```
**********************************************************
   73461 Wed May  1 11:13:42 2013
new/usr/src/uts/common/fs/zfs/dbuf.c
3756 want lz4 support for metadata compression
**********************************************************
```

```
/*
 * 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) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 Martin Matuska. All rights reserved.
#endif /* ! codereview */
 */

#include <sys/zfs_context.h>
#include <sys/dmu.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>

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);

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

/* 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);
}

/* ARGSUSED */
static void
dbuf_dest(void *vdb, void *unused)
{
        dmu_buf_impl_t *db = vdb;
        mutex_destroy(&db->db_mtx);
        cv_destroy(&db->db_changed);
        refcount_destroy(&db->db_holds);
}

/*
 * dbuf hash table routines
 */
static dbuf_hash_table_t dbuf_hash_table;

static uint64_t dbuf_hash_count;

static uint64_t
dbuf_hash(void *os, uint64_t obj, uint8_t lvl, uint64_t blkid)
{
        uintptr_t osv = (uintptr_t)os;
        uint64_t crc = -1ULL;

        ASSERT(zfs_crc64_table[128] == ZFS_CRC64_POLY);
        crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (lvl)) & 0xFF];
        crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (osv >> 6)) & 0xFF];
        crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (obj >> 0)) & 0xFF];
        crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (obj >> 8)) & 0xFF];
        crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (blkid >> 0)) & 0xFF];
        crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (blkid >> 8)) & 0xFF];

        crc ^= (osv>>14) ^ (obj>>16) ^ (blkid>>16);

        return (crc);
}

#define DBUF_HASH(os, obj, level, blkid) dbuf_hash(os, obj, level, blkid);

#define DBUF_EQUAL(dbuf, os, obj, level, blkid)         \
        ((dbuf)->db.db_object == (obj) &&               \
        (dbuf)->db_objset == (os) &&                    \
        (dbuf)->db_level == (level) &&                  \
        (dbuf)->db_blkid == (blkid))

dmu_buf_impl_t *
dbuf_find(dnode_t *dn, uint8_t level, uint64_t blkid)
{
        dbuf_hash_table_t *h = &dbuf_hash_table;
        objset_t *os = dn->dn_objset;
        uint64_t obj = dn->dn_object;
        uint64_t hv = DBUF_HASH(os, obj, level, blkid);
        uint64_t idx = hv & h->hash_table_mask;
        dmu_buf_impl_t *db;

        mutex_enter(DBUF_HASH_MUTEX(h, idx));
        for (db = h->hash_table[idx]; db != NULL; db = db->db_hash_next) {
                if (DBUF_EQUAL(db, os, obj, level, blkid)) {
                        mutex_enter(&db->db_mtx);
                        if (db->db_state != DB_EVICTING) {
                                mutex_exit(DBUF_HASH_MUTEX(h, idx));
                                return (db);
                        }
                        mutex_exit(&db->db_mtx);
```

```
128                 }
129         }
130         mutex_exit(DBUF_HASH_MUTEX(h, idx));
131         return (NULL);
132 }

133 /*
134  * Insert an entry into the hash table.  If there is already an element
135  * equal to elem in the hash table, then the already existing element
136  * will be returned and the new element will not be inserted.
137  * Otherwise returns NULL.
138  */
139 static dmu_buf_impl_t *
140 dbuf_hash_insert(dmu_buf_impl_t *db)
141 {
142         dbuf_hash_table_t *h = &dbuf_hash_table;
143         objset_t *os = db->db_objset;
144         uint64_t obj = db->db.db_object;
145         int level = db->db_level;
146         uint64_t blkid = db->db_blkid;
147         uint64_t hv = DBUF_HASH(os, obj, level, blkid);
148         uint64_t idx = hv & h->hash_table_mask;
149         dmu_buf_impl_t *dbf;

150         mutex_enter(DBUF_HASH_MUTEX(h, idx));
151         for (dbf = h->hash_table[idx]; dbf != NULL; dbf = dbf->db_hash_next) {
152                 if (DBUF_EQUAL(dbf, os, obj, level, blkid)) {
153                         mutex_enter(&dbf->db_mtx);
154                         if (dbf->db_state != DB_EVICTING) {
155                                 mutex_exit(DBUF_HASH_MUTEX(h, idx));
156                                 return (dbf);
157                         }
158                         mutex_exit(&dbf->db_mtx);
159                 }
160         }

161         mutex_enter(&db->db_mtx);
162         db->db_hash_next = h->hash_table[idx];
163         h->hash_table[idx] = db;
164         mutex_exit(DBUF_HASH_MUTEX(h, idx));
165         atomic_add_64(&dbuf_hash_count, 1);

166         return (NULL);
167 }

168 /*
169  * Remove an entry from the hash table.  This operation will
170  * fail if there are any existing holds on the db.
171  */
172 static void
173 dbuf_hash_remove(dmu_buf_impl_t *db)
174 {
175         dbuf_hash_table_t *h = &dbuf_hash_table;
176         uint64_t hv = DBUF_HASH(db->db_objset, db->db.db_object,
177             db->db_level, db->db_blkid);
178         uint64_t idx = hv & h->hash_table_mask;
179         dmu_buf_impl_t *dbf, **dbp;

180         /*
181          * We musn't hold db_mtx to maintin lock ordering:
182          * DBUF_HASH_MUTEX > db_mtx.
183          */
184         ASSERT(refcount_is_zero(&db->db_holds));
185         ASSERT(db->db_state == DB_EVICTING);
186         ASSERT(!MUTEX_HELD(&db->db_mtx));
```



```
187         mutex_enter(DBUF_HASH_MUTEX(h, idx));
188         dbp = &h->hash_table[idx];
189         while ((dbf = *dbp) != db) {
190                 dbp = &dbf->db_hash_next;
191                 ASSERT(dbf != NULL);
192         }
193         *dbp = db->db_hash_next;
194         db->db_hash_next = NULL;
195         mutex_exit(DBUF_HASH_MUTEX(h, idx));
196         atomic_add_64(&dbuf_hash_count, -1);
197 }

198 static arc_evict_func_t dbuf_do_evict;

199 static void
200 dbuf_evict_user(dmu_buf_impl_t *db)
201 {
202         ASSERT(MUTEX_HELD(&db->db_mtx));

203         if (db->db_level != 0 || db->db_evict_func == NULL)
204                 return;

205         if (db->db_user_data_ptr_ptr)
206                 *db->db_user_data_ptr_ptr = db->db.db_data;
207         db->db_evict_func(&db->db, db->db_user_ptr);
208         db->db_user_ptr = NULL;
209         db->db_user_data_ptr_ptr = NULL;
210         db->db_evict_func = NULL;
211 }

212 boolean_t
213 dbuf_is_metadata(dmu_buf_impl_t *db)
214 {
215         if (db->db_level > 0) {
216                 return (B_TRUE);
217         } else {
218                 boolean_t is_metadata;

219                 DB_DNODE_ENTER(db);
220                 is_metadata = DMU_OT_IS_METADATA(DB_DNODE(db)->dn_type);
221                 DB_DNODE_EXIT(db);

222                 return (is_metadata);
223         }
224 }

225 void
226 dbuf_evict(dmu_buf_impl_t *db)
227 {
228         ASSERT(MUTEX_HELD(&db->db_mtx));
229         ASSERT(db->db_buf == NULL);
230         ASSERT(db->db_data_pending == NULL);

231         dbuf_clear(db);
232         dbuf_destroy(db);
233 }

234 void
235 dbuf_init(void)
236 {
237         uint64_t hsize = 1ULL << 16;
238         dbuf_hash_table_t *h = &dbuf_hash_table;
239         int i;

240         /*
241          * The hash table is big enough to fill all of physical memory
```

```
260          * with an average 4K block size.  The table will take up
261          * totalmem*sizeof(void*)/4K (i.e. 2MB/GB with 8-byte pointers).
262          */
263         while (hsize * 4096 < physmem * PAGESIZE)
264                 hsize <<= 1;

266 retry:
267         h->hash_table_mask = hsize - 1;
268         h->hash_table = kmem_zalloc(hsize * sizeof (void *), KM_NOSLEEP);
269         if (h->hash_table == NULL) {
270                 /* XXX - we should really return an error instead of assert */
271                 ASSERT(hsize > (1ULL << 10));
272                 hsize >>= 1;
273                 goto retry;
274         }

276         dbuf_cache = kmem_cache_create("dmu_buf_impl_t",
277             sizeof (dmu_buf_impl_t),
278             0, dbuf_cons, dbuf_dest, NULL, NULL, 0);

280         for (i = 0; i < DBUF_MUTEXES; i++)
281                 mutex_init(&h->hash_mutexes[i], NULL, MUTEX_DEFAULT, NULL);
282 }

284 void
285 dbuf_fini(void)
286 {
287         dbuf_hash_table_t *h = &dbuf_hash_table;
288         int i;

290         for (i = 0; i < DBUF_MUTEXES; i++)
291                 mutex_destroy(&h->hash_mutexes[i]);
292         kmem_free(h->hash_table, (h->hash_table_mask + 1) * sizeof (void *));
293         kmem_cache_destroy(dbuf_cache);
294 }

296 /*
297  * Other stuff.
298  */

300 #ifdef ZFS_DEBUG
301 static void
302 dbuf_verify(dmu_buf_impl_t *db)
303 {
304         dnode_t *dn;
305         dbuf_dirty_record_t *dr;

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

309         if (!(zfs_flags & ZFS_DEBUG_DBUF_VERIFY))
310                 return;

312         ASSERT(db->db_objset != NULL);
313         DB_DNODE_ENTER(db);
314         dn = DB_DNODE(db);
315         if (dn == NULL) {
316                 ASSERT(db->db_parent == NULL);
317                 ASSERT(db->db_blkptr == NULL);
318         } else {
319                 ASSERT3U(db->db.db_object, ==, dn->dn_object);
320                 ASSERT3P(db->db_objset, ==, dn->dn_objset);
321                 ASSERT3U(db->db_level, <, dn->dn_nlevels);
322                 ASSERT(db->db_blkid == DMU_BONUS_BLKID ||
323                     db->db_blkid == DMU_SPILL_BLKID ||
324                     !list_is_empty(&dn->dn_dbufs));
325         }
```



```
326         if (db->db_blkid == DMU_BONUS_BLKID) {
327                 ASSERT(dn != NULL);
328                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
329                 ASSERT3U(db->db.db_offset, ==, DMU_BONUS_BLKID);
330         } else if (db->db_blkid == DMU_SPILL_BLKID) {
331                 ASSERT(dn != NULL);
332                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
333                 ASSERT0(db->db.db_offset);
334         } else {
335                 ASSERT3U(db->db.db_offset, ==, db->db_blkid * db->db.db_size);
336         }

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

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

344         /*
345          * We can't assert that db_size matches dn_datablksz because it
346          * can be momentarily different when another thread is doing
347          * dnode_set_blksz().
348          */
349         if (db->db_level == 0 && db->db.db_object == DMU_META_DNODE_OBJECT) {
350                 dr = db->db_data_pending;
351                 /*
352                  * It should only be modified in syncing context, so
353                  * make sure we only have one copy of the data.
354                  */
355                 ASSERT(dr == NULL || dr->dt.dl.dr_data == db->db_buf);
356         }

358         /* verify db->db_blkptr */
359         if (db->db_blkptr) {
360                 if (db->db_parent == dn->dn_dbuf) {
361                         /* db is pointed to by the dnode */
362                         /* ASSERT3U(db->db_blkid, <, dn->dn_nblkptr); */
363                         if (DMU_OBJECT_IS_SPECIAL(db->db.db_object))
364                                 ASSERT(db->db_parent == NULL);
365                         else
366                                 ASSERT(db->db_parent != NULL);
367                         if (db->db_blkid != DMU_SPILL_BLKID)
368                                 ASSERT3P(db->db_blkptr, ==,
369                                     &dn->dn_phys->dn_blkptr[db->db_blkid]);
370                 } else {
371                         /* db is pointed to by an indirect block */
372                         int epb = db->db_parent->db.db_size >> SPA_BLKPTRSHIFT;
373                         ASSERT3U(db->db_parent->db_level, ==, db->db_level+1);
374                         ASSERT3U(db->db_parent->db.db_object, ==,
375                             db->db.db_object);
376                         /*
377                          * dnode_grow_indblksz() can make this fail if we don't
378                          * have the struct_rwlock.  XXX indblksz no longer
379                          * grows.  safe to do this now?
380                          */
381                         if (RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
382                                 ASSERT3P(db->db_blkptr, ==,
383                                     ((blkptr_t *)db->db_parent->db.db_data +
384                                     db->db_blkid % epb));
385                         }
386                 }
387         }
388         if ((db->db_blkptr == NULL || BP_IS_HOLE(db->db_blkptr)) &&
389             (db->db_buf == NULL || db->db_buf->b_data) &&
390             db->db.db_data && db->db_blkid != DMU_BONUS_BLKID &&
391             db->db_state != DB_FILL && !dn->dn_free_txg) {
```

```
392                 /*
393                  * If the blkptr isn't set but they have nonzero data,
394                  * it had better be dirty, otherwise we'll lose that
395                  * data when we evict this buffer.
396                  */
397                 if (db->db_dirtycnt == 0) {
398                         uint64_t *buf = db->db.db_data;
399                         int i;

401                         for (i = 0; i < db->db.db_size >> 3; i++) {
402                                 ASSERT(buf[i] == 0);
403                         }
404                 }
405         }
406         DB_DNODE_EXIT(db);
407 }
408 #endif

410 static void
411 dbuf_update_data(dmu_buf_impl_t *db)
412 {
413         ASSERT(MUTEX_HELD(&db->db_mtx));
414         if (db->db_level == 0 && db->db_user_data_ptr_ptr) {
415                 ASSERT(!refcount_is_zero(&db->db_holds));
416                 *db->db_user_data_ptr_ptr = db->db.db_data;
417         }
418 }

420 static void
421 dbuf_set_data(dmu_buf_impl_t *db, arc_buf_t *buf)
422 {
423         ASSERT(MUTEX_HELD(&db->db_mtx));
424         ASSERT(db->db_buf == NULL || !arc_has_callback(db->db_buf));
425         db->db_buf = buf;
426         if (buf != NULL) {
427                 ASSERT(buf->b_data != NULL);
428                 db->db.db_data = buf->b_data;
429                 if (!arc_released(buf))
430                         arc_set_callback(buf, dbuf_do_evict, db);
431                 dbuf_update_data(db);
432         } else {
433                 dbuf_evict_user(db);
434                 db->db.db_data = NULL;
435                 if (db->db_state != DB_NOFILL)
436                         db->db_state = DB_UNCACHED;
437         }
438 }

440 /*
441  * Loan out an arc_buf for read.  Return the loaned arc_buf.
442  */
443 arc_buf_t *
444 dbuf_loan_arcbuf(dmu_buf_impl_t *db)
445 {
446         arc_buf_t *abuf;

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

453                 mutex_exit(&db->db_mtx);
454                 DB_GET_SPA(&spa, db);
455                 abuf = arc_loan_buf(spa, blksz);
456                 bcopy(db->db.db_data, abuf->b_data, blksz);
457         } else {
```



```
458                 abuf = db->db_buf;
459                 arc_loan_inuse_buf(abuf, db);
460                 dbuf_set_data(db, NULL);
461                 mutex_exit(&db->db_mtx);
462         }
463         return (abuf);
464 }

466 uint64_t
467 dbuf_whichblock(dnode_t *dn, uint64_t offset)
468 {
469         if (dn->dn_datablkshift) {
470                 return (offset >> dn->dn_datablkshift);
471         } else {
472                 ASSERT3U(offset, <, dn->dn_datablksz);
473                 return (0);
474         }
475 }

477 static void
478 dbuf_read_done(zio_t *zio, arc_buf_t *buf, void *vdb)
479 {
480         dmu_buf_impl_t *db = vdb;

482         mutex_enter(&db->db_mtx);
483         ASSERT3U(db->db_state, ==, DB_READ);
484         /*
485          * All reads are synchronous, so we must have a hold on the dbuf
486          */
487         ASSERT(refcount_count(&db->db_holds) > 0);
488         ASSERT(db->db_buf == NULL);
489         ASSERT(db->db.db_data == NULL);
490         if (db->db_level == 0 && db->db_freed_in_flight) {
491                 /* we were freed in flight; disregard any error */
492                 arc_release(buf, db);
493                 bzero(buf->b_data, db->db.db_size);
494                 arc_buf_freeze(buf);
495                 db->db_freed_in_flight = FALSE;
496                 dbuf_set_data(db, buf);
497                 db->db_state = DB_CACHED;
498         } else if (zio == NULL || zio->io_error == 0) {
499                 dbuf_set_data(db, buf);
500                 db->db_state = DB_CACHED;
501         } else {
502                 ASSERT(db->db_blkid != DMU_BONUS_BLKID);
503                 ASSERT3P(db->db_buf, ==, NULL);
504                 VERIFY(arc_buf_remove_ref(buf, db));
505                 db->db_state = DB_UNCACHED;
506         }
507         cv_broadcast(&db->db_changed);
508         dbuf_rele_and_unlock(db, NULL);
509 }

511 static void
512 dbuf_read_impl(dmu_buf_impl_t *db, zio_t *zio, uint32_t *flags)
513 {
514         dnode_t *dn;
515         spa_t *spa;
516         zbookmark_t zb;
517         uint32_t aflags = ARC_NOWAIT;

519         DB_DNODE_ENTER(db);
520         dn = DB_DNODE(db);
521         ASSERT(!refcount_is_zero(&db->db_holds));
522         /* We need the struct_rwlock to prevent db_blkptr from changing. */
523         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
```

```
524         ASSERT(MUTEX_HELD(&db->db_mtx));
525         ASSERT(db->db_state == DB_UNCACHED);
526         ASSERT(db->db_buf == NULL);

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

531                 ASSERT3U(bonuslen, <=, db->db.db_size);
532                 db->db.db_data = zio_buf_alloc(DN_MAX_BONUSLEN);
533                 arc_space_consume(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
534                 if (bonuslen < DN_MAX_BONUSLEN)
535                         bzero(db->db.db_data, DN_MAX_BONUSLEN);
536                 if (bonuslen)
537                         bcopy(DN_BONUS(dn->dn_phys), db->db.db_data, bonuslen);
538                 DB_DNODE_EXIT(db);
539                 dbuf_update_data(db);
540                 db->db_state = DB_CACHED;
541                 mutex_exit(&db->db_mtx);
542                 return;
543         }

545         /*
546          * Recheck BP_IS_HOLE() after dnode_block_freed() in case dnode_sync()
547          * processes the delete record and clears the bp while we are waiting
548          * for the dn_mtx (resulting in a "no" from block_freed).
549          */
550         if (db->db_blkptr == NULL || BP_IS_HOLE(db->db_blkptr) ||
551             (db->db_level == 0 && (dnode_block_freed(dn, db->db_blkid) ||
552             BP_IS_HOLE(db->db_blkptr)))) {
553                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);

555                 dbuf_set_data(db, arc_buf_alloc(dn->dn_objset->os_spa,
556                     db->db.db_size, db, type));
557                 DB_DNODE_EXIT(db);
558                 bzero(db->db.db_data, db->db.db_size);
559                 db->db_state = DB_CACHED;
560                 *flags |= DB_RF_CACHED;
561                 mutex_exit(&db->db_mtx);
562                 return;
563         }

565         spa = dn->dn_objset->os_spa;
566         DB_DNODE_EXIT(db);

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

571         if (DBUF_IS_L2CACHEABLE(db))
572                 aflags |= ARC_L2CACHE;

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

578         dbuf_add_ref(db, NULL);

580         (void) arc_read(zio, spa, db->db_blkptr,
581             dbuf_read_done, db, ZIO_PRIORITY_SYNC_READ,
582             (*flags & DB_RF_CANFAIL) ? ZIO_FLAG_CANFAIL : ZIO_FLAG_MUSTSUCCEED,
583             &aflags, &zb);
584         if (aflags & ARC_CACHED)
585                 *flags |= DB_RF_CACHED;
586 }

588 int
589 dbuf_read(dmu_buf_impl_t *db, zio_t *zio, uint32_t flags)
```



```
590 {
591         int err = 0;
592         int havepzio = (zio != NULL);
593         int prefetch;
594         dnode_t *dn;

596         /*
597          * We don't have to hold the mutex to check db_state because it
598          * can't be freed while we have a hold on the buffer.
599          */
600         ASSERT(!refcount_is_zero(&db->db_holds));

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

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

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

614         mutex_enter(&db->db_mtx);
615         if (db->db_state == DB_CACHED) {
616                 mutex_exit(&db->db_mtx);
617                 if (prefetch)
618                         dmu_zfetch(&dn->dn_zfetch, db->db.db_offset,
619                             db->db.db_size, TRUE);
620                 if ((flags & DB_RF_HAVESTRUCT) == 0)
621                         rw_exit(&dn->dn_struct_rwlock);
622                 DB_DNODE_EXIT(db);
623         } else if (db->db_state == DB_UNCACHED) {
624                 spa_t *spa = dn->dn_objset->os_spa;

626                 if (zio == NULL)
627                         zio = zio_root(spa, NULL, NULL, ZIO_FLAG_CANFAIL);
628                 dbuf_read_impl(db, zio, &flags);

630                 /* dbuf_read_impl has dropped db_mtx for us */

632                 if (prefetch)
633                         dmu_zfetch(&dn->dn_zfetch, db->db.db_offset,
634                             db->db.db_size, flags & DB_RF_CACHED);

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

640                 if (!havepzio)
641                         err = zio_wait(zio);
642         } else {
643                 mutex_exit(&db->db_mtx);
644                 if (prefetch)
645                         dmu_zfetch(&dn->dn_zfetch, db->db.db_offset,
646                             db->db.db_size, TRUE);
647                 if ((flags & DB_RF_HAVESTRUCT) == 0)
648                         rw_exit(&dn->dn_struct_rwlock);
649                 DB_DNODE_EXIT(db);

651                 mutex_enter(&db->db_mtx);
652                 if ((flags & DB_RF_NEVERWAIT) == 0) {
653                         while (db->db_state == DB_READ ||
654                             db->db_state == DB_FILL) {
655                                 ASSERT(db->db_state == DB_READ ||
```

```
656                                     (flags & DB_RF_HAVESTRUCT) == 0);
657                                 cv_wait(&db->db_changed, &db->db_mtx);
658                         }
659                         if (db->db_state == DB_UNCACHED)
660                                 err = SET_ERROR(EIO);
661                 }
662                 mutex_exit(&db->db_mtx);
663         }

665         ASSERT(err || havepzio || db->db_state == DB_CACHED);
666         return (err);
667 }

669 static void
670 dbuf_noread(dmu_buf_impl_t *db)
671 {
672         ASSERT(!refcount_is_zero(&db->db_holds));
673         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
674         mutex_enter(&db->db_mtx);
675         while (db->db_state == DB_READ || db->db_state == DB_FILL)
676                 cv_wait(&db->db_changed, &db->db_mtx);
677         if (db->db_state == DB_UNCACHED) {
678                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
679                 spa_t *spa;

681                 ASSERT(db->db_buf == NULL);
682                 ASSERT(db->db.db_data == NULL);
683                 DB_GET_SPA(&spa, db);
684                 dbuf_set_data(db, arc_buf_alloc(spa, db->db.db_size, db, type));
685                 db->db_state = DB_FILL;
686         } else if (db->db_state == DB_NOFILL) {
687                 dbuf_set_data(db, NULL);
688         } else {
689                 ASSERT3U(db->db_state, ==, DB_CACHED);
690         }
691         mutex_exit(&db->db_mtx);
692 }

694 /*
695  * This is our just-in-time copy function.  It makes a copy of
696  * buffers, that have been modified in a previous transaction
697  * group, before we modify them in the current active group.
698  *
699  * This function is used in two places: when we are dirtying a
700  * buffer for the first time in a txg, and when we are freeing
701  * a range in a dnode that includes this buffer.
702  *
703  * Note that when we are called from dbuf_free_range() we do
704  * not put a hold on the buffer, we just traverse the active
705  * dbuf list for the dnode.
706  */
707 static void
708 dbuf_fix_old_data(dmu_buf_impl_t *db, uint64_t txg)
709 {
710         dbuf_dirty_record_t *dr = db->db_last_dirty;

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

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



```
722         /*
723          * If the last dirty record for this dbuf has not yet synced
724          * and its referencing the dbuf data, either:
725          *      reset the reference to point to a new copy,
726          * or (if there a no active holders)
727          *      just null out the current db_data pointer.
728          */
729         ASSERT(dr->dr_txg >= txg - 2);
730         if (db->db_blkid == DMU_BONUS_BLKID) {
731                 /* Note that the data bufs here are zio_bufs */
732                 dr->dt.dl.dr_data = zio_buf_alloc(DN_MAX_BONUSLEN);
733                 arc_space_consume(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
734                 bcopy(db->db.db_data, dr->dt.dl.dr_data, DN_MAX_BONUSLEN);
735         } else if (refcount_count(&db->db_holds) > db->db_dirtycnt) {
736                 int size = db->db.db_size;
737                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
738                 spa_t *spa;

740                 DB_GET_SPA(&spa, db);
741                 dr->dt.dl.dr_data = arc_buf_alloc(spa, size, db, type);
742                 bcopy(db->db.db_data, dr->dt.dl.dr_data->b_data, size);
743         } else {
744                 dbuf_set_data(db, NULL);
745         }
746 }

748 void
749 dbuf_unoverride(dbuf_dirty_record_t *dr)
750 {
751         dmu_buf_impl_t *db = dr->dr_dbuf;
752         blkptr_t *bp = &dr->dt.dl.dr_overridden_by;
753         uint64_t txg = dr->dr_txg;

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

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

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

765         /* free this block */
766         if (!BP_IS_HOLE(bp) && !dr->dt.dl.dr_nopwrite) {
767                 spa_t *spa;

769                 DB_GET_SPA(&spa, db);
770                 zio_free(spa, txg, bp);
771         }
772         dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
773         dr->dt.dl.dr_nopwrite = B_FALSE;

775         /*
776          * Release the already-written buffer, so we leave it in
777          * a consistent dirty state.  Note that all callers are
778          * modifying the buffer, so they will immediately do
779          * another (redundant) arc_release().  Therefore, leave
780          * the buf thawed to save the effort of freezing &
781          * immediately re-thawing it.
782          */
783         arc_release(dr->dt.dl.dr_data, db);
784 }

786 /*
787  * 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.  Also, if we happen accross any level-1 dbufs in the
 * range that have not already been marked dirty, mark them dirty so
 * they stay in memory.
 */
void
dbuf_free_range(dnode_t *dn, uint64_t start, uint64_t end, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db, *db_next;
        uint64_t txg = tx->tx_txg;
        int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
        uint64_t first_l1 = start >> epbs;
        uint64_t last_l1 = end >> epbs;

        if (end > dn->dn_maxblkid && (end != DMU_SPILL_BLKID)) {
                end = dn->dn_maxblkid;
                last_l1 = end >> epbs;
        }
        dprintf_dnode(dn, "start=%llu end=%llu\n", start, end);
        mutex_enter(&dn->dn_dbufs_mtx);
        for (db = list_head(&dn->dn_dbufs); db; db = db_next) {
                db_next = list_next(&dn->dn_dbufs, db);
                ASSERT(db->db_blkid != DMU_BONUS_BLKID);

                if (db->db_level == 1 &&
                    db->db_blkid >= first_l1 && db->db_blkid <= last_l1) {
                        mutex_enter(&db->db_mtx);
                        if (db->db_last_dirty &&
                            db->db_last_dirty->dr_txg < txg) {
                                dbuf_add_ref(db, FTAG);
                                mutex_exit(&db->db_mtx);
                                dbuf_will_dirty(db, tx);
                                dbuf_rele(db, FTAG);
                        } else {
                                mutex_exit(&db->db_mtx);
                        }
                }

                if (db->db_level != 0)
                        continue;
                dprintf_dbuf(db, "found buf %s\n", "");
                if (db->db_blkid < start || db->db_blkid > end)
                        continue;

                /* 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.
         */
        ASSERT(MUTEX_HELD(&db->db_mtx));
        if (db->db_last_dirty)
                birth_txg = db->db_last_dirty->dr_txg;
        else if (db->db_blkptr)
                birth_txg = db->db_blkptr->blk_birth;

        /*
         * If we don't exist or are in a snapshot, we 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)
                return (ds == NULL ||
```

```
920                     dsl_dataset_block_freeable(ds, NULL, birth_txg));
921         else
922                 return (FALSE);
923 }
924
925 void
926 dbuf_new_size(dmu_buf_impl_t *db, int size, dmu_tx_t *tx)
927 {
928         arc_buf_t *buf, *obuf;
929         int osize = db->db.db_size;
930         arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
931         dnode_t *dn;
932
933         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
934
935         DB_DNODE_ENTER(db);
936         dn = DB_DNODE(db);
937
938         /* XXX does *this* func really need the lock? */
939         ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));
940
941         /*
942          * This call to dbuf_will_dirty() with the dn_struct_rwlock held
943          * is OK, because there can be no other references to the db
944          * when we are changing its size, so no concurrent DB_FILL can
945          * be happening.
946          */
947         /*
948          * XXX we should be doing a dbuf_read, checking the return
949          * value and returning that up to our callers
950          */
951         dbuf_will_dirty(db, tx);
952
953         /* create the data buffer for the new block */
954         buf = arc_buf_alloc(dn->dn_objset->os_spa, size, db, type);
955
956         /* copy old block data to the new block */
957         obuf = db->db_buf;
958         bcopy(obuf->b_data, buf->b_data, MIN(osize, size));
959         /* zero the remainder */
960         if (size > osize)
961                 bzero((uint8_t *)buf->b_data + osize, size - osize);
962
963         mutex_enter(&db->db_mtx);
964         dbuf_set_data(db, buf);
965         VERIFY(arc_buf_remove_ref(obuf, db));
966         db->db.db_size = size;
967
968         if (db->db_level == 0) {
969                 ASSERT3U(db->db_last_dirty->dr_txg, ==, tx->tx_txg);
970                 db->db_last_dirty->dt.dl.dr_data = buf;
971         }
972         mutex_exit(&db->db_mtx);
973
974         dnode_willuse_space(dn, size-osize, tx);
975         DB_DNODE_EXIT(db);
976 }
977
978 void
979 dbuf_release_bp(dmu_buf_impl_t *db)
980 {
981         objset_t *os;
982
983         DB_GET_OBJSET(&os, db);
984         ASSERT(dsl_pool_sync_context(dmu_objset_pool(os)));
985         ASSERT(arc_released(os->os_phys_buf) ||
```



```
986             list_link_active(&os->os_dsl_dataset->ds_synced_link));
987         ASSERT(db->db_parent == NULL || arc_released(db->db_parent->db_buf));
988
989         (void) arc_release(db->db_buf, db);
990 }
991
992 dbuf_dirty_record_t *
993 dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
994 {
995         dnode_t *dn;
996         objset_t *os;
997         dbuf_dirty_record_t **drp, *dr;
998         int drop_struct_lock = FALSE;
999         boolean_t do_free_accounting = B_FALSE;
1000        int txgoff = tx->tx_txg & TXG_MASK;
1001
1002        ASSERT(tx->tx_txg != 0);
1003        ASSERT(!refcount_is_zero(&db->db_holds));
1004        DMU_TX_DIRTY_BUF(tx, db);
1005
1006        DB_DNODE_ENTER(db);
1007        dn = DB_DNODE(db);
1008        /*
1009         * Shouldn't dirty a regular buffer in syncing context.  Private
1010         * objects may be dirtied in syncing context, but only if they
1011         * were already pre-dirtied in open context.
1012         */
1013        ASSERT(!dmu_tx_is_syncing(tx) ||
1014            BP_IS_HOLE(dn->dn_objset->os_rootbp) ||
1015            DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1016            dn->dn_objset->os_dsl_dataset == NULL);
1017        /*
1018         * We make this assert for private objects as well, but after we
1019         * check if we're already dirty.  They are allowed to re-dirty
1020         * in syncing context.
1021         */
1022        ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1023            dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1024            (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));
1025
1026        mutex_enter(&db->db_mtx);
1027        /*
1028         * XXX make this true for indirects too?  The problem is that
1029         * transactions created with dmu_tx_create_assigned() from
1030         * syncing context don't bother holding ahead.
1031         */
1032        ASSERT(db->db_level != 0 ||
1033            db->db_state == DB_CACHED || db->db_state == DB_FILL ||
1034            db->db_state == DB_NOFILL);
1035
1036        mutex_enter(&dn->dn_mtx);
1037        /*
1038         * Don't set dirtyctx to SYNC if we're just modifying this as we
1039         * initialize the objset.
1040         */
1041        if (dn->dn_dirtyctx == DN_UNDIRTIED &&
1042            !BP_IS_HOLE(dn->dn_objset->os_rootbp)) {
1043                dn->dn_dirtyctx =
1044                    (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN);
1045                ASSERT(dn->dn_dirtyctx_firstset == NULL);
1046                dn->dn_dirtyctx_firstset = kmem_alloc(1, KM_SLEEP);
1047        }
1048        mutex_exit(&dn->dn_mtx);
1049
1050        if (db->db_blkid == DMU_SPILL_BLKID)
1051                dn->dn_have_spill = B_TRUE;
```

```
1053         /*
1054          * If this buffer is already dirty, we're done.
1055          */
1056         drp = &db->db_last_dirty;
1057         ASSERT(*drp == NULL || (*drp)->dr_txg <= tx->tx_txg ||
1058             db->db.db_object == DMU_META_DNODE_OBJECT);
1059         while ((dr = *drp) != NULL && dr->dr_txg > tx->tx_txg)
1060                 drp = &dr->dr_next;
1061         if (dr && dr->dr_txg == tx->tx_txg) {
1062                 DB_DNODE_EXIT(db);
1063
1064                 if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID) {
1065                         /*
1066                          * If this buffer has already been written out,
1067                          * we now need to reset its state.
1068                          */
1069                         dbuf_unoverride(dr);
1070                         if (db->db.db_object != DMU_META_DNODE_OBJECT &&
1071                             db->db_state != DB_NOFILL)
1072                                 arc_buf_thaw(db->db_buf);
1073                 }
1074                 mutex_exit(&db->db_mtx);
1075                 return (dr);
1076         }
1077
1078         /*
1079          * Only valid if not already dirty.
1080          */
1081         ASSERT(dn->dn_object == 0 ||
1082             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1083             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));
1084
1085         ASSERT3U(dn->dn_nlevels, >, db->db_level);
1086         ASSERT((dn->dn_phys->dn_nlevels == 0 && db->db_level == 0) ||
1087             dn->dn_phys->dn_nlevels > db->db_level ||
1088             dn->dn_next_nlevels[txgoff] > db->db_level ||
1089             dn->dn_next_nlevels[(tx->tx_txg-1) & TXG_MASK] > db->db_level ||
1090             dn->dn_next_nlevels[(tx->tx_txg-2) & TXG_MASK] > db->db_level);
1091
1092         /*
1093          * We should only be dirtying in syncing context if it's the
1094          * mos or we're initializing the os or it's a special object.
1095          * However, we are allowed to dirty in syncing context provided
1096          * we already dirtied it in open context.  Hence we must make
1097          * this assertion only if we're not already dirty.
1098          */
1099         os = dn->dn_objset;
1100         ASSERT(!dmu_tx_is_syncing(tx) || DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1101             os->os_dsl_dataset == NULL || BP_IS_HOLE(os->os_rootbp));
1102         ASSERT(db->db.db_size != 0);
1103
1104         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);
1105
1106         if (db->db_blkid != DMU_BONUS_BLKID) {
1107                 /*
1108                  * Update the accounting.
1109                  * Note: we delay "free accounting" until after we drop
1110                  * the db_mtx.  This keeps us from grabbing other locks
1111                  * (and possibly deadlocking) in bp_get_dsize() while
1112                  * also holding the db_mtx.
1113                  */
1114                 dnode_willuse_space(dn, db->db.db_size, tx);
1115                 do_free_accounting = dbuf_block_freeable(db);
1116         }
```



```
1118         /*
1119          * If this buffer is dirty in an old transaction group we need
1120          * to make a copy of it so that the changes we make in this
1121          * transaction group won't leak out when we sync the older txg.
1122          */
1123         dr = kmem_zalloc(sizeof (dbuf_dirty_record_t), KM_SLEEP);
1124         if (db->db_level == 0) {
1125                 void *data_old = db->db_buf;
1126
1127                 if (db->db_state != DB_NOFILL) {
1128                         if (db->db_blkid == DMU_BONUS_BLKID) {
1129                                 dbuf_fix_old_data(db, tx->tx_txg);
1130                                 data_old = db->db.db_data;
1131                         } else if (db->db.db_object != DMU_META_DNODE_OBJECT) {
1132                                 /*
1133                                  * Release the data buffer from the cache so
1134                                  * that we can modify it without impacting
1135                                  * possible other users of this cached data
1136                                  * block.  Note that indirect blocks and
1137                                  * private objects are not released until the
1138                                  * syncing state (since they are only modified
1139                                  * then).
1140                                  */
1141                                 arc_release(db->db_buf, db);
1142                                 dbuf_fix_old_data(db, tx->tx_txg);
1143                                 data_old = db->db_buf;
1144                         }
1145                         ASSERT(data_old != NULL);
1146                 }
1147                 dr->dt.dl.dr_data = data_old;
1148         } else {
1149                 mutex_init(&dr->dt.di.dr_mtx, NULL, MUTEX_DEFAULT, NULL);
1150                 list_create(&dr->dt.di.dr_children,
1151                     sizeof (dbuf_dirty_record_t),
1152                     offsetof(dbuf_dirty_record_t, dr_dirty_node));
1153         }
1154         dr->dr_dbuf = db;
1155         dr->dr_txg = tx->tx_txg;
1156         dr->dr_next = *drp;
1157         *drp = dr;
1158
1159         /*
1160          * We could have been freed_in_flight between the dbuf_noread
1161          * and dbuf_dirty.  We win, as though the dbuf_noread() had
1162          * happened after the free.
1163          */
1164         if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
1165             db->db_blkid != DMU_SPILL_BLKID) {
1166                 mutex_enter(&dn->dn_mtx);
1167                 dnode_clear_range(dn, db->db_blkid, 1, tx);
1168                 mutex_exit(&dn->dn_mtx);
1169                 db->db_freed_in_flight = FALSE;
1170         }
1171
1172         /*
1173          * This buffer is now part of this txg
1174          */
1175         dbuf_add_ref(db, (void *)(uintptr_t)tx->tx_txg);
1176         db->db_dirtycnt += 1;
1177         ASSERT3U(db->db_dirtycnt, <=, 3);
1178
1179         mutex_exit(&db->db_mtx);
1180
1181         if (db->db_blkid == DMU_BONUS_BLKID ||
1182             db->db_blkid == DMU_SPILL_BLKID) {
1183                 mutex_enter(&dn->dn_mtx);
```

```
1184                ASSERT(!list_link_active(&dr->dr_dirty_node));
1185                list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1186                mutex_exit(&dn->dn_mtx);
1187                dnode_setdirty(dn, tx);
1188                DB_DNODE_EXIT(db);
1189                return (dr);
1190        } else if (do_free_accounting) {
1191                blkptr_t *bp = db->db_blkptr;
1192                int64_t willfree = (bp && !BP_IS_HOLE(bp)) ?
1193                    bp_get_dsize(os->os_spa, bp) : db->db.db_size;
1194                /*
1195                 * This is only a guess -- if the dbuf is dirty
1196                 * in a previous txg, we don't know how much
1197                 * space it will use on disk yet.  We should
1198                 * really have the struct_rwlock to access
1199                 * db_blkptr, but since this is just a guess,
1200                 * it's OK if we get an odd answer.
1201                 */
1202                ddt_prefetch(os->os_spa, bp);
1203                dnode_willuse_space(dn, -willfree, tx);
1204        }

1206        if (!RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
1207                rw_enter(&dn->dn_struct_rwlock, RW_READER);
1208                drop_struct_lock = TRUE;
1209        }

1211        if (db->db_level == 0) {
1212                dnode_new_blkid(dn, db->db_blkid, tx, drop_struct_lock);
1213                ASSERT(dn->dn_maxblkid >= db->db_blkid);
1214        }

1216        if (db->db_level+1 < dn->dn_nlevels) {
1217                dmu_buf_impl_t *parent = db->db_parent;
1218                dbuf_dirty_record_t *di;
1219                int parent_held = FALSE;

1221                if (db->db_parent == NULL || db->db_parent == dn->dn_dbuf) {
1222                        int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;

1224                        parent = dbuf_hold_level(dn, db->db_level+1,
1225                            db->db_blkid >> epbs, FTAG);
1226                        ASSERT(parent != NULL);
1227                        parent_held = TRUE;
1228                }
1229                if (drop_struct_lock)
1230                        rw_exit(&dn->dn_struct_rwlock);
1231                ASSERT3U(db->db_level+1, ==, parent->db_level);
1232                di = dbuf_dirty(parent, tx);
1233                if (parent_held)
1234                        dbuf_rele(parent, FTAG);

1236                mutex_enter(&db->db_mtx);
1237                /*  possible race with dbuf_undirty() */
1238                if (db->db_last_dirty == dr ||
1239                    dn->dn_object == DMU_META_DNODE_OBJECT) {
1240                        mutex_enter(&di->dt.di.dr_mtx);
1241                        ASSERT3U(di->dr_txg, ==, tx->tx_txg);
1242                        ASSERT(!list_link_active(&dr->dr_dirty_node));
1243                        list_insert_tail(&di->dt.di.dr_children, dr);
1244                        mutex_exit(&di->dt.di.dr_mtx);
1245                        dr->dr_parent = di;
1246                }
1247                mutex_exit(&db->db_mtx);
1248        } else {
1249                ASSERT(db->db_level+1 == dn->dn_nlevels);
```



```
1250                ASSERT(db->db_blkid < dn->dn_nblkptr);
1251                ASSERT(db->db_parent == NULL || db->db_parent == dn->dn_dbuf);
1252                mutex_enter(&dn->dn_mtx);
1253                ASSERT(!list_link_active(&dr->dr_dirty_node));
1254                list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1255                mutex_exit(&dn->dn_mtx);
1256                if (drop_struct_lock)
1257                        rw_exit(&dn->dn_struct_rwlock);
1258        }

1260        dnode_setdirty(dn, tx);
1261        DB_DNODE_EXIT(db);
1262        return (dr);
1263 }

1265 /*
1266  * Return TRUE if this evicted the dbuf.
1267  */
1268 static boolean_t
1269 dbuf_undirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1270 {
1271        dnode_t *dn;
1272        uint64_t txg = tx->tx_txg;
1273        dbuf_dirty_record_t *dr, **drp;

1275        ASSERT(txg != 0);
1276        ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1277        ASSERT0(db->db_level);
1278        ASSERT(MUTEX_HELD(&db->db_mtx));

1280        /*
1281         * If this buffer is not dirty, we're done.
1282         */
1283        for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1284                if (dr->dr_txg <= txg)
1285                        break;
1286        if (dr == NULL || dr->dr_txg < txg)
1287                return (B_FALSE);
1288        ASSERT(dr->dr_txg == txg);
1289        ASSERT(dr->dr_dbuf == db);

1291        DB_DNODE_ENTER(db);
1292        dn = DB_DNODE(db);

1294        /*
1295         * Note:  This code will probably work even if there are concurrent
1296         * holders, but it is untested in that scenerio, as the ZPL and
1297         * ztest have additional locking (the range locks) that prevents
1298         * that type of concurrent access.
1299         */
1300        ASSERT3U(refcount_count(&db->db_holds), ==, db->db_dirtycnt);

1302        dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1304        ASSERT(db->db.db_size != 0);

1306        /* XXX would be nice to fix up dn_towrite_space[] */

1308        *drp = dr->dr_next;

1310        /*
1311         * Note that there are three places in dbuf_dirty()
1312         * where this dirty record may be put on a list.
1313         * Make sure to do a list_remove corresponding to
1314         * every one of those list_insert calls.
1315         */
```

```
1316         if (dr->dr_parent) {
1317                 mutex_enter(&dr->dr_parent->dt.di.dr_mtx);
1318                 list_remove(&dr->dr_parent->dt.di.dr_children, dr);
1319                 mutex_exit(&dr->dr_parent->dt.di.dr_mtx);
1320         } else if (db->db_blkid == DMU_SPILL_BLKID ||
1321             db->db_level+1 == dn->dn_nlevels) {
1322                 ASSERT(db->db_blkptr == NULL || db->db_parent == dn->dn_dbuf);
1323                 mutex_enter(&dn->dn_mtx);
1324                 list_remove(&dn->dn_dirty_records[txg & TXG_MASK], dr);
1325                 mutex_exit(&dn->dn_mtx);
1326         }
1327         DB_DNODE_EXIT(db);

1329         if (db->db_state != DB_NOFILL) {
1330                 dbuf_unoverride(dr);

1332                 ASSERT(db->db_buf != NULL);
1333                 ASSERT(dr->dt.dl.dr_data != NULL);
1334                 if (dr->dt.dl.dr_data != db->db_buf)
1335                         VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data, db));
1336         }
1337         kmem_free(dr, sizeof (dbuf_dirty_record_t));

1339         ASSERT(db->db_dirtycnt > 0);
1340         db->db_dirtycnt -= 1;

1342         if (refcount_remove(&db->db_holds, (void *)(uintptr_t)txg) == 0) {
1343                 arc_buf_t *buf = db->db_buf;

1345                 ASSERT(db->db_state == DB_NOFILL || arc_released(buf));
1346                 dbuf_set_data(db, NULL);
1347                 VERIFY(arc_buf_remove_ref(buf, db));
1348                 dbuf_evict(db);
1349                 return (B_TRUE);
1350         }

1352         return (B_FALSE);
1353 }

1355 #pragma weak dmu_buf_will_dirty = dbuf_will_dirty
1356 void
1357 dbuf_will_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1358 {
1359         int rf = DB_RF_MUST_SUCCEED | DB_RF_NOPREFETCH;

1361         ASSERT(tx->tx_txg != 0);
1362         ASSERT(!refcount_is_zero(&db->db_holds));

1364         DB_DNODE_ENTER(db);
1365         if (RW_WRITE_HELD(&DB_DNODE(db)->dn_struct_rwlock))
1366                 rf |= DB_RF_HAVESTRUCT;
1367         DB_DNODE_EXIT(db);
1368         (void) dbuf_read(db, NULL, rf);
1369         (void) dbuf_dirty(db, tx);
1370 }

1372 void
1373 dmu_buf_will_not_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
1374 {
1375         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1377         db->db_state = DB_NOFILL;

1379         dmu_buf_will_fill(db_fake, tx);
1380 }
```



```
1382 void
1383 dmu_buf_will_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
1384 {
1385         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1387         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1388         ASSERT(tx->tx_txg != 0);
1389         ASSERT(db->db_level == 0);
1390         ASSERT(!refcount_is_zero(&db->db_holds));

1392         ASSERT(db->db.db_object != DMU_META_DNODE_OBJECT ||
1393             dmu_tx_private_ok(tx));

1395         dbuf_noread(db);
1396         (void) dbuf_dirty(db, tx);
1397 }

1399 #pragma weak dmu_buf_fill_done = dbuf_fill_done
1400 /* ARGSUSED */
1401 void
1402 dbuf_fill_done(dmu_buf_impl_t *db, dmu_tx_t *tx)
1403 {
1404         mutex_enter(&db->db_mtx);
1405         DBUF_VERIFY(db);

1407         if (db->db_state == DB_FILL) {
1408                 if (db->db_level == 0 && db->db_freed_in_flight) {
1409                         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1410                         /* we were freed while filling */
1411                         /* XXX dbuf_undirty? */
1412                         bzero(db->db.db_data, db->db.db_size);
1413                         db->db_freed_in_flight = FALSE;
1414                 }
1415                 db->db_state = DB_CACHED;
1416                 cv_broadcast(&db->db_changed);
1417         }
1418         mutex_exit(&db->db_mtx);
1419 }

1421 /*
1422  * Directly assign a provided arc buf to a given dbuf if it's not referenced
1423  * by anybody except our caller. Otherwise copy arcbuf's contents to dbuf.
1424  */
1425 void
1426 dbuf_assign_arcbuf(dmu_buf_impl_t *db, arc_buf_t *buf, dmu_tx_t *tx)
1427 {
1428         ASSERT(!refcount_is_zero(&db->db_holds));
1429         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1430         ASSERT(db->db_level == 0);
1431         ASSERT(DBUF_GET_BUFC_TYPE(db) == ARC_BUFC_DATA);
1432         ASSERT(buf != NULL);
1433         ASSERT(arc_buf_size(buf) == db->db.db_size);
1434         ASSERT(tx->tx_txg != 0);

1436         arc_return_buf(buf, db);
1437         ASSERT(arc_released(buf));

1439         mutex_enter(&db->db_mtx);

1441         while (db->db_state == DB_READ || db->db_state == DB_FILL)
1442                 cv_wait(&db->db_changed, &db->db_mtx);

1444         ASSERT(db->db_state == DB_CACHED || db->db_state == DB_UNCACHED);

1446         if (db->db_state == DB_CACHED &&
1447             refcount_count(&db->db_holds) - 1 > db->db_dirtycnt) {
```

```
1448                 mutex_exit(&db->db_mtx);
1449                 (void) dbuf_dirty(db, tx);
1450                 bcopy(buf->b_data, db->db.db_data, db->db.db_size);
1451                 VERIFY(arc_buf_remove_ref(buf, db));
1452                 xuio_stat_wbuf_copied();
1453                 return;
1454         }

1456         xuio_stat_wbuf_nocopy();
1457         if (db->db_state == DB_CACHED) {
1458                 dbuf_dirty_record_t *dr = db->db_last_dirty;

1460                 ASSERT(db->db_buf != NULL);
1461                 if (dr != NULL && dr->dr_txg == tx->tx_txg) {
1462                         ASSERT(dr->dt.dl.dr_data == db->db_buf);
1463                         if (!arc_released(db->db_buf)) {
1464                                 ASSERT(dr->dt.dl.dr_override_state ==
1465                                     DR_OVERRIDDEN);
1466                                 arc_release(db->db_buf, db);
1467                         }
1468                         dr->dt.dl.dr_data = buf;
1469                         VERIFY(arc_buf_remove_ref(db->db_buf, db));
1470                 } else if (dr == NULL || dr->dt.dl.dr_data != db->db_buf) {
1471                         arc_release(db->db_buf, db);
1472                         VERIFY(arc_buf_remove_ref(db->db_buf, db));
1473                 }
1474                 db->db_buf = NULL;
1475         }
1476         ASSERT(db->db_buf == NULL);
1477         dbuf_set_data(db, buf);
1478         db->db_state = DB_FILL;
1479         mutex_exit(&db->db_mtx);
1480         (void) dbuf_dirty(db, tx);
1481         dbuf_fill_done(db, tx);
1482 }

1484 /*
1485  * "Clear" the contents of this dbuf.  This will mark the dbuf
1486  * EVICTING and clear *most* of its references.  Unfortunetely,
1487  * when we are not holding the dn_dbufs_mtx, we can't clear the
1488  * entry in the dn_dbufs list.  We have to wait until dbuf_destroy()
1489  * in this case.  For callers from the DMU we will usually see:
1490  *      dbuf_clear()->arc_buf_evict()->dbuf_do_evict()->dbuf_destroy()
1491  * For the arc callback, we will usually see:
1492  *      dbuf_do_evict()->dbuf_clear();dbuf_destroy()
1493  * Sometimes, though, we will get a mix of these two:
1494  *      DMU: dbuf_clear()->arc_buf_evict()
1495  *      ARC: dbuf_do_evict()->dbuf_destroy()
1496  */
1497 void
1498 dbuf_clear(dmu_buf_impl_t *db)
1499 {
1500         dnode_t *dn;
1501         dmu_buf_impl_t *parent = db->db_parent;
1502         dmu_buf_impl_t *dndb;
1503         int dbuf_gone = FALSE;

1505         ASSERT(MUTEX_HELD(&db->db_mtx));
1506         ASSERT(refcount_is_zero(&db->db_holds));

1508         dbuf_evict_user(db);

1510         if (db->db_state == DB_CACHED) {
1511                 ASSERT(db->db.db_data != NULL);
1512                 if (db->db_blkid == DMU_BONUS_BLKID) {
1513                         zio_buf_free(db->db.db_data, DN_MAX_BONUSLEN);
```



```
1514                         arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
1515                 }
1516                 db->db.db_data = NULL;
1517                 db->db_state = DB_UNCACHED;
1518         }

1520         ASSERT(db->db_state == DB_UNCACHED || db->db_state == DB_NOFILL);
1521         ASSERT(db->db_data_pending == NULL);

1523         db->db_state = DB_EVICTING;
1524         db->db_blkptr = NULL;

1526         DB_DNODE_ENTER(db);
1527         dn = DB_DNODE(db);
1528         dndb = dn->dn_dbuf;
1529         if (db->db_blkid != DMU_BONUS_BLKID && MUTEX_HELD(&dn->dn_dbufs_mtx)) {
1530                 list_remove(&dn->dn_dbufs, db);
1531                 (void) atomic_dec_32_nv(&dn->dn_dbufs_count);
1532                 membar_producer();
1533                 DB_DNODE_EXIT(db);
1534                 /*
1535                  * Decrementing the dbuf count means that the hold corresponding
1536                  * to the removed dbuf is no longer discounted in dnode_move(),
1537                  * so the dnode cannot be moved until after we release the hold.
1538                  * The membar_producer() ensures visibility of the decremented
1539                  * value in dnode_move(), since DB_DNODE_EXIT doesn't actually
1540                  * release any lock.
1541                  */
1542                 dnode_rele(dn, db);
1543                 db->db_dnode_handle = NULL;
1544         } else {
1545                 DB_DNODE_EXIT(db);
1546         }

1548         if (db->db_buf)
1549                 dbuf_gone = arc_buf_evict(db->db_buf);

1551         if (!dbuf_gone)
1552                 mutex_exit(&db->db_mtx);

1554         /*
1555          * If this dbuf is referenced from an indirect dbuf,
1556          * decrement the ref count on the indirect dbuf.
1557          */
1558         if (parent && parent != dndb)
1559                 dbuf_rele(parent, db);
1560 }

1562 static int
1563 dbuf_findbp(dnode_t *dn, int level, uint64_t blkid, int fail_sparse,
1564     dmu_buf_impl_t **parentp, blkptr_t **bpp)
1565 {
1566         int nlevels, epbs;

1568         *parentp = NULL;
1569         *bpp = NULL;

1571         ASSERT(blkid != DMU_BONUS_BLKID);

1573         if (blkid == DMU_SPILL_BLKID) {
1574                 mutex_enter(&dn->dn_mtx);
1575                 if (dn->dn_have_spill &&
1576                     (dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR))
1577                         *bpp = &dn->dn_phys->dn_spill;
1578                 else
1579                         *bpp = NULL;
```

```
1580                dbuf_add_ref(dn->dn_dbuf, NULL);
1581                *parentp = dn->dn_dbuf;
1582                mutex_exit(&dn->dn_mtx);
1583                return (0);
1584        }

1586        if (dn->dn_phys->dn_nlevels == 0)
1587                nlevels = 1;
1588        else
1589                nlevels = dn->dn_phys->dn_nlevels;

1591        epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;

1593        ASSERT3U(level * epbs, <, 64);
1594        ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1595        if (level >= nlevels ||
1596            (blkid > (dn->dn_phys->dn_maxblkid >> (level * epbs)))) {
1597                /* the buffer has no parent yet */
1598                return (SET_ERROR(ENOENT));
1599        } else if (level < nlevels-1) {
1600                /* this block is referenced from an indirect block */
1601                int err = dbuf_hold_impl(dn, level+1,
1602                    blkid >> epbs, fail_sparse, NULL, parentp);
1603                if (err)
1604                        return (err);
1605                err = dbuf_read(*parentp, NULL,
1606                    (DB_RF_HAVESTRUCT | DB_RF_NOPREFETCH | DB_RF_CANFAIL));
1607                if (err) {
1608                        dbuf_rele(*parentp, NULL);
1609                        *parentp = NULL;
1610                        return (err);
1611                }
1612                *bpp = ((blkptr_t *)(*parentp)->db.db_data) +
1613                    (blkid & ((1ULL << epbs) - 1));
1614                return (0);
1615        } else {
1616                /* the block is referenced from the dnode */
1617                ASSERT3U(level, ==, nlevels-1);
1618                ASSERT(dn->dn_phys->dn_nblkptr == 0 ||
1619                    blkid < dn->dn_phys->dn_nblkptr);
1620                if (dn->dn_dbuf) {
1621                        dbuf_add_ref(dn->dn_dbuf, NULL);
1622                        *parentp = dn->dn_dbuf;
1623                }
1624                *bpp = &dn->dn_phys->dn_blkptr[blkid];
1625                return (0);
1626        }
1627 }

1629 static dmu_buf_impl_t *
1630 dbuf_create(dnode_t *dn, uint8_t level, uint64_t blkid,
1631     dmu_buf_impl_t *parent, blkptr_t *blkptr)
1632 {
1633        objset_t *os = dn->dn_objset;
1634        dmu_buf_impl_t *db, *odb;

1636        ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1637        ASSERT(dn->dn_type != DMU_OT_NONE);

1639        db = kmem_cache_alloc(dbuf_cache, KM_SLEEP);

1641        db->db_objset = os;
1642        db->db.db_object = dn->dn_object;
1643        db->db_level = level;
1644        db->db_blkid = blkid;
1645        db->db_last_dirty = NULL;
```



```
1646        db->db_dirtycnt = 0;
1647        db->db_dnode_handle = dn->dn_handle;
1648        db->db_parent = parent;
1649        db->db_blkptr = blkptr;

1651        db->db_user_ptr = NULL;
1652        db->db_user_data_ptr_ptr = NULL;
1653        db->db_evict_func = NULL;
1654        db->db_immediate_evict = 0;
1655        db->db_freed_in_flight = 0;

1657        if (blkid == DMU_BONUS_BLKID) {
1658                ASSERT3P(parent, ==, dn->dn_dbuf);
1659                db->db.db_size = DN_MAX_BONUSLEN -
1660                    (dn->dn_nblkptr-1) * sizeof (blkptr_t);
1661                ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
1662                db->db.db_offset = DMU_BONUS_BLKID;
1663                db->db_state = DB_UNCACHED;
1664                /* the bonus dbuf is not placed in the hash table */
1665                arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1666                return (db);
1667        } else if (blkid == DMU_SPILL_BLKID) {
1668                db->db.db_size = (blkptr != NULL) ?
1669                    BP_GET_LSIZE(blkptr) : SPA_MINBLOCKSIZE;
1670                db->db.db_offset = 0;
1671        } else {
1672                int blocksize =
1673                    db->db_level ? 1<<dn->dn_indblkshift :  dn->dn_datablksz;
1674                db->db.db_size = blocksize;
1675                db->db.db_offset = db->db_blkid * blocksize;
1676        }

1678        /*
1679         * Hold the dn_dbufs_mtx while we get the new dbuf
1680         * in the hash table *and* added to the dbufs list.
1681         * This prevents a possible deadlock with someone
1682         * trying to look up this dbuf before its added to the
1683         * dn_dbufs list.
1684         */
1685        mutex_enter(&dn->dn_dbufs_mtx);
1686        db->db_state = DB_EVICTING;
1687        if ((odb = dbuf_hash_insert(db)) != NULL) {
1688                /* someone else inserted it first */
1689                kmem_cache_free(dbuf_cache, db);
1690                mutex_exit(&dn->dn_dbufs_mtx);
1691                return (odb);
1692        }
1693        list_insert_head(&dn->dn_dbufs, db);
1694        db->db_state = DB_UNCACHED;
1695        mutex_exit(&dn->dn_dbufs_mtx);
1696        arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);

1698        if (parent && parent != dn->dn_dbuf)
1699                dbuf_add_ref(parent, db);

1701        ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1702            refcount_count(&dn->dn_holds) > 0);
1703        (void) refcount_add(&dn->dn_holds, db);
1704        (void) atomic_inc_32_nv(&dn->dn_dbufs_count);

1706        dprintf_dbuf(db, "db=%p\n", db);

1708        return (db);
1709 }

1711 static int
```

```
1712 dbuf_do_evict(void *private)
1713 {
1714         arc_buf_t *buf = private;
1715         dmu_buf_impl_t *db = buf->b_private;
1716
1717         if (!MUTEX_HELD(&db->db_mtx))
1718                 mutex_enter(&db->db_mtx);
1719
1720         ASSERT(refcount_is_zero(&db->db_holds));
1721
1722         if (db->db_state != DB_EVICTING) {
1723                 ASSERT(db->db_state == DB_CACHED);
1724                 DBUF_VERIFY(db);
1725                 db->db_buf = NULL;
1726                 dbuf_evict(db);
1727         } else {
1728                 mutex_exit(&db->db_mtx);
1729                 dbuf_destroy(db);
1730         }
1731         return (0);
1732 }
1733
1734 static void
1735 dbuf_destroy(dmu_buf_impl_t *db)
1736 {
1737         ASSERT(refcount_is_zero(&db->db_holds));
1738
1739         if (db->db_blkid != DMU_BONUS_BLKID) {
1740                 /*
1741                  * If this dbuf is still on the dn_dbufs list,
1742                  * remove it from that list.
1743                  */
1744                 if (db->db_dnode_handle != NULL) {
1745                         dnode_t *dn;
1746
1747                         DB_DNODE_ENTER(db);
1748                         dn = DB_DNODE(db);
1749                         mutex_enter(&dn->dn_dbufs_mtx);
1750                         list_remove(&dn->dn_dbufs, db);
1751                         (void) atomic_dec_32_nv(&dn->dn_dbufs_count);
1752                         mutex_exit(&dn->dn_dbufs_mtx);
1753                         DB_DNODE_EXIT(db);
1754                         /*
1755                          * Decrementing the dbuf count means that the hold
1756                          * corresponding to the removed dbuf is no longer
1757                          * discounted in dnode_move(), so the dnode cannot be
1758                          * moved until after we release the hold.
1759                          */
1760                         dnode_rele(dn, db);
1761                         db->db_dnode_handle = NULL;
1762                 }
1763                 dbuf_hash_remove(db);
1764         }
1765         db->db_parent = NULL;
1766         db->db_buf = NULL;
1767
1768         ASSERT(!list_link_active(&db->db_link));
1769         ASSERT(db->db.db_data == NULL);
1770         ASSERT(db->db_hash_next == NULL);
1771         ASSERT(db->db_blkptr == NULL);
1772         ASSERT(db->db_data_pending == NULL);
1773
1774         kmem_cache_free(dbuf_cache, db);
1775         arc_space_return(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1776 }
```



```
1778 void
1779 dbuf_prefetch(dnode_t *dn, uint64_t blkid)
1780 {
1781         dmu_buf_impl_t *db = NULL;
1782         blkptr_t *bp = NULL;
1783
1784         ASSERT(blkid != DMU_BONUS_BLKID);
1785         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1786
1787         if (dnode_block_freed(dn, blkid))
1788                 return;
1789
1790         /* dbuf_find() returns with db_mtx held */
1791         if (db = dbuf_find(dn, 0, blkid)) {
1792                 /*
1793                  * This dbuf is already in the cache.  We assume that
1794                  * it is already CACHED, or else about to be either
1795                  * read or filled.
1796                  */
1797                 mutex_exit(&db->db_mtx);
1798                 return;
1799         }
1800
1801         if (dbuf_findbp(dn, 0, blkid, TRUE, &db, &bp) == 0) {
1802                 if (bp && !BP_IS_HOLE(bp)) {
1803                         int priority = dn->dn_type == DMU_OT_DDT_ZAP ?
1804                             ZIO_PRIORITY_DDT_PREFETCH : ZIO_PRIORITY_ASYNC_READ;
1805                         dsl_dataset_t *ds = dn->dn_objset->os_dsl_dataset;
1806                         uint32_t aflags = ARC_NOWAIT | ARC_PREFETCH;
1807                         zbookmark_t zb;
1808
1809                         SET_BOOKMARK(&zb, ds ? ds->ds_object : DMU_META_OBJSET,
1810                             dn->dn_object, 0, blkid);
1811
1812                         (void) arc_read(NULL, dn->dn_objset->os_spa,
1813                             bp, NULL, NULL, priority,
1814                             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE,
1815                             &aflags, &zb);
1816                 }
1817                 if (db)
1818                         dbuf_rele(db, NULL);
1819         }
1820 }
1821
1822 /*
1823  * Returns with db_holds incremented, and db_mtx not held.
1824  * Note: dn_struct_rwlock must be held.
1825  */
1826 int
1827 dbuf_hold_impl(dnode_t *dn, uint8_t level, uint64_t blkid, int fail_sparse,
1828     void *tag, dmu_buf_impl_t **dbp)
1829 {
1830         dmu_buf_impl_t *db, *parent = NULL;
1831
1832         ASSERT(blkid != DMU_BONUS_BLKID);
1833         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1834         ASSERT3U(dn->dn_nlevels, >, level);
1835
1836         *dbp = NULL;
1837 top:
1838         /* dbuf_find() returns with db_mtx held */
1839         db = dbuf_find(dn, level, blkid);
1840
1841         if (db == NULL) {
1842                 blkptr_t *bp = NULL;
1843                 int err;
```

```
1845                ASSERT3P(parent, ==, NULL);
1846                err = dbuf_findbp(dn, level, blkid, fail_sparse, &parent, &bp);
1847                if (fail_sparse) {
1848                        if (err == 0 && bp && BP_IS_HOLE(bp))
1849                                err = SET_ERROR(ENOENT);
1850                        if (err) {
1851                                if (parent)
1852                                        dbuf_rele(parent, NULL);
1853                                return (err);
1854                        }
1855                }
1856                if (err && err != ENOENT)
1857                        return (err);
1858                db = dbuf_create(dn, level, blkid, parent, bp);
1859        }

1861        if (db->db_buf && refcount_is_zero(&db->db_holds)) {
1862                arc_buf_add_ref(db->db_buf, db);
1863                if (db->db_buf->b_data == NULL) {
1864                        dbuf_clear(db);
1865                        if (parent) {
1866                                dbuf_rele(parent, NULL);
1867                                parent = NULL;
1868                        }
1869                        goto top;
1870                }
1871                ASSERT3P(db->db.db_data, ==, db->db_buf->b_data);
1872        }

1874        ASSERT(db->db_buf == NULL || arc_referenced(db->db_buf));

1876        /*
1877         * If this buffer is currently syncing out, and we are are
1878         * still referencing it from db_data, we need to make a copy
1879         * of it in case we decide we want to dirty it again in this txg.
1880         */
1881        if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
1882            dn->dn_object != DMU_META_DNODE_OBJECT &&
1883            db->db_state == DB_CACHED && db->db_data_pending) {
1884                dbuf_dirty_record_t *dr = db->db_data_pending;

1886                if (dr->dt.dl.dr_data == db->db_buf) {
1887                        arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);

1889                        dbuf_set_data(db,
1890                            arc_buf_alloc(dn->dn_objset->os_spa,
1891                            db->db.db_size, db, type));
1892                        bcopy(dr->dt.dl.dr_data->b_data, db->db.db_data,
1893                            db->db.db_size);
1894                }
1895        }

1897        (void) refcount_add(&db->db_holds, tag);
1898        dbuf_update_data(db);
1899        DBUF_VERIFY(db);
1900        mutex_exit(&db->db_mtx);

1902        /* NOTE: we can't rele the parent until after we drop the db_mtx */
1903        if (parent)
1904                dbuf_rele(parent, NULL);

1906        ASSERT3P(DB_DNODE(db), ==, dn);
1907        ASSERT3U(db->db_blkid, ==, blkid);
1908        ASSERT3U(db->db_level, ==, level);
1909        *dbp = db;
```



```
1911        return (0);
1912 }

1914 dmu_buf_impl_t *
1915 dbuf_hold(dnode_t *dn, uint64_t blkid, void *tag)
1916 {
1917        dmu_buf_impl_t *db;
1918        int err = dbuf_hold_impl(dn, 0, blkid, FALSE, tag, &db);
1919        return (err ? NULL : db);
1920 }

1922 dmu_buf_impl_t *
1923 dbuf_hold_level(dnode_t *dn, int level, uint64_t blkid, void *tag)
1924 {
1925        dmu_buf_impl_t *db;
1926        int err = dbuf_hold_impl(dn, level, blkid, FALSE, tag, &db);
1927        return (err ? NULL : db);
1928 }

1930 void
1931 dbuf_create_bonus(dnode_t *dn)
1932 {
1933        ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));

1935        ASSERT(dn->dn_bonus == NULL);
1936        dn->dn_bonus = dbuf_create(dn, 0, DMU_BONUS_BLKID, dn->dn_dbuf, NULL);
1937 }

1939 int
1940 dbuf_spill_set_blksz(dmu_buf_t *db_fake, uint64_t blksz, dmu_tx_t *tx)
1941 {
1942        dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
1943        dnode_t *dn;

1945        if (db->db_blkid != DMU_SPILL_BLKID)
1946                return (SET_ERROR(ENOTSUP));
1947        if (blksz == 0)
1948                blksz = SPA_MINBLOCKSIZE;
1949        if (blksz > SPA_MAXBLOCKSIZE)
1950                blksz = SPA_MAXBLOCKSIZE;
1951        else
1952                blksz = P2ROUNDUP(blksz, SPA_MINBLOCKSIZE);

1954        DB_DNODE_ENTER(db);
1955        dn = DB_DNODE(db);
1956        rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
1957        dbuf_new_size(db, blksz, tx);
1958        rw_exit(&dn->dn_struct_rwlock);
1959        DB_DNODE_EXIT(db);

1961        return (0);
1962 }

1964 void
1965 dbuf_rm_spill(dnode_t *dn, dmu_tx_t *tx)
1966 {
1967        dbuf_free_range(dn, DMU_SPILL_BLKID, DMU_SPILL_BLKID, tx);
1968 }

1970 #pragma weak dmu_buf_add_ref = dbuf_add_ref
1971 void
1972 dbuf_add_ref(dmu_buf_impl_t *db, void *tag)
1973 {
1974        int64_t holds = refcount_add(&db->db_holds, tag);
1975        ASSERT(holds > 1);
```

```
1976 }
1978 /*
1979  * If you call dbuf_rele() you had better not be referencing the dnode handle
1980  * unless you have some other direct or indirect hold on the dnode. (An indirect
1981  * hold is a hold on one of the dnode's dbufs, including the bonus buffer.)
1982  * Without that, the dbuf_rele() could lead to a dnode_rele() followed by the
1983  * dnode's parent dbuf evicting its dnode handles.
1984  */
1985 #pragma weak dmu_buf_rele = dbuf_rele
1986 void
1987 dbuf_rele(dmu_buf_impl_t *db, void *tag)
1988 {
1989         mutex_enter(&db->db_mtx);
1990         dbuf_rele_and_unlock(db, tag);
1991 }
1993 /*
1994  * dbuf_rele() for an already-locked dbuf.  This is necessary to allow
1995  * db_dirtycnt and db_holds to be updated atomically.
1996  */
1997 void
1998 dbuf_rele_and_unlock(dmu_buf_impl_t *db, void *tag)
1999 {
2000         int64_t holds;
2002         ASSERT(MUTEX_HELD(&db->db_mtx));
2003         DBUF_VERIFY(db);
2005         /*
2006          * Remove the reference to the dbuf before removing its hold on the
2007          * dnode so we can guarantee in dnode_move() that a referenced bonus
2008          * buffer has a corresponding dnode hold.
2009          */
2010         holds = refcount_remove(&db->db_holds, tag);
2011         ASSERT(holds >= 0);
2013         /*
2014          * We can't freeze indirects if there is a possibility that they
2015          * may be modified in the current syncing context.
2016          */
2017         if (db->db_buf && holds == (db->db_level == 0 ? db->db_dirtycnt : 0))
2018                 arc_buf_freeze(db->db_buf);
2020         if (holds == db->db_dirtycnt &&
2021             db->db_level == 0 && db->db_immediate_evict)
2022                 dbuf_evict_user(db);
2024         if (holds == 0) {
2025                 if (db->db_blkid == DMU_BONUS_BLKID) {
2026                         mutex_exit(&db->db_mtx);
2028                         /*
2029                          * If the dnode moves here, we cannot cross this barrier
2030                          * until the move completes.
2031                          */
2032                         DB_DNODE_ENTER(db);
2033                         (void) atomic_dec_32_nv(&DB_DNODE(db)->dn_dbufs_count);
2034                         DB_DNODE_EXIT(db);
2035                         /*
2036                          * The bonus buffer's dnode hold is no longer discounted
2037                          * in dnode_move(). The dnode cannot move until after
2038                          * the dnode_rele().
2039                          */
2040                         dnode_rele(DB_DNODE(db), db);
2041                 } else if (db->db_buf == NULL) {
```



```
2042                         /*
2043                          * This is a special case: we never associated this
2044                          * dbuf with any data allocated from the ARC.
2045                          */
2046                         ASSERT(db->db_state == DB_UNCACHED ||
2047                             db->db_state == DB_NOFILL);
2048                         dbuf_evict(db);
2049                 } else if (arc_released(db->db_buf)) {
2050                         arc_buf_t *buf = db->db_buf;
2051                         /*
2052                          * This dbuf has anonymous data associated with it.
2053                          */
2054                         dbuf_set_data(db, NULL);
2055                         VERIFY(arc_buf_remove_ref(buf, db));
2056                         dbuf_evict(db);
2057                 } else {
2058                         VERIFY(!arc_buf_remove_ref(db->db_buf, db));
2060                         /*
2061                          * A dbuf will be eligible for eviction if either the
2062                          * 'primarycache' property is set or a duplicate
2063                          * copy of this buffer is already cached in the arc.
2064                          *
2065                          * In the case of the 'primarycache' a buffer
2066                          * is considered for eviction if it matches the
2067                          * criteria set in the property.
2068                          *
2069                          * To decide if our buffer is considered a
2070                          * duplicate, we must call into the arc to determine
2071                          * if multiple buffers are referencing the same
2072                          * block on-disk. If so, then we simply evict
2073                          * ourselves.
2074                          */
2075                         if (!DBUF_IS_CACHEABLE(db) ||
2076                             arc_buf_eviction_needed(db->db_buf))
2077                                 dbuf_clear(db);
2078                         else
2079                                 mutex_exit(&db->db_mtx);
2080                 }
2081         } else {
2082                 mutex_exit(&db->db_mtx);
2083         }
2084 }
2086 #pragma weak dmu_buf_refcount = dbuf_refcount
2087 uint64_t
2088 dbuf_refcount(dmu_buf_impl_t *db)
2089 {
2090         return (refcount_count(&db->db_holds));
2091 }
2093 void *
2094 dmu_buf_set_user(dmu_buf_t *db_fake, void *user_ptr, void *user_data_ptr_ptr,
2095     dmu_buf_evict_func_t *evict_func)
2096 {
2097         return (dmu_buf_update_user(db_fake, NULL, user_ptr,
2098             user_data_ptr_ptr, evict_func));
2099 }
2101 void *
2102 dmu_buf_set_user_ie(dmu_buf_t *db_fake, void *user_ptr, void *user_data_ptr_ptr,
2103     dmu_buf_evict_func_t *evict_func)
2104 {
2105         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
2107         db->db_immediate_evict = TRUE;
```

```
2108         return (dmu_buf_update_user(db_fake, NULL, user_ptr,
2109             user_data_ptr_ptr, evict_func));
2110 }

2112 void *
2113 dmu_buf_update_user(dmu_buf_t *db_fake, void *old_user_ptr, void *user_ptr,
2114     void *user_data_ptr_ptr, dmu_buf_evict_func_t *evict_func)
2115 {
2116         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
2117         ASSERT(db->db_level == 0);

2119         ASSERT((user_ptr == NULL) == (evict_func == NULL));

2121         mutex_enter(&db->db_mtx);

2123         if (db->db_user_ptr == old_user_ptr) {
2124                 db->db_user_ptr = user_ptr;
2125                 db->db_user_data_ptr_ptr = user_data_ptr_ptr;
2126                 db->db_evict_func = evict_func;

2128                 dbuf_update_data(db);
2129         } else {
2130                 old_user_ptr = db->db_user_ptr;
2131         }

2133         mutex_exit(&db->db_mtx);
2134         return (old_user_ptr);
2135 }

2137 void *
2138 dmu_buf_get_user(dmu_buf_t *db_fake)
2139 {
2140         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
2141         ASSERT(!refcount_is_zero(&db->db_holds));

2143         return (db->db_user_ptr);
2144 }

2146 boolean_t
2147 dmu_buf_freeable(dmu_buf_t *dbuf)
2148 {
2149         boolean_t res = B_FALSE;
2150         dmu_buf_impl_t *db = (dmu_buf_impl_t *)dbuf;

2152         if (db->db_blkptr)
2153                 res = dsl_dataset_block_freeable(db->db_objset->os_dsl_dataset,
2154                     db->db_blkptr, db->db_blkptr->blk_birth);

2156         return (res);
2157 }

2159 blkptr_t *
2160 dmu_buf_get_blkptr(dmu_buf_t *db)
2161 {
2162         dmu_buf_impl_t *dbi = (dmu_buf_impl_t *)db;
2163         return (dbi->db_blkptr);
2164 }

2166 static void
2167 dbuf_check_blkptr(dnode_t *dn, dmu_buf_impl_t *db)
2168 {
2169         /* ASSERT(dmu_tx_is_syncing(tx) */
2170         ASSERT(MUTEX_HELD(&db->db_mtx));

2172         if (db->db_blkptr != NULL)
2173                 return;
```



```
2175         if (db->db_blkid == DMU_SPILL_BLKID) {
2176                 db->db_blkptr = &dn->dn_phys->dn_spill;
2177                 BP_ZERO(db->db_blkptr);
2178                 return;
2179         }
2180         if (db->db_level == dn->dn_phys->dn_nlevels-1) {
2181                 /*
2182                  * This buffer was allocated at a time when there was
2183                  * no available blkptrs from the dnode, or it was
2184                  * inappropriate to hook it in (i.e., nlevels mis-match).
2185                  */
2186                 ASSERT(db->db_blkid < dn->dn_phys->dn_nblkptr);
2187                 ASSERT(db->db_parent == NULL);
2188                 db->db_parent = dn->dn_dbuf;
2189                 db->db_blkptr = &dn->dn_phys->dn_blkptr[db->db_blkid];
2190                 DBUF_VERIFY(db);
2191         } else {
2192                 dmu_buf_impl_t *parent = db->db_parent;
2193                 int epbs = dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;

2195                 ASSERT(dn->dn_phys->dn_nlevels > 1);
2196                 if (parent == NULL) {
2197                         mutex_exit(&db->db_mtx);
2198                         rw_enter(&dn->dn_struct_rwlock, RW_READER);
2199                         (void) dbuf_hold_impl(dn, db->db_level+1,
2200                             db->db_blkid >> epbs, FALSE, db, &parent);
2201                         rw_exit(&dn->dn_struct_rwlock);
2202                         mutex_enter(&db->db_mtx);
2203                         db->db_parent = parent;
2204                 }
2205                 db->db_blkptr = (blkptr_t *)parent->db.db_data +
2206                     (db->db_blkid & ((1ULL << epbs) - 1));
2207                 DBUF_VERIFY(db);
2208         }
2209 }

2211 static void
2212 dbuf_sync_indirect(dbuf_dirty_record_t *dr, dmu_tx_t *tx)
2213 {
2214         dmu_buf_impl_t *db = dr->dr_dbuf;
2215         dnode_t *dn;
2216         zio_t *zio;

2218         ASSERT(dmu_tx_is_syncing(tx));

2220         dprintf_dbuf_bp(db, db->db_blkptr, "blkptr=%p", db->db_blkptr);

2222         mutex_enter(&db->db_mtx);

2224         ASSERT(db->db_level > 0);
2225         DBUF_VERIFY(db);

2227         if (db->db_buf == NULL) {
2228                 mutex_exit(&db->db_mtx);
2229                 (void) dbuf_read(db, NULL, DB_RF_MUST_SUCCEED);
2230                 mutex_enter(&db->db_mtx);
2231         }
2232         ASSERT3U(db->db_state, ==, DB_CACHED);
2233         ASSERT(db->db_buf != NULL);

2235         DB_DNODE_ENTER(db);
2236         dn = DB_DNODE(db);
2237         ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2238         dbuf_check_blkptr(dn, db);
2239         DB_DNODE_EXIT(db);
```

```
2241         db->db_data_pending = dr;

2243         mutex_exit(&db->db_mtx);
2244         dbuf_write(dr, db->db_buf, tx);

2246         zio = dr->dr_zio;
2247         mutex_enter(&dr->dt.di.dr_mtx);
2248         dbuf_sync_list(&dr->dt.di.dr_children, tx);
2249         ASSERT(list_head(&dr->dt.di.dr_children) == NULL);
2250         mutex_exit(&dr->dt.di.dr_mtx);
2251         zio_nowait(zio);
2252 }

2254 static void
2255 dbuf_sync_leaf(dbuf_dirty_record_t *dr, dmu_tx_t *tx)
2256 {
2257         arc_buf_t **datap = &dr->dt.dl.dr_data;
2258         dmu_buf_impl_t *db = dr->dr_dbuf;
2259         dnode_t *dn;
2260         objset_t *os;
2261         uint64_t txg = tx->tx_txg;

2263         ASSERT(dmu_tx_is_syncing(tx));

2265         dprintf_dbuf_bp(db, db->db_blkptr, "blkptr=%p", db->db_blkptr);

2267         mutex_enter(&db->db_mtx);
2268         /*
2269          * To be synced, we must be dirtied.  But we
2270          * might have been freed after the dirty.
2271          */
2272         if (db->db_state == DB_UNCACHED) {
2273                 /* This buffer has been freed since it was dirtied */
2274                 ASSERT(db->db.db_data == NULL);
2275         } else if (db->db_state == DB_FILL) {
2276                 /* This buffer was freed and is now being re-filled */
2277                 ASSERT(db->db.db_data != dr->dt.dl.dr_data);
2278         } else {
2279                 ASSERT(db->db_state == DB_CACHED || db->db_state == DB_NOFILL);
2280         }
2281         DBUF_VERIFY(db);

2283         DB_DNODE_ENTER(db);
2284         dn = DB_DNODE(db);

2286         if (db->db_blkid == DMU_SPILL_BLKID) {
2287                 mutex_enter(&dn->dn_mtx);
2288                 dn->dn_phys->dn_flags |= DNODE_FLAG_SPILL_BLKPTR;
2289                 mutex_exit(&dn->dn_mtx);
2290         }

2292         /*
2293          * If this is a bonus buffer, simply copy the bonus data into the
2294          * dnode.  It will be written out when the dnode is synced (and it
2295          * will be synced, since it must have been dirty for dbuf_sync to
2296          * be called).
2297          */
2298         if (db->db_blkid == DMU_BONUS_BLKID) {
2299                 dbuf_dirty_record_t **drp;

2301                 ASSERT(*datap != NULL);
2302                 ASSERT0(db->db_level);
2303                 ASSERT3U(dn->dn_phys->dn_bonuslen, <=, DN_MAX_BONUSLEN);
2304                 bcopy(*datap, DN_BONUS(dn->dn_phys), dn->dn_phys->dn_bonuslen);
2305                 DB_DNODE_EXIT(db);
```



```
2307                 if (*datap != db->db.db_data) {
2308                         zio_buf_free(*datap, DN_MAX_BONUSLEN);
2309                         arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
2310                 }
2311                 db->db_data_pending = NULL;
2312                 drp = &db->db_last_dirty;
2313                 while (*drp != dr)
2314                         drp = &(*drp)->dr_next;
2315                 ASSERT(dr->dr_next == NULL);
2316                 ASSERT(dr->dr_dbuf == db);
2317                 *drp = dr->dr_next;
2318                 kmem_free(dr, sizeof (dbuf_dirty_record_t));
2319                 ASSERT(db->db_dirtycnt > 0);
2320                 db->db_dirtycnt -= 1;
2321                 dbuf_rele_and_unlock(db, (void *)(uintptr_t)txg);
2322                 return;
2323         }

2325         os = dn->dn_objset;

2327         /*
2328          * This function may have dropped the db_mtx lock allowing a dmu_sync
2329          * operation to sneak in. As a result, we need to ensure that we
2330          * don't check the dr_override_state until we have returned from
2331          * dbuf_check_blkptr.
2332          */
2333         dbuf_check_blkptr(dn, db);

2335         /*
2336          * If this buffer is in the middle of an immediate write,
2337          * wait for the synchronous IO to complete.
2338          */
2339         while (dr->dt.dl.dr_override_state == DR_IN_DMU_SYNC) {
2340                 ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
2341                 cv_wait(&db->db_changed, &db->db_mtx);
2342                 ASSERT(dr->dt.dl.dr_override_state != DR_NOT_OVERRIDDEN);
2343         }

2345         if (db->db_state != DB_NOFILL &&
2346             dn->dn_object != DMU_META_DNODE_OBJECT &&
2347             refcount_count(&db->db_holds) > 1 &&
2348             dr->dt.dl.dr_override_state != DR_OVERRIDDEN &&
2349             *datap == db->db_buf) {
2350                 /*
2351                  * If this buffer is currently "in use" (i.e., there
2352                  * are active holds and db_data still references it),
2353                  * then make a copy before we start the write so that
2354                  * any modifications from the open txg will not leak
2355                  * into this write.
2356                  *
2357                  * NOTE: this copy does not need to be made for
2358                  * objects only modified in the syncing context (e.g.
2359                  * DNONE_DNODE blocks).
2360                  */
2361                 int blksz = arc_buf_size(*datap);
2362                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
2363                 *datap = arc_buf_alloc(os->os_spa, blksz, db, type);
2364                 bcopy(db->db.db_data, (*datap)->b_data, blksz);
2365         }
2366         db->db_data_pending = dr;

2368         mutex_exit(&db->db_mtx);

2370         dbuf_write(dr, *datap, tx);
```

```
2372         ASSERT(!list_link_active(&dr->dr_dirty_node));
2373         if (dn->dn_object == DMU_META_DNODE_OBJECT) {
2374                 list_insert_tail(&dn->dn_dirty_records[txg&TXG_MASK], dr);
2375                 DB_DNODE_EXIT(db);
2376         } else {
2377                 /*
2378                  * Although zio_nowait() does not "wait for an IO", it does
2379                  * initiate the IO. If this is an empty write it seems plausible
2380                  * that the IO could actually be completed before the nowait
2381                  * returns. We need to DB_DNODE_EXIT() first in case
2382                  * zio_nowait() invalidates the dbuf.
2383                  */
2384                 DB_DNODE_EXIT(db);
2385                 zio_nowait(dr->dr_zio);
2386         }
2387 }

2389 void
2390 dbuf_sync_list(list_t *list, dmu_tx_t *tx)
2391 {
2392         dbuf_dirty_record_t *dr;

2394         while (dr = list_head(list)) {
2395                 if (dr->dr_zio != NULL) {
2396                         /*
2397                          * If we find an already initialized zio then we
2398                          * are processing the meta-dnode, and we have finished.
2399                          * The dbufs for all dnodes are put back on the list
2400                          * during processing, so that we can zio_wait()
2401                          * these IOs after initiating all child IOs.
2402                          */
2403                         ASSERT3U(dr->dr_dbuf->db.db_object, ==,
2404                             DMU_META_DNODE_OBJECT);
2405                         break;
2406                 }
2407                 list_remove(list, dr);
2408                 if (dr->dr_dbuf->db_level > 0)
2409                         dbuf_sync_indirect(dr, tx);
2410                 else
2411                         dbuf_sync_leaf(dr, tx);
2412         }
2413 }

2415 /* ARGSUSED */
2416 static void
2417 dbuf_write_ready(zio_t *zio, arc_buf_t *buf, void *vdb)
2418 {
2419         dmu_buf_impl_t *db = vdb;
2420         dnode_t *dn;
2421         blkptr_t *bp = zio->io_bp;
2422         blkptr_t *bp_orig = &zio->io_bp_orig;
2423         spa_t *spa = zio->io_spa;
2424         int64_t delta;
2425         uint64_t fill = 0;
2426         int i;

2428         ASSERT(db->db_blkptr == bp);

2430         DB_DNODE_ENTER(db);
2431         dn = DB_DNODE(db);
2432         delta = bp_get_dsize_sync(spa, bp) - bp_get_dsize_sync(spa, bp_orig);
2433         dnode_diduse_space(dn, delta - zio->io_prev_space_delta);
2434         zio->io_prev_space_delta = delta;

2436         if (BP_IS_HOLE(bp)) {
2437                 ASSERT(bp->blk_fill == 0);
```



```
2438                 DB_DNODE_EXIT(db);
2439                 return;
2440         }

2442         ASSERT((db->db_blkid != DMU_SPILL_BLKID &&
2443             BP_GET_TYPE(bp) == dn->dn_type) ||
2444             (db->db_blkid == DMU_SPILL_BLKID &&
2445             BP_GET_TYPE(bp) == dn->dn_bonustype));
2446         ASSERT(BP_GET_LEVEL(bp) == db->db_level);

2448         mutex_enter(&db->db_mtx);

2450 #ifdef ZFS_DEBUG
2451         if (db->db_blkid == DMU_SPILL_BLKID) {
2452                 ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
2453                 ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
2454                     db->db_blkptr == &dn->dn_phys->dn_spill);
2455         }
2456 #endif

2458         if (db->db_level == 0) {
2459                 mutex_enter(&dn->dn_mtx);
2460                 if (db->db_blkid > dn->dn_phys->dn_maxblkid &&
2461                     db->db_blkid != DMU_SPILL_BLKID)
2462                         dn->dn_phys->dn_maxblkid = db->db_blkid;
2463                 mutex_exit(&dn->dn_mtx);

2465                 if (dn->dn_type == DMU_OT_DNODE) {
2466                         dnode_phys_t *dnp = db->db.db_data;
2467                         for (i = db->db.db_size >> DNODE_SHIFT; i > 0;
2468                             i--, dnp++) {
2469                                 if (dnp->dn_type != DMU_OT_NONE)
2470                                         fill++;
2471                         }
2472                 } else {
2473                         fill = 1;
2474                 }
2475         } else {
2476                 blkptr_t *ibp = db->db.db_data;
2477                 ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2478                 for (i = db->db.db_size >> SPA_BLKPTRSHIFT; i > 0; i--, ibp++) {
2479                         if (BP_IS_HOLE(ibp))
2480                                 continue;
2481                         fill += ibp->blk_fill;
2482                 }
2483         }
2484         DB_DNODE_EXIT(db);

2486         bp->blk_fill = fill;

2488         mutex_exit(&db->db_mtx);
2489 }

2491 /* ARGSUSED */
2492 static void
2493 dbuf_write_done(zio_t *zio, arc_buf_t *buf, void *vdb)
2494 {
2495         dmu_buf_impl_t *db = vdb;
2496         blkptr_t *bp = zio->io_bp;
2497         blkptr_t *bp_orig = &zio->io_bp_orig;
2498         uint64_t txg = zio->io_txg;
2499         dbuf_dirty_record_t **drp, *dr;

2501         ASSERT0(zio->io_error);
2502         ASSERT(db->db_blkptr == bp);
```

```
2504         /*
2505          * For nopwrites and rewrites we ensure that the bp matches our
2506          * original and bypass all the accounting.
2507          */
2508         if (zio->io_flags & (ZIO_FLAG_IO_REWRITE | ZIO_FLAG_NOPWRITE)) {
2509                 ASSERT(BP_EQUAL(bp, bp_orig));
2510         } else {
2511                 objset_t *os;
2512                 dsl_dataset_t *ds;
2513                 dmu_tx_t *tx;
2515                 DB_GET_OBJSET(&os, db);
2516                 ds = os->os_dsl_dataset;
2517                 tx = os->os_synctx;
2519                 (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
2520                 dsl_dataset_block_born(ds, bp, tx);
2521         }
2523         mutex_enter(&db->db_mtx);
2525         DBUF_VERIFY(db);
2527         drp = &db->db_last_dirty;
2528         while ((dr = *drp) != db->db_data_pending)
2529                 drp = &dr->dr_next;
2530         ASSERT(!list_link_active(&dr->dr_dirty_node));
2531         ASSERT(dr->dr_txg == txg);
2532         ASSERT(dr->dr_dbuf == db);
2533         ASSERT(dr->dr_next == NULL);
2534         *drp = dr->dr_next;
2536 #ifdef ZFS_DEBUG
2537         if (db->db_blkid == DMU_SPILL_BLKID) {
2538                 dnode_t *dn;
2540                 DB_DNODE_ENTER(db);
2541                 dn = DB_DNODE(db);
2542                 ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
2543                 ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
2544                     db->db_blkptr == &dn->dn_phys->dn_spill);
2545                 DB_DNODE_EXIT(db);
2546         }
2547 #endif
2549         if (db->db_level == 0) {
2550                 ASSERT(db->db_blkid != DMU_BONUS_BLKID);
2551                 ASSERT(dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN);
2552                 if (db->db_state != DB_NOFILL) {
2553                         if (dr->dt.dl.dr_data != db->db_buf)
2554                                 VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data,
2555                                     db));
2556                         else if (!arc_released(db->db_buf))
2557                                 arc_set_callback(db->db_buf, dbuf_do_evict, db);
2558                 }
2559         } else {
2560                 dnode_t *dn;
2562                 DB_DNODE_ENTER(db);
2563                 dn = DB_DNODE(db);
2564                 ASSERT(list_head(&dr->dt.di.dr_children) == NULL);
2565                 ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2566                 if (!BP_IS_HOLE(db->db_blkptr)) {
2567                         int epbs =
2568                             dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;
2569                         ASSERT3U(BP_GET_LSIZE(db->db_blkptr), ==,
```



```
2570                             db->db.db_size);
2571                         ASSERT3U(dn->dn_phys->dn_maxblkid
2572                             >> (db->db_level * epbs), >=, db->db_blkid);
2573                         arc_set_callback(db->db_buf, dbuf_do_evict, db);
2574                 }
2575                 DB_DNODE_EXIT(db);
2576                 mutex_destroy(&dr->dt.di.dr_mtx);
2577                 list_destroy(&dr->dt.di.dr_children);
2578         }
2579         kmem_free(dr, sizeof (dbuf_dirty_record_t));
2581         cv_broadcast(&db->db_changed);
2582         ASSERT(db->db_dirtycnt > 0);
2583         db->db_dirtycnt -= 1;
2584         db->db_data_pending = NULL;
2585         dbuf_rele_and_unlock(db, (void *)(uintptr_t)txg);
2586 }
2588 static void
2589 dbuf_write_nofill_ready(zio_t *zio)
2590 {
2591         dbuf_write_ready(zio, NULL, zio->io_private);
2592 }
2594 static void
2595 dbuf_write_nofill_done(zio_t *zio)
2596 {
2597         dbuf_write_done(zio, NULL, zio->io_private);
2598 }
2600 static void
2601 dbuf_write_override_ready(zio_t *zio)
2602 {
2603         dbuf_dirty_record_t *dr = zio->io_private;
2604         dmu_buf_impl_t *db = dr->dr_dbuf;
2606         dbuf_write_ready(zio, NULL, db);
2607 }
2609 static void
2610 dbuf_write_override_done(zio_t *zio)
2611 {
2612         dbuf_dirty_record_t *dr = zio->io_private;
2613         dmu_buf_impl_t *db = dr->dr_dbuf;
2614         blkptr_t *obp = &dr->dt.dl.dr_overridden_by;
2616         mutex_enter(&db->db_mtx);
2617         if (!BP_EQUAL(zio->io_bp, obp)) {
2618                 if (!BP_IS_HOLE(obp))
2619                         dsl_free(spa_get_dsl(zio->io_spa), zio->io_txg, obp);
2620                 arc_release(dr->dt.dl.dr_data, db);
2621         }
2622         mutex_exit(&db->db_mtx);
2624         dbuf_write_done(zio, NULL, db);
2625 }
2627 static void
2628 dbuf_write(dbuf_dirty_record_t *dr, arc_buf_t *data, dmu_tx_t *tx)
2629 {
2630         dmu_buf_impl_t *db = dr->dr_dbuf;
2631         dnode_t *dn;
2632         objset_t *os;
2633         dmu_buf_impl_t *parent = db->db_parent;
2634         uint64_t txg = tx->tx_txg;
2635         zbookmark_t zb;
```

```
2636         zio_prop_t zp;
2637         zio_t *zio;
2638         int wp_flag = 0;

2640         DB_DNODE_ENTER(db);
2641         dn = DB_DNODE(db);
2642         os = dn->dn_objset;

2644         if (db->db_state != DB_NOFILL) {
2645                 if (db->db_level > 0 || dn->dn_type == DMU_OT_DNODE) {
2646                         /*
2647                          * Private object buffers are released here rather
2648                          * than in dbuf_dirty() since they are only modified
2649                          * in the syncing context and we don't want the
2650                          * overhead of making multiple copies of the data.
2651                          */
2652                         if (BP_IS_HOLE(db->db_blkptr)) {
2653                                 arc_buf_thaw(data);
2654                         } else {
2655                                 dbuf_release_bp(db);
2656                         }
2657                 }
2658         }

2660         if (parent != dn->dn_dbuf) {
2661                 ASSERT(parent && parent->db_data_pending);
2662                 ASSERT(db->db_level == parent->db_level-1);
2663                 ASSERT(arc_released(parent->db_buf));
2664                 zio = parent->db_data_pending->dr_zio;
2665         } else {
2666                 ASSERT((db->db_level == dn->dn_phys->dn_nlevels-1 &&
2667                     db->db_blkid != DMU_SPILL_BLKID) ||
2668                     (db->db_blkid == DMU_SPILL_BLKID && db->db_level == 0));
2669                 if (db->db_blkid != DMU_SPILL_BLKID)
2670                         ASSERT3P(db->db_blkptr, ==,
2671                             &dn->dn_phys->dn_blkptr[db->db_blkid]);
2672                 zio = dn->dn_zio;
2673         }

2675         ASSERT(db->db_level == 0 || data == db->db_buf);
2676         ASSERT3U(db->db_blkptr->blk_birth, <=, txg);
2677         ASSERT(zio);

2679         SET_BOOKMARK(&zb, os->os_dsl_dataset ?
2680             os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
2681             db->db.db_object, db->db_level, db->db_blkid);

2683         if (db->db_blkid == DMU_SPILL_BLKID)
2684                 wp_flag = WP_SPILL;
2685         wp_flag |= (db->db_state == DB_NOFILL) ? WP_NOFILL : 0;

2687         dmu_write_policy(os, dn, db->db_level, wp_flag, &zp, txg);
  25         dmu_write_policy(os, dn, db->db_level, wp_flag, &zp);
2688         DB_DNODE_EXIT(db);

2690         if (db->db_level == 0 && dr->dt.dl.dr_override_state == DR_OVERRIDDEN) {
2691                 ASSERT(db->db_state != DB_NOFILL);
2692                 dr->dr_zio = zio_write(zio, os->os_spa, txg,
2693                     db->db_blkptr, data->b_data, arc_buf_size(data), &zp,
2694                     dbuf_write_override_ready, dbuf_write_override_done, dr,
2695                     ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
2696                 mutex_enter(&db->db_mtx);
2697                 dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
2698                 zio_write_override(dr->dr_zio, &dr->dt.dl.dr_overridden_by,
2699                     dr->dt.dl.dr_copies, dr->dt.dl.dr_nopwrite);
2700                 mutex_exit(&db->db_mtx);
```



```
2701         } else if (db->db_state == DB_NOFILL) {
2702                 ASSERT(zp.zp_checksum == ZIO_CHECKSUM_OFF);
2703                 dr->dr_zio = zio_write(zio, os->os_spa, txg,
2704                     db->db_blkptr, NULL, db->db.db_size, &zp,
2705                     dbuf_write_nofill_ready, dbuf_write_nofill_done, db,
2706                     ZIO_PRIORITY_ASYNC_WRITE,
2707                     ZIO_FLAG_MUSTSUCCEED | ZIO_FLAG_NODATA, &zb);
2708         } else {
2709                 ASSERT(arc_released(data));
2710                 dr->dr_zio = arc_write(zio, os->os_spa, txg,
2711                     db->db_blkptr, data, DBUF_IS_L2CACHEABLE(db), &zp,
2712                     dbuf_write_ready, dbuf_write_done, db,
2713                     ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
2714         }
2715 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   44752 Wed May  1 11:13:43 2013
new/usr/src/uts/common/fs/zfs/dmu.c
3756 want lz4 support for metadata compression
**********************************************************
```

```
/*
 * 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 (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 Martin Matuska. All rights reserved.
#endif /* ! codereview */
 */

#include <sys/dmu.h>
#include <sys/dmu_impl.h>
#include <sys/dmu_tx.h>
#include <sys/dbuf.h>
#include <sys/dnode.h>
#include <sys/zfs_context.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_traverse.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_prop.h>
#include <sys/dmu_zfetch.h>
#include <sys/zfs_ioctl.h>
#include <sys/zap.h>
#include <sys/zio_checksum.h>
#include <sys/zio_compress.h>
#include <sys/sa.h>
#include <sys/zfeature.h>
#endif /* ! codereview */
#ifdef _KERNEL
#include <sys/vmsystm.h>
#include <sys/zfs_znode.h>
#endif

/*
 * Enable/disable nopwrite feature.
 */
int zfs_nopwrite_enabled = 1;

const dmu_object_type_info_t dmu_ot[DMU_OT_NUMTYPES] = {
        {       DMU_BSWAP_UINT8,        TRUE,   "unallocated"           },
        {       DMU_BSWAP_ZAP,          TRUE,   "object directory"      },
```



```
        {       DMU_BSWAP_UINT64,       TRUE,   "object array"          },
        {       DMU_BSWAP_UINT8,        TRUE,   "packed nvlist"         },
        {       DMU_BSWAP_UINT64,       TRUE,   "packed nvlist size"    },
        {       DMU_BSWAP_UINT64,       TRUE,   "bpobj"                 },
        {       DMU_BSWAP_UINT64,       TRUE,   "bpobj header"          },
        {       DMU_BSWAP_UINT64,       TRUE,   "SPA space map header"  },
        {       DMU_BSWAP_UINT64,       TRUE,   "SPA space map"         },
        {       DMU_BSWAP_UINT64,       TRUE,   "ZIL intent log"        },
        {       DMU_BSWAP_DNODE,        TRUE,   "DMU dnode"             },
        {       DMU_BSWAP_OBJSET,       TRUE,   "DMU objset"            },
        {       DMU_BSWAP_UINT64,       TRUE,   "DSL directory"         },
        {       DMU_BSWAP_ZAP,          TRUE,   "DSL directory child map"},
        {       DMU_BSWAP_ZAP,          TRUE,   "DSL dataset snap map"  },
        {       DMU_BSWAP_ZAP,          TRUE,   "DSL props"             },
        {       DMU_BSWAP_UINT64,       TRUE,   "DSL dataset"           },
        {       DMU_BSWAP_ZNODE,        TRUE,   "ZFS znode"             },
        {       DMU_BSWAP_OLDACL,       TRUE,   "ZFS V0 ACL"            },
        {       DMU_BSWAP_UINT8,        FALSE,  "ZFS plain file"        },
        {       DMU_BSWAP_ZAP,          TRUE,   "ZFS directory"         },
        {       DMU_BSWAP_ZAP,          TRUE,   "ZFS master node"       },
        {       DMU_BSWAP_ZAP,          TRUE,   "ZFS delete queue"      },
        {       DMU_BSWAP_UINT8,        FALSE,  "zvol object"           },
        {       DMU_BSWAP_ZAP,          TRUE,   "zvol prop"             },
        {       DMU_BSWAP_UINT8,        FALSE,  "other uint8[]"         },
        {       DMU_BSWAP_UINT64,       FALSE,  "other uint64[]"        },
        {       DMU_BSWAP_ZAP,          TRUE,   "other ZAP"             },
        {       DMU_BSWAP_ZAP,          TRUE,   "persistent error log"  },
        {       DMU_BSWAP_UINT8,        TRUE,   "SPA history"           },
        {       DMU_BSWAP_UINT64,       TRUE,   "SPA history offsets"   },
        {       DMU_BSWAP_ZAP,          TRUE,   "Pool properties"       },
        {       DMU_BSWAP_ZAP,          TRUE,   "DSL permissions"       },
        {       DMU_BSWAP_ACL,          TRUE,   "ZFS ACL"               },
        {       DMU_BSWAP_UINT8,        TRUE,   "ZFS SYSACL"            },
        {       DMU_BSWAP_UINT8,        TRUE,   "FUID table"            },
        {       DMU_BSWAP_UINT64,       TRUE,   "FUID table size"       },
        {       DMU_BSWAP_ZAP,          TRUE,   "DSL dataset next clones"},
        {       DMU_BSWAP_ZAP,          TRUE,   "scan work queue"       },
        {       DMU_BSWAP_ZAP,          TRUE,   "ZFS user/group used"   },
        {       DMU_BSWAP_ZAP,          TRUE,   "ZFS user/group quota"  },
        {       DMU_BSWAP_ZAP,          TRUE,   "snapshot refcount tags"},
        {       DMU_BSWAP_ZAP,          TRUE,   "DDT ZAP algorithm"     },
        {       DMU_BSWAP_ZAP,          TRUE,   "DDT statistics"        },
        {       DMU_BSWAP_UINT8,        TRUE,   "System attributes"     },
        {       DMU_BSWAP_ZAP,          TRUE,   "SA master node"        },
        {       DMU_BSWAP_ZAP,          TRUE,   "SA attr registration"  },
        {       DMU_BSWAP_ZAP,          TRUE,   "SA attr layouts"       },
        {       DMU_BSWAP_ZAP,          TRUE,   "scan translations"     },
        {       DMU_BSWAP_UINT8,        FALSE,  "deduplicated block"    },
        {       DMU_BSWAP_ZAP,          TRUE,   "DSL deadlist map"      },
        {       DMU_BSWAP_UINT64,       TRUE,   "DSL deadlist map hdr"  },
        {       DMU_BSWAP_ZAP,          TRUE,   "DSL dir clones"        },
        {       DMU_BSWAP_UINT64,       TRUE,   "bpobj subobj"          }
};

const dmu_object_byteswap_info_t dmu_ot_byteswap[DMU_BSWAP_NUMFUNCS] = {
        {       byteswap_uint8_array,   "uint8"         },
        {       byteswap_uint16_array,  "uint16"        },
        {       byteswap_uint32_array,  "uint32"        },
        {       byteswap_uint64_array,  "uint64"        },
        {       zap_byteswap,           "zap"           },
        {       dnode_buf_byteswap,     "dnode"         },
        {       dmu_objset_byteswap,    "objset"        },
        {       zfs_znode_byteswap,     "znode"         },
        {       zfs_oldacl_byteswap,    "oldacl"        },
        {       zfs_acl_byteswap,       "acl"           }
};
```

```
129 int
130 dmu_buf_hold(objset_t *os, uint64_t object, uint64_t offset,
131     void *tag, dmu_buf_t **dbp, int flags)
132 {
133         dnode_t *dn;
134         uint64_t blkid;
135         dmu_buf_impl_t *db;
136         int err;
137         int db_flags = DB_RF_CANFAIL;
138
139         if (flags & DMU_READ_NO_PREFETCH)
140                 db_flags |= DB_RF_NOPREFETCH;
141
142         err = dnode_hold(os, object, FTAG, &dn);
143         if (err)
144                 return (err);
145         blkid = dbuf_whichblock(dn, offset);
146         rw_enter(&dn->dn_struct_rwlock, RW_READER);
147         db = dbuf_hold(dn, blkid, tag);
148         rw_exit(&dn->dn_struct_rwlock);
149         if (db == NULL) {
150                 err = SET_ERROR(EIO);
151         } else {
152                 err = dbuf_read(db, NULL, db_flags);
153                 if (err) {
154                         dbuf_rele(db, tag);
155                         db = NULL;
156                 }
157         }
158
159         dnode_rele(dn, FTAG);
160         *dbp = &db->db; /* NULL db plus first field offset is NULL */
161         return (err);
162 }
163
164 int
165 dmu_bonus_max(void)
166 {
167         return (DN_MAX_BONUSLEN);
168 }
169
170 int
171 dmu_set_bonus(dmu_buf_t *db_fake, int newsize, dmu_tx_t *tx)
172 {
173         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
174         dnode_t *dn;
175         int error;
176
177         DB_DNODE_ENTER(db);
178         dn = DB_DNODE(db);
179
180         if (dn->dn_bonus != db) {
181                 error = SET_ERROR(EINVAL);
182         } else if (newsize < 0 || newsize > db_fake->db_size) {
183                 error = SET_ERROR(EINVAL);
184         } else {
185                 dnode_setbonuslen(dn, newsize, tx);
186                 error = 0;
187         }
188
189         DB_DNODE_EXIT(db);
190         return (error);
191 }
192
193 int
```



```
194 dmu_set_bonustype(dmu_buf_t *db_fake, dmu_object_type_t type, dmu_tx_t *tx)
195 {
196         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
197         dnode_t *dn;
198         int error;
199
200         DB_DNODE_ENTER(db);
201         dn = DB_DNODE(db);
202
203         if (!DMU_OT_IS_VALID(type)) {
204                 error = SET_ERROR(EINVAL);
205         } else if (dn->dn_bonus != db) {
206                 error = SET_ERROR(EINVAL);
207         } else {
208                 dnode_setbonus_type(dn, type, tx);
209                 error = 0;
210         }
211
212         DB_DNODE_EXIT(db);
213         return (error);
214 }
215
216 dmu_object_type_t
217 dmu_get_bonustype(dmu_buf_t *db_fake)
218 {
219         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
220         dnode_t *dn;
221         dmu_object_type_t type;
222
223         DB_DNODE_ENTER(db);
224         dn = DB_DNODE(db);
225         type = dn->dn_bonustype;
226         DB_DNODE_EXIT(db);
227
228         return (type);
229 }
230
231 int
232 dmu_rm_spill(objset_t *os, uint64_t object, dmu_tx_t *tx)
233 {
234         dnode_t *dn;
235         int error;
236
237         error = dnode_hold(os, object, FTAG, &dn);
238         dbuf_rm_spill(dn, tx);
239         rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
240         dnode_rm_spill(dn, tx);
241         rw_exit(&dn->dn_struct_rwlock);
242         dnode_rele(dn, FTAG);
243         return (error);
244 }
245
246 /*
247  * returns ENOENT, EIO, or 0.
248  */
249 int
250 dmu_bonus_hold(objset_t *os, uint64_t object, void *tag, dmu_buf_t **dbp)
251 {
252         dnode_t *dn;
253         dmu_buf_impl_t *db;
254         int error;
255
256         error = dnode_hold(os, object, FTAG, &dn);
257         if (error)
258                 return (error);
```

```
260         rw_enter(&dn->dn_struct_rwlock, RW_READER);
261         if (dn->dn_bonus == NULL) {
262                 rw_exit(&dn->dn_struct_rwlock);
263                 rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
264                 if (dn->dn_bonus == NULL)
265                         dbuf_create_bonus(dn);
266         }
267         db = dn->dn_bonus;
268
269         /* as long as the bonus buf is held, the dnode will be held */
270         if (refcount_add(&db->db_holds, tag) == 1) {
271                 VERIFY(dnode_add_ref(dn, db));
272                 (void) atomic_inc_32_nv(&dn->dn_dbufs_count);
273         }
274
275         /*
276          * Wait to drop dn_struct_rwlock until after adding the bonus dbuf's
277          * hold and incrementing the dbuf count to ensure that dnode_move() sees
278          * a dnode hold for every dbuf.
279          */
280         rw_exit(&dn->dn_struct_rwlock);
281
282         dnode_rele(dn, FTAG);
283
284         VERIFY(0 == dbuf_read(db, NULL, DB_RF_MUST_SUCCEED | DB_RF_NOPREFETCH));
285
286         *dbp = &db->db;
287         return (0);
288 }
289
290 /*
291  * returns ENOENT, EIO, or 0.
292  *
293  * This interface will allocate a blank spill dbuf when a spill blk
294  * doesn't already exist on the dnode.
295  *
296  * if you only want to find an already existing spill db, then
297  * dmu_spill_hold_existing() should be used.
298  */
299 int
300 dmu_spill_hold_by_dnode(dnode_t *dn, uint32_t flags, void *tag, dmu_buf_t **dbp)
301 {
302         dmu_buf_impl_t *db = NULL;
303         int err;
304
305         if ((flags & DB_RF_HAVESTRUCT) == 0)
306                 rw_enter(&dn->dn_struct_rwlock, RW_READER);
307
308         db = dbuf_hold(dn, DMU_SPILL_BLKID, tag);
309
310         if ((flags & DB_RF_HAVESTRUCT) == 0)
311                 rw_exit(&dn->dn_struct_rwlock);
312
313         ASSERT(db != NULL);
314         err = dbuf_read(db, NULL, flags);
315         if (err == 0)
316                 *dbp = &db->db;
317         else
318                 dbuf_rele(db, tag);
319         return (err);
320 }
321
322 int
323 dmu_spill_hold_existing(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp)
324 {
325         dmu_buf_impl_t *db = (dmu_buf_impl_t *)bonus;
```



```
326         dnode_t *dn;
327         int err;
328
329         DB_DNODE_ENTER(db);
330         dn = DB_DNODE(db);
331
332         if (spa_version(dn->dn_objset->os_spa) < SPA_VERSION_SA) {
333                 err = SET_ERROR(EINVAL);
334         } else {
335                 rw_enter(&dn->dn_struct_rwlock, RW_READER);
336
337                 if (!dn->dn_have_spill) {
338                         err = SET_ERROR(ENOENT);
339                 } else {
340                         err = dmu_spill_hold_by_dnode(dn,
341                             DB_RF_HAVESTRUCT | DB_RF_CANFAIL, tag, dbp);
342                 }
343
344                 rw_exit(&dn->dn_struct_rwlock);
345         }
346
347         DB_DNODE_EXIT(db);
348         return (err);
349 }
350
351 int
352 dmu_spill_hold_by_bonus(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp)
353 {
354         dmu_buf_impl_t *db = (dmu_buf_impl_t *)bonus;
355         dnode_t *dn;
356         int err;
357
358         DB_DNODE_ENTER(db);
359         dn = DB_DNODE(db);
360         err = dmu_spill_hold_by_dnode(dn, DB_RF_CANFAIL, tag, dbp);
361         DB_DNODE_EXIT(db);
362
363         return (err);
364 }
365
366 /*
367  * Note: longer-term, we should modify all of the dmu_buf_*() interfaces
368  * to take a held dnode rather than <os, object> -- the lookup is wasteful,
369  * and can induce severe lock contention when writing to several files
370  * whose dnodes are in the same block.
371  */
372 static int
373 dmu_buf_hold_array_by_dnode(dnode_t *dn, uint64_t offset, uint64_t length,
374     int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp, uint32_t flags)
375 {
376         dsl_pool_t *dp = NULL;
377         dmu_buf_t **dbp;
378         uint64_t blkid, nblks, i;
379         uint32_t dbuf_flags;
380         int err;
381         zio_t *zio;
382         hrtime_t start;
383
384         ASSERT(length <= DMU_MAX_ACCESS);
385
386         dbuf_flags = DB_RF_CANFAIL | DB_RF_NEVERWAIT | DB_RF_HAVESTRUCT;
387         if (flags & DMU_READ_NO_PREFETCH || length > zfetch_array_rd_sz)
388                 dbuf_flags |= DB_RF_NOPREFETCH;
389
390         rw_enter(&dn->dn_struct_rwlock, RW_READER);
391         if (dn->dn_datablkshift) {
```

```
392                 int blkshift = dn->dn_datablkshift;
393                 nblks = (P2ROUNDUP(offset+length, 1ULL<<blkshift) -
394                     P2ALIGN(offset, 1ULL<<blkshift)) >> blkshift;
395         } else {
396                 if (offset + length > dn->dn_datablksz) {
397                         zfs_panic_recover("zfs: accessing past end of object "
398                             "%llx/%llx (size=%u access=%llu+%llu)",
399                             (longlong_t)dn->dn_objset->
400                             os_dsl_dataset->ds_object,
401                             (longlong_t)dn->dn_object, dn->dn_datablksz,
402                             (longlong_t)offset, (longlong_t)length);
403                         rw_exit(&dn->dn_struct_rwlock);
404                         return (SET_ERROR(EIO));
405                 }
406                 nblks = 1;
407         }
408         dbp = kmem_zalloc(sizeof (dmu_buf_t *) * nblks, KM_SLEEP);
409
410         if (dn->dn_objset->os_dsl_dataset)
411                 dp = dn->dn_objset->os_dsl_dataset->ds_dir->dd_pool;
412         start = gethrtime();
413         zio = zio_root(dn->dn_objset->os_spa, NULL, NULL, ZIO_FLAG_CANFAIL);
414         blkid = dbuf_whichblock(dn, offset);
415         for (i = 0; i < nblks; i++) {
416                 dmu_buf_impl_t *db = dbuf_hold(dn, blkid+i, tag);
417                 if (db == NULL) {
418                         rw_exit(&dn->dn_struct_rwlock);
419                         dmu_buf_rele_array(dbp, nblks, tag);
420                         zio_nowait(zio);
421                         return (SET_ERROR(EIO));
422                 }
423                 /* initiate async i/o */
424                 if (read) {
425                         (void) dbuf_read(db, zio, dbuf_flags);
426                 }
427                 dbp[i] = &db->db;
428         }
429         rw_exit(&dn->dn_struct_rwlock);
430
431         /* wait for async i/o */
432         err = zio_wait(zio);
433         /* track read overhead when we are in sync context */
434         if (dp && dsl_pool_sync_context(dp))
435                 dp->dp_read_overhead += gethrtime() - start;
436         if (err) {
437                 dmu_buf_rele_array(dbp, nblks, tag);
438                 return (err);
439         }
440
441         /* wait for other io to complete */
442         if (read) {
443                 for (i = 0; i < nblks; i++) {
444                         dmu_buf_impl_t *db = (dmu_buf_impl_t *)dbp[i];
445                         mutex_enter(&db->db_mtx);
446                         while (db->db_state == DB_READ ||
447                             db->db_state == DB_FILL)
448                                 cv_wait(&db->db_changed, &db->db_mtx);
449                         if (db->db_state == DB_UNCACHED)
450                                 err = SET_ERROR(EIO);
451                         mutex_exit(&db->db_mtx);
452                         if (err) {
453                                 dmu_buf_rele_array(dbp, nblks, tag);
454                                 return (err);
455                         }
456                 }
457         }
```



```
459         *numbufsp = nblks;
460         *dbpp = dbp;
461         return (0);
462 }
463
464 static int
465 dmu_buf_hold_array(objset_t *os, uint64_t object, uint64_t offset,
466     uint64_t length, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp)
467 {
468         dnode_t *dn;
469         int err;
470
471         err = dnode_hold(os, object, FTAG, &dn);
472         if (err)
473                 return (err);
474
475         err = dmu_buf_hold_array_by_dnode(dn, offset, length, read, tag,
476             numbufsp, dbpp, DMU_READ_PREFETCH);
477
478         dnode_rele(dn, FTAG);
479
480         return (err);
481 }
482
483 int
484 dmu_buf_hold_array_by_bonus(dmu_buf_t *db_fake, uint64_t offset,
485     uint64_t length, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp)
486 {
487         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
488         dnode_t *dn;
489         int err;
490
491         DB_DNODE_ENTER(db);
492         dn = DB_DNODE(db);
493         err = dmu_buf_hold_array_by_dnode(dn, offset, length, read, tag,
494             numbufsp, dbpp, DMU_READ_PREFETCH);
495         DB_DNODE_EXIT(db);
496
497         return (err);
498 }
499
500 void
501 dmu_buf_rele_array(dmu_buf_t **dbp_fake, int numbufs, void *tag)
502 {
503         int i;
504         dmu_buf_impl_t **dbp = (dmu_buf_impl_t **)dbp_fake;
505
506         if (numbufs == 0)
507                 return;
508
509         for (i = 0; i < numbufs; i++) {
510                 if (dbp[i])
511                         dbuf_rele(dbp[i], tag);
512         }
513
514         kmem_free(dbp, sizeof (dmu_buf_t *) * numbufs);
515 }
516
517 void
518 dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset, uint64_t len)
519 {
520         dnode_t *dn;
521         uint64_t blkid;
522         int nblks, i, err;
```

```
524         if (zfs_prefetch_disable)
525                 return;
526
527         if (len == 0) {  /* they're interested in the bonus buffer */
528                 dn = DMU_META_DNODE(os);
529
530                 if (object == 0 || object >= DN_MAX_OBJECT)
531                         return;
532
533                 rw_enter(&dn->dn_struct_rwlock, RW_READER);
534                 blkid = dbuf_whichblock(dn, object * sizeof (dnode_phys_t));
535                 dbuf_prefetch(dn, blkid);
536                 rw_exit(&dn->dn_struct_rwlock);
537                 return;
538         }
539
540         /*
541          * XXX - Note, if the dnode for the requested object is not
542          * already cached, we will do a *synchronous* read in the
543          * dnode_hold() call.  The same is true for any indirects.
544          */
545         err = dnode_hold(os, object, FTAG, &dn);
546         if (err != 0)
547                 return;
548
549         rw_enter(&dn->dn_struct_rwlock, RW_READER);
550         if (dn->dn_datablkshift) {
551                 int blkshift = dn->dn_datablkshift;
552                 nblks = (P2ROUNDUP(offset+len, 1<<blkshift) -
553                     P2ALIGN(offset, 1<<blkshift)) >> blkshift;
554         } else {
555                 nblks = (offset < dn->dn_datablksz);
556         }
557
558         if (nblks != 0) {
559                 blkid = dbuf_whichblock(dn, offset);
560                 for (i = 0; i < nblks; i++)
561                         dbuf_prefetch(dn, blkid+i);
562         }
563
564         rw_exit(&dn->dn_struct_rwlock);
565
566         dnode_rele(dn, FTAG);
567 }
568
569 /*
570  * Get the next "chunk" of file data to free.  We traverse the file from
571  * the end so that the file gets shorter over time (if we crashes in the
572  * middle, this will leave us in a better state).  We find allocated file
573  * data by simply searching the allocated level 1 indirects.
574  */
575 static int
576 get_next_chunk(dnode_t *dn, uint64_t *start, uint64_t limit)
577 {
578         uint64_t len = *start - limit;
579         uint64_t blkcnt = 0;
580         uint64_t maxblks = DMU_MAX_ACCESS / (1ULL << (dn->dn_indblkshift + 1));
581         uint64_t iblkrange =
582             dn->dn_datablksz * EPB(dn->dn_indblkshift, SPA_BLKPTRSHIFT);
583
584         ASSERT(limit <= *start);
585
586         if (len <= iblkrange * maxblks) {
587                 *start = limit;
588                 return (0);
589         }
```



```
590         ASSERT(ISP2(iblkrange));
591
592         while (*start > limit && blkcnt < maxblks) {
593                 int err;
594
595                 /* find next allocated L1 indirect */
596                 err = dnode_next_offset(dn,
597                     DNODE_FIND_BACKWARDS, start, 2, 1, 0);
598
599                 /* if there are no more, then we are done */
600                 if (err == ESRCH) {
601                         *start = limit;
602                         return (0);
603                 } else if (err) {
604                         return (err);
605                 }
606                 blkcnt += 1;
607
608                 /* reset offset to end of "next" block back */
609                 *start = P2ALIGN(*start, iblkrange);
610                 if (*start <= limit)
611                         *start = limit;
612                 else
613                         *start -= 1;
614         }
615         return (0);
616 }
617
618 static int
619 dmu_free_long_range_impl(objset_t *os, dnode_t *dn, uint64_t offset,
620     uint64_t length, boolean_t free_dnode)
621 {
622         dmu_tx_t *tx;
623         uint64_t object_size, start, end, len;
624         boolean_t trunc = (length == DMU_OBJECT_END);
625         int align, err;
626
627         align = 1 << dn->dn_datablkshift;
628         ASSERT(align > 0);
629         object_size = align == 1 ? dn->dn_datablksz :
630             (dn->dn_maxblkid + 1) << dn->dn_datablkshift;
631
632         end = offset + length;
633         if (trunc || end > object_size)
634                 end = object_size;
635         if (end <= offset)
636                 return (0);
637         length = end - offset;
638
639         while (length) {
640                 start = end;
641                 /* assert(offset <= start) */
642                 err = get_next_chunk(dn, &start, offset);
643                 if (err)
644                         return (err);
645                 len = trunc ? DMU_OBJECT_END : end - start;
646
647                 tx = dmu_tx_create(os);
648                 dmu_tx_hold_free(tx, dn->dn_object, start, len);
649                 err = dmu_tx_assign(tx, TXG_WAIT);
650                 if (err) {
651                         dmu_tx_abort(tx);
652                         return (err);
653                 }
654
655                 dnode_free_range(dn, start, trunc ? -1 : len, tx);
```

```
657                 if (start == 0 && free_dnode) {
658                         ASSERT(trunc);
659                         dnode_free(dn, tx);
660                 }

662                 length -= end - start;

664                 dmu_tx_commit(tx);
665                 end = start;
666         }
667         return (0);
668 }

670 int
671 dmu_free_long_range(objset_t *os, uint64_t object,
672     uint64_t offset, uint64_t length)
673 {
674         dnode_t *dn;
675         int err;

677         err = dnode_hold(os, object, FTAG, &dn);
678         if (err != 0)
679                 return (err);
680         err = dmu_free_long_range_impl(os, dn, offset, length, FALSE);
681         dnode_rele(dn, FTAG);
682         return (err);
683 }

685 int
686 dmu_free_object(objset_t *os, uint64_t object)
687 {
688         dnode_t *dn;
689         dmu_tx_t *tx;
690         int err;

692         err = dnode_hold_impl(os, object, DNODE_MUST_BE_ALLOCATED,
693             FTAG, &dn);
694         if (err != 0)
695                 return (err);
696         if (dn->dn_nlevels == 1) {
697                 tx = dmu_tx_create(os);
698                 dmu_tx_hold_bonus(tx, object);
699                 dmu_tx_hold_free(tx, dn->dn_object, 0, DMU_OBJECT_END);
700                 err = dmu_tx_assign(tx, TXG_WAIT);
701                 if (err == 0) {
702                         dnode_free_range(dn, 0, DMU_OBJECT_END, tx);
703                         dnode_free(dn, tx);
704                         dmu_tx_commit(tx);
705                 } else {
706                         dmu_tx_abort(tx);
707                 }
708         } else {
709                 err = dmu_free_long_range_impl(os, dn, 0, DMU_OBJECT_END, TRUE);
710         }
711         dnode_rele(dn, FTAG);
712         return (err);
713 }

715 int
716 dmu_free_range(objset_t *os, uint64_t object, uint64_t offset,
717     uint64_t size, dmu_tx_t *tx)
718 {
719         dnode_t *dn;
720         int err = dnode_hold(os, object, FTAG, &dn);
721         if (err)
```



```
722                 return (err);
723         ASSERT(offset < UINT64_MAX);
724         ASSERT(size == -1ULL || size <= UINT64_MAX - offset);
725         dnode_free_range(dn, offset, size, tx);
726         dnode_rele(dn, FTAG);
727         return (0);
728 }

730 int
731 dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
732     void *buf, uint32_t flags)
733 {
734         dnode_t *dn;
735         dmu_buf_t **dbp;
736         int numbufs, err;

738         err = dnode_hold(os, object, FTAG, &dn);
739         if (err)
740                 return (err);

742         /*
743          * Deal with odd block sizes, where there can't be data past the first
744          * block.  If we ever do the tail block optimization, we will need to
745          * handle that here as well.
746          */
747         if (dn->dn_maxblkid == 0) {
748                 int newsz = offset > dn->dn_datablksz ? 0 :
749                     MIN(size, dn->dn_datablksz - offset);
750                 bzero((char *)buf + newsz, size - newsz);
751                 size = newsz;
752         }

754         while (size > 0) {
755                 uint64_t mylen = MIN(size, DMU_MAX_ACCESS / 2);
756                 int i;

758                 /*
759                  * NB: we could do this block-at-a-time, but it's nice
760                  * to be reading in parallel.
761                  */
762                 err = dmu_buf_hold_array_by_dnode(dn, offset, mylen,
763                     TRUE, FTAG, &numbufs, &dbp, flags);
764                 if (err)
765                         break;

767                 for (i = 0; i < numbufs; i++) {
768                         int tocpy;
769                         int bufoff;
770                         dmu_buf_t *db = dbp[i];

772                         ASSERT(size > 0);

774                         bufoff = offset - db->db_offset;
775                         tocpy = (int)MIN(db->db_size - bufoff, size);

777                         bcopy((char *)db->db_data + bufoff, buf, tocpy);

779                         offset += tocpy;
780                         size -= tocpy;
781                         buf = (char *)buf + tocpy;
782                 }
783                 dmu_buf_rele_array(dbp, numbufs, FTAG);
784         }
785         dnode_rele(dn, FTAG);
786         return (err);
787 }
```

```
789 void
790 dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
791     const void *buf, dmu_tx_t *tx)
792 {
793         dmu_buf_t **dbp;
794         int numbufs, i;

796         if (size == 0)
797                 return;

799         VERIFY(0 == dmu_buf_hold_array(os, object, offset, size,
800             FALSE, FTAG, &numbufs, &dbp));

802         for (i = 0; i < numbufs; i++) {
803                 int tocpy;
804                 int bufoff;
805                 dmu_buf_t *db = dbp[i];

807                 ASSERT(size > 0);

809                 bufoff = offset - db->db_offset;
810                 tocpy = (int)MIN(db->db_size - bufoff, size);

812                 ASSERT(i == 0 || i == numbufs-1 || tocpy == db->db_size);

814                 if (tocpy == db->db_size)
815                         dmu_buf_will_fill(db, tx);
816                 else
817                         dmu_buf_will_dirty(db, tx);

819                 bcopy(buf, (char *)db->db_data + bufoff, tocpy);

821                 if (tocpy == db->db_size)
822                         dmu_buf_fill_done(db, tx);

824                 offset += tocpy;
825                 size -= tocpy;
826                 buf = (char *)buf + tocpy;
827         }
828         dmu_buf_rele_array(dbp, numbufs, FTAG);
829 }

831 void
832 dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
833     dmu_tx_t *tx)
834 {
835         dmu_buf_t **dbp;
836         int numbufs, i;

838         if (size == 0)
839                 return;

841         VERIFY(0 == dmu_buf_hold_array(os, object, offset, size,
842             FALSE, FTAG, &numbufs, &dbp));

844         for (i = 0; i < numbufs; i++) {
845                 dmu_buf_t *db = dbp[i];

847                 dmu_buf_will_not_fill(db, tx);
848         }
849         dmu_buf_rele_array(dbp, numbufs, FTAG);
850 }

852 /*
853  * DMU support for xuio
```



```
854  */
855 kstat_t *xuio_ksp = NULL;

857 int
858 dmu_xuio_init(xuio_t *xuio, int nblk)
859 {
860         dmu_xuio_t *priv;
861         uio_t *uio = &xuio->xu_uio;

863         uio->uio_iovcnt = nblk;
864         uio->uio_iov = kmem_zalloc(nblk * sizeof (iovec_t), KM_SLEEP);

866         priv = kmem_zalloc(sizeof (dmu_xuio_t), KM_SLEEP);
867         priv->cnt = nblk;
868         priv->bufs = kmem_zalloc(nblk * sizeof (arc_buf_t *), KM_SLEEP);
869         priv->iovp = uio->uio_iov;
870         XUIO_XUZC_PRIV(xuio) = priv;

872         if (XUIO_XUZC_RW(xuio) == UIO_READ)
873                 XUIOSTAT_INCR(xuiostat_onloan_rbuf, nblk);
874         else
875                 XUIOSTAT_INCR(xuiostat_onloan_wbuf, nblk);

877         return (0);
878 }

880 void
881 dmu_xuio_fini(xuio_t *xuio)
882 {
883         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);
884         int nblk = priv->cnt;

886         kmem_free(priv->iovp, nblk * sizeof (iovec_t));
887         kmem_free(priv->bufs, nblk * sizeof (arc_buf_t *));
888         kmem_free(priv, sizeof (dmu_xuio_t));

890         if (XUIO_XUZC_RW(xuio) == UIO_READ)
891                 XUIOSTAT_INCR(xuiostat_onloan_rbuf, -nblk);
892         else
893                 XUIOSTAT_INCR(xuiostat_onloan_wbuf, -nblk);
894 }

896 /*
897  * Initialize iov[priv->next] and priv->bufs[priv->next] with { off, n, abuf }
898  * and increase priv->next by 1.
899  */
900 int
901 dmu_xuio_add(xuio_t *xuio, arc_buf_t *abuf, offset_t off, size_t n)
902 {
903         struct iovec *iov;
904         uio_t *uio = &xuio->xu_uio;
905         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);
906         int i = priv->next++;

908         ASSERT(i < priv->cnt);
909         ASSERT(off + n <= arc_buf_size(abuf));
910         iov = uio->uio_iov + i;
911         iov->iov_base = (char *)abuf->b_data + off;
912         iov->iov_len = n;
913         priv->bufs[i] = abuf;
914         return (0);
915 }

917 int
918 dmu_xuio_cnt(xuio_t *xuio)
919 {
```

```
920         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);
921         return (priv->cnt);
922 }

924 arc_buf_t *
925 dmu_xuio_arcbuf(xuio_t *xuio, int i)
926 {
927         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);

929         ASSERT(i < priv->cnt);
930         return (priv->bufs[i]);
931 }

933 void
934 dmu_xuio_clear(xuio_t *xuio, int i)
935 {
936         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);

938         ASSERT(i < priv->cnt);
939         priv->bufs[i] = NULL;
940 }

942 static void
943 xuio_stat_init(void)
944 {
945         xuio_ksp = kstat_create("zfs", 0, "xuio_stats", "misc",
946             KSTAT_TYPE_NAMED, sizeof (xuio_stats) / sizeof (kstat_named_t),
947             KSTAT_FLAG_VIRTUAL);
948         if (xuio_ksp != NULL) {
949                 xuio_ksp->ks_data = &xuio_stats;
950                 kstat_install(xuio_ksp);
951         }
952 }

954 static void
955 xuio_stat_fini(void)
956 {
957         if (xuio_ksp != NULL) {
958                 kstat_delete(xuio_ksp);
959                 xuio_ksp = NULL;
960         }
961 }

963 void
964 xuio_stat_wbuf_copied()
965 {
966         XUIOSTAT_BUMP(xuiostat_wbuf_copied);
967 }

969 void
970 xuio_stat_wbuf_nocopy()
971 {
972         XUIOSTAT_BUMP(xuiostat_wbuf_nocopy);
973 }

975 #ifdef _KERNEL
976 int
977 dmu_read_uio(objset_t *os, uint64_t object, uio_t *uio, uint64_t size)
978 {
979         dmu_buf_t **dbp;
980         int numbufs, i, err;
981         xuio_t *xuio = NULL;

983         /*
984          * NB: we could do this block-at-a-time, but it's nice
985          * to be reading in parallel.
```



```
986          */
987         err = dmu_buf_hold_array(os, object, uio->uio_loffset, size, TRUE, FTAG,
988             &numbufs, &dbp);
989         if (err)
990                 return (err);

992         if (uio->uio_extflg == UIO_XUIO)
993                 xuio = (xuio_t *)uio;

995         for (i = 0; i < numbufs; i++) {
996                 int tocpy;
997                 int bufoff;
998                 dmu_buf_t *db = dbp[i];

1000                ASSERT(size > 0);

1002                bufoff = uio->uio_loffset - db->db_offset;
1003                tocpy = (int)MIN(db->db_size - bufoff, size);

1005                if (xuio) {
1006                        dmu_buf_impl_t *dbi = (dmu_buf_impl_t *)db;
1007                        arc_buf_t *dbuf_abuf = dbi->db_buf;
1008                        arc_buf_t *abuf = dbuf_loan_arcbuf(dbi);
1009                        err = dmu_xuio_add(xuio, abuf, bufoff, tocpy);
1010                        if (!err) {
1011                                uio->uio_resid -= tocpy;
1012                                uio->uio_loffset += tocpy;
1013                        }

1015                        if (abuf == dbuf_abuf)
1016                                XUIOSTAT_BUMP(xuiostat_rbuf_nocopy);
1017                        else
1018                                XUIOSTAT_BUMP(xuiostat_rbuf_copied);
1019                } else {
1020                        err = uiomove((char *)db->db_data + bufoff, tocpy,
1021                            UIO_READ, uio);
1022                }
1023                if (err)
1024                        break;

1026                size -= tocpy;
1027        }
1028        dmu_buf_rele_array(dbp, numbufs, FTAG);

1030        return (err);
1031 }

1033 static int
1034 dmu_write_uio_dnode(dnode_t *dn, uio_t *uio, uint64_t size, dmu_tx_t *tx)
1035 {
1036         dmu_buf_t **dbp;
1037         int numbufs;
1038         int err = 0;
1039         int i;

1041         err = dmu_buf_hold_array_by_dnode(dn, uio->uio_loffset, size,
1042             FALSE, FTAG, &numbufs, &dbp, DMU_READ_PREFETCH);
1043         if (err)
1044                 return (err);

1046         for (i = 0; i < numbufs; i++) {
1047                 int tocpy;
1048                 int bufoff;
1049                 dmu_buf_t *db = dbp[i];

1051                 ASSERT(size > 0);
```

```
1053                 bufoff = uio->uio_loffset - db->db_offset;
1054                 tocpy = (int)MIN(db->db_size - bufoff, size);

1056                 ASSERT(i == 0 || i == numbufs-1 || tocpy == db->db_size);

1058                 if (tocpy == db->db_size)
1059                         dmu_buf_will_fill(db, tx);
1060                 else
1061                         dmu_buf_will_dirty(db, tx);

1063                 /*
1064                  * XXX uiomove could block forever (eg. nfs-backed
1065                  * pages).  There needs to be a uiolockdown() function
1066                  * to lock the pages in memory, so that uiomove won't
1067                  * block.
1068                  */
1069                 err = uiomove((char *)db->db_data + bufoff, tocpy,
1070                     UIO_WRITE, uio);

1072                 if (tocpy == db->db_size)
1073                         dmu_buf_fill_done(db, tx);

1075                 if (err)
1076                         break;

1078                 size -= tocpy;
1079         }

1081         dmu_buf_rele_array(dbp, numbufs, FTAG);
1082         return (err);
1083 }

1085 int
1086 dmu_write_uio_dbuf(dmu_buf_t *zdb, uio_t *uio, uint64_t size,
1087     dmu_tx_t *tx)
1088 {
1089         dmu_buf_impl_t *db = (dmu_buf_impl_t *)zdb;
1090         dnode_t *dn;
1091         int err;

1093         if (size == 0)
1094                 return (0);

1096         DB_DNODE_ENTER(db);
1097         dn = DB_DNODE(db);
1098         err = dmu_write_uio_dnode(dn, uio, size, tx);
1099         DB_DNODE_EXIT(db);

1101         return (err);
1102 }

1104 int
1105 dmu_write_uio(objset_t *os, uint64_t object, uio_t *uio, uint64_t size,
1106     dmu_tx_t *tx)
1107 {
1108         dnode_t *dn;
1109         int err;

1111         if (size == 0)
1112                 return (0);

1114         err = dnode_hold(os, object, FTAG, &dn);
1115         if (err)
1116                 return (err);
```



```
1118         err = dmu_write_uio_dnode(dn, uio, size, tx);

1120         dnode_rele(dn, FTAG);

1122         return (err);
1123 }

1125 int
1126 dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
1127     page_t *pp, dmu_tx_t *tx)
1128 {
1129         dmu_buf_t **dbp;
1130         int numbufs, i;
1131         int err;

1133         if (size == 0)
1134                 return (0);

1136         err = dmu_buf_hold_array(os, object, offset, size,
1137             FALSE, FTAG, &numbufs, &dbp);
1138         if (err)
1139                 return (err);

1141         for (i = 0; i < numbufs; i++) {
1142                 int tocpy, copied, thiscpy;
1143                 int bufoff;
1144                 dmu_buf_t *db = dbp[i];
1145                 caddr_t va;

1147                 ASSERT(size > 0);
1148                 ASSERT3U(db->db_size, >=, PAGESIZE);

1150                 bufoff = offset - db->db_offset;
1151                 tocpy = (int)MIN(db->db_size - bufoff, size);

1153                 ASSERT(i == 0 || i == numbufs-1 || tocpy == db->db_size);

1155                 if (tocpy == db->db_size)
1156                         dmu_buf_will_fill(db, tx);
1157                 else
1158                         dmu_buf_will_dirty(db, tx);

1160                 for (copied = 0; copied < tocpy; copied += PAGESIZE) {
1161                         ASSERT3U(pp->p_offset, ==, db->db_offset + bufoff);
1162                         thiscpy = MIN(PAGESIZE, tocpy - copied);
1163                         va = zfs_map_page(pp, S_READ);
1164                         bcopy(va, (char *)db->db_data + bufoff, thiscpy);
1165                         zfs_unmap_page(pp, va);
1166                         pp = pp->p_next;
1167                         bufoff += PAGESIZE;
1168                 }

1170                 if (tocpy == db->db_size)
1171                         dmu_buf_fill_done(db, tx);

1173                 offset += tocpy;
1174                 size -= tocpy;
1175         }
1176         dmu_buf_rele_array(dbp, numbufs, FTAG);
1177         return (err);
1178 }
1179 #endif

1181 /*
1182  * Allocate a loaned anonymous arc buffer.
1183  */
```

```
1184 arc_buf_t *
1185 dmu_request_arcbuf(dmu_buf_t *handle, int size)
1186 {
1187         dmu_buf_impl_t *db = (dmu_buf_impl_t *)handle;
1188         spa_t *spa;
1189
1190         DB_GET_SPA(&spa, db);
1191         return (arc_loan_buf(spa, size));
1192 }
1193
1194 /*
1195  * Free a loaned arc buffer.
1196  */
1197 void
1198 dmu_return_arcbuf(arc_buf_t *buf)
1199 {
1200         arc_return_buf(buf, FTAG);
1201         VERIFY(arc_buf_remove_ref(buf, FTAG));
1202 }
1203
1204 /*
1205  * When possible directly assign passed loaned arc buffer to a dbuf.
1206  * If this is not possible copy the contents of passed arc buf via
1207  * dmu_write().
1208  */
1209 void
1210 dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, arc_buf_t *buf,
1211     dmu_tx_t *tx)
1212 {
1213         dmu_buf_impl_t *dbuf = (dmu_buf_impl_t *)handle;
1214         dnode_t *dn;
1215         dmu_buf_impl_t *db;
1216         uint32_t blksz = (uint32_t)arc_buf_size(buf);
1217         uint64_t blkid;
1218
1219         DB_DNODE_ENTER(dbuf);
1220         dn = DB_DNODE(dbuf);
1221         rw_enter(&dn->dn_struct_rwlock, RW_READER);
1222         blkid = dbuf_whichblock(dn, offset);
1223         VERIFY((db = dbuf_hold(dn, blkid, FTAG)) != NULL);
1224         rw_exit(&dn->dn_struct_rwlock);
1225         DB_DNODE_EXIT(dbuf);
1226
1227         if (offset == db->db.db_offset && blksz == db->db.db_size) {
1228                 dbuf_assign_arcbuf(db, buf, tx);
1229                 dbuf_rele(db, FTAG);
1230         } else {
1231                 objset_t *os;
1232                 uint64_t object;
1233
1234                 DB_DNODE_ENTER(dbuf);
1235                 dn = DB_DNODE(dbuf);
1236                 os = dn->dn_objset;
1237                 object = dn->dn_object;
1238                 DB_DNODE_EXIT(dbuf);
1239
1240                 dbuf_rele(db, FTAG);
1241                 dmu_write(os, object, offset, blksz, buf->b_data, tx);
1242                 dmu_return_arcbuf(buf);
1243                 XUIOSTAT_BUMP(xuiostat_wbuf_copied);
1244         }
1245 }
1246
1247 typedef struct {
1248         dbuf_dirty_record_t     *dsa_dr;
1249         dmu_sync_cb_t           *dsa_done;
```



```
1250         zgd_t                   *dsa_zgd;
1251         dmu_tx_t                *dsa_tx;
1252 } dmu_sync_arg_t;
1253
1254 /* ARGSUSED */
1255 static void
1256 dmu_sync_ready(zio_t *zio, arc_buf_t *buf, void *varg)
1257 {
1258         dmu_sync_arg_t *dsa = varg;
1259         dmu_buf_t *db = dsa->dsa_zgd->zgd_db;
1260         blkptr_t *bp = zio->io_bp;
1261
1262         if (zio->io_error == 0) {
1263                 if (BP_IS_HOLE(bp)) {
1264                         /*
1265                          * A block of zeros may compress to a hole, but the
1266                          * block size still needs to be known for replay.
1267                          */
1268                         BP_SET_LSIZE(bp, db->db_size);
1269                 } else {
1270                         ASSERT(BP_GET_LEVEL(bp) == 0);
1271                         bp->blk_fill = 1;
1272                 }
1273         }
1274 }
1275
1276 static void
1277 dmu_sync_late_arrival_ready(zio_t *zio)
1278 {
1279         dmu_sync_ready(zio, NULL, zio->io_private);
1280 }
1281
1282 /* ARGSUSED */
1283 static void
1284 dmu_sync_done(zio_t *zio, arc_buf_t *buf, void *varg)
1285 {
1286         dmu_sync_arg_t *dsa = varg;
1287         dbuf_dirty_record_t *dr = dsa->dsa_dr;
1288         dmu_buf_impl_t *db = dr->dr_dbuf;
1289
1290         mutex_enter(&db->db_mtx);
1291         ASSERT(dr->dt.dl.dr_override_state == DR_IN_DMU_SYNC);
1292         if (zio->io_error == 0) {
1293                 dr->dt.dl.dr_nopwrite = !!(zio->io_flags & ZIO_FLAG_NOPWRITE);
1294                 if (dr->dt.dl.dr_nopwrite) {
1295                         blkptr_t *bp = zio->io_bp;
1296                         blkptr_t *bp_orig = &zio->io_bp_orig;
1297                         uint8_t chksum = BP_GET_CHECKSUM(bp_orig);
1298
1299                         ASSERT(BP_EQUAL(bp, bp_orig));
1300                         ASSERT(zio->io_prop.zp_compress != ZIO_COMPRESS_OFF);
1301                         ASSERT(zio_checksum_table[chksum].ci_dedup);
1302                 }
1303                 dr->dt.dl.dr_overridden_by = *zio->io_bp;
1304                 dr->dt.dl.dr_override_state = DR_OVERRIDDEN;
1305                 dr->dt.dl.dr_copies = zio->io_prop.zp_copies;
1306                 if (BP_IS_HOLE(&dr->dt.dl.dr_overridden_by))
1307                         BP_ZERO(&dr->dt.dl.dr_overridden_by);
1308         } else {
1309                 dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
1310         }
1311         cv_broadcast(&db->db_changed);
1312         mutex_exit(&db->db_mtx);
1313
1314         dsa->dsa_done(dsa->dsa_zgd, zio->io_error);
```

```
1316         kmem_free(dsa, sizeof (*dsa));
1317 }

1319 static void
1320 dmu_sync_late_arrival_done(zio_t *zio)
1321 {
1322         blkptr_t *bp = zio->io_bp;
1323         dmu_sync_arg_t *dsa = zio->io_private;
1324         blkptr_t *bp_orig = &zio->io_bp_orig;

1326         if (zio->io_error == 0 && !BP_IS_HOLE(bp)) {
1327                 /*
1328                  * If we didn't allocate a new block (i.e. ZIO_FLAG_NOPWRITE)
1329                  * then there is nothing to do here. Otherwise, free the
1330                  * newly allocated block in this txg.
1331                  */
1332                 if (zio->io_flags & ZIO_FLAG_NOPWRITE) {
1333                         ASSERT(BP_EQUAL(bp, bp_orig));
1334                 } else {
1335                         ASSERT(BP_IS_HOLE(bp_orig) || !BP_EQUAL(bp, bp_orig));
1336                         ASSERT(zio->io_bp->blk_birth == zio->io_txg);
1337                         ASSERT(zio->io_txg > spa_syncing_txg(zio->io_spa));
1338                         zio_free(zio->io_spa, zio->io_txg, zio->io_bp);
1339                 }
1340         }

1342         dmu_tx_commit(dsa->dsa_tx);

1344         dsa->dsa_done(dsa->dsa_zgd, zio->io_error);

1346         kmem_free(dsa, sizeof (*dsa));
1347 }

1349 static int
1350 dmu_sync_late_arrival(zio_t *pio, objset_t *os, dmu_sync_cb_t *done, zgd_t *zgd,
1351     zio_prop_t *zp, zbookmark_t *zb)
1352 {
1353         dmu_sync_arg_t *dsa;
1354         dmu_tx_t *tx;

1356         tx = dmu_tx_create(os);
1357         dmu_tx_hold_space(tx, zgd->zgd_db->db_size);
1358         if (dmu_tx_assign(tx, TXG_WAIT) != 0) {
1359                 dmu_tx_abort(tx);
1360                 /* Make zl_get_data do txg_waited_synced() */
1361                 return (SET_ERROR(EIO));
1362         }

1364         dsa = kmem_alloc(sizeof (dmu_sync_arg_t), KM_SLEEP);
1365         dsa->dsa_dr = NULL;
1366         dsa->dsa_done = done;
1367         dsa->dsa_zgd = zgd;
1368         dsa->dsa_tx = tx;

1370         zio_nowait(zio_write(pio, os->os_spa, dmu_tx_get_txg(tx), zgd->zgd_bp,
1371             zgd->zgd_db->db_data, zgd->zgd_db->db_size, zp,
1372             dmu_sync_late_arrival_ready, dmu_sync_late_arrival_done, dsa,
1373             ZIO_PRIORITY_SYNC_WRITE, ZIO_FLAG_CANFAIL, zb));

1375         return (0);
1376 }

1378 /*
1379  * Intent log support: sync the block associated with db to disk.
1380  * N.B. and XXX: the caller is responsible for making sure that the
1381  * data isn't changing while dmu_sync() is writing it.
```



```
1382  *
1383  * Return values:
1384  *
1385  *      EEXIST: this txg has already been synced, so there's nothing to do.
1386  *              The caller should not log the write.
1387  *
1388  *      ENOENT: the block was dbuf_free_range()'d, so there's nothing to do.
1389  *              The caller should not log the write.
1390  *
1391  *      EALREADY: this block is already in the process of being synced.
1392  *              The caller should track its progress (somehow).
1393  *
1394  *      EIO: could not do the I/O.
1395  *              The caller should do a txg_wait_synced().
1396  *
1397  *      0: the I/O has been initiated.
1398  *              The caller should log this blkptr in the done callback.
1399  *              It is possible that the I/O will fail, in which case
1400  *              the error will be reported to the done callback and
1401  *              propagated to pio from zio_done().
1402  */
1403 int
1404 dmu_sync(zio_t *pio, uint64_t txg, dmu_sync_cb_t *done, zgd_t *zgd)
1405 {
1406         blkptr_t *bp = zgd->zgd_bp;
1407         dmu_buf_impl_t *db = (dmu_buf_impl_t *)zgd->zgd_db;
1408         objset_t *os = db->db_objset;
1409         dsl_dataset_t *ds = os->os_dsl_dataset;
1410         dbuf_dirty_record_t *dr;
1411         dmu_sync_arg_t *dsa;
1412         zbookmark_t zb;
1413         zio_prop_t zp;
1414         dnode_t *dn;

1416         ASSERT(pio != NULL);
1417         ASSERT(txg != 0);

1419         SET_BOOKMARK(&zb, ds->ds_object,
1420             db->db.db_object, db->db_level, db->db_blkid);

1422         DB_DNODE_ENTER(db);
1423         dn = DB_DNODE(db);
1424         dmu_write_policy(os, dn, db->db_level, WP_DMU_SYNC, &zp, txg);
  24         dmu_write_policy(os, dn, db->db_level, WP_DMU_SYNC, &zp);
1425         DB_DNODE_EXIT(db);

1427         /*
1428          * If we're frozen (running ziltest), we always need to generate a bp.
1429          */
1430         if (txg > spa_freeze_txg(os->os_spa))
1431                 return (dmu_sync_late_arrival(pio, os, done, zgd, &zp, &zb));

1433         /*
1434          * Grabbing db_mtx now provides a barrier between dbuf_sync_leaf()
1435          * and us.  If we determine that this txg is not yet syncing,
1436          * but it begins to sync a moment later, that's OK because the
1437          * sync thread will block in dbuf_sync_leaf() until we drop db_mtx.
1438          */
1439         mutex_enter(&db->db_mtx);

1441         if (txg <= spa_last_synced_txg(os->os_spa)) {
1442                 /*
1443                  * This txg has already synced.  There's nothing to do.
1444                  */
1445                 mutex_exit(&db->db_mtx);
1446                 return (SET_ERROR(EEXIST));
```

```
1447         }

1449         if (txg <= spa_syncing_txg(os->os_spa)) {
1450                 /*
1451                  * This txg is currently syncing, so we can't mess with
1452                  * the dirty record anymore; just write a new log block.
1453                  */
1454                 mutex_exit(&db->db_mtx);
1455                 return (dmu_sync_late_arrival(pio, os, done, zgd, &zp, &zb));
1456         }

1458         dr = db->db_last_dirty;
1459         while (dr && dr->dr_txg != txg)
1460                 dr = dr->dr_next;

1462         if (dr == NULL) {
1463                 /*
1464                  * There's no dr for this dbuf, so it must have been freed.
1465                  * There's no need to log writes to freed blocks, so we're done.
1466                  */
1467                 mutex_exit(&db->db_mtx);
1468                 return (SET_ERROR(ENOENT));
1469         }

1471         ASSERT(dr->dr_next == NULL || dr->dr_next->dr_txg < txg);

1473         /*
1474          * Assume the on-disk data is X, the current syncing data is Y,
1475          * and the current in-memory data is Z (currently in dmu_sync).
1476          * X and Z are identical but Y is has been modified. Normally,
1477          * when X and Z are the same we will perform a nopwrite but if Y
1478          * is different we must disable nopwrite since the resulting write
1479          * of Y to disk can free the block containing X. If we allowed a
1480          * nopwrite to occur the block pointing to Z would reference a freed
1481          * block. Since this is a rare case we simplify this by disabling
1482          * nopwrite if the current dmu_sync-ing dbuf has been modified in
1483          * a previous transaction.
1484          */
1485         if (dr->dr_next)
1486                 zp.zp_nopwrite = B_FALSE;

1488         ASSERT(dr->dr_txg == txg);
1489         if (dr->dt.dl.dr_override_state == DR_IN_DMU_SYNC ||
1490             dr->dt.dl.dr_override_state == DR_OVERRIDDEN) {
1491                 /*
1492                  * We have already issued a sync write for this buffer,
1493                  * or this buffer has already been synced.  It could not
1494                  * have been dirtied since, or we would have cleared the state.
1495                  */
1496                 mutex_exit(&db->db_mtx);
1497                 return (SET_ERROR(EALREADY));
1498         }

1500         ASSERT(dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN);
1501         dr->dt.dl.dr_override_state = DR_IN_DMU_SYNC;
1502         mutex_exit(&db->db_mtx);

1504         dsa = kmem_alloc(sizeof (dmu_sync_arg_t), KM_SLEEP);
1505         dsa->dsa_dr = dr;
1506         dsa->dsa_done = done;
1507         dsa->dsa_zgd = zgd;
1508         dsa->dsa_tx = NULL;

1510         zio_nowait(arc_write(pio, os->os_spa, txg,
1511             bp, dr->dt.dl.dr_data, DBUF_IS_L2CACHEABLE(db), &zp,
1512             dmu_sync_ready, dmu_sync_done, dsa,
```



```
1513             ZIO_PRIORITY_SYNC_WRITE, ZIO_FLAG_CANFAIL, &zb));

1515         return (0);
1516 }
_____unchanged_portion_omitted_

1561 int zfs_mdcomp_disable = 0;
1562 int zfs_mdcomp_lz4 = 0;
1563 #endif /* ! codereview */

1565 void
1566 dmu_write_policy(objset_t *os, dnode_t *dn, int level, int wp, zio_prop_t *zp,
1567     uint64_t txg)
 162 dmu_write_policy(objset_t *os, dnode_t *dn, int level, int wp, zio_prop_t *zp)
1568 {
1569         dmu_object_type_t type = dn ? dn->dn_type : DMU_OT_OBJSET;
1570         boolean_t ismd = (level > 0 || DMU_OT_IS_METADATA(type) ||
1571             (wp & WP_SPILL));
1572         enum zio_checksum checksum = os->os_checksum;
1573         enum zio_compress compress = os->os_compress;
1574         enum zio_checksum dedup_checksum = os->os_dedup_checksum;
1575         boolean_t dedup = B_FALSE;
1576         boolean_t nopwrite = B_FALSE;
1577         boolean_t dedup_verify = os->os_dedup_verify;
1578         int copies = os->os_copies;

1580         /*
1581          * We maintain different write policies for each of the following
1582          * types of data:
1583          *       1. metadata
1584          *       2. preallocated blocks (i.e. level-0 blocks of a dump device)
1585          *       3. all other level 0 blocks
1586          */
1587         if (ismd) {
1588                 /*
1589                  * XXX -- we should design a compression algorithm
1590                  * that specializes in arrays of bps.
1591                  */
1592                 if (zfs_mdcomp_disable)
1593                         compress = ZIO_COMPRESS_EMPTY;
1594                 else if (zfs_mdcomp_lz4 && os->os_spa != NULL) {
1595                         zfeature_info_t *feat = &spa_feature_table
1596                             [SPA_FEATURE_LZ4_COMPRESS];

1598                         if (spa_feature_is_active(os->os_spa, feat))
1599                                 compress = ZIO_COMPRESS_LZ4;
1600                         else if (spa_feature_is_enabled(os->os_spa, feat)) {
1601                                 dmu_tx_t *tx;

1603                                 tx = dmu_tx_create_assigned(
1604                                     spa_get_dsl(os->os_spa), txg);
1605                                 spa_feature_incr(os->os_spa, feat, tx);
1606                                 dmu_tx_commit(tx);
1607                                 compress = ZIO_COMPRESS_LZ4;
1608                         } else
1609                                 compress = ZIO_COMPRESS_LZJB;
1610                 } else
1611                         compress = ZIO_COMPRESS_LZJB;
 187                 compress = zfs_mdcomp_disable ? ZIO_COMPRESS_EMPTY :
 188                     ZIO_COMPRESS_LZJB;

1613                 /*
1614                  * Metadata always gets checksummed.  If the data
1615                  * checksum is multi-bit correctable, and it's not a
1616                  * ZBT-style checksum, then it's suitable for metadata
1617                  * as well.  Otherwise, the metadata checksum defaults
```

```
1618                  * to fletcher4.
1619                  */
1620                 if (zio_checksum_table[checksum].ci_correctable < 1 ||
1621                     zio_checksum_table[checksum].ci_eck)
1622                         checksum = ZIO_CHECKSUM_FLETCHER_4;
1623         } else if (wp & WP_NOFILL) {
1624                 ASSERT(level == 0);

1626                 /*
1627                  * If we're writing preallocated blocks, we aren't actually
1628                  * writing them so don't set any policy properties.  These
1629                  * blocks are currently only used by an external subsystem
1630                  * outside of zfs (i.e. dump) and not written by the zio
1631                  * pipeline.
1632                  */
1633                 compress = ZIO_COMPRESS_OFF;
1634                 checksum = ZIO_CHECKSUM_OFF;
1635         } else {
1636                 compress = zio_compress_select(dn->dn_compress, compress);

1638                 checksum = (dedup_checksum == ZIO_CHECKSUM_OFF) ?
1639                     zio_checksum_select(dn->dn_checksum, checksum) :
1640                     dedup_checksum;

1642                 /*
1643                  * Determine dedup setting.  If we are in dmu_sync(),
1644                  * we won't actually dedup now because that's all
1645                  * done in syncing context; but we do want to use the
1646                  * dedup checkum.  If the checksum is not strong
1647                  * enough to ensure unique signatures, force
1648                  * dedup_verify.
1649                  */
1650                 if (dedup_checksum != ZIO_CHECKSUM_OFF) {
1651                         dedup = (wp & WP_DMU_SYNC) ? B_FALSE : B_TRUE;
1652                         if (!zio_checksum_table[checksum].ci_dedup)
1653                                 dedup_verify = B_TRUE;
1654                 }

1656                 /*
1657                  * Enable nopwrite if we have a cryptographically secure
1658                  * checksum that has no known collisions (i.e. SHA-256)
1659                  * and compression is enabled.  We don't enable nopwrite if
1660                  * dedup is enabled as the two features are mutually exclusive.
1661                  */
1662                 nopwrite = (!dedup && zio_checksum_table[checksum].ci_dedup &&
1663                     compress != ZIO_COMPRESS_OFF && zfs_nopwrite_enabled);
1664         }

1666         zp->zp_checksum = checksum;
1667         zp->zp_compress = compress;
1668         zp->zp_type = (wp & WP_SPILL) ? dn->dn_bonustype : type;
1669         zp->zp_level = level;
1670         zp->zp_copies = MIN(copies + ismd, spa_max_replication(os->os_spa));
1671         zp->zp_dedup = dedup;
1672         zp->zp_dedup_verify = dedup && dedup_verify;
1673         zp->zp_nopwrite = nopwrite;
1674 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   42891 Wed May  1 11:13:43 2013
new/usr/src/uts/common/fs/zfs/dmu_objset.c
3756 want lz4 support for metadata compression
**********************************************************
```

```
/*
 * 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 (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 Martin Matuska. All rights reserved.
#endif /* ! codereview */
 */

/* Portions Copyright 2010 Robert Milkowski */

#include <sys/cred.h>
#include <sys/zfs_context.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_deleg.h>
#include <sys/dnode.h>
#include <sys/dbuf.h>
#include <sys/zvol.h>
#include <sys/dmu_tx.h>
#include <sys/zap.h>
#include <sys/zil.h>
#include <sys/dmu_impl.h>
#include <sys/zfs_ioctl.h>
#include <sys/sa.h>
#include <sys/zfs_onexit.h>
#include <sys/dsl_destroy.h>

/*
 * Needed to close a window in dnode_move() that allows the objset to be freed
 * before it can be safely accessed.
 */
krwlock_t os_lock;

void
dmu_objset_init(void)
{
        rw_init(&os_lock, NULL, RW_DEFAULT, NULL);
}
```



```
void
dmu_objset_fini(void)
{
        rw_destroy(&os_lock);
}

spa_t *
dmu_objset_spa(objset_t *os)
{
        return (os->os_spa);
}

zilog_t *
dmu_objset_zil(objset_t *os)
{
        return (os->os_zil);
}

dsl_pool_t *
dmu_objset_pool(objset_t *os)
{
        dsl_dataset_t *ds;

        if ((ds = os->os_dsl_dataset) != NULL && ds->ds_dir)
                return (ds->ds_dir->dd_pool);
        else
                return (spa_get_dsl(os->os_spa));
}

dsl_dataset_t *
dmu_objset_ds(objset_t *os)
{
        return (os->os_dsl_dataset);
}

dmu_objset_type_t
dmu_objset_type(objset_t *os)
{
        return (os->os_phys->os_type);
}

void
dmu_objset_name(objset_t *os, char *buf)
{
        dsl_dataset_name(os->os_dsl_dataset, buf);
}

uint64_t
dmu_objset_id(objset_t *os)
{
        dsl_dataset_t *ds = os->os_dsl_dataset;

        return (ds ? ds->ds_object : 0);
}

uint64_t
dmu_objset_syncprop(objset_t *os)
{
        return (os->os_sync);
}

uint64_t
dmu_objset_logbias(objset_t *os)
{
        return (os->os_logbias);
```

```
128 }
130 static void
131 checksum_changed_cb(void *arg, uint64_t newval)
132 {
133         objset_t *os = arg;
135         /*
136          * Inheritance should have been done by now.
137          */
138         ASSERT(newval != ZIO_CHECKSUM_INHERIT);
140         os->os_checksum = zio_checksum_select(newval, ZIO_CHECKSUM_ON_VALUE);
141 }
143 static void
144 compression_changed_cb(void *arg, uint64_t newval)
145 {
146         objset_t *os = arg;
148         /*
149          * Inheritance and range checking should have been done by now.
150          */
151         ASSERT(newval != ZIO_COMPRESS_INHERIT);
153         os->os_compress = zio_compress_select(newval, ZIO_COMPRESS_ON_VALUE);
154 }
156 static void
157 copies_changed_cb(void *arg, uint64_t newval)
158 {
159         objset_t *os = arg;
161         /*
162          * Inheritance and range checking should have been done by now.
163          */
164         ASSERT(newval > 0);
165         ASSERT(newval <= spa_max_replication(os->os_spa));
167         os->os_copies = newval;
168 }
170 static void
171 dedup_changed_cb(void *arg, uint64_t newval)
172 {
173         objset_t *os = arg;
174         spa_t *spa = os->os_spa;
175         enum zio_checksum checksum;
177         /*
178          * Inheritance should have been done by now.
179          */
180         ASSERT(newval != ZIO_CHECKSUM_INHERIT);
182         checksum = zio_checksum_dedup_select(spa, newval, ZIO_CHECKSUM_OFF);
184         os->os_dedup_checksum = checksum & ZIO_CHECKSUM_MASK;
185         os->os_dedup_verify = !!(checksum & ZIO_CHECKSUM_VERIFY);
186 }
188 static void
189 primary_cache_changed_cb(void *arg, uint64_t newval)
190 {
191         objset_t *os = arg;
193         /*
```



```
194          * Inheritance and range checking should have been done by now.
195          */
196         ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
197             newval == ZFS_CACHE_METADATA);
199         os->os_primary_cache = newval;
200 }
202 static void
203 secondary_cache_changed_cb(void *arg, uint64_t newval)
204 {
205         objset_t *os = arg;
207         /*
208          * Inheritance and range checking should have been done by now.
209          */
210         ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
211             newval == ZFS_CACHE_METADATA);
213         os->os_secondary_cache = newval;
214 }
216 static void
217 sync_changed_cb(void *arg, uint64_t newval)
218 {
219         objset_t *os = arg;
221         /*
222          * Inheritance and range checking should have been done by now.
223          */
224         ASSERT(newval == ZFS_SYNC_STANDARD || newval == ZFS_SYNC_ALWAYS ||
225             newval == ZFS_SYNC_DISABLED);
227         os->os_sync = newval;
228         if (os->os_zil)
229                 zil_set_sync(os->os_zil, newval);
230 }
232 static void
233 logbias_changed_cb(void *arg, uint64_t newval)
234 {
235         objset_t *os = arg;
237         ASSERT(newval == ZFS_LOGBIAS_LATENCY ||
238             newval == ZFS_LOGBIAS_THROUGHPUT);
239         os->os_logbias = newval;
240         if (os->os_zil)
241                 zil_set_logbias(os->os_zil, newval);
242 }
244 void
245 dmu_objset_byteswap(void *buf, size_t size)
246 {
247         objset_phys_t *osp = buf;
249         ASSERT(size == OBJSET_OLD_PHYS_SIZE || size == sizeof (objset_phys_t));
250         dnode_byteswap(&osp->os_meta_dnode);
251         byteswap_uint64_array(&osp->os_zil_header, sizeof (zil_header_t));
252         osp->os_type = BSWAP_64(osp->os_type);
253         osp->os_flags = BSWAP_64(osp->os_flags);
254         if (size == sizeof (objset_phys_t)) {
255                 dnode_byteswap(&osp->os_userused_dnode);
256                 dnode_byteswap(&osp->os_groupused_dnode);
257         }
258 }
```

```
260 int
261 dmu_objset_open_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
262     objset_t **osp)
263 {
264         objset_t *os;
265         int i, err;
267         ASSERT(ds == NULL || MUTEX_HELD(&ds->ds_opening_lock));
269         os = kmem_zalloc(sizeof (objset_t), KM_SLEEP);
270         os->os_dsl_dataset = ds;
271         os->os_spa = spa;
272         os->os_rootbp = bp;
273         if (!BP_IS_HOLE(os->os_rootbp)) {
274                 uint32_t aflags = ARC_WAIT;
275                 zbookmark_t zb;
276                 SET_BOOKMARK(&zb, ds ? ds->ds_object : DMU_META_OBJSET,
277                     ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
279                 if (DMU_OS_IS_L2CACHEABLE(os))
280                         aflags |= ARC_L2CACHE;
282                 dprintf_bp(os->os_rootbp, "reading %s", "");
283                 err = arc_read(NULL, spa, os->os_rootbp,
284                     arc_getbuf_func, &os->os_phys_buf,
285                     ZIO_PRIORITY_SYNC_READ, ZIO_FLAG_CANFAIL, &aflags, &zb);
286                 if (err != 0) {
287                         kmem_free(os, sizeof (objset_t));
288                         /* convert checksum errors into IO errors */
289                         if (err == ECKSUM)
290                                 err = SET_ERROR(EIO);
291                         return (err);
292                 }
294                 /* Increase the blocksize if we are permitted. */
295                 if (spa_version(spa) >= SPA_VERSION_USERSPACE &&
296                     arc_buf_size(os->os_phys_buf) < sizeof (objset_phys_t)) {
297                         arc_buf_t *buf = arc_buf_alloc(spa,
298                             sizeof (objset_phys_t), &os->os_phys_buf,
299                             ARC_BUFC_METADATA);
300                         bzero(buf->b_data, sizeof (objset_phys_t));
301                         bcopy(os->os_phys_buf->b_data, buf->b_data,
302                             arc_buf_size(os->os_phys_buf));
303                         (void) arc_buf_remove_ref(os->os_phys_buf,
304                             &os->os_phys_buf);
305                         os->os_phys_buf = buf;
306                 }
308                 os->os_phys = os->os_phys_buf->b_data;
309                 os->os_flags = os->os_phys->os_flags;
310         } else {
311                 int size = spa_version(spa) >= SPA_VERSION_USERSPACE ?
312                     sizeof (objset_phys_t) : OBJSET_OLD_PHYS_SIZE;
313                 os->os_phys_buf = arc_buf_alloc(spa, size,
314                     &os->os_phys_buf, ARC_BUFC_METADATA);
315                 os->os_phys = os->os_phys_buf->b_data;
316                 bzero(os->os_phys, size);
317         }
319         /*
320          * Note: the changed_cb will be called once before the register
321          * func returns, thus changing the checksum/compression from the
322          * default (fletcher2/off).  Snapshots don't need to know about
323          * checksum/compression/copies.
324          */
325         if (ds) {
```



```
326                 err = dsl_prop_register(ds,
327                     zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
328                     primary_cache_changed_cb, os);
329                 if (err == 0) {
330                         err = dsl_prop_register(ds,
331                             zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
332                             secondary_cache_changed_cb, os);
333                 }
334                 if (!dsl_dataset_is_snapshot(ds)) {
335                         if (err == 0) {
336                                 err = dsl_prop_register(ds,
337                                     zfs_prop_to_name(ZFS_PROP_CHECKSUM),
338                                     checksum_changed_cb, os);
339                         }
340                         if (err == 0) {
341                                 err = dsl_prop_register(ds,
342                                     zfs_prop_to_name(ZFS_PROP_COMPRESSION),
343                                     compression_changed_cb, os);
344                         }
345                         if (err == 0) {
346                                 err = dsl_prop_register(ds,
347                                     zfs_prop_to_name(ZFS_PROP_COPIES),
348                                     copies_changed_cb, os);
349                         }
350                         if (err == 0) {
351                                 err = dsl_prop_register(ds,
352                                     zfs_prop_to_name(ZFS_PROP_DEDUP),
353                                     dedup_changed_cb, os);
354                         }
355                         if (err == 0) {
356                                 err = dsl_prop_register(ds,
357                                     zfs_prop_to_name(ZFS_PROP_LOGBIAS),
358                                     logbias_changed_cb, os);
359                         }
360                         if (err == 0) {
361                                 err = dsl_prop_register(ds,
362                                     zfs_prop_to_name(ZFS_PROP_SYNC),
363                                     sync_changed_cb, os);
364                         }
365                 }
366                 if (err != 0) {
367                         VERIFY(arc_buf_remove_ref(os->os_phys_buf,
368                             &os->os_phys_buf));
369                         kmem_free(os, sizeof (objset_t));
370                         return (err);
371                 }
372         } else if (ds == NULL) {
373                 /* It's the meta-objset. */
374                 os->os_checksum = ZIO_CHECKSUM_FLETCHER_4;
375                 os->os_compress = ZIO_COMPRESS_LZJB;
376                 os->os_copies = spa_max_replication(spa);
377                 os->os_dedup_checksum = ZIO_CHECKSUM_OFF;
378                 os->os_dedup_verify = 0;
379                 os->os_logbias = 0;
380                 os->os_sync = 0;
381                 os->os_primary_cache = ZFS_CACHE_ALL;
382                 os->os_secondary_cache = ZFS_CACHE_ALL;
383         }
385         if (ds == NULL || !dsl_dataset_is_snapshot(ds))
386                 os->os_zil_header = os->os_phys->os_zil_header;
387         os->os_zil = zil_alloc(os, &os->os_zil_header);
389         for (i = 0; i < TXG_SIZE; i++) {
390                 list_create(&os->os_dirty_dnodes[i], sizeof (dnode_t),
391                     offsetof(dnode_t, dn_dirty_link[i]));
```

```
392                 list_create(&os->os_free_dnodes[i], sizeof (dnode_t),
393                     offsetof(dnode_t, dn_dirty_link[i]));
394         }
395         list_create(&os->os_dnodes, sizeof (dnode_t),
396             offsetof(dnode_t, dn_link));
397         list_create(&os->os_downgraded_dbufs, sizeof (dmu_buf_impl_t),
398             offsetof(dmu_buf_impl_t, db_link));

400         mutex_init(&os->os_lock, NULL, MUTEX_DEFAULT, NULL);
401         mutex_init(&os->os_obj_lock, NULL, MUTEX_DEFAULT, NULL);
402         mutex_init(&os->os_user_ptr_lock, NULL, MUTEX_DEFAULT, NULL);

404         DMU_META_DNODE(os) = dnode_special_open(os,
405             &os->os_phys->os_meta_dnode, DMU_META_DNODE_OBJECT,
406             &os->os_meta_dnode);
407         if (arc_buf_size(os->os_phys_buf) >= sizeof (objset_phys_t)) {
408                 DMU_USERUSED_DNODE(os) = dnode_special_open(os,
409                     &os->os_phys->os_userused_dnode, DMU_USERUSED_OBJECT,
410                     &os->os_userused_dnode);
411                 DMU_GROUPUSED_DNODE(os) = dnode_special_open(os,
412                     &os->os_phys->os_groupused_dnode, DMU_GROUPUSED_OBJECT,
413                     &os->os_groupused_dnode);
414         }

416         /*
417          * We should be the only thread trying to do this because we
418          * have ds_opening_lock
419          */
420         if (ds) {
421                 mutex_enter(&ds->ds_lock);
422                 ASSERT(ds->ds_objset == NULL);
423                 ds->ds_objset = os;
424                 mutex_exit(&ds->ds_lock);
425         }

427         *osp = os;
428         return (0);
429 }

431 int
432 dmu_objset_from_ds(dsl_dataset_t *ds, objset_t **osp)
433 {
434         int err = 0;

436         mutex_enter(&ds->ds_opening_lock);
437         *osp = ds->ds_objset;
438         if (*osp == NULL) {
439                 err = dmu_objset_open_impl(dsl_dataset_get_spa(ds),
440                     ds, dsl_dataset_get_blkptr(ds), osp);
441         }
442         mutex_exit(&ds->ds_opening_lock);
443         return (err);
444 }

446 /*
447  * Holds the pool while the objset is held.  Therefore only one objset
448  * can be held at a time.
449  */
450 int
451 dmu_objset_hold(const char *name, void *tag, objset_t **osp)
452 {
453         dsl_pool_t *dp;
454         dsl_dataset_t *ds;
455         int err;

457         err = dsl_pool_hold(name, tag, &dp);
```



```
458         if (err != 0)
459                 return (err);
460         err = dsl_dataset_hold(dp, name, tag, &ds);
461         if (err != 0) {
462                 dsl_pool_rele(dp, tag);
463                 return (err);
464         }

466         err = dmu_objset_from_ds(ds, osp);
467         if (err != 0) {
468                 dsl_dataset_rele(ds, tag);
469                 dsl_pool_rele(dp, tag);
470         }

472         return (err);
473 }

475 /*
476  * dsl_pool must not be held when this is called.
477  * Upon successful return, there will be a longhold on the dataset,
478  * and the dsl_pool will not be held.
479  */
480 int
481 dmu_objset_own(const char *name, dmu_objset_type_t type,
482     boolean_t readonly, void *tag, objset_t **osp)
483 {
484         dsl_pool_t *dp;
485         dsl_dataset_t *ds;
486         int err;

488         err = dsl_pool_hold(name, FTAG, &dp);
489         if (err != 0)
490                 return (err);
491         err = dsl_dataset_own(dp, name, tag, &ds);
492         if (err != 0) {
493                 dsl_pool_rele(dp, FTAG);
494                 return (err);
495         }

497         err = dmu_objset_from_ds(ds, osp);
498         dsl_pool_rele(dp, FTAG);
499         if (err != 0) {
500                 dsl_dataset_disown(ds, tag);
501         } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
502                 dsl_dataset_disown(ds, tag);
503                 return (SET_ERROR(EINVAL));
504         } else if (!readonly && dsl_dataset_is_snapshot(ds)) {
505                 dsl_dataset_disown(ds, tag);
506                 return (SET_ERROR(EROFS));
507         }
508         return (err);
509 }

511 void
512 dmu_objset_rele(objset_t *os, void *tag)
513 {
514         dsl_pool_t *dp = dmu_objset_pool(os);
515         dsl_dataset_rele(os->os_dsl_dataset, tag);
516         dsl_pool_rele(dp, tag);
517 }

519 void
520 dmu_objset_disown(objset_t *os, void *tag)
521 {
522         dsl_dataset_disown(os->os_dsl_dataset, tag);
523 }
```

```
525 void
526 dmu_objset_evict_dbufs(objset_t *os)
527 {
528         dnode_t *dn;

530         mutex_enter(&os->os_lock);

532         /* process the mdn last, since the other dnodes have holds on it */
533         list_remove(&os->os_dnodes, DMU_META_DNODE(os));
534         list_insert_tail(&os->os_dnodes, DMU_META_DNODE(os));

536         /*
537          * Find the first dnode with holds.  We have to do this dance
538          * because dnode_add_ref() only works if you already have a
539          * hold.  If there are no holds then it has no dbufs so OK to
540          * skip.
541          */
542         for (dn = list_head(&os->os_dnodes);
543             dn && !dnode_add_ref(dn, FTAG);
544             dn = list_next(&os->os_dnodes, dn))
545                 continue;

547         while (dn) {
548                 dnode_t *next_dn = dn;

550                 do {
551                         next_dn = list_next(&os->os_dnodes, next_dn);
552                 } while (next_dn && !dnode_add_ref(next_dn, FTAG));

554                 mutex_exit(&os->os_lock);
555                 dnode_evict_dbufs(dn);
556                 dnode_rele(dn, FTAG);
557                 mutex_enter(&os->os_lock);
558                 dn = next_dn;
559         }
560         mutex_exit(&os->os_lock);
561 }

563 void
564 dmu_objset_evict(objset_t *os)
565 {
566         dsl_dataset_t *ds = os->os_dsl_dataset;

568         for (int t = 0; t < TXG_SIZE; t++)
569                 ASSERT(!dmu_objset_is_dirty(os, t));

571         if (ds) {
572                 if (!dsl_dataset_is_snapshot(ds)) {
573                         VERIFY0(dsl_prop_unregister(ds,
574                             zfs_prop_to_name(ZFS_PROP_CHECKSUM),
575                             checksum_changed_cb, os));
576                         VERIFY0(dsl_prop_unregister(ds,
577                             zfs_prop_to_name(ZFS_PROP_COMPRESSION),
578                             compression_changed_cb, os));
579                         VERIFY0(dsl_prop_unregister(ds,
580                             zfs_prop_to_name(ZFS_PROP_COPIES),
581                             copies_changed_cb, os));
582                         VERIFY0(dsl_prop_unregister(ds,
583                             zfs_prop_to_name(ZFS_PROP_DEDUP),
584                             dedup_changed_cb, os));
585                         VERIFY0(dsl_prop_unregister(ds,
586                             zfs_prop_to_name(ZFS_PROP_LOGBIAS),
587                             logbias_changed_cb, os));
588                         VERIFY0(dsl_prop_unregister(ds,
589                             zfs_prop_to_name(ZFS_PROP_SYNC),
```



```
590                             sync_changed_cb, os));
591                 }
592                 VERIFY0(dsl_prop_unregister(ds,
593                     zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
594                     primary_cache_changed_cb, os));
595                 VERIFY0(dsl_prop_unregister(ds,
596                     zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
597                     secondary_cache_changed_cb, os));
598         }

600         if (os->os_sa)
601                 sa_tear_down(os);

603         dmu_objset_evict_dbufs(os);

605         dnode_special_close(&os->os_meta_dnode);
606         if (DMU_USERUSED_DNODE(os)) {
607                 dnode_special_close(&os->os_userused_dnode);
608                 dnode_special_close(&os->os_groupused_dnode);
609         }
610         zil_free(os->os_zil);

612         ASSERT3P(list_head(&os->os_dnodes), ==, NULL);

614         VERIFY(arc_buf_remove_ref(os->os_phys_buf, &os->os_phys_buf));

616         /*
617          * This is a barrier to prevent the objset from going away in
618          * dnode_move() until we can safely ensure that the objset is still in
619          * use. We consider the objset valid before the barrier and invalid
620          * after the barrier.
621          */
622         rw_enter(&os_lock, RW_READER);
623         rw_exit(&os_lock);

625         mutex_destroy(&os->os_lock);
626         mutex_destroy(&os->os_obj_lock);
627         mutex_destroy(&os->os_user_ptr_lock);
628         kmem_free(os, sizeof (objset_t));
629 }

631 timestruc_t
632 dmu_objset_snap_cmtime(objset_t *os)
633 {
634         return (dsl_dir_snap_cmtime(os->os_dsl_dataset->ds_dir));
635 }

637 /* called from dsl for meta-objset */
638 objset_t *
639 dmu_objset_create_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
640     dmu_objset_type_t type, dmu_tx_t *tx)
641 {
642         objset_t *os;
643         dnode_t *mdn;

645         ASSERT(dmu_tx_is_syncing(tx));

647         if (ds != NULL)
648                 VERIFY0(dmu_objset_from_ds(ds, &os));
649         else
650                 VERIFY0(dmu_objset_open_impl(spa, NULL, bp, &os));

652         mdn = DMU_META_DNODE(os);

654         dnode_allocate(mdn, DMU_OT_DNODE, 1 << DNODE_BLOCK_SHIFT,
655             DN_MAX_INDBLKSHIFT, DMU_OT_NONE, 0, tx);
```

```
657         /*
658          * We don't want to have to increase the meta-dnode's nlevels
659          * later, because then we could do it in quescing context while
660          * we are also accessing it in open context.
661          *
662          * This precaution is not necessary for the MOS (ds == NULL),
663          * because the MOS is only updated in syncing context.
664          * This is most fortunate: the MOS is the only objset that
665          * needs to be synced multiple times as spa_sync() iterates
666          * to convergence, so minimizing its dn_nlevels matters.
667          */
668         if (ds != NULL) {
669                 int levels = 1;

671                 /*
672                  * Determine the number of levels necessary for the meta-dnode
673                  * to contain DN_MAX_OBJECT dnodes.
674                  */
675                 while ((uint64_t)mdn->dn_nblkptr << (mdn->dn_datablkshift +
676                     (levels - 1) * (mdn->dn_indblkshift - SPA_BLKPTRSHIFT)) <
677                     DN_MAX_OBJECT * sizeof (dnode_phys_t))
678                         levels++;

680                 mdn->dn_next_nlevels[tx->tx_txg & TXG_MASK] =
681                     mdn->dn_nlevels = levels;
682         }

684         ASSERT(type != DMU_OST_NONE);
685         ASSERT(type != DMU_OST_ANY);
686         ASSERT(type < DMU_OST_NUMTYPES);
687         os->os_phys->os_type = type;
688         if (dmu_objset_userused_enabled(os)) {
689                 os->os_phys->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
690                 os->os_flags = os->os_phys->os_flags;
691         }

693         dsl_dataset_dirty(ds, tx);

695         return (os);
696 }

698 typedef struct dmu_objset_create_arg {
699         const char *doca_name;
700         cred_t *doca_cred;
701         void (*doca_userfunc)(objset_t *os, void *arg,
702             cred_t *cr, dmu_tx_t *tx);
703         void *doca_userarg;
704         dmu_objset_type_t doca_type;
705         uint64_t doca_flags;
706 } dmu_objset_create_arg_t;

708 /*ARGSUSED*/
709 static int
710 dmu_objset_create_check(void *arg, dmu_tx_t *tx)
711 {
712         dmu_objset_create_arg_t *doca = arg;
713         dsl_pool_t *dp = dmu_tx_pool(tx);
714         dsl_dir_t *pdd;
715         const char *tail;
716         int error;

718         if (strchr(doca->doca_name, '@') != NULL)
719                 return (SET_ERROR(EINVAL));

721         error = dsl_dir_hold(dp, doca->doca_name, FTAG, &pdd, &tail);
```



```
722         if (error != 0)
723                 return (error);
724         if (tail == NULL) {
725                 dsl_dir_rele(pdd, FTAG);
726                 return (SET_ERROR(EEXIST));
727         }
728         dsl_dir_rele(pdd, FTAG);

730         return (0);
731 }

733 static void
734 dmu_objset_create_sync(void *arg, dmu_tx_t *tx)
735 {
736         dmu_objset_create_arg_t *doca = arg;
737         dsl_pool_t *dp = dmu_tx_pool(tx);
738         dsl_dir_t *pdd;
739         const char *tail;
740         dsl_dataset_t *ds;
741         uint64_t obj;
742         blkptr_t *bp;
743         objset_t *os;

745         VERIFY0(dsl_dir_hold(dp, doca->doca_name, FTAG, &pdd, &tail));

747         obj = dsl_dataset_create_sync(pdd, tail, NULL, doca->doca_flags,
748             doca->doca_cred, tx);

750         VERIFY0(dsl_dataset_hold_obj(pdd->dd_pool, obj, FTAG, &ds));
751         bp = dsl_dataset_get_blkptr(ds);
752         os = dmu_objset_create_impl(pdd->dd_pool->dp_spa,
753             ds, bp, doca->doca_type, tx);

755         if (doca->doca_userfunc != NULL) {
756                 doca->doca_userfunc(os, doca->doca_userarg,
757                     doca->doca_cred, tx);
758         }

760         spa_history_log_internal_ds(ds, "create", tx, "");
761         dsl_dataset_rele(ds, FTAG);
762         dsl_dir_rele(pdd, FTAG);
763 }

765 int
766 dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
767     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg)
768 {
769         dmu_objset_create_arg_t doca;

771         doca.doca_name = name;
772         doca.doca_cred = CRED();
773         doca.doca_flags = flags;
774         doca.doca_userfunc = func;
775         doca.doca_userarg = arg;
776         doca.doca_type = type;

778         return (dsl_sync_task(name,
779             dmu_objset_create_check, dmu_objset_create_sync, &doca, 5));
780 }

782 typedef struct dmu_objset_clone_arg {
783         const char *doca_clone;
784         const char *doca_origin;
785         cred_t *doca_cred;
786 } dmu_objset_clone_arg_t;
```

```
/*ARGSUSED*/
static int
dmu_objset_clone_check(void *arg, dmu_tx_t *tx)
{
        dmu_objset_clone_arg_t *doca = arg;
        dsl_dir_t *pdd;
        const char *tail;
        int error;
        dsl_dataset_t *origin;
        dsl_pool_t *dp = dmu_tx_pool(tx);

        if (strchr(doca->doca_clone, '@') != NULL)
                return (SET_ERROR(EINVAL));

        error = dsl_dir_hold(dp, doca->doca_clone, FTAG, &pdd, &tail);
        if (error != 0)
                return (error);
        if (tail == NULL) {
                dsl_dir_rele(pdd, FTAG);
                return (SET_ERROR(EEXIST));
        }
        /* You can't clone across pools. */
        if (pdd->dd_pool != dp) {
                dsl_dir_rele(pdd, FTAG);
                return (SET_ERROR(EXDEV));
        }
        dsl_dir_rele(pdd, FTAG);

        error = dsl_dataset_hold(dp, doca->doca_origin, FTAG, &origin);
        if (error != 0)
                return (error);

        /* You can't clone across pools. */
        if (origin->ds_dir->dd_pool != dp) {
                dsl_dataset_rele(origin, FTAG);
                return (SET_ERROR(EXDEV));
        }

        /* You can only clone snapshots, not the head datasets. */
        if (!dsl_dataset_is_snapshot(origin)) {
                dsl_dataset_rele(origin, FTAG);
                return (SET_ERROR(EINVAL));
        }
        dsl_dataset_rele(origin, FTAG);

        return (0);
}

static void
dmu_objset_clone_sync(void *arg, dmu_tx_t *tx)
{
        dmu_objset_clone_arg_t *doca = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dir_t *pdd;
        const char *tail;
        dsl_dataset_t *origin, *ds;
        uint64_t obj;
        char namebuf[MAXNAMELEN];

        VERIFY0(dsl_dir_hold(dp, doca->doca_clone, FTAG, &pdd, &tail));
        VERIFY0(dsl_dataset_hold(dp, doca->doca_origin, FTAG, &origin));

        obj = dsl_dataset_create_sync(pdd, tail, origin, 0,
            doca->doca_cred, tx);

        VERIFY0(dsl_dataset_hold_obj(pdd->dd_pool, obj, FTAG, &ds));
```



```
        dsl_dataset_name(origin, namebuf);
        spa_history_log_internal_ds(ds, "clone", tx,
            "origin=%s (%llu)", namebuf, origin->ds_object);
        dsl_dataset_rele(ds, FTAG);
        dsl_dataset_rele(origin, FTAG);
        dsl_dir_rele(pdd, FTAG);
}

int
dmu_objset_clone(const char *clone, const char *origin)
{
        dmu_objset_clone_arg_t doca;

        doca.doca_clone = clone;
        doca.doca_origin = origin;
        doca.doca_cred = CRED();

        return (dsl_sync_task(clone,
            dmu_objset_clone_check, dmu_objset_clone_sync, &doca, 5));
}

int
dmu_objset_snapshot_one(const char *fsname, const char *snapname)
{
        int err;
        char *longsnap = kmem_asprintf("%s@%s", fsname, snapname);
        nvlist_t *snaps = fnvlist_alloc();

        fnvlist_add_boolean(snaps, longsnap);
        strfree(longsnap);
        err = dsl_dataset_snapshot(snaps, NULL, NULL);
        fnvlist_free(snaps);
        return (err);
}

static void
dmu_objset_sync_dnodes(list_t *list, list_t *newlist, dmu_tx_t *tx)
{
        dnode_t *dn;

        while (dn = list_head(list)) {
                ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
                ASSERT(dn->dn_dbuf->db_data_pending);
                /*
                 * Initialize dn_zio outside dnode_sync() because the
                 * meta-dnode needs to set it ouside dnode_sync().
                 */
                dn->dn_zio = dn->dn_dbuf->db_data_pending->dr_zio;
                ASSERT(dn->dn_zio);

                ASSERT3U(dn->dn_nlevels, <=, DN_MAX_LEVELS);
                list_remove(list, dn);

                if (newlist) {
                        (void) dnode_add_ref(dn, newlist);
                        list_insert_tail(newlist, dn);
                }

                dnode_sync(dn, tx);
        }
}

/* ARGSUSED */
static void
dmu_objset_write_ready(zio_t *zio, arc_buf_t *abuf, void *arg)
{
```

```
        blkptr_t *bp = zio->io_bp;
        objset_t *os = arg;
        dnode_phys_t *dnp = &os->os_phys->os_meta_dnode;

        ASSERT3P(bp, ==, os->os_rootbp);
        ASSERT3U(BP_GET_TYPE(bp), ==, DMU_OT_OBJSET);
        ASSERT0(BP_GET_LEVEL(bp));

        /*
         * Update rootbp fill count: it should be the number of objects
         * allocated in the object set (not counting the "special"
         * objects that are stored in the objset_phys_t -- the meta
         * dnode and user/group accounting objects).
         */
        bp->blk_fill = 0;
        for (int i = 0; i < dnp->dn_nblkptr; i++)
                bp->blk_fill += dnp->dn_blkptr[i].blk_fill;
}

/* ARGSUSED */
static void
dmu_objset_write_done(zio_t *zio, arc_buf_t *abuf, void *arg)
{
        blkptr_t *bp = zio->io_bp;
        blkptr_t *bp_orig = &zio->io_bp_orig;
        objset_t *os = arg;

        if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {
                ASSERT(BP_EQUAL(bp, bp_orig));
        } else {
                dsl_dataset_t *ds = os->os_dsl_dataset;
                dmu_tx_t *tx = os->os_synctx;

                (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
                dsl_dataset_block_born(ds, bp, tx);
        }
}

/* called from dsl */
void
dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
{
        int txgoff;
        zbookmark_t zb;
        zio_prop_t zp;
        zio_t *zio;
        list_t *list;
        list_t *newlist = NULL;
        dbuf_dirty_record_t *dr;

        dprintf_ds(os->os_dsl_dataset, "txg=%llu\n", tx->tx_txg);

        ASSERT(dmu_tx_is_syncing(tx));
        /* XXX the write_done callback should really give us the tx... */
        os->os_synctx = tx;

        if (os->os_dsl_dataset == NULL) {
                /*
                 * This is the MOS.  If we have upgraded,
                 * spa_max_replication() could change, so reset
                 * os_copies here.
                 */
                os->os_copies = spa_max_replication(os->os_spa);
        }

        /*
```



```
         * Create the root block IO
         */
        SET_BOOKMARK(&zb, os->os_dsl_dataset ?
            os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
            ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
        arc_release(os->os_phys_buf, &os->os_phys_buf);

        dmu_write_policy(os, NULL, 0, 0, &zp, tx->tx_txg);
        dmu_write_policy(os, NULL, 0, 0, &zp);

        zio = arc_write(pio, os->os_spa, tx->tx_txg,
            os->os_rootbp, os->os_phys_buf, DMU_OS_IS_L2CACHEABLE(os), &zp,
            dmu_objset_write_ready, dmu_objset_write_done, os,
            ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);

        /*
         * Sync special dnodes - the parent IO for the sync is the root block
         */
        DMU_META_DNODE(os)->dn_zio = zio;
        dnode_sync(DMU_META_DNODE(os), tx);

        os->os_phys->os_flags = os->os_flags;

        if (DMU_USERUSED_DNODE(os) &&
            DMU_USERUSED_DNODE(os)->dn_type != DMU_OT_NONE) {
                DMU_USERUSED_DNODE(os)->dn_zio = zio;
                dnode_sync(DMU_USERUSED_DNODE(os), tx);
                DMU_GROUPUSED_DNODE(os)->dn_zio = zio;
                dnode_sync(DMU_GROUPUSED_DNODE(os), tx);
        }

        txgoff = tx->tx_txg & TXG_MASK;

        if (dmu_objset_userused_enabled(os)) {
                newlist = &os->os_synced_dnodes;
                /*
                 * We must create the list here because it uses the
                 * dn_dirty_link[] of this txg.
                 */
                list_create(newlist, sizeof (dnode_t),
                    offsetof(dnode_t, dn_dirty_link[txgoff]));
        }

        dmu_objset_sync_dnodes(&os->os_free_dnodes[txgoff], newlist, tx);
        dmu_objset_sync_dnodes(&os->os_dirty_dnodes[txgoff], newlist, tx);

        list = &DMU_META_DNODE(os)->dn_dirty_records[txgoff];
        while (dr = list_head(list)) {
                ASSERT0(dr->dr_dbuf->db_level);
                list_remove(list, dr);
                if (dr->dr_zio)
                        zio_nowait(dr->dr_zio);
        }
        /*
         * Free intent log blocks up to this tx.
         */
        zil_sync(os->os_zil, tx);
        os->os_phys->os_zil_header = os->os_zil_header;
        zio_nowait(zio);
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   28641 Wed May  1 11:13:44 2013
new/usr/src/uts/common/fs/zfs/sys/dmu.h
3756 want lz4 support for metadata compression
**********************************************************
```

```
/*
 * 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 (c) 2012 by Delphix. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 Martin Matuska. All rights reserved.
#endif /* ! codereview */
 */

/* Portions Copyright 2010 Robert Milkowski */

#ifndef _SYS_DMU_H
#define _SYS_DMU_H

/*
 * This file describes the interface that the DMU provides for its
 * consumers.
 *
 * The DMU also interacts with the SPA.  That interface is described in
 * dmu_spa.h.
 */

#include <sys/inttypes.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/cred.h>
#include <sys/time.h>
#include <sys/fs/zfs.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct uio;
struct xuio;
struct page;
struct vnode;
struct spa;
struct zilog;
struct zio;
```



```
struct blkptr;
struct zap_cursor;
struct dsl_dataset;
struct dsl_pool;
struct dnode;
struct drr_begin;
struct drr_end;
struct zbookmark;
struct spa;
struct nvlist;
struct arc_buf;
struct zio_prop;
struct sa_handle;

typedef struct objset objset_t;
typedef struct dmu_tx dmu_tx_t;
typedef struct dsl_dir dsl_dir_t;

typedef enum dmu_object_byteswap {
        DMU_BSWAP_UINT8,
        DMU_BSWAP_UINT16,
        DMU_BSWAP_UINT32,
        DMU_BSWAP_UINT64,
        DMU_BSWAP_ZAP,
        DMU_BSWAP_DNODE,
        DMU_BSWAP_OBJSET,
        DMU_BSWAP_ZNODE,
        DMU_BSWAP_OLDACL,
        DMU_BSWAP_ACL,
        /*
         * Allocating a new byteswap type number makes the on-disk format
         * incompatible with any other format that uses the same number.
         *
         * Data can usually be structured to work with one of the
         * DMU_BSWAP_UINT* or DMU_BSWAP_ZAP types.
         */
        DMU_BSWAP_NUMFUNCS
} dmu_object_byteswap_t;

#define DMU_OT_NEWTYPE 0x80
#define DMU_OT_METADATA 0x40
#define DMU_OT_BYTESWAP_MASK 0x3f

/*
 * Defines a uint8_t object type. Object types specify if the data
 * in the object is metadata (boolean) and how to byteswap the data
 * (dmu_object_byteswap_t).
 */
#define DMU_OT(byteswap, metadata) \
        (DMU_OT_NEWTYPE | \
        ((metadata) ? DMU_OT_METADATA : 0) | \
        ((byteswap) & DMU_OT_BYTESWAP_MASK))

#define DMU_OT_IS_VALID(ot) (((ot) & DMU_OT_NEWTYPE) ? \
        ((ot) & DMU_OT_BYTESWAP_MASK) < DMU_BSWAP_NUMFUNCS : \
        (ot) < DMU_OT_NUMTYPES)

#define DMU_OT_IS_METADATA(ot) (((ot) & DMU_OT_NEWTYPE) ? \
        ((ot) & DMU_OT_METADATA) : \
        dmu_ot[(ot)].ot_metadata)

#define DMU_OT_BYTESWAP(ot) (((ot) & DMU_OT_NEWTYPE) ? \
        ((ot) & DMU_OT_BYTESWAP_MASK) : \
        dmu_ot[(ot)].ot_byteswap)

typedef enum dmu_object_type {
```

```
128         DMU_OT_NONE,
129         /* general: */
130         DMU_OT_OBJECT_DIRECTORY,        /* ZAP */
131         DMU_OT_OBJECT_ARRAY,            /* UINT64 */
132         DMU_OT_PACKED_NVLIST,           /* UINT8 (XDR by nvlist_pack/unpack) */
133         DMU_OT_PACKED_NVLIST_SIZE,      /* UINT64 */
134         DMU_OT_BPOBJ,                   /* UINT64 */
135         DMU_OT_BPOBJ_HDR,               /* UINT64 */
136         /* spa: */
137         DMU_OT_SPACE_MAP_HEADER,        /* UINT64 */
138         DMU_OT_SPACE_MAP,               /* UINT64 */
139         /* zil: */
140         DMU_OT_INTENT_LOG,              /* UINT64 */
141         /* dmu: */
142         DMU_OT_DNODE,                   /* DNODE */
143         DMU_OT_OBJSET,                  /* OBJSET */
144         /* dsl: */
145         DMU_OT_DSL_DIR,                 /* UINT64 */
146         DMU_OT_DSL_DIR_CHILD_MAP,       /* ZAP */
147         DMU_OT_DSL_DS_SNAP_MAP,         /* ZAP */
148         DMU_OT_DSL_PROPS,               /* ZAP */
149         DMU_OT_DSL_DATASET,             /* UINT64 */
150         /* zpl: */
151         DMU_OT_ZNODE,                   /* ZNODE */
152         DMU_OT_OLDACL,                  /* Old ACL */
153         DMU_OT_PLAIN_FILE_CONTENTS,     /* UINT8 */
154         DMU_OT_DIRECTORY_CONTENTS,      /* ZAP */
155         DMU_OT_MASTER_NODE,             /* ZAP */
156         DMU_OT_UNLINKED_SET,            /* ZAP */
157         /* zvol: */
158         DMU_OT_ZVOL,                    /* UINT8 */
159         DMU_OT_ZVOL_PROP,               /* ZAP */
160         /* other; for testing only! */
161         DMU_OT_PLAIN_OTHER,             /* UINT8 */
162         DMU_OT_UINT64_OTHER,            /* UINT64 */
163         DMU_OT_ZAP_OTHER,               /* ZAP */
164         /* new object types: */
165         DMU_OT_ERROR_LOG,               /* ZAP */
166         DMU_OT_SPA_HISTORY,             /* UINT8 */
167         DMU_OT_SPA_HISTORY_OFFSETS,     /* spa_his_phys_t */
168         DMU_OT_POOL_PROPS,              /* ZAP */
169         DMU_OT_DSL_PERMS,               /* ZAP */
170         DMU_OT_ACL,                     /* ACL */
171         DMU_OT_SYSACL,                  /* SYSACL */
172         DMU_OT_FUID,                    /* FUID table (Packed NVLIST UINT8) */
173         DMU_OT_FUID_SIZE,               /* FUID table size UINT64 */
174         DMU_OT_NEXT_CLONES,             /* ZAP */
175         DMU_OT_SCAN_QUEUE,              /* ZAP */
176         DMU_OT_USERGROUP_USED,          /* ZAP */
177         DMU_OT_USERGROUP_QUOTA,         /* ZAP */
178         DMU_OT_USERREFS,                /* ZAP */
179         DMU_OT_DDT_ZAP,                 /* ZAP */
180         DMU_OT_DDT_STATS,               /* ZAP */
181         DMU_OT_SA,                      /* System attr */
182         DMU_OT_SA_MASTER_NODE,          /* ZAP */
183         DMU_OT_SA_ATTR_REGISTRATION,    /* ZAP */
184         DMU_OT_SA_ATTR_LAYOUTS,         /* ZAP */
185         DMU_OT_SCAN_XLATE,              /* ZAP */
186         DMU_OT_DEDUP,                   /* fake dedup BP from ddt_bp_create() */
187         DMU_OT_DEADLIST,                /* ZAP */
188         DMU_OT_DEADLIST_HDR,            /* UINT64 */
189         DMU_OT_DSL_CLONES,              /* ZAP */
190         DMU_OT_BPOBJ_SUBOBJ,            /* UINT64 */
191         /*
192          * Do not allocate new object types here. Doing so makes the on-disk
193          * format incompatible with any other format that uses the same object
```



```
194          * type number.
195          *
196          * When creating an object which does not have one of the above types
197          * use the DMU_OTN_* type with the correct byteswap and metadata
198          * values.
199          *
200          * The DMU_OTN_* types do not have entries in the dmu_ot table,
201          * use the DMU_OT_IS_METDATA() and DMU_OT_BYTESWAP() macros instead
202          * of indexing into dmu_ot directly (this works for both DMU_OT_* types
203          * and DMU_OTN_* types).
204          */
205         DMU_OT_NUMTYPES,

207         /*
208          * Names for valid types declared with DMU_OT().
209          */
210         DMU_OTN_UINT8_DATA = DMU_OT(DMU_BSWAP_UINT8, B_FALSE),
211         DMU_OTN_UINT8_METADATA = DMU_OT(DMU_BSWAP_UINT8, B_TRUE),
212         DMU_OTN_UINT16_DATA = DMU_OT(DMU_BSWAP_UINT16, B_FALSE),
213         DMU_OTN_UINT16_METADATA = DMU_OT(DMU_BSWAP_UINT16, B_TRUE),
214         DMU_OTN_UINT32_DATA = DMU_OT(DMU_BSWAP_UINT32, B_FALSE),
215         DMU_OTN_UINT32_METADATA = DMU_OT(DMU_BSWAP_UINT32, B_TRUE),
216         DMU_OTN_UINT64_DATA = DMU_OT(DMU_BSWAP_UINT64, B_FALSE),
217         DMU_OTN_UINT64_METADATA = DMU_OT(DMU_BSWAP_UINT64, B_TRUE),
218         DMU_OTN_ZAP_DATA = DMU_OT(DMU_BSWAP_ZAP, B_FALSE),
219         DMU_OTN_ZAP_METADATA = DMU_OT(DMU_BSWAP_ZAP, B_TRUE),
220 } dmu_object_type_t;

222 typedef enum txg_how {
223         TXG_WAIT = 1,
224         TXG_NOWAIT,
225 } txg_how_t;

227 void byteswap_uint64_array(void *buf, size_t size);
228 void byteswap_uint32_array(void *buf, size_t size);
229 void byteswap_uint16_array(void *buf, size_t size);
230 void byteswap_uint8_array(void *buf, size_t size);
231 void zap_byteswap(void *buf, size_t size);
232 void zfs_oldacl_byteswap(void *buf, size_t size);
233 void zfs_acl_byteswap(void *buf, size_t size);
234 void zfs_znode_byteswap(void *buf, size_t size);

236 #define DS_FIND_SNAPSHOTS       (1<<0)
237 #define DS_FIND_CHILDREN        (1<<1)

239 /*
240  * The maximum number of bytes that can be accessed as part of one
241  * operation, including metadata.
242  */
243 #define DMU_MAX_ACCESS (10<<20) /* 10MB */
244 #define DMU_MAX_DELETEBLKCNT (20480) /* ~5MB of indirect blocks */

246 #define DMU_USERUSED_OBJECT     (-1ULL)
247 #define DMU_GROUPUSED_OBJECT    (-2ULL)
248 #define DMU_DEADLIST_OBJECT     (-3ULL)

250 /*
251  * artificial blkids for bonus buffer and spill blocks
252  */
253 #define DMU_BONUS_BLKID         (-1ULL)
254 #define DMU_SPILL_BLKID         (-2ULL)
255 /*
256  * Public routines to create, destroy, open, and close objsets.
257  */
258 int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
259 int dmu_objset_own(const char *name, dmu_objset_type_t type,
```

```
260     boolean_t readonly, void *tag, objset_t **osp);
261 void dmu_objset_rele(objset_t *os, void *tag);
262 void dmu_objset_disown(objset_t *os, void *tag);
263 int dmu_objset_open_ds(struct dsl_dataset *ds, objset_t **osp);

265 void dmu_objset_evict_dbufs(objset_t *os);
266 int dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
267     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg);
268 int dmu_objset_clone(const char *name, const char *origin);
269 int dsl_destroy_snapshots_nvl(struct nvlist *snaps, boolean_t defer,
270     struct nvlist *errlist);
271 int dmu_objset_snapshot_one(const char *fsname, const char *snapname);
272 int dmu_objset_snapshot_tmp(const char *, const char *, int);
273 int dmu_objset_find(char *name, int func(const char *, void *), void *arg,
274     int flags);
275 void dmu_objset_byteswap(void *buf, size_t size);
276 int dsl_dataset_rename_snapshot(const char *fsname,
277     const char *oldsnapname, const char *newsnapname, boolean_t recursive);

279 typedef struct dmu_buf {
280         uint64_t db_object;             /* object that this buffer is part of */
281         uint64_t db_offset;             /* byte offset in this object */
282         uint64_t db_size;               /* size of buffer in bytes */
283         void *db_data;                  /* data in buffer */
284 } dmu_buf_t;

286 typedef void dmu_buf_evict_func_t(struct dmu_buf *db, void *user_ptr);

288 /*
289  * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
290  */
291 #define DMU_POOL_DIRECTORY_OBJECT       1
292 #define DMU_POOL_CONFIG                 "config"
293 #define DMU_POOL_FEATURES_FOR_WRITE     "features_for_write"
294 #define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
295 #define DMU_POOL_FEATURE_DESCRIPTIONS   "feature_descriptions"
296 #define DMU_POOL_ROOT_DATASET           "root_dataset"
297 #define DMU_POOL_SYNC_BPOBJ             "sync_bplist"
298 #define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
299 #define DMU_POOL_ERRLOG_LAST            "errlog_last"
300 #define DMU_POOL_SPARES                 "spares"
301 #define DMU_POOL_DEFLATE                "deflate"
302 #define DMU_POOL_HISTORY                "history"
303 #define DMU_POOL_PROPS                  "pool_props"
304 #define DMU_POOL_L2CACHE                "l2cache"
305 #define DMU_POOL_TMP_USERREFS           "tmp_userrefs"
306 #define DMU_POOL_DDT                    "DDT-%s-%s-%s"
307 #define DMU_POOL_DDT_STATS              "DDT-statistics"
308 #define DMU_POOL_CREATION_VERSION       "creation_version"
309 #define DMU_POOL_SCAN                   "scan"
310 #define DMU_POOL_FREE_BPOBJ             "free_bpobj"
311 #define DMU_POOL_BPTREE_OBJ             "bptree_obj"
312 #define DMU_POOL_EMPTY_BPOBJ            "empty_bpobj"

314 /*
315  * Allocate an object from this objset.  The range of object numbers
316  * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
317  *
318  * The transaction must be assigned to a txg.  The newly allocated
319  * object will be "held" in the transaction (ie. you can modify the
320  * newly allocated object in this transaction).
321  *
322  * dmu_object_alloc() chooses an object and returns it in *objectp.
323  *
324  * dmu_object_claim() allocates a specific object number.  If that
325  * number is already allocated, it fails and returns EEXIST.
```



```
326  *
327  * Return 0 on success, or ENOSPC or EEXIST as specified above.
328  */
329 uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
330     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
331 int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
332     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
333 int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
334     int blocksize, dmu_object_type_t bonustype, int bonuslen);

336 /*
337  * Free an object from this objset.
338  *
339  * The object's data will be freed as well (ie. you don't need to call
340  * dmu_free(object, 0, -1, tx)).
341  *
342  * The object need not be held in the transaction.
343  *
344  * If there are any holds on this object's buffers (via dmu_buf_hold()),
345  * or tx holds on the object (via dmu_tx_hold_object()), you can not
346  * free it; it fails and returns EBUSY.
347  *
348  * If the object is not allocated, it fails and returns ENOENT.
349  *
350  * Return 0 on success, or EBUSY or ENOENT as specified above.
351  */
352 int dmu_object_free(objset_t *os, uint64_t object, dmu_tx_t *tx);

354 /*
355  * Find the next allocated or free object.
356  *
357  * The objectp parameter is in-out.  It will be updated to be the next
358  * object which is allocated.  Ignore objects which have not been
359  * modified since txg.
360  *
361  * XXX Can only be called on a objset with no dirty data.
362  *
363  * Returns 0 on success, or ENOENT if there are no more objects.
364  */
365 int dmu_object_next(objset_t *os, uint64_t *objectp,
366     boolean_t hole, uint64_t txg);

368 /*
369  * Set the data blocksize for an object.
370  *
371  * The object cannot have any blocks allcated beyond the first.  If
372  * the first block is allocated already, the new size must be greater
373  * than the current block size.  If these conditions are not met,
374  * ENOTSUP will be returned.
375  *
376  * Returns 0 on success, or EBUSY if there are any holds on the object
377  * contents, or ENOTSUP as described above.
378  */
379 int dmu_object_set_blocksize(objset_t *os, uint64_t object, uint64_t size,
380     int ibs, dmu_tx_t *tx);

382 /*
383  * Set the checksum property on a dnode.  The new checksum algorithm will
384  * apply to all newly written blocks; existing blocks will not be affected.
385  */
386 void dmu_object_set_checksum(objset_t *os, uint64_t object, uint8_t checksum,
387     dmu_tx_t *tx);

389 /*
390  * Set the compress property on a dnode.  The new compression algorithm will
391  * 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);

/*
 * 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, uint64_t txg);
    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().
 */
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);
void dmu_buf_add_ref(dmu_buf_t *db, 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, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp);
void dmu_buf_rele_array(dmu_buf_t **, int numbufs, void *tag);

/*
 * Returns NULL on success, or the existing user ptr if it's already
 * been set.
 *
 * user_ptr is for use by the user and can be obtained via dmu_buf_get_user().
 *
 * user_data_ptr_ptr should be NULL, or a pointer to a pointer which
 * will be set to db->db_data when you are allowed to access it.  Note
 * that db->db_data (the pointer) can change when you do dmu_buf_read(),
 * dmu_buf_tryupgrade(), dmu_buf_will_dirty(), or dmu_buf_will_fill().
 * *user_data_ptr_ptr will be set to the new value when it changes.
 *
 * If non-NULL, pageout func will be called when this buffer is being
 * excised from the cache, so that you can clean up the data structure
 * pointed to by user_ptr.
 *
 * dmu_evict_user() will call the pageout func for all buffers in a
 * objset with a given pageout func.
 */
void *dmu_buf_set_user(dmu_buf_t *db, void *user_ptr, void *user_data_ptr_ptr,
    dmu_buf_evict_func_t *pageout_func);
/*
 * set_user_ie is the same as set_user, but request immediate eviction
 * when hold count goes to zero.
 */
void *dmu_buf_set_user_ie(dmu_buf_t *db, void *user_ptr,
    void *user_data_ptr_ptr, dmu_buf_evict_func_t *pageout_func);
void *dmu_buf_update_user(dmu_buf_t *db_fake, void *old_user_ptr,
    void *user_ptr, void *user_data_ptr_ptr,
    dmu_buf_evict_func_t *pageout_func);
void dmu_evict_user(objset_t *os, dmu_buf_evict_func_t *func);

/*
 * Returns the user_ptr set with dmu_buf_set_user(), or NULL if not set.
 */
void *dmu_buf_get_user(dmu_buf_t *db);

/*
 * 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);

/*
 * 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_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_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 int zfs_prefetch_disable;

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

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_pad[5];
        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;
```

unchanged_portion_omitted_