WRITE_HELD(&DB_DNODE(db)->dn_struct_rwlock))
                rf |= DB_RF_HAVESTRUCT;
        DB_DNODE_EXIT(db);
        (void) dbuf_read(db, NULL, rf);
        (void) dbuf_dirty(db, tx);
}

void
dmu_buf_will_not_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

        db->db_state = DB_NOFILL;

        dmu_buf_will_fill(db_fake, tx);
}

void
dmu_buf_will_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

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

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

```

        dbuf_noread(db);
        (void) dbuf_dirty(db, tx);
}

#pragma weak dmu_buf_fill_done = dbuf_fill_done
/* ARGSUSED */
void
dbuf_fill_done(dmu_buf_impl_t *db, dmu_tx_t *tx)
{
        mutex_enter(&db->db_mtx);
        DBUF_VERIFY(db);

        if (db->db_state == DB_FILL) {
                if (db->db_level == 0 && db->db_freed_in_flight) {
                        ASSERT(db->db_blkid != DMU_BONUS_BLKID);
                        /* we were freed while filling */
                        /* XXX dbuf_undirty? */
                        bzero(db->db.db_data, db->db.db_size);
                        db->db_freed_in_flight = FALSE;
                }
                db->db_state = DB_CACHED;
                cv_broadcast(&db->db_changed);
        }
        mutex_exit(&db->db_mtx);
}

void
dmu_buf_write_embedded(dmu_buf_t *dbuf, void *data,
    bp_embedded_type_t etype, enum zio_compress comp,
    int uncompressed_size, int compressed_size, int byteorder,
    dmu_tx_t *tx)
{
        dmu_buf_impl_t *db = (dmu_buf_impl_t *)dbuf;
        struct dirty_leaf *dl;
        dmu_object_type_t type;

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

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

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

        dmu_buf_will_not_fill(dbuf, tx);

        ASSERT3U(db->db_last_dirty->dr_txg, ==, tx->tx_txg);
        dl = &db->db_last_dirty->dt.dl;
        encode_embedded_bp_compressed(&dl->dr_overridden_by,
            data, comp, uncompressed_size, compressed_size);
        BPE_SET_ETYPE(&dl->dr_overridden_by, etype);
        BP_SET_TYPE(&dl->dr_overridden_by, type);
        BP_SET_LEVEL(&dl->dr_overridden_by, 0);
        BP_SET_BYTEORDER(&dl->dr_overridden_by, byteorder);

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

/*
 * 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)
```

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

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

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

        *parentp = NULL;
        *bpp = NULL;

        ASSERT(blkid != DMU_BONUS_BLKID);

        if (blkid == DMU_SPILL_BLKID) {
                mutex_enter(&dn->dn_mtx);
                if (dn->dn_have_spill &&
                    (dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR))
                        *bpp = &dn->dn_phys->dn_spill;
                else
                        *bpp = NULL;
                dbuf_add_ref(dn->dn_dbuf, NULL);
                *parentp = dn->dn_dbuf;
                mutex_exit(&dn->dn_mtx);
                return (0);
        }

        if (dn->dn_phys->dn_nlevels == 0)
                nlevels = 1;
        else
                nlevels = dn->dn_phys->dn_nlevels;

        epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;

        ASSERT3U(level * epbs, <, 64);
        ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
        if (level >= nlevels ||
            (blkid > (dn->dn_phys->dn_maxblkid >> (level * epbs)))) {
                /* the buffer has no parent yet */
                return (SET_ERROR(ENOENT));
        } else if (level < nlevels-1) {
                /* this block is referenced from an indirect block */
                int err = dbuf_hold_impl(dn, level+1,
                    blkid >> epbs, fail_sparse, FALSE, NULL, parentp);
                if (err)
                        return (err);
                err = dbuf_read(*parentp, NULL,
                    (DB_RF_HAVESTRUCT | DB_RF_NOPREFETCH | DB_RF_CANFAIL));
                if (err) {
                        dbuf_rele(*parentp, NULL);
                        *parentp = NULL;
                        return (err);
                }
```

```
                *bpp = ((blkptr_t *)(*parentp)->db.db_data) +
                    (blkid & ((1ULL << epbs) - 1));
                return (0);
        } else {
                /* the block is referenced from the dnode */
                ASSERT3U(level, ==, nlevels-1);
                ASSERT(dn->dn_phys->dn_nblkptr == 0 ||
                    blkid < dn->dn_phys->dn_nblkptr);
                if (dn->dn_dbuf) {
                        dbuf_add_ref(dn->dn_dbuf, NULL);
                        *parentp = dn->dn_dbuf;
                }
                *bpp = &dn->dn_phys->dn_blkptr[blkid];
                return (0);
        }
}

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

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

        db = kmem_cache_alloc(dbuf_cache, KM_SLEEP);

        db->db_objset = os;
        db->db.db_object = dn->dn_object;
        db->db_level = level;
        db->db_blkid = blkid;
        db->db_last_dirty = NULL;
        db->db_dirtycnt = 0;
        db->db_dnode_handle = dn->dn_handle;
        db->db_parent = parent;
        db->db_blkptr = blkptr;

        db->db_user = NULL;
        db->db_user_immediate_evict = FALSE;
        db->db_freed_in_flight = FALSE;
        db->db_pending_evict = FALSE;

        if (blkid == DMU_BONUS_BLKID) {
                ASSERT3P(parent, ==, dn->dn_dbuf);
                db->db.db_size = DN_MAX_BONUSLEN -
                    (dn->dn_nblkptr-1) * sizeof (blkptr_t);
                ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
                db->db.db_offset = DMU_BONUS_BLKID;
                db->db_state = DB_UNCACHED;
                /* the bonus dbuf is not placed in the hash table */
                arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
                return (db);
        } else if (blkid == DMU_SPILL_BLKID) {
                db->db.db_size = (blkptr != NULL) ?
                    BP_GET_LSIZE(blkptr) : SPA_MINBLOCKSIZE;
                db->db.db_offset = 0;
        } else {
                int blocksize =
                    db->db_level ? 1 << dn->dn_indblkshift : dn->dn_datablksz;
                db->db.db_size = blocksize;
                db->db.db_offset = db->db_blkid * blocksize;
        }

        /*
```

```
         * Hold the dn_dbufs_mtx while we get the new dbuf
         * in the hash table *and* added to the dbufs list.
         * This prevents a possible deadlock with someone
         * trying to look up this dbuf before its added to the
         * dn_dbufs list.
         */
        mutex_enter(&dn->dn_dbufs_mtx);
        db->db_state = DB_EVICTING;
        if ((odb = dbuf_hash_insert(db)) != NULL) {
                /* someone else inserted it first */
                kmem_cache_free(dbuf_cache, db);
                mutex_exit(&dn->dn_dbufs_mtx);
                return (odb);
        }
        avl_add(&dn->dn_dbufs, db);
        if (db->db_level == 0 && db->db_blkid >=
            dn->dn_unlisted_l0_blkid)
                dn->dn_unlisted_l0_blkid = db->db_blkid + 1;
        db->db_state = DB_UNCACHED;
        mutex_exit(&dn->dn_dbufs_mtx);
        arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);

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

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

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

        return (db);
}

static int
dbuf_do_evict(void *private)
{
        dmu_buf_impl_t *db = private;

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

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

        if (db->db_state != DB_EVICTING) {
                ASSERT(db->db_state == DB_CACHED);
                DBUF_VERIFY(db);
                db->db_buf = NULL;
                dbuf_evict(db);
        } else {
                mutex_exit(&db->db_mtx);
                dbuf_destroy(db);
        }
        return (0);
}

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

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

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

                        DB_DNODE_ENTER(db);
                        dn = DB_DNODE(db);
                        mutex_enter(&dn->dn_dbufs_mtx);
                        avl_remove(&dn->dn_dbufs, db);
                        atomic_dec_32(&dn->dn_dbufs_count);
                        mutex_exit(&dn->dn_dbufs_mtx);
                        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.
                         */
                        dnode_rele(dn, db);
                        db->db_dnode_handle = NULL;
                }
                dbuf_hash_remove(db);
        }
        db->db_parent = NULL;
        db->db_buf = NULL;

        ASSERT(db->db.db_data == NULL);
        ASSERT(db->db_hash_next == NULL);
        ASSERT(db->db_blkptr == NULL);
        ASSERT(db->db_data_pending == NULL);

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

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

/*
 * Actually issue the prefetch read for the block given.
 */
static void
dbuf_issue_final_prefetch(dbuf_prefetch_arg_t *dpa, blkptr_t *bp)
{
        if (BP_IS_HOLE(bp) || BP_IS_EMBEDDED(bp))
                return;

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

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

/*
 * 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);
        dpa->dpa_curlevel = curlevel;
        dpa->dpa_prio = prio;
        dpa->dpa_aflags = aflags;
        dpa->dpa_spa = dn->dn_objset->os_spa;
        dpa->dpa_epbs = epbs;
        dpa->dpa_zio = pio;

        /*
         * If we have the indirect just above us, no need to do the asynchronous
         * prefetch chain; we'll just run the last step ourselves.  If we're at
         * a higher level, though, we want to issue the prefetches for all the
         * indirect blocks asynchronously, so we can go on with whatever we were
         * doing.
         */
        if (curlevel == level) {
                ASSERT3U(curblkid, ==, 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;

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

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

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

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

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

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

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

```
                        if (err == 0 && bp && BP_IS_HOLE(bp))
                                err = SET_ERROR(ENOENT);
                        if (err) {
                                if (parent)
                                        dbuf_rele(parent, NULL);
                                return (err);
                        }
                }
                if (err && err != ENOENT)
                        return (err);
                db = dbuf_create(dn, level, blkid, parent, bp);
        }

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

        if (db->db_buf && refcount_is_zero(&db->db_holds)) {
                arc_buf_add_ref(db->db_buf, db);
                if (db->db_buf->b_data == NULL) {
                        dbuf_clear(db);
                        if (parent) {
                                dbuf_rele(parent, NULL);
                                parent = NULL;
                        }
                        goto top;
                }
                ASSERT3P(db->db.db_data, ==, db->db_buf->b_data);
        }

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

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

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

                        dbuf_set_data(db,
                            arc_buf_alloc(dn->dn_objset->os_spa,
                            db->db.db_size, db, type));
                        bcopy(dr->dt.dl.dr_data->b_data, db->db.db_data,
                            db->db.db_size);
                }
        }

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

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

        ASSERT3P(DB_DNODE(db), ==, dn);
        ASSERT3U(db->db_blkid, ==, blkid);
        ASSERT3U(db->db_level, ==, level);
        *dbp = db;
        return (0);
}

dmu_buf_impl_t *
dbuf_hold(dnode_t *dn, uint64_t blkid, void *tag)
{
        return (dbuf_hold_level(dn, 0, blkid, tag));
}

dmu_buf_impl_t *
dbuf_hold_level(dnode_t *dn, int level, uint64_t blkid, void *tag)
{
        dmu_buf_impl_t *db;
        int err = dbuf_hold_impl(dn, level, blkid, FALSE, FALSE, tag, &db);
        return (err ? NULL : db);
}

void
dbuf_create_bonus(dnode_t *dn)
{
        ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));

        ASSERT(dn->dn_bonus == NULL);
        dn->dn_bonus = dbuf_create(dn, 0, DMU_BONUS_BLKID, dn->dn_dbuf, NULL);
}

int
dbuf_spill_set_blksz(dmu_buf_t *db_fake, uint64_t blksz, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
        dnode_t *dn;

        if (db->db_blkid != DMU_SPILL_BLKID)
                return (SET_ERROR(ENOTSUP));
        if (blksz == 0)
                blksz = SPA_MINBLOCKSIZE;
        ASSERT3U(blksz, <=, spa_maxblocksize(dmu_objset_spa(db->db_objset)));
        blksz = P2ROUNDUP(blksz, SPA_MINBLOCKSIZE);

        DB_DNODE_ENTER(db);
        dn = DB_DNODE(db);
        rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
        dbuf_new_size(db, blksz, tx);
        rw_exit(&dn->dn_struct_rwlock);
        DB_DNODE_EXIT(db);

        return (0);
}

void
dbuf_rm_spill(dnode_t *dn, dmu_tx_t *tx)
{
        dbuf_free_range(dn, DMU_SPILL_BLKID, DMU_SPILL_BLKID, tx);
}

#pragma weak dmu_buf_add_ref = dbuf_add_ref
void
dbuf_add_ref(dmu_buf_impl_t *db, void *tag)
{
        int64_t holds = refcount_add(&db->db_holds, tag);
        ASSERT(holds > 1);
}

#pragma weak dmu_buf_try_add_ref = dbuf_try_add_ref
boolean_t
```

```
dbuf_try_add_ref(dmu_buf_t *db_fake, objset_t *os, uint64_t obj, uint64_t blkid,
    void *tag)
{
        dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
        dmu_buf_impl_t *found_db;
        boolean_t result = B_FALSE;

        if (db->db_blkid == DMU_BONUS_BLKID)
                found_db = dbuf_find_bonus(os, obj);
        else
                found_db = dbuf_find(os, obj, 0, blkid);

        if (found_db != NULL) {
                if (db == found_db && dbuf_refcount(db) > db->db_dirtycnt) {
                        (void) refcount_add(&db->db_holds, tag);
                        result = B_TRUE;
                }
                mutex_exit(&db->db_mtx);
        }
        return (result);
}

/*
 * If you call dbuf_rele() you had better not be referencing the dnode handle
 * unless you have some other direct or indirect hold on the dnode. (An indirect
 * hold is a hold on one of the dnode’s dbufs, including the bonus buffer.)
 * Without that, the dbuf_rele() could lead to a dnode_rele() followed by the
 * dnode’s parent dbuf evicting its dnode handles.
 */
void
dbuf_rele(dmu_buf_impl_t *db, void *tag)
{
        mutex_enter(&db->db_mtx);
        dbuf_rele_and_unlock(db, tag);
}

void
dmu_buf_rele(dmu_buf_t *db, void *tag)
{
        dbuf_rele((dmu_buf_impl_t *)db, tag);
}

/*
 * dbuf_rele() for an already-locked dbuf.  This is necessary to allow
 * db_dirtycnt and db_holds to be updated atomically.
 */
void
dbuf_rele_and_unlock(dmu_buf_impl_t *db, void *tag)
{
        int64_t holds;

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

        /*
         * Remove the reference to the dbuf before removing its hold on the
         * dnode so we can guarantee in dnode_move() that a referenced bonus
         * buffer has a corresponding dnode hold.
         */
        holds = refcount_remove(&db->db_holds, tag);
        ASSERT(holds >= 0);

        /*
         * We can’t freeze indirects if there is a possibility that they
         * may be modified in the current syncing context.
         */
        if (db->db_buf && holds == (db->db_level == 0 ? db->db_dirtycnt : 0))
                arc_buf_freeze(db->db_buf);

        if (holds == db->db_dirtycnt &&
            db->db_level == 0 && db->db_user_immediate_evict)
                dbuf_evict_user(db);

        if (holds == 0) {
                if (db->db_blkid == DMU_BONUS_BLKID) {
                        dnode_t *dn;
                        boolean_t evict_dbuf = db->db_pending_evict;

                        /*
                         * If the dnode moves here, we cannot cross this
                         * barrier until the move completes.
                         */
                        DB_DNODE_ENTER(db);

                        dn = DB_DNODE(db);
                        atomic_dec_32(&dn->dn_dbufs_count);

                        /*
                         * Decrementing the dbuf count means that the bonus
                         * buffer’s dnode hold is no longer discounted in
                         * dnode_move(). The dnode cannot move until after
                         * the dnode_rele() below.
                         */
                        DB_DNODE_EXIT(db);

                        /*
                         * Do not reference db after its lock is dropped.
                         * Another thread may evict it.
                         */
                        mutex_exit(&db->db_mtx);

                        if (evict_dbuf)
                                dnode_evict_bonus(dn);

                        dnode_rele(dn, db);
                } else if (db->db_buf == NULL) {
                        /*
                         * This is a special case: we never associated this
                         * dbuf with any data allocated from the ARC.
                         */
                        ASSERT(db->db_state == DB_UNCACHED ||
                            db->db_state == DB_NOFILL);
                        dbuf_evict(db);
                } else if (arc_released(db->db_buf)) {
                        arc_buf_t *buf = db->db_buf;
                        /*
                         * This dbuf has anonymous data associated with it.
                         */
                        dbuf_clear_data(db);
                        VERIFY(arc_buf_remove_ref(buf, db));
                        dbuf_evict(db);
                } else {
                        VERIFY(!arc_buf_remove_ref(db->db_buf, db));

                        /*
                         * A dbuf will be eligible for eviction if either the
                         * ’primarycache’ property is set or a duplicate
                         * copy of this buffer is already cached in the arc.
                         *
                         * In the case of the ’primarycache’ a buffer
                         * is considered for eviction if it matches the
                         * criteria set in the property.
```

```
                         *
                         * To decide if our buffer is considered a
                         * duplicate, we must call into the arc to determine
                         * if multiple buffers are referencing the same
                         * block on-disk. If so, then we simply evict
                         * ourselves.
                         */
                        if (!DBUF_IS_CACHEABLE(db)) {
                                if (db->db_blkptr != NULL &&
                                    !BP_IS_HOLE(db->db_blkptr) &&
                                    !BP_IS_EMBEDDED(db->db_blkptr)) {
                                        spa_t *spa =
                                            dmu_objset_spa(db->db_objset);
                                        blkptr_t bp = *db->db_blkptr;
                                        dbuf_clear(db);
                                        arc_freed(spa, &bp);
                                } else {
                                        dbuf_clear(db);
                                }
                        } else if (db->db_pending_evict ||
                            arc_buf_eviction_needed(db->db_buf)) {
                                dbuf_clear(db);
                        } else {
                                mutex_exit(&db->db_mtx);
                        }
                }
        } else {
                mutex_exit(&db->db_mtx);
        }
}

#pragma weak dmu_buf_refcount = dbuf_refcount
uint64_t
dbuf_refcount(dmu_buf_impl_t *db)
{
        return (refcount_count(&db->db_holds));
}

void *
dmu_buf_replace_user(dmu_buf_t *db_fake, dmu_buf_user_t *old_user,
    dmu_buf_user_t *new_user)
{
        dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

        mutex_enter(&db->db_mtx);
        dbuf_verify_user(db, DBVU_NOT_EVICTING);
        if (db->db_user == old_user)
                db->db_user = new_user;
        else
                old_user = db->db_user;
        dbuf_verify_user(db, DBVU_NOT_EVICTING);
        mutex_exit(&db->db_mtx);

        return (old_user);
}

void *
dmu_buf_set_user(dmu_buf_t *db_fake, dmu_buf_user_t *user)
{
        return (dmu_buf_replace_user(db_fake, NULL, user));
}

void *
dmu_buf_set_user_ie(dmu_buf_t *db_fake, dmu_buf_user_t *user)
{
        dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
        db->db_user_immediate_evict = TRUE;
        return (dmu_buf_set_user(db_fake, user));
}

void *
dmu_buf_remove_user(dmu_buf_t *db_fake, dmu_buf_user_t *user)
{
        return (dmu_buf_replace_user(db_fake, user, NULL));
}

void *
dmu_buf_get_user(dmu_buf_t *db_fake)
{
        dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

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

void
dmu_buf_user_evict_wait()
{
        taskq_wait(dbu_evict_taskq);
}

boolean_t
dmu_buf_freeable(dmu_buf_t *dbuf)
{
        boolean_t res = B_FALSE;
        dmu_buf_impl_t *db = (dmu_buf_impl_t *)dbuf;

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

        return (res);
}

blkptr_t *
dmu_buf_get_blkptr(dmu_buf_t *db)
{
        dmu_buf_impl_t *dbi = (dmu_buf_impl_t *)db;
        return (dbi->db_blkptr);
}

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

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

        if (db->db_blkid == DMU_SPILL_BLKID) {
                db->db_blkptr = &dn->dn_phys->dn_spill;
                BP_ZERO(db->db_blkptr);
                return;
        }
        if (db->db_level == dn->dn_phys->dn_nlevels-1) {
                /*
                 * This buffer was allocated at a time when there was
                 * no available blkptrs from the dnode, or it was
                 * inappropriate to hook it in (i.e., nlevels mis-match).
                 */
                ASSERT(db->db_blkid < dn->dn_phys->dn_nblkptr);
                ASSERT(db->db_parent == NULL);
                db->db_parent = dn->dn_dbuf;
                db->db_blkptr = &dn->dn_phys->dn_blkptr[db->db_blkid];
                DBUF_VERIFY(db);
        } else {
                dmu_buf_impl_t *parent = db->db_parent;
                int epbs = dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;

                ASSERT(dn->dn_phys->dn_nlevels > 1);
                if (parent == NULL) {
                        mutex_exit(&db->db_mtx);
                        rw_enter(&dn->dn_struct_rwlock, RW_READER);
                        parent = dbuf_hold_level(dn, db->db_level + 1,
                            db->db_blkid >> epbs, db);
                        rw_exit(&dn->dn_struct_rwlock);
                        mutex_enter(&db->db_mtx);
                        db->db_parent = parent;
                }
                db->db_blkptr = (blkptr_t *)parent->db.db_data +
                    (db->db_blkid & ((1ULL << epbs) - 1));
                DBUF_VERIFY(db);
        }
}

static void
dbuf_sync_indirect(dbuf_dirty_record_t *dr, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db = dr->dr_dbuf;
        dnode_t *dn;
        zio_t *zio;

        ASSERT(dmu_tx_is_syncing(tx));

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

        mutex_enter(&db->db_mtx);

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

        /* Read the block if it hasn’t been read yet. */
        if (db->db_buf == NULL) {
                mutex_exit(&db->db_mtx);
                (void) dbuf_read(db, NULL, DB_RF_MUST_SUCCEED);
                mutex_enter(&db->db_mtx);
        }
        ASSERT3U(db->db_state, ==, DB_CACHED);
        ASSERT(db->db_buf != NULL);

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

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

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

        zio = dr->dr_zio;
        mutex_enter(&dr->dt.di.dr_mtx);
        dbuf_sync_list(&dr->dt.di.dr_children, db->db_level - 1, tx);
        ASSERT(list_head(&dr->dt.di.dr_children) == NULL);
        mutex_exit(&dr->dt.di.dr_mtx);
        zio_nowait(zio);
}

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

        ASSERT(dmu_tx_is_syncing(tx));

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

        mutex_enter(&db->db_mtx);
        /*
         * To be synced, we must be dirtied.  But we
         * might have been freed after the dirty.
         */
        if (db->db_state == DB_UNCACHED) {
                /* This buffer has been freed since it was dirtied */
                ASSERT(db->db.db_data == NULL);
        } else if (db->db_state == DB_FILL) {
                /* This buffer was freed and is now being re-filled */
                ASSERT(db->db.db_data != dr->dt.dl.dr_data);
        } else {
                ASSERT(db->db_state == DB_CACHED || db->db_state == DB_NOFILL);
        }
        DBUF_VERIFY(db);

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

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

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

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

                if (*datap != db->db.db_data) {
                        zio_buf_free(*datap, DN_MAX_BONUSLEN);
                        arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
                }
                db->db_data_pending = NULL;
                drp = &db->db_last_dirty;
                while (*drp != dr)
                        drp = &(*drp)->dr_next;
```

```
                ASSERT(dr->dr_next == NULL);
                ASSERT(dr->dr_dbuf == db);
                *drp = dr->dr_next;
                kmem_free(dr, sizeof (dbuf_dirty_record_t));
                ASSERT(db->db_dirtycnt > 0);
                db->db_dirtycnt -= 1;
                dbuf_rele_and_unlock(db, (void *)(uintptr_t)txg);
                return;
        }

        os = dn->dn_objset;

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

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

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

        mutex_exit(&db->db_mtx);

        dbuf_write(dr, *datap, tx);

        ASSERT(!list_link_active(&dr->dr_dirty_node));
        if (dn->dn_object == DMU_META_DNODE_OBJECT) {
                list_insert_tail(&dn->dn_dirty_records[txg&TXG_MASK], dr);
                DB_DNODE_EXIT(db);
        } else {
                /*
                 * Although zio_nowait() does not "wait for an IO", it does
                 * initiate the IO. If this is an empty write it seems plausible
                 * that the IO could actually be completed before the nowait
                 * returns. We need to DB_DNODE_EXIT() first in case
                 * zio_nowait() invalidates the dbuf.
                 */
                DB_DNODE_EXIT(db);
                zio_nowait(dr->dr_zio);
        }
}

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

        while (dr = list_head(list)) {
                if (dr->dr_zio != NULL) {
                        /*
                         * If we find an already initialized zio then we
                         * are processing the meta-dnode, and we have finished.
                         * The dbufs for all dnodes are put back on the list
                         * during processing, so that we can zio_wait()
                         * these IOs after initiating all child IOs.
                         */
                        ASSERT3U(dr->dr_dbuf->db.db_object, ==,
                            DMU_META_DNODE_OBJECT);
                        break;
                }
                if (dr->dr_dbuf->db_blkid != DMU_BONUS_BLKID &&
                    dr->dr_dbuf->db_blkid != DMU_SPILL_BLKID) {
                        VERIFY3U(dr->dr_dbuf->db_level, ==, level);
                }
                list_remove(list, dr);
                if (dr->dr_dbuf->db_level > 0)
                        dbuf_sync_indirect(dr, tx);
                else
                        dbuf_sync_leaf(dr, tx);
        }
}

/* ARGSUSED */
static void
dbuf_write_ready(zio_t *zio, arc_buf_t *buf, void *vdb)
{
        dmu_buf_impl_t *db = vdb;
        dnode_t *dn;
        blkptr_t *bp = zio->io_bp;
        blkptr_t *bp_orig = &zio->io_bp_orig;
        spa_t *spa = zio->io_spa;
        int64_t delta;
        uint64_t fill = 0;
        int i;

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

        DB_DNODE_ENTER(db);
        dn = DB_DNODE(db);
        delta = bp_get_dsize_sync(spa, bp) - bp_get_dsize_sync(spa, bp_orig);
        dnode_diduse_space(dn, delta - zio->io_prev_space_delta);
        zio->io_prev_space_delta = delta;

        if (bp->blk_birth != 0) {
                ASSERT((db->db_blkid != DMU_SPILL_BLKID &&
                    BP_GET_TYPE(bp) == dn->dn_type) ||
                    (db->db_blkid == DMU_SPILL_BLKID &&
                    BP_GET_TYPE(bp) == dn->dn_bonustype) ||
                    BP_IS_EMBEDDED(bp));
                ASSERT(BP_GET_LEVEL(bp) == db->db_level);
```

```
        }

        mutex_enter(&db->db_mtx);

#ifdef ZFS_DEBUG
        if (db->db_blkid == DMU_SPILL_BLKID) {
                ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
                ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
                    db->db_blkptr == &dn->dn_phys->dn_spill);
        }
#endif

        if (db->db_level == 0) {
                mutex_enter(&dn->dn_mtx);
                if (db->db_blkid > dn->dn_phys->dn_maxblkid &&
                    db->db_blkid != DMU_SPILL_BLKID)
                        dn->dn_phys->dn_maxblkid = db->db_blkid;
                mutex_exit(&dn->dn_mtx);

                if (dn->dn_type == DMU_OT_DNODE) {
                        dnode_phys_t *dnp = db->db.db_data;
                        for (i = db->db.db_size >> DNODE_SHIFT; i > 0;
                            i--, dnp++) {
                                if (dnp->dn_type != DMU_OT_NONE)
                                        fill++;
                        }
                } else {
                        if (BP_IS_HOLE(bp)) {
                                fill = 0;
                        } else {
                                fill = 1;
                        }
                }
        } else {
                blkptr_t *ibp = db->db.db_data;
                ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
                for (i = db->db.db_size >> SPA_BLKPTRSHIFT; i > 0; i--, ibp++) {
                        if (BP_IS_HOLE(ibp))
                                continue;
                        fill += BP_GET_FILL(ibp);
                }
        }
        DB_DNODE_EXIT(db);

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

        mutex_exit(&db->db_mtx);
}

/*
 * The SPA will call this callback several times for each zio - once
 * for every physical child i/o (zio->io_phys_children times).  This
 * allows the DMU to monitor the progress of each logical i/o.  For example,
 * there may be 2 copies of an indirect block, or many fragments of a RAID-Z
 * block.  There may be a long delay before all copies/fragments are completed,
 * so this callback allows us to retire dirty space gradually, as the physical
 * i/os complete.
 */
/* ARGSUSED */
static void
dbuf_write_physdone(zio_t *zio, arc_buf_t *buf, void *arg)
{
        dmu_buf_impl_t *db = arg;
        objset_t *os = db->db_objset;
        dsl_pool_t *dp = dmu_objset_pool(os);
        dbuf_dirty_record_t *dr;
        int delta = 0;

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

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

/* ARGSUSED */
static void
dbuf_write_done(zio_t *zio, arc_buf_t *buf, void *vdb)
{
        dmu_buf_impl_t *db = vdb;
        blkptr_t *bp_orig = &zio->io_bp_orig;
        blkptr_t *bp = db->db_blkptr;
        objset_t *os = db->db_objset;
        dmu_tx_t *tx = os->os_synctx;
        dbuf_dirty_record_t **drp, *dr;

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

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

        mutex_enter(&db->db_mtx);

        DBUF_VERIFY(db);

        drp = &db->db_last_dirty;
        while ((dr = *drp) != db->db_data_pending)
                drp = &dr->dr_next;
        ASSERT(!list_link_active(&dr->dr_dirty_node));
        ASSERT(dr->dr_dbuf == db);
        ASSERT(dr->dr_next == NULL);
        *drp = dr->dr_next;

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

                DB_DNODE_ENTER(db);
                dn = DB_DNODE(db);
                ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
                ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
                    db->db_blkptr == &dn->dn_phys->dn_spill);
                DB_DNODE_EXIT(db);
        }
#endif
```

```
        if (db->db_level == 0) {
                ASSERT(db->db_blkid != DMU_BONUS_BLKID);
                ASSERT(dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN);
                if (db->db_state != DB_NOFILL) {
                        if (dr->dt.dl.dr_data != db->db_buf)
                                VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data,
                                    db));
                        else if (!arc_released(db->db_buf))
                                arc_set_callback(db->db_buf, dbuf_do_evict, db);
                }
        } else {
                dnode_t *dn;

                DB_DNODE_ENTER(db);
                dn = DB_DNODE(db);
                ASSERT(list_head(&dr->dt.di.dr_children) == NULL);
                ASSERT3U(db->db.db_size, ==, 1 << dn->dn_phys->dn_indblkshift);
                if (!BP_IS_HOLE(db->db_blkptr)) {
                        int epbs =
                            dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;
                        ASSERT3U(db->db_blkid, <=,
                            dn->dn_phys->dn_maxblkid >> (db->db_level * epbs));
                        ASSERT3U(BP_GET_LSIZE(db->db_blkptr), ==,
                            db->db.db_size);
                        if (!arc_released(db->db_buf))
                                arc_set_callback(db->db_buf, dbuf_do_evict, db);
                }
                DB_DNODE_EXIT(db);
                mutex_destroy(&dr->dt.di.dr_mtx);
                list_destroy(&dr->dt.di.dr_children);
        }
        kmem_free(dr, sizeof (dbuf_dirty_record_t));

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

static void
dbuf_write_nofill_ready(zio_t *zio)
{
        dbuf_write_ready(zio, NULL, zio->io_private);
}

static void
dbuf_write_nofill_done(zio_t *zio)
{
        dbuf_write_done(zio, NULL, zio->io_private);
}

static void
dbuf_write_override_ready(zio_t *zio)
{
        dbuf_dirty_record_t *dr = zio->io_private;
        dmu_buf_impl_t *db = dr->dr_dbuf;

        dbuf_write_ready(zio, NULL, db);
}

static void
dbuf_write_override_done(zio_t *zio)
{
        dbuf_dirty_record_t *dr = zio->io_private;
        dmu_buf_impl_t *db = dr->dr_dbuf;
        blkptr_t *obp = &dr->dt.dl.dr_overridden_by;

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

        dbuf_write_done(zio, NULL, db);
}

/* Issue I/O to commit a dirty buffer to disk. */
static void
dbuf_write(dbuf_dirty_record_t *dr, arc_buf_t *data, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db = dr->dr_dbuf;
        dnode_t *dn;
        objset_t *os;
        dmu_buf_impl_t *parent = db->db_parent;
        uint64_t txg = tx->tx_txg;
        zbookmark_phys_t zb;
        zio_prop_t zp;
        zio_t *zio;
        int wp_flag = 0;

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

        if (db->db_state != DB_NOFILL) {
                if (db->db_level > 0 || dn->dn_type == DMU_OT_DNODE) {
                        /*
                         * Private object buffers are released here rather
                         * than in dbuf_dirty() since they are only modified
                         * in the syncing context and we don’t want the
                         * overhead of making multiple copies of the data.
                         */
                        if (BP_IS_HOLE(db->db_blkptr)) {
                                arc_buf_thaw(data);
                        } else {
                                dbuf_release_bp(db);
                        }
                }
        }

        if (parent != dn->dn_dbuf) {
                /* Our parent is an indirect block. */
                /* We have a dirty parent that has been scheduled for write. */
                ASSERT(parent && parent->db_data_pending);
                /* Our parent’s buffer is one level closer to the dnode. */
                ASSERT(db->db_level == parent->db_level-1);
                /*
                 * We’re about to modify our parent’s db_data by modifying
                 * our block pointer, so the parent must be released.
                 */
                ASSERT(arc_released(parent->db_buf));
                zio = parent->db_data_pending->dr_zio;
        } else {
                /* Our parent is the dnode itself. */
                ASSERT((db->db_level == dn->dn_phys->dn_nlevels-1 &&
                    db->db_blkid != DMU_SPILL_BLKID) ||
                    (db->db_blkid == DMU_SPILL_BLKID && db->db_level == 0));
                if (db->db_blkid != DMU_SPILL_BLKID)
```

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

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

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

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

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

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

                dr->dr_zio = zio_write(zio, os->os_spa, txg,
                    db->db_blkptr, contents, db->db.db_size, &zp,
                    dbuf_write_override_ready, NULL, dbuf_write_override_done,
                    dr, ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
                mutex_enter(&db->db_mtx);
                dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
                zio_write_override(dr->dr_zio, &dr->dt.dl.dr_overridden_by,
                    dr->dt.dl.dr_copies, dr->dt.dl.dr_nopwrite);
                mutex_exit(&db->db_mtx);
        } else if (db->db_state == DB_NOFILL) {
                ASSERT(zp.zp_checksum == ZIO_CHECKSUM_OFF ||
                    zp.zp_checksum == ZIO_CHECKSUM_NOPARITY);
                dr->dr_zio = zio_write(zio, os->os_spa, txg,
                    db->db_blkptr, NULL, db->db.db_size, &zp,
                    dbuf_write_nofill_ready, NULL, dbuf_write_nofill_done, db,
                    ZIO_PRIORITY_ASYNC_WRITE,
                    ZIO_FLAG_MUSTSUCCEED | ZIO_FLAG_NODATA, &zb);
        } else {
                ASSERT(arc_released(data));
                dr->dr_zio = arc_write(zio, os->os_spa, txg,
                    db->db_blkptr, data, DBUF_IS_L2CACHEABLE(db),
                    DBUF_IS_L2COMPRESSIBLE(db), &zp, dbuf_write_ready,
                    dbuf_write_physdone, dbuf_write_done, db,
                    ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
        }
}
```

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

```
/*
 * errors:
 * EINVAL - invalid object number.
 * EIO - i/o error.
 * succeeds even for free dnodes.
 */
int
dnode_hold_impl(objset_t *os, uint64_t object, int flag,
    void *tag, dnode_t **dnp)
{
        int epb, idx, err;
        int drop_struct_lock = FALSE;
        int type;
        uint64_t blk;
        dnode_t *mdn, *dn;
        dmu_buf_impl_t *db;
        dnode_children_t *children_dnodes;
        dnode_handle_t *dnh;

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

        if (object == DMU_USERUSED_OBJECT || object == DMU_GROUPUSED_OBJECT) {
                dn = (object == DMU_USERUSED_OBJECT) ?
                    DMU_USERUSED_DNODE(os) : DMU_GROUPUSED_DNODE(os);
                if (dn == NULL)
                        return (SET_ERROR(ENOENT));
                type = dn->dn_type;
                if ((flag & DNODE_MUST_BE_ALLOCATED) && type == DMU_OT_NONE)
                        return (SET_ERROR(ENOENT));
                if ((flag & DNODE_MUST_BE_FREE) && type != DMU_OT_NONE)
                        return (SET_ERROR(EEXIST));
                DNODE_VERIFY(dn);
                (void) refcount_add(&dn->dn_holds, tag);
                *dnp = dn;
                return (0);
        }

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

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

        DNODE_VERIFY(mdn);

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

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

```

        db = dbuf_hold(mdn, blk, FTAG);
        if (drop_struct_lock)
                rw_exit(&mdn->dn_struct_rwlock);
        if (db == NULL)
                return (SET_ERROR(EIO));
        err = dbuf_read(db, NULL, DB_RF_CANFAIL);
        if (err) {
                dbuf_rele(db, FTAG);
                return (err);
        }

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

        idx = object & (epb-1);

        ASSERT(DB_DNODE(db)->dn_type == DMU_OT_DNODE);
        children_dnodes = dmu_buf_get_user(&db->db);
        if (children_dnodes == NULL) {
                int i;
                dnode_children_t *winner;
                children_dnodes = kmem_zalloc(sizeof (dnode_children_t) +
                    epb * sizeof (dnode_handle_t), KM_SLEEP);
                children_dnodes->dnc_count = epb;
                dnh = &children_dnodes->dnc_children[0];
                for (i = 0; i < epb; i++) {
                        zrl_init(&dnh[i].dnh_zrlock);
                }
                dmu_buf_init_user(&children_dnodes->dnc_dbu, NULL,
                dmu_buf_init_user(&children_dnodes->dnc_dbu,
                    dnode_buf_pageout, NULL);
                winner = dmu_buf_set_user(&db->db, &children_dnodes->dnc_dbu);
                if (winner != NULL) {

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

                        kmem_free(children_dnodes, sizeof (dnode_children_t) +
                            epb * sizeof (dnode_handle_t));
                        children_dnodes = winner;
                }
        }
        ASSERT(children_dnodes->dnc_count == epb);

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

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

        mutex_enter(&dn->dn_mtx);
        type = dn->dn_type;
        if (dn->dn_free_txg ||
            ((flag & DNODE_MUST_BE_ALLOCATED) && type == DMU_OT_NONE) ||
            ((flag & DNODE_MUST_BE_FREE) &&
            (type != DMU_OT_NONE || !refcount_is_zero(&dn->dn_holds)))) {
                mutex_exit(&dn->dn_mtx);
                zrl_remove(&dnh->dnh_zrlock);
                dbuf_rele(db, FTAG);
                return (type == DMU_OT_NONE ? ENOENT : EEXIST);
        }
        if (refcount_add(&dn->dn_holds, tag) == 1)
                dbuf_add_ref(db, dnh);
        mutex_exit(&dn->dn_mtx);

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

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

        *dnp = dn;
        return (0);
}
```

______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_

static void
dsl_dataset_evict_prep(void *dbu)
{
        dsl_dataset_t *ds = dbu;

        ASSERT(ds->ds_owner == NULL);

        unique_remove(ds->ds_fsid_guid);
}

static void
#endif /* ! codereview */
dsl_dataset_evict(void *dbu)
{
        dsl_dataset_t *ds = dbu;

        ASSERT(ds->ds_owner == NULL);

        ds->ds_dbuf = NULL;

        unique_remove(ds->ds_fsid_guid);

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

        if (ds->ds_prev) {
                dsl_dataset_rele(ds->ds_prev, ds);
                ds->ds_prev = NULL;
        }

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

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

        list_destroy(&ds->ds_prop_cbs);
        mutex_destroy(&ds->ds_lock);
        mutex_destroy(&ds->ds_opening_lock);
        mutex_destroy(&ds->ds_sendstream_lock);
        refcount_destroy(&ds->ds_longholds);

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

int
dsl_dataset_hold_obj(dsl_pool_t *dp, uint64_t dsobj, void *tag,
    dsl_dataset_t **dsp)
{
        objset_t *mos = dp->dp_meta_objset;
        dmu_buf_t *dbuf;
        dsl_dataset_t *ds;
        int err;
        dmu_object_info_t doi;

        ASSERT(dsl_pool_config_held(dp));
```

```

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

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

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

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

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

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

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

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

                if (doi.doi_type == DMU_OTN_ZAP_METADATA) {
                        for (spa_feature_t f = 0; f < SPA_FEATURES; f++) {
                                if (!(spa_feature_table[f].fi_flags &
                                    ZFEATURE_FLAG_PER_DATASET))
                                        continue;
                                err = zap_contains(mos, dsobj,
                                    spa_feature_table[f].fi_guid);
                                if (err == 0) {
                                        ds->ds_feature_inuse[f] = B_TRUE;
                                } else {
                                        ASSERT3U(err, ==, ENOENT);
                                        err = 0;
                                }
                        }
                }

                err = dsl_dir_hold_obj(dp,
                    dsl_dataset_phys(ds)->ds_dir_obj, NULL, ds, &ds->ds_dir);
                if (err != 0) {
                        mutex_destroy(&ds->ds_lock);
                        mutex_destroy(&ds->ds_opening_lock);
                        mutex_destroy(&ds->ds_sendstream_lock);
                        refcount_destroy(&ds->ds_longholds);
                        bplist_destroy(&ds->ds_pending_deadlist);
                        dsl_deadlist_close(&ds->ds_deadlist);
                        kmem_free(ds, sizeof (dsl_dataset_t));
                        dmu_buf_rele(dbuf, tag);
                        return (err);
                }
                if (!ds->ds_is_snapshot) {
                        ds->ds_snapname[0] = '\0';
                        if (dsl_dataset_phys(ds)->ds_prev_snap_obj != 0) {
                                err = dsl_dataset_hold_obj(dp,
                                    dsl_dataset_phys(ds)->ds_prev_snap_obj,
                                    ds, &ds->ds_prev);
                        }
                        if (doi.doi_type == DMU_OTN_ZAP_METADATA) {
                                int zaperr = zap_lookup(mos, ds->ds_object,
                                    DS_FIELD_BOOKMARK_NAMES,
                                    sizeof (ds->ds_bookmarks), 1,
                                    &ds->ds_bookmarks);
                                if (zaperr != ENOENT)
                                        VERIFY0(zaperr);
                        }
                } else {
                        if (zfs_flags & ZFS_DEBUG_SNAPNAMES)
                                err = dsl_dataset_get_snapname(ds);
                        if (err == 0 &&
                            dsl_dataset_phys(ds)->ds_userrefs_obj != 0) {
                                err = zap_count(
                                    ds->ds_dir->dd_pool->dp_meta_objset,
                                    dsl_dataset_phys(ds)->ds_userrefs_obj,
                                    &ds->ds_userrefs);
                        }
                }

                if (err == 0 && !ds->ds_is_snapshot) {
                        err = dsl_prop_get_int_ds(ds,
                            zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
                            &ds->ds_reserved);
                        if (err == 0) {
                                err = dsl_prop_get_int_ds(ds,
                                    zfs_prop_to_name(ZFS_PROP_REFQUOTA),
                                    &ds->ds_quota);
                        }
                } else {
                        ds->ds_reserved = ds->ds_quota = 0;
                }

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

                if (err != 0 || winner != NULL) {
                        bplist_destroy(&ds->ds_pending_deadlist);
                        dsl_deadlist_close(&ds->ds_deadlist);
                        if (ds->ds_prev)
                                dsl_dataset_rele(ds->ds_prev, ds);
                        dsl_dir_rele(ds->ds_dir, ds);
                        mutex_destroy(&ds->ds_lock);
                        mutex_destroy(&ds->ds_opening_lock);
                        mutex_destroy(&ds->ds_sendstream_lock);
                        refcount_destroy(&ds->ds_longholds);
                        kmem_free(ds, sizeof (dsl_dataset_t));
                        if (err != 0) {
                                dmu_buf_rele(dbuf, tag);
                                return (err);
                        }
                        ds = winner;
                } else {
                        ds->ds_fsid_guid =
                            unique_insert(dsl_dataset_phys(ds)->ds_fsid_guid);
                }
```

```
        }
        ASSERT3P(ds->ds_dbuf, ==, dbuf);
        ASSERT3P(dsl_dataset_phys(ds), ==, dbuf->db_data);
        ASSERT(dsl_dataset_phys(ds)->ds_prev_snap_obj != 0 ||
            spa_version(dp->dp_spa) < SPA_VERSION_ORIGIN ||
            dp->dp_origin_snap == NULL || ds == dp->dp_origin_snap);
        *dsp = ds;
        return (0);
}
```

______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_

int
dsl_dir_hold_obj(dsl_pool_t *dp, uint64_t ddobj,
    const char *tail, void *tag, dsl_dir_t **ddp)
{
        dmu_buf_t *dbuf;
        dsl_dir_t *dd;
        int err;

        ASSERT(dsl_pool_config_held(dp));

        err = dmu_bonus_hold(dp->dp_meta_objset, ddobj, tag, &dbuf);
        if (err != 0)
                return (err);
        dd = dmu_buf_get_user(dbuf);
#ifdef ZFS_DEBUG
        {
                dmu_object_info_t doi;
                dmu_object_info_from_db(dbuf, &doi);
                ASSERT3U(doi.doi_bonus_type, ==, DMU_OT_DSL_DIR);
                ASSERT3U(doi.doi_bonus_size, >=, sizeof (dsl_dir_phys_t));
        }
#endif
        if (dd == NULL) {
                dsl_dir_t *winner;

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

                dsl_dir_snap_cmtime_update(dd);

                if (dsl_dir_phys(dd)->dd_parent_obj) {
                        err = dsl_dir_hold_obj(dp,
                            dsl_dir_phys(dd)->dd_parent_obj, NULL, dd,
                            &dd->dd_parent);
                        if (err != 0)
                                goto errout;
                        if (tail) {
#ifdef ZFS_DEBUG
                                uint64_t foundobj;

                                err = zap_lookup(dp->dp_meta_objset,
                                    dsl_dir_phys(dd->dd_parent)->
                                    dd_child_dir_zapobj, tail,
                                    sizeof (foundobj), 1, &foundobj);
                                ASSERT(err || foundobj == ddobj);
#endif
                                (void) strcpy(dd->dd_myname, tail);
                        } else {
                                err = zap_value_search(dp->dp_meta_objset,
                                    dsl_dir_phys(dd->dd_parent)->
                                    dd_child_dir_zapobj,
                                    ddobj, 0, dd->dd_myname);
                        }
                        if (err != 0)
                                goto errout;
```

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

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

                        /*
                         * We can't open the origin dataset, because
                         * that would require opening this dsl_dir.
                         * Just look at its phys directly instead.
                         */
                        err = dmu_bonus_hold(dp->dp_meta_objset,
                            dsl_dir_phys(dd)->dd_origin_obj, FTAG,
                            &origin_bonus);
                        if (err != 0)
                                goto errout;
                        origin_phys = origin_bonus->db_data;
                        dd->dd_origin_txg =
                            origin_phys->ds_creation_txg;
                        dmu_buf_rele(origin_bonus, FTAG);
                }

                dmu_buf_init_user(&dd->dd_dbu, NULL, dsl_dir_evict,
                    &dd->dd_dbuf);
                dmu_buf_init_user(&dd->dd_dbu, dsl_dir_evict, &dd->dd_dbuf);
                winner = dmu_buf_set_user_ie(dbuf, &dd->dd_dbu);
                if (winner != NULL) {
                        if (dd->dd_parent)
                                dsl_dir_rele(dd->dd_parent, dd);
                        dsl_prop_fini(dd);
                        mutex_destroy(&dd->dd_lock);
                        kmem_free(dd, sizeof (dsl_dir_t));
                        dd = winner;
                } else {
                        spa_open_ref(dp->dp_spa, dd);
                }
        }

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

errout:
        if (dd->dd_parent)
                dsl_dir_rele(dd->dd_parent, dd);
        dsl_prop_fini(dd);
        mutex_destroy(&dd->dd_lock);
        kmem_free(dd, sizeof (dsl_dir_t));
        dmu_buf_rele(dbuf, tag);
        return (err);
}
______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_

int
sa_handle_get_from_db(objset_t *os, dmu_buf_t *db, void *userp,
    sa_handle_type_t hdl_type, sa_handle_t **handlepp)
{
        int error = 0;
        dmu_object_info_t doi;
        sa_handle_t *handle = NULL;

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

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

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

                handle = kmem_cache_alloc(sa_cache, KM_SLEEP);
                handle->sa_dbu.dbu_evict_func_prep = NULL;
#endif /* ! codereview */
                handle->sa_dbu.dbu_evict_func = NULL;
                handle->sa_userp = userp;
                handle->sa_bonus = db;
                handle->sa_os = os;
                handle->sa_spill = NULL;
                handle->sa_bonus_tab = NULL;
                handle->sa_spill_tab = NULL;

                error = sa_build_index(handle, SA_BONUS);

                if (hdl_type == SA_HDL_SHARED) {
                        dmu_buf_init_user(&handle->sa_dbu, NULL, sa_evict,
                            NULL);
                        dmu_buf_init_user(&handle->sa_dbu, sa_evict, NULL);
                        winner = dmu_buf_set_user_ie(db, &handle->sa_dbu);
                }

                if (winner != NULL) {
                        kmem_cache_free(sa_cache, handle);
                        handle = winner;
                }
        }
        *handlepp = handle;

        return (error);
}
______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_

/*
 * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
 */
#define DMU_POOL_DIRECTORY_OBJECT      1
#define DMU_POOL_CONFIG                 "config"
#define DMU_POOL_FEATURES_FOR_WRITE     "features_for_write"
#define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
#define DMU_POOL_FEATURE_DESCRIPTIONS   "feature_descriptions"
#define DMU_POOL_FEATURE_ENABLED_TXG    "feature_enabled_txg"
#define DMU_POOL_ROOT_DATASET           "root_dataset"
#define DMU_POOL_SYNC_BPOBJ             "sync_bplist"
#define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
#define DMU_POOL_ERRLOG_LAST            "errlog_last"
#define DMU_POOL_SPARES                 "spares"
#define DMU_POOL_DEFLATE                "deflate"
#define DMU_POOL_HISTORY                "history"
#define DMU_POOL_PROPS                  "pool_props"
#define DMU_POOL_L2CACHE                "l2cache"
#define DMU_POOL_TMP_USERREFS           "tmp_userrefs"
#define DMU_POOL_DDT                    "DDT-%s-%s-%s"
#define DMU_POOL_DDT_STATS              "DDT-statistics"
#define DMU_POOL_CREATION_VERSION       "creation_version"
#define DMU_POOL_SCAN                   "scan"
#define DMU_POOL_FREE_BPOBJ             "free_bpobj"
#define DMU_POOL_BPTREE_OBJ             "bptree_obj"
#define DMU_POOL_EMPTY_BPOBJ            "empty_bpobj"
#define DMU_POOL_CHECKSUM_SALT          "org.illumos:checksum_salt"

/*
 * Allocate an object from this objset.  The range of object numbers
 * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
 *
 * The transaction must be assigned to a txg.  The newly allocated
 * object will be "held" in the transaction (ie. you can modify the
 * newly allocated object in this transaction).
 *
 * dmu_object_alloc() chooses an object and returns it in *objectp.
 *
 * dmu_object_claim() allocates a specific object number.  If that
 * number is already allocated, it fails and returns EEXIST.
 *
 * Return 0 on success, or ENOSPC or EEXIST as specified above.
 */
uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
    int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
    int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
    int blocksize, dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *txp);

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

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

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

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

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

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

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

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

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

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

/*
 * Special spill buffer support used by "SA" framework
 */

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

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

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

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

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

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

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

typedef void dmu_buf_evict_func_t(void *user_ptr);

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

        /*
         * This instance's eviction function pointers.
         *
         * dbu_evict_func_prep is called synchronously while dbu_evict_func
         * is executed asynchronously on a taskq.
         */
        dmu_buf_evict_func_t *dbu_evict_func_prep;
        /* This instance's eviction function pointer. */
        dmu_buf_evict_func_t *dbu_evict_func;
#ifdef ZFS_DEBUG
        /*
         * Pointer to user's dbuf pointer.  NULL for clients that do
         * not associate a dbuf with their user data.
         *
         * The dbuf pointer is cleared upon eviction so as to catch
         * use-after-evict bugs in clients.
         */
        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 {
        uint64_t dds_num_clones; /* number of clones of this */
        uint64_t dds_creation_txg;
        uint64_t dds_guid;
        dmu_objset_type_t dds_type;
        uint8_t dds_is_snapshot;
        uint8_t dds_inconsistent;
        char dds_origin[MAXNAMELEN];
} dmu_objset_stats_t;

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

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

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

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

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

int dmu_objset_is_snapshot(objset_t *os);

extern struct spa *dmu_objset_spa(objset_t *os);
extern struct zilog *dmu_objset_zil(objset_t *os);
extern struct dsl_pool *dmu_objset_pool(objset_t *os);
extern struct dsl_dataset *dmu_objset_ds(objset_t *os);
extern void dmu_objset_name(objset_t *os, char *buf);
extern dmu_objset_type_t dmu_objset_type(objset_t *os);
extern uint64_t dmu_objset_id(objset_t *os);
extern zfs_sync_type_t dmu_objset_syncprop(objset_t *os);
extern zfs_logbias_op_t dmu_objset_logbias(objset_t *os);
extern int dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
    uint64_t *id, uint64_t *offp, boolean_t *case_conflict);
extern int dmu_snapshot_realname(objset_t *os, char *name, char *real,
    int maxlen, boolean_t *conflict);
extern int dmu_dir_list_next(objset_t *os, int namelen, char *name,
    uint64_t *idp, uint64_t *offp);

typedef int objset_used_cb_t(dmu_object_type_t bonustype,
```

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

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

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

/*
 * {zfs,zvol,ztest}_get_done() args
 */
typedef struct zgd {
        struct zilog    *zgd_zilog;
        struct blkptr   *zgd_bp;
        dmu_buf_t       *zgd_db;
        struct rl       *zgd_rl;
        void            *zgd_private;
} zgd_t;

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

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

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

/*
 * Initial setup and final teardown.
 */
extern void dmu_init(void);
extern void dmu_fini(void);

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

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

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

extern int zfs_mdcomp_disable;

#ifdef  __cplusplus
}
#endif

#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_
```

```
void
fzap_upgrade(zap_t *zap, dmu_tx_t *tx, zap_flags_t flags)
{
        dmu_buf_t *db;
        zap_leaf_t *l;
        int i;
        zap_phys_t *zp;

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

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

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

        zp = zap_f_phys(zap);
        /*
         * explicitly zero it since it might be coming from an
         * initialized microzap
         */
        bzero(zap->zap_dbuf->db_data, zap->zap_dbuf->db_size);
        zp->zap_block_type = ZBT_HEADER;
        zp->zap_magic = ZAP_MAGIC;

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

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

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

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

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

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

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

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

```
        if (RW_WRITE_HELD(&zap->zap_rwlock))
                return (1);
        if (rw_tryupgrade(&zap->zap_rwlock)) {
                dmu_buf_will_dirty(zap->zap_dbuf, tx);
                return (1);
        }
        return (0);
}

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

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

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

        if (tbl->zt_nextblk != 0) {
                newblk = tbl->zt_nextblk;
        } else {
                newblk = zap_allocate_blocks(zap, tbl->zt_numblks * 2);
                tbl->zt_nextblk = newblk;
                ASSERT0(tbl->zt_blks_copied);
                dmu_prefetch(zap->zap_objset, zap->zap_object, 0,
                    tbl->zt_blk << bs, tbl->zt_numblks << bs,
                    ZIO_PRIORITY_SYNC_READ);
        }

        /*
         * Copy the ptrtbl from the old to new location.
         */

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

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

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

        dmu_buf_rele(db_old, FTAG);
        tbl->zt_blks_copied++;

        dprintf("copied block %llu of %llu\n",
            tbl->zt_blks_copied, tbl->zt_numblks);

        if (tbl->zt_blks_copied == tbl->zt_numblks) {
                (void) dmu_free_range(zap->zap_objset, zap->zap_object,
                    tbl->zt_blk << bs, tbl->zt_numblks << bs, tx);

                tbl->zt_blk = newblk;
                tbl->zt_numblks *= 2;
                tbl->zt_shift++;
                tbl->zt_nextblk = 0;
                tbl->zt_blks_copied = 0;

                dprintf("finished; numblocks now %llu (%lluk entries)\n",
                    tbl->zt_numblks, 1<<(tbl->zt_shift-10));
        }

        return (0);
}

static int
zap_table_store(zap_t *zap, zap_table_phys_t *tbl, uint64_t idx, uint64_t val,
    dmu_tx_t *tx)
{
        int err;
        uint64_t blk, off;
        int bs = FZAP_BLOCK_SHIFT(zap);
        dmu_buf_t *db;

        ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));
        ASSERT(tbl->zt_blk != 0);

        dprintf("storing %llx at index %llx\n", val, idx);

        blk = idx >> (bs-3);
        off = idx & ((1<<(bs-3))-1);

        err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
            (tbl->zt_blk + blk) << bs, FTAG, &db, DMU_READ_NO_PREFETCH);
        if (err)
                return (err);
        dmu_buf_will_dirty(db, tx);

        if (tbl->zt_nextblk != 0) {
                uint64_t idx2 = idx * 2;
                uint64_t blk2 = idx2 >> (bs-3);
                uint64_t off2 = idx2 & ((1<<(bs-3))-1);
                dmu_buf_t *db2;

                err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
                    (tbl->zt_nextblk + blk2) << bs, FTAG, &db2,
                    DMU_READ_NO_PREFETCH);
                if (err) {
                        dmu_buf_rele(db, FTAG);
                        return (err);
                }
                dmu_buf_will_dirty(db2, tx);
                ((uint64_t *)db2->db_data)[off2] = val;
                ((uint64_t *)db2->db_data)[off2+1] = val;
                dmu_buf_rele(db2, FTAG);
        }

        ((uint64_t *)db->db_data)[off] = val;
```

```
        dmu_buf_rele(db, FTAG);

        return (0);
}

static int
zap_table_load(zap_t *zap, zap_table_phys_t *tbl, uint64_t idx, uint64_t *valp)
{
        uint64_t blk, off;
        int err;
        dmu_buf_t *db;
        int bs = FZAP_BLOCK_SHIFT(zap);

        ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));

        blk = idx >> (bs-3);
        off = idx & ((1<<(bs-3))-1);

        err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
            (tbl->zt_blk + blk) << bs, FTAG, &db, DMU_READ_NO_PREFETCH);
        if (err)
                return (err);
        *valp = ((uint64_t *)db->db_data)[off];
        dmu_buf_rele(db, FTAG);

        if (tbl->zt_nextblk != 0) {
                /*
                 * read the nextblk for the sake of i/o error checking,
                 * so that zap_table_load() will catch errors for
                 * zap_table_store.
                 */
                blk = (idx*2) >> (bs-3);

                err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
                    (tbl->zt_nextblk + blk) << bs, FTAG, &db,
                    DMU_READ_NO_PREFETCH);
                if (err == 0)
                        dmu_buf_rele(db, FTAG);
        }
        return (err);
}

/*
 * Routines for growing the ptrtbl.
 */

static void
zap_ptrtbl_transfer(const uint64_t *src, uint64_t *dst, int n)
{
        int i;
        for (i = 0; i < n; i++) {
                uint64_t lb = src[i];
                dst[2*i+0] = lb;
                dst[2*i+1] = lb;
        }
}

static int
zap_grow_ptrtbl(zap_t *zap, dmu_tx_t *tx)
{
        /*
         * The pointer table should never use more hash bits than we
         * have (otherwise we’d be using useless zero bits to index it).
         * If we are within 2 bits of running out, stop growing, since
         * this is already an aberrant condition.
         */
        if (zap_f_phys(zap)->zap_ptrtbl.zt_shift >= zap_hashbits(zap) - 2)
                return (SET_ERROR(ENOSPC));

        if (zap_f_phys(zap)->zap_ptrtbl.zt_numblks == 0) {
                /*
                 * We are outgrowing the "embedded" ptrtbl (the one
                 * stored in the header block).  Give it its own entire
                 * block, which will double the size of the ptrtbl.
                 */
                uint64_t newblk;
                dmu_buf_t *db_new;
                int err;

                ASSERT3U(zap_f_phys(zap)->zap_ptrtbl.zt_shift, ==,
                    ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
                ASSERT0(zap_f_phys(zap)->zap_ptrtbl.zt_blk);

                newblk = zap_allocate_blocks(zap, 1);
                err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
                    newblk << FZAP_BLOCK_SHIFT(zap), FTAG, &db_new,
                    DMU_READ_NO_PREFETCH);
                if (err)
                        return (err);
                dmu_buf_will_dirty(db_new, tx);
                zap_ptrtbl_transfer(&ZAP_EMBEDDED_PTRTBL_ENT(zap, 0),
                    db_new->db_data, 1 << ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
                dmu_buf_rele(db_new, FTAG);

                zap_f_phys(zap)->zap_ptrtbl.zt_blk = newblk;
                zap_f_phys(zap)->zap_ptrtbl.zt_numblks = 1;
                zap_f_phys(zap)->zap_ptrtbl.zt_shift++;

                ASSERT3U(1ULL << zap_f_phys(zap)->zap_ptrtbl.zt_shift, ==,
                    zap_f_phys(zap)->zap_ptrtbl.zt_numblks <<
                    (FZAP_BLOCK_SHIFT(zap)-3));

                return (0);
        } else {
                return (zap_table_grow(zap, &zap_f_phys(zap)->zap_ptrtbl,
                    zap_ptrtbl_transfer, tx));
        }
}

static void
zap_increment_num_entries(zap_t *zap, int delta, dmu_tx_t *tx)
{
        dmu_buf_will_dirty(zap->zap_dbuf, tx);
        mutex_enter(&zap->zap_f.zap_num_entries_mtx);
        ASSERT(delta > 0 || zap_f_phys(zap)->zap_num_entries >= -delta);
        zap_f_phys(zap)->zap_num_entries += delta;
        mutex_exit(&zap->zap_f.zap_num_entries_mtx);
}

static uint64_t
zap_allocate_blocks(zap_t *zap, int nblocks)
{
        uint64_t newblk;
        ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
        newblk = zap_f_phys(zap)->zap_freeblk;
        zap_f_phys(zap)->zap_freeblk += nblocks;
        return (newblk);
}

static void
zap_leaf_pageout(void *dbu)
{
```

```
        zap_leaf_t *l = dbu;

        rw_destroy(&l->l_rwlock);
        kmem_free(l, sizeof (zap_leaf_t));
}

static zap_leaf_t *
zap_create_leaf(zap_t *zap, dmu_tx_t *tx)
{
        void *winner;
        zap_leaf_t *l = kmem_zalloc(sizeof (zap_leaf_t), KM_SLEEP);

        ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));

        rw_init(&l->l_rwlock, 0, 0, 0);
        rw_enter(&l->l_rwlock, RW_WRITER);
        l->l_blkid = zap_allocate_blocks(zap, 1);
        l->l_dbuf = NULL;

        VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
            l->l_blkid << FZAP_BLOCK_SHIFT(zap), NULL, &l->l_dbuf,
            DMU_READ_NO_PREFETCH));
        dmu_buf_init_user(&l->l_dbu, NULL, zap_leaf_pageout, &l->l_dbuf);
        dmu_buf_init_user(&l->l_dbu, zap_leaf_pageout, &l->l_dbuf);
        winner = dmu_buf_set_user(l->l_dbuf, &l->l_dbu);
        ASSERT(winner == NULL);
        dmu_buf_will_dirty(l->l_dbuf, tx);

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

        zap_f_phys(zap)->zap_num_leafs++;

        return (l);
}
______unchanged_portion_omitted_

static zap_leaf_t *
zap_open_leaf(uint64_t blkid, dmu_buf_t *db)
{
        zap_leaf_t *l, *winner;

        ASSERT(blkid != 0);

        l = kmem_zalloc(sizeof (zap_leaf_t), KM_SLEEP);
        rw_init(&l->l_rwlock, 0, 0, 0);
        rw_enter(&l->l_rwlock, RW_WRITER);
        l->l_blkid = blkid;
        l->l_bs = highbit64(db->db_size) - 1;
        l->l_dbuf = db;

        dmu_buf_init_user(&l->l_dbu, NULL, zap_leaf_pageout, &l->l_dbuf);
        dmu_buf_init_user(&l->l_dbu, zap_leaf_pageout, &l->l_dbuf);
        winner = dmu_buf_set_user(db, &l->l_dbu);

        rw_exit(&l->l_rwlock);
        if (winner != NULL) {
                /* someone else set it first */
                zap_leaf_pageout(&l->l_dbu);
                l = winner;
        }

        /*
         * lhr_pad was previously used for the next leaf in the leaf
         * chain.  There should be no chained leafs (as we have removed
         * support for them).
         */
        ASSERT0(zap_leaf_phys(l)->l_hdr.lh_pad1);

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

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

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

        return (l);
}
```

______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_

static zap_t *
mzap_open(objset_t *os, uint64_t obj, dmu_buf_t *db)
{
        zap_t *winner;
        zap_t *zap;
        int i;

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

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

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

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

        if (winner != NULL) {
                rw_exit(&zap->zap_rwlock);
                rw_destroy(&zap->zap_rwlock);
                if (!zap->zap_ismicro)
                        mutex_destroy(&zap->zap_f.zap_num_entries_mtx);
                kmem_free(zap, sizeof (zap_t));
                return (winner);
        }

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

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

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

```
                        }
                }
        } else {
                zap->zap_salt = zap_f_phys(zap)->zap_salt;
                zap->zap_normflags = zap_f_phys(zap)->zap_normflags;

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

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

                /*
                 * The embedded pointer table should end at the end of
                 * the block
                 */
                ASSERT3U((uintptr_t)&ZAP_EMBEDDED_PTRTBL_ENT(zap,
                    1<<ZAP_EMBEDDED_PTRTBL_SHIFT(zap)) -
                    (uintptr_t)zap_f_phys(zap), ==,
                    zap->zap_dbuf->db_size);
        }
        rw_exit(&zap->zap_rwlock);
        return (zap);
}
______unchanged_portion_omitted_
```