```
**********************************************************
   47304 Fri Oct 31 10:14:51 2014
new/usr/src/uts/common/fs/zfs/dmu_objset.c
5269 zfs: zpool import slow
While importing a pool all objsets are enumerated twice, once to check
the zil log chains and once to claim them. On pools with many datasets
this process might take a substantial amount of time.
Speed up the process by parallelizing it utilizing a taskq. The number
of parallel tasks is limited to 4 times the number of leaf vdevs.
**********************************************************
```

```
/*
 * 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, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright (c) 2014, STRATO AG, Inc. 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>
#include <sys/vdev.h>
#endif /* ! codereview */

/*
 * 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);
}

zfs_sync_type_t
```

```
123 dmu_objset_syncprop(objset_t *os)
124 {
125         return (os->os_sync);
126 }
127
128 zfs_logbias_op_t
129 dmu_objset_logbias(objset_t *os)
130 {
131         return (os->os_logbias);
132 }
133
134 static void
135 checksum_changed_cb(void *arg, uint64_t newval)
136 {
137         objset_t *os = arg;
138
139         /*
140          * Inheritance should have been done by now.
141          */
142         ASSERT(newval != ZIO_CHECKSUM_INHERIT);
143
144         os->os_checksum = zio_checksum_select(newval, ZIO_CHECKSUM_ON_VALUE);
145 }
146
147 static void
148 compression_changed_cb(void *arg, uint64_t newval)
149 {
150         objset_t *os = arg;
151
152         /*
153          * Inheritance and range checking should have been done by now.
154          */
155         ASSERT(newval != ZIO_COMPRESS_INHERIT);
156
157         os->os_compress = zio_compress_select(newval, ZIO_COMPRESS_ON_VALUE);
158 }
159
160 static void
161 copies_changed_cb(void *arg, uint64_t newval)
162 {
163         objset_t *os = arg;
164
165         /*
166          * Inheritance and range checking should have been done by now.
167          */
168         ASSERT(newval > 0);
169         ASSERT(newval <= spa_max_replication(os->os_spa));
170
171         os->os_copies = newval;
172 }
173
174 static void
175 dedup_changed_cb(void *arg, uint64_t newval)
176 {
177         objset_t *os = arg;
178         spa_t *spa = os->os_spa;
179         enum zio_checksum checksum;
180
181         /*
182          * Inheritance should have been done by now.
183          */
184         ASSERT(newval != ZIO_CHECKSUM_INHERIT);
185
186         checksum = zio_checksum_dedup_select(spa, newval, ZIO_CHECKSUM_OFF);
187
188         os->os_dedup_checksum = checksum & ZIO_CHECKSUM_MASK;
```



```
189         os->os_dedup_verify = !!(checksum & ZIO_CHECKSUM_VERIFY);
190 }
191
192 static void
193 primary_cache_changed_cb(void *arg, uint64_t newval)
194 {
195         objset_t *os = arg;
196
197         /*
198          * Inheritance and range checking should have been done by now.
199          */
200         ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
201             newval == ZFS_CACHE_METADATA);
202
203         os->os_primary_cache = newval;
204 }
205
206 static void
207 secondary_cache_changed_cb(void *arg, uint64_t newval)
208 {
209         objset_t *os = arg;
210
211         /*
212          * Inheritance and range checking should have been done by now.
213          */
214         ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
215             newval == ZFS_CACHE_METADATA);
216
217         os->os_secondary_cache = newval;
218 }
219
220 static void
221 sync_changed_cb(void *arg, uint64_t newval)
222 {
223         objset_t *os = arg;
224
225         /*
226          * Inheritance and range checking should have been done by now.
227          */
228         ASSERT(newval == ZFS_SYNC_STANDARD || newval == ZFS_SYNC_ALWAYS ||
229             newval == ZFS_SYNC_DISABLED);
230
231         os->os_sync = newval;
232         if (os->os_zil)
233                 zil_set_sync(os->os_zil, newval);
234 }
235
236 static void
237 redundant_metadata_changed_cb(void *arg, uint64_t newval)
238 {
239         objset_t *os = arg;
240
241         /*
242          * Inheritance and range checking should have been done by now.
243          */
244         ASSERT(newval == ZFS_REDUNDANT_METADATA_ALL ||
245             newval == ZFS_REDUNDANT_METADATA_MOST);
246
247         os->os_redundant_metadata = newval;
248 }
249
250 static void
251 logbias_changed_cb(void *arg, uint64_t newval)
252 {
253         objset_t *os = arg;
```

```
255         ASSERT(newval == ZFS_LOGBIAS_LATENCY ||
256             newval == ZFS_LOGBIAS_THROUGHPUT);
257         os->os_logbias = newval;
258         if (os->os_zil)
259                 zil_set_logbias(os->os_zil, newval);
260 }

262 void
263 dmu_objset_byteswap(void *buf, size_t size)
264 {
265         objset_phys_t *osp = buf;

267         ASSERT(size == OBJSET_OLD_PHYS_SIZE || size == sizeof (objset_phys_t));
268         dnode_byteswap(&osp->os_meta_dnode);
269         byteswap_uint64_array(&osp->os_zil_header, sizeof (zil_header_t));
270         osp->os_type = BSWAP_64(osp->os_type);
271         osp->os_flags = BSWAP_64(osp->os_flags);
272         if (size == sizeof (objset_phys_t)) {
273                 dnode_byteswap(&osp->os_userused_dnode);
274                 dnode_byteswap(&osp->os_groupused_dnode);
275         }
276 }

278 int
279 dmu_objset_open_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
280     objset_t **osp)
281 {
282         objset_t *os;
283         int i, err;

285         ASSERT(ds == NULL || MUTEX_HELD(&ds->ds_opening_lock));

287         os = kmem_zalloc(sizeof (objset_t), KM_SLEEP);
288         os->os_dsl_dataset = ds;
289         os->os_spa = spa;
290         os->os_rootbp = bp;
291         if (!BP_IS_HOLE(os->os_rootbp)) {
292                 uint32_t aflags = ARC_WAIT;
293                 zbookmark_t zb;
294                 SET_BOOKMARK(&zb, ds ? ds->ds_object : DMU_META_OBJSET,
295                     ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);

297                 if (DMU_OS_IS_L2CACHEABLE(os))
298                         aflags |= ARC_L2CACHE;
299                 if (DMU_OS_IS_L2COMPRESSIBLE(os))
300                         aflags |= ARC_L2COMPRESS;

302                 dprintf_bp(os->os_rootbp, "reading %s", "");
303                 err = arc_read(NULL, spa, os->os_rootbp,
304                     arc_getbuf_func, &os->os_phys_buf,
305                     ZIO_PRIORITY_SYNC_READ, ZIO_FLAG_CANFAIL, &aflags, &zb);
306                 if (err != 0) {
307                         kmem_free(os, sizeof (objset_t));
308                         /* convert checksum errors into IO errors */
309                         if (err == ECKSUM)
310                                 err = SET_ERROR(EIO);
311                         return (err);
312                 }

314                 /* Increase the blocksize if we are permitted. */
315                 if (spa_version(spa) >= SPA_VERSION_USERSPACE &&
316                     arc_buf_size(os->os_phys_buf) < sizeof (objset_phys_t)) {
317                         arc_buf_t *buf = arc_buf_alloc(spa,
318                             sizeof (objset_phys_t), &os->os_phys_buf,
319                             ARC_BUFC_METADATA);
320                         bzero(buf->b_data, sizeof (objset_phys_t));
```



```
321                         bcopy(os->os_phys_buf->b_data, buf->b_data,
322                             arc_buf_size(os->os_phys_buf));
323                         (void) arc_buf_remove_ref(os->os_phys_buf,
324                             &os->os_phys_buf);
325                         os->os_phys_buf = buf;
326                 }

328                 os->os_phys = os->os_phys_buf->b_data;
329                 os->os_flags = os->os_phys->os_flags;
330         } else {
331                 int size = spa_version(spa) >= SPA_VERSION_USERSPACE ?
332                     sizeof (objset_phys_t) : OBJSET_OLD_PHYS_SIZE;
333                 os->os_phys_buf = arc_buf_alloc(spa, size,
334                     &os->os_phys_buf, ARC_BUFC_METADATA);
335                 os->os_phys = os->os_phys_buf->b_data;
336                 bzero(os->os_phys, size);
337         }

339         /*
340          * Note: the changed_cb will be called once before the register
341          * func returns, thus changing the checksum/compression from the
342          * default (fletcher2/off).  Snapshots don't need to know about
343          * checksum/compression/copies.
344          */
345         if (ds != NULL) {
346                 err = dsl_prop_register(ds,
347                     zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
348                     primary_cache_changed_cb, os);
349                 if (err == 0) {
350                         err = dsl_prop_register(ds,
351                             zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
352                             secondary_cache_changed_cb, os);
353                 }
354                 if (!dsl_dataset_is_snapshot(ds)) {
355                         if (err == 0) {
356                                 err = dsl_prop_register(ds,
357                                     zfs_prop_to_name(ZFS_PROP_CHECKSUM),
358                                     checksum_changed_cb, os);
359                         }
360                         if (err == 0) {
361                                 err = dsl_prop_register(ds,
362                                     zfs_prop_to_name(ZFS_PROP_COMPRESSION),
363                                     compression_changed_cb, os);
364                         }
365                         if (err == 0) {
366                                 err = dsl_prop_register(ds,
367                                     zfs_prop_to_name(ZFS_PROP_COPIES),
368                                     copies_changed_cb, os);
369                         }
370                         if (err == 0) {
371                                 err = dsl_prop_register(ds,
372                                     zfs_prop_to_name(ZFS_PROP_DEDUP),
373                                     dedup_changed_cb, os);
374                         }
375                         if (err == 0) {
376                                 err = dsl_prop_register(ds,
377                                     zfs_prop_to_name(ZFS_PROP_LOGBIAS),
378                                     logbias_changed_cb, os);
379                         }
380                         if (err == 0) {
381                                 err = dsl_prop_register(ds,
382                                     zfs_prop_to_name(ZFS_PROP_SYNC),
383                                     sync_changed_cb, os);
384                         }
385                         if (err == 0) {
386                                 err = dsl_prop_register(ds,
```

```
387                                     zfs_prop_to_name(
388                                     ZFS_PROP_REDUNDANT_METADATA),
389                                     redundant_metadata_changed_cb, os);
390                             }
391                     }
392                     if (err != 0) {
393                             VERIFY(arc_buf_remove_ref(os->os_phys_buf,
394                                 &os->os_phys_buf));
395                             kmem_free(os, sizeof (objset_t));
396                             return (err);
397                     }
398             } else {
399                     /* It's the meta-objset. */
400                     os->os_checksum = ZIO_CHECKSUM_FLETCHER_4;
401                     os->os_compress = ZIO_COMPRESS_LZJB;
402                     os->os_copies = spa_max_replication(spa);
403                     os->os_dedup_checksum = ZIO_CHECKSUM_OFF;
404                     os->os_dedup_verify = B_FALSE;
405                     os->os_logbias = ZFS_LOGBIAS_LATENCY;
406                     os->os_sync = ZFS_SYNC_STANDARD;
407                     os->os_primary_cache = ZFS_CACHE_ALL;
408                     os->os_secondary_cache = ZFS_CACHE_ALL;
409             }

411             if (ds == NULL || !dsl_dataset_is_snapshot(ds))
412                     os->os_zil_header = os->os_phys->os_zil_header;
413             os->os_zil = zil_alloc(os, &os->os_zil_header);

415             for (i = 0; i < TXG_SIZE; i++) {
416                     list_create(&os->os_dirty_dnodes[i], sizeof (dnode_t),
417                         offsetof(dnode_t, dn_dirty_link[i]));
418                     list_create(&os->os_free_dnodes[i], sizeof (dnode_t),
419                         offsetof(dnode_t, dn_dirty_link[i]));
420             }
421             list_create(&os->os_dnodes, sizeof (dnode_t),
422                 offsetof(dnode_t, dn_link));
423             list_create(&os->os_downgraded_dbufs, sizeof (dmu_buf_impl_t),
424                 offsetof(dmu_buf_impl_t, db_link));

426             mutex_init(&os->os_lock, NULL, MUTEX_DEFAULT, NULL);
427             mutex_init(&os->os_obj_lock, NULL, MUTEX_DEFAULT, NULL);
428             mutex_init(&os->os_user_ptr_lock, NULL, MUTEX_DEFAULT, NULL);

430             DMU_META_DNODE(os) = dnode_special_open(os,
431                 &os->os_phys->os_meta_dnode, DMU_META_DNODE_OBJECT,
432                 &os->os_meta_dnode);
433             if (arc_buf_size(os->os_phys_buf) >= sizeof (objset_phys_t)) {
434                     DMU_USERUSED_DNODE(os) = dnode_special_open(os,
435                         &os->os_phys->os_userused_dnode, DMU_USERUSED_OBJECT,
436                         &os->os_userused_dnode);
437                     DMU_GROUPUSED_DNODE(os) = dnode_special_open(os,
438                         &os->os_phys->os_groupused_dnode, DMU_GROUPUSED_OBJECT,
439                         &os->os_groupused_dnode);
440             }

442             *osp = os;
443             return (0);
444     }

446     int
447     dmu_objset_from_ds(dsl_dataset_t *ds, objset_t **osp)
448     {
449             int err = 0;

451             mutex_enter(&ds->ds_opening_lock);
452             if (ds->ds_objset == NULL) {
```



```
453                     objset_t *os;
454                     err = dmu_objset_open_impl(dsl_dataset_get_spa(ds),
455                         ds, dsl_dataset_get_blkptr(ds), &os);

457                     if (err == 0) {
458                             mutex_enter(&ds->ds_lock);
459                             ASSERT(ds->ds_objset == NULL);
460                             ds->ds_objset = os;
461                             mutex_exit(&ds->ds_lock);
462                     }
463             }
464             *osp = ds->ds_objset;
465             mutex_exit(&ds->ds_opening_lock);
466             return (err);
467     }

469     /*
470      * Holds the pool while the objset is held.  Therefore only one objset
471      * can be held at a time.
472      */
473     int
474     dmu_objset_hold(const char *name, void *tag, objset_t **osp)
475     {
476             dsl_pool_t *dp;
477             dsl_dataset_t *ds;
478             int err;

480             err = dsl_pool_hold(name, tag, &dp);
481             if (err != 0)
482                     return (err);
483             err = dsl_dataset_hold(dp, name, tag, &ds);
484             if (err != 0) {
485                     dsl_pool_rele(dp, tag);
486                     return (err);
487             }

489             err = dmu_objset_from_ds(ds, osp);
490             if (err != 0) {
491                     dsl_dataset_rele(ds, tag);
492                     dsl_pool_rele(dp, tag);
493             }

495             return (err);
496     }

498     static int
499     dmu_objset_own_common(dsl_dataset_t *ds, dmu_objset_type_t type,
500         boolean_t readonly, void *tag, objset_t **osp)
501     {
502             int err;

504             err = dmu_objset_from_ds(ds, osp);
505             if (err != 0) {
506                     dsl_dataset_disown(ds, tag);
507             } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
508                     dsl_dataset_disown(ds, tag);
509                     return (SET_ERROR(EINVAL));
510             } else if (!readonly && dsl_dataset_is_snapshot(ds)) {
511                     dsl_dataset_disown(ds, tag);
512                     return (SET_ERROR(EROFS));
513             }
514             return (err);
515     }

517     #endif /* ! codereview */
518     /*
```

```
 * dsl_pool must not be held when this is called.
 * Upon successful return, there will be a longhold on the dataset,
 * and the dsl_pool will not be held.
 */
int
dmu_objset_own(const char *name, dmu_objset_type_t type,
    boolean_t readonly, void *tag, objset_t **osp)
{
        dsl_pool_t *dp;
        dsl_dataset_t *ds;
        int err;

        err = dsl_pool_hold(name, FTAG, &dp);
        if (err != 0)
                return (err);
        err = dsl_dataset_own(dp, name, tag, &ds);
        if (err != 0) {
                dsl_pool_rele(dp, FTAG);
                return (err);
        }
        err = dmu_objset_own_common(ds, type, readonly, tag, osp);
        dsl_pool_rele(dp, FTAG);

        return (err);
}
#endif /* ! codereview */

int
dmu_objset_own_obj(dsl_pool_t *dp, uint64_t obj, dmu_objset_type_t type,
    boolean_t readonly, void *tag, objset_t **osp)
{
        dsl_dataset_t *ds;
        int err;

        err = dsl_dataset_own_obj(dp, obj, tag, &ds);
        if (err != 0)
        err = dmu_objset_from_ds(ds, osp);
        dsl_pool_rele(dp, FTAG);
        if (err != 0) {
                dsl_dataset_disown(ds, tag);
        } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
                dsl_dataset_disown(ds, tag);
                return (SET_ERROR(EINVAL));
        } else if (!readonly && dsl_dataset_is_snapshot(ds)) {
                dsl_dataset_disown(ds, tag);
                return (SET_ERROR(EROFS));
        }
                return (err);

        return (dmu_objset_own_common(ds, type, readonly, tag, osp));
#endif /* ! codereview */
}

void
dmu_objset_rele(objset_t *os, void *tag)
{
        dsl_pool_t *dp = dmu_objset_pool(os);
        dsl_dataset_rele(os->os_dsl_dataset, tag);
        dsl_pool_rele(dp, tag);
}

void
dmu_objset_rele_obj(objset_t *os, void *tag)
{
        dsl_dataset_rele(os->os_dsl_dataset, tag);
}
```



```
#endif /* ! codereview */
/*
 * When we are called, os MUST refer to an objset associated with a dataset
 * that is owned by 'tag'; that is, is held and long held by 'tag' and ds_owner
 * == tag.  We will then release and reacquire ownership of the dataset while
 * holding the pool config_rwlock to avoid intervening namespace or ownership
 * changes may occur.
 *
 * This exists solely to accommodate zfs_ioc_userspace_upgrade()'s desire to
 * release the hold on its dataset and acquire a new one on the dataset of the
 * same name so that it can be partially torn down and reconstructed.
 */
void
dmu_objset_refresh_ownership(objset_t *os, void *tag)
{
        dsl_pool_t *dp;
        dsl_dataset_t *ds, *newds;
        char name[MAXNAMELEN];

        ds = os->os_dsl_dataset;
        VERIFY3P(ds, !=, NULL);
        VERIFY3P(ds->ds_owner, ==, tag);
        VERIFY(dsl_dataset_long_held(ds));

        dsl_dataset_name(ds, name);
        dp = dmu_objset_pool(os);
        dsl_pool_config_enter(dp, FTAG);
        dmu_objset_disown(os, tag);
        VERIFY0(dsl_dataset_own(dp, name, tag, &newds));
        VERIFY3P(newds, ==, os->os_dsl_dataset);
        dsl_pool_config_exit(dp, FTAG);
}

void
dmu_objset_disown(objset_t *os, void *tag)
{
        dsl_dataset_disown(os->os_dsl_dataset, tag);
}

void
dmu_objset_evict_dbufs(objset_t *os)
{
        dnode_t *dn;

        mutex_enter(&os->os_lock);

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

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

        while (dn) {
                dnode_t *next_dn = dn;

                do {
```

```
640                         next_dn = list_next(&os->os_dnodes, next_dn);
641                 } while (next_dn && !dnode_add_ref(next_dn, FTAG));

643                 mutex_exit(&os->os_lock);
644                 dnode_evict_dbufs(dn);
645                 dnode_rele(dn, FTAG);
646                 mutex_enter(&os->os_lock);
647                 dn = next_dn;
648         }
649         mutex_exit(&os->os_lock);
650 }

652 void
653 dmu_objset_evict(objset_t *os)
654 {
655         dsl_dataset_t *ds = os->os_dsl_dataset;

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

660         if (ds) {
661                 if (!dsl_dataset_is_snapshot(ds)) {
662                         VERIFY0(dsl_prop_unregister(ds,
663                             zfs_prop_to_name(ZFS_PROP_CHECKSUM),
664                             checksum_changed_cb, os));
665                         VERIFY0(dsl_prop_unregister(ds,
666                             zfs_prop_to_name(ZFS_PROP_COMPRESSION),
667                             compression_changed_cb, os));
668                         VERIFY0(dsl_prop_unregister(ds,
669                             zfs_prop_to_name(ZFS_PROP_COPIES),
670                             copies_changed_cb, os));
671                         VERIFY0(dsl_prop_unregister(ds,
672                             zfs_prop_to_name(ZFS_PROP_DEDUP),
673                             dedup_changed_cb, os));
674                         VERIFY0(dsl_prop_unregister(ds,
675                             zfs_prop_to_name(ZFS_PROP_LOGBIAS),
676                             logbias_changed_cb, os));
677                         VERIFY0(dsl_prop_unregister(ds,
678                             zfs_prop_to_name(ZFS_PROP_SYNC),
679                             sync_changed_cb, os));
680                         VERIFY0(dsl_prop_unregister(ds,
681                             zfs_prop_to_name(ZFS_PROP_REDUNDANT_METADATA),
682                             redundant_metadata_changed_cb, os));
683                 }
684                 VERIFY0(dsl_prop_unregister(ds,
685                     zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
686                     primary_cache_changed_cb, os));
687                 VERIFY0(dsl_prop_unregister(ds,
688                     zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
689                     secondary_cache_changed_cb, os));
690         }

692         if (os->os_sa)
693                 sa_tear_down(os);

695         dmu_objset_evict_dbufs(os);

697         dnode_special_close(&os->os_meta_dnode);
698         if (DMU_USERUSED_DNODE(os)) {
699                 dnode_special_close(&os->os_userused_dnode);
700                 dnode_special_close(&os->os_groupused_dnode);
701         }
702         zil_free(os->os_zil);

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



```
706         VERIFY(arc_buf_remove_ref(os->os_phys_buf, &os->os_phys_buf));

708         /*
709          * This is a barrier to prevent the objset from going away in
710          * dnode_move() until we can safely ensure that the objset is still in
711          * use. We consider the objset valid before the barrier and invalid
712          * after the barrier.
713          */
714         rw_enter(&os_lock, RW_READER);
715         rw_exit(&os_lock);

717         mutex_destroy(&os->os_lock);
718         mutex_destroy(&os->os_obj_lock);
719         mutex_destroy(&os->os_user_ptr_lock);
720         kmem_free(os, sizeof (objset_t));
721 }

723 timestruc_t
724 dmu_objset_snap_cmtime(objset_t *os)
725 {
726         return (dsl_dir_snap_cmtime(os->os_dsl_dataset->ds_dir));
727 }

729 /* called from dsl for meta-objset */
730 objset_t *
731 dmu_objset_create_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
732     dmu_objset_type_t type, dmu_tx_t *tx)
733 {
734         objset_t *os;
735         dnode_t *mdn;

737         ASSERT(dmu_tx_is_syncing(tx));

739         if (ds != NULL)
740                 VERIFY0(dmu_objset_from_ds(ds, &os));
741         else
742                 VERIFY0(dmu_objset_open_impl(spa, NULL, bp, &os));

744         mdn = DMU_META_DNODE(os);

746         dnode_allocate(mdn, DMU_OT_DNODE, 1 << DNODE_BLOCK_SHIFT,
747             DN_MAX_INDBLKSHIFT, DMU_OT_NONE, 0, tx);

749         /*
750          * We don't want to have to increase the meta-dnode's nlevels
751          * later, because then we could do it in quescing context while
752          * we are also accessing it in open context.
753          *
754          * This precaution is not necessary for the MOS (ds == NULL),
755          * because the MOS is only updated in syncing context.
756          * This is most fortunate: the MOS is the only objset that
757          * needs to be synced multiple times as spa_sync() iterates
758          * to convergence, so minimizing its dn_nlevels matters.
759          */
760         if (ds != NULL) {
761                 int levels = 1;

763                 /*
764                  * Determine the number of levels necessary for the meta-dnode
765                  * to contain DN_MAX_OBJECT dnodes.
766                  */
767                 while ((uint64_t)mdn->dn_nblkptr << (mdn->dn_datablkshift +
768                     (levels - 1) * (mdn->dn_indblkshift - SPA_BLKPTRSHIFT)) <
769                     DN_MAX_OBJECT * sizeof (dnode_phys_t))
770                         levels++;
```

```
772                 mdn->dn_next_nlevels[tx->tx_txg & TXG_MASK] =
773                     mdn->dn_nlevels = levels;
774         }

776         ASSERT(type != DMU_OST_NONE);
777         ASSERT(type != DMU_OST_ANY);
778         ASSERT(type < DMU_OST_NUMTYPES);
779         os->os_phys->os_type = type;
780         if (dmu_objset_userused_enabled(os)) {
781                 os->os_phys->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
782                 os->os_flags = os->os_phys->os_flags;
783         }

785         dsl_dataset_dirty(ds, tx);

787         return (os);
788 }

790 typedef struct dmu_objset_create_arg {
791         const char *doca_name;
792         cred_t *doca_cred;
793         void (*doca_userfunc)(objset_t *os, void *arg,
794             cred_t *cr, dmu_tx_t *tx);
795         void *doca_userarg;
796         dmu_objset_type_t doca_type;
797         uint64_t doca_flags;
798 } dmu_objset_create_arg_t;

800 /*ARGSUSED*/
801 static int
802 dmu_objset_create_check(void *arg, dmu_tx_t *tx)
803 {
804         dmu_objset_create_arg_t *doca = arg;
805         dsl_pool_t *dp = dmu_tx_pool(tx);
806         dsl_dir_t *pdd;
807         const char *tail;
808         int error;

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

813         error = dsl_dir_hold(dp, doca->doca_name, FTAG, &pdd, &tail);
814         if (error != 0)
815                 return (error);
816         if (tail == NULL) {
817                 dsl_dir_rele(pdd, FTAG);
818                 return (SET_ERROR(EEXIST));
819         }
820         error = dsl_fs_ss_limit_check(pdd, 1, ZFS_PROP_FILESYSTEM_LIMIT, NULL,
821             doca->doca_cred);
822         dsl_dir_rele(pdd, FTAG);

824         return (error);
825 }

827 static void
828 dmu_objset_create_sync(void *arg, dmu_tx_t *tx)
829 {
830         dmu_objset_create_arg_t *doca = arg;
831         dsl_pool_t *dp = dmu_tx_pool(tx);
832         dsl_dir_t *pdd;
833         const char *tail;
834         dsl_dataset_t *ds;
835         uint64_t obj;
836         blkptr_t *bp;
837         objset_t *os;
```



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

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

844         VERIFY0(dsl_dataset_hold_obj(pdd->dd_pool, obj, FTAG, &ds));
845         bp = dsl_dataset_get_blkptr(ds);
846         os = dmu_objset_create_impl(pdd->dd_pool->dp_spa,
847             ds, bp, doca->doca_type, tx);

849         if (doca->doca_userfunc != NULL) {
850                 doca->doca_userfunc(os, doca->doca_userarg,
851                     doca->doca_cred, tx);
852         }

854         spa_history_log_internal_ds(ds, "create", tx, "");
855         dsl_dataset_rele(ds, FTAG);
856         dsl_dir_rele(pdd, FTAG);
857 }

859 int
860 dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
861     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg)
862 {
863         dmu_objset_create_arg_t doca;

865         doca.doca_name = name;
866         doca.doca_cred = CRED();
867         doca.doca_flags = flags;
868         doca.doca_userfunc = func;
869         doca.doca_userarg = arg;
870         doca.doca_type = type;

872         return (dsl_sync_task(name,
873             dmu_objset_create_check, dmu_objset_create_sync, &doca, 5));
874 }

876 typedef struct dmu_objset_clone_arg {
877         const char *doca_clone;
878         const char *doca_origin;
879         cred_t *doca_cred;
880 } dmu_objset_clone_arg_t;

882 /*ARGSUSED*/
883 static int
884 dmu_objset_clone_check(void *arg, dmu_tx_t *tx)
885 {
886         dmu_objset_clone_arg_t *doca = arg;
887         dsl_dir_t *pdd;
888         const char *tail;
889         int error;
890         dsl_dataset_t *origin;
891         dsl_pool_t *dp = dmu_tx_pool(tx);

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

896         error = dsl_dir_hold(dp, doca->doca_clone, FTAG, &pdd, &tail);
897         if (error != 0)
898                 return (error);
899         if (tail == NULL) {
900                 dsl_dir_rele(pdd, FTAG);
901                 return (SET_ERROR(EEXIST));
902         }
903         /* You can't clone across pools. */
```

```
904         if (pdd->dd_pool != dp) {
905                 dsl_dir_rele(pdd, FTAG);
906                 return (SET_ERROR(EXDEV));
907         }
908         error = dsl_fs_ss_limit_check(pdd, 1, ZFS_PROP_FILESYSTEM_LIMIT, NULL,
909             doca->doca_cred);
910         if (error != 0) {
911                 dsl_dir_rele(pdd, FTAG);
912                 return (SET_ERROR(EDQUOT));
913         }
914         dsl_dir_rele(pdd, FTAG);

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

920         /* You can't clone across pools. */
921         if (origin->ds_dir->dd_pool != dp) {
922                 dsl_dataset_rele(origin, FTAG);
923                 return (SET_ERROR(EXDEV));
924         }

926         /* You can only clone snapshots, not the head datasets. */
927         if (!dsl_dataset_is_snapshot(origin)) {
928                 dsl_dataset_rele(origin, FTAG);
929                 return (SET_ERROR(EINVAL));
930         }
931         dsl_dataset_rele(origin, FTAG);

933         return (0);
934 }

936 static void
937 dmu_objset_clone_sync(void *arg, dmu_tx_t *tx)
938 {
939         dmu_objset_clone_arg_t *doca = arg;
940         dsl_pool_t *dp = dmu_tx_pool(tx);
941         dsl_dir_t *pdd;
942         const char *tail;
943         dsl_dataset_t *origin, *ds;
944         uint64_t obj;
945         char namebuf[MAXNAMELEN];

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

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

953         VERIFY0(dsl_dataset_hold_obj(pdd->dd_pool, obj, FTAG, &ds));
954         dsl_dataset_name(origin, namebuf);
955         spa_history_log_internal_ds(ds, "clone", tx,
956             "origin=%s (%llu)", namebuf, origin->ds_object);
957         dsl_dataset_rele(ds, FTAG);
958         dsl_dataset_rele(origin, FTAG);
959         dsl_dir_rele(pdd, FTAG);
960 }

962 int
963 dmu_objset_clone(const char *clone, const char *origin)
964 {
965         dmu_objset_clone_arg_t doca;

967         doca.doca_clone = clone;
968         doca.doca_origin = origin;
969         doca.doca_cred = CRED();
```



```
971         return (dsl_sync_task(clone,
972             dmu_objset_clone_check, dmu_objset_clone_sync, &doca, 5));
973 }

975 int
976 dmu_objset_snapshot_one(const char *fsname, const char *snapname)
977 {
978         int err;
979         char *longsnap = kmem_asprintf("%s@%s", fsname, snapname);
980         nvlist_t *snaps = fnvlist_alloc();

982         fnvlist_add_boolean(snaps, longsnap);
983         strfree(longsnap);
984         err = dsl_dataset_snapshot(snaps, NULL, NULL);
985         fnvlist_free(snaps);
986         return (err);
987 }

989 static void
990 dmu_objset_sync_dnodes(list_t *list, list_t *newlist, dmu_tx_t *tx)
991 {
992         dnode_t *dn;

994         while (dn = list_head(list)) {
995                 ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
996                 ASSERT(dn->dn_dbuf->db_data_pending);
997                 /*
998                  * Initialize dn_zio outside dnode_sync() because the
999                  * meta-dnode needs to set it ouside dnode_sync().
1000                 */
1001                dn->dn_zio = dn->dn_dbuf->db_data_pending->dr_zio;
1002                ASSERT(dn->dn_zio);

1004                ASSERT3U(dn->dn_nlevels, <=, DN_MAX_LEVELS);
1005                list_remove(list, dn);

1007                if (newlist) {
1008                        (void) dnode_add_ref(dn, newlist);
1009                        list_insert_tail(newlist, dn);
1010                }

1012                dnode_sync(dn, tx);
1013        }
1014 }

1016 /* ARGSUSED */
1017 static void
1018 dmu_objset_write_ready(zio_t *zio, arc_buf_t *abuf, void *arg)
1019 {
1020         blkptr_t *bp = zio->io_bp;
1021         objset_t *os = arg;
1022         dnode_phys_t *dnp = &os->os_phys->os_meta_dnode;

1024         ASSERT(!BP_IS_EMBEDDED(bp));
1025         ASSERT3P(bp, ==, os->os_rootbp);
1026         ASSERT3U(BP_GET_TYPE(bp), ==, DMU_OT_OBJSET);
1027         ASSERT0(BP_GET_LEVEL(bp));

1029         /*
1030          * Update rootbp fill count: it should be the number of objects
1031          * allocated in the object set (not counting the "special"
1032          * objects that are stored in the objset_phys_t -- the meta
1033          * dnode and user/group accounting objects).
1034          */
1035         bp->blk_fill = 0;
```

```
1036         for (int i = 0; i < dnp->dn_nblkptr; i++)
1037                 bp->blk_fill += BP_GET_FILL(&dnp->dn_blkptr[i]);
1038 }

1040 /* ARGSUSED */
1041 static void
1042 dmu_objset_write_done(zio_t *zio, arc_buf_t *abuf, void *arg)
1043 {
1044         blkptr_t *bp = zio->io_bp;
1045         blkptr_t *bp_orig = &zio->io_bp_orig;
1046         objset_t *os = arg;

1048         if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {
1049                 ASSERT(BP_EQUAL(bp, bp_orig));
1050         } else {
1051                 dsl_dataset_t *ds = os->os_dsl_dataset;
1052                 dmu_tx_t *tx = os->os_synctx;

1054                 (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
1055                 dsl_dataset_block_born(ds, bp, tx);
1056         }
1057 }

1059 /* called from dsl */
1060 void
1061 dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
1062 {
1063         int txgoff;
1064         zbookmark_t zb;
1065         zio_prop_t zp;
1066         zio_t *zio;
1067         list_t *list;
1068         list_t *newlist = NULL;
1069         dbuf_dirty_record_t *dr;

1071         dprintf_ds(os->os_dsl_dataset, "txg=%llu\n", tx->tx_txg);

1073         ASSERT(dmu_tx_is_syncing(tx));
1074         /* XXX the write_done callback should really give us the tx... */
1075         os->os_synctx = tx;

1077         if (os->os_dsl_dataset == NULL) {
1078                 /*
1079                  * This is the MOS.  If we have upgraded,
1080                  * spa_max_replication() could change, so reset
1081                  * os_copies here.
1082                  */
1083                 os->os_copies = spa_max_replication(os->os_spa);
1084         }

1086         /*
1087          * Create the root block IO
1088          */
1089         SET_BOOKMARK(&zb, os->os_dsl_dataset ?
1090             os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
1091             ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
1092         arc_release(os->os_phys_buf, &os->os_phys_buf);

1094         dmu_write_policy(os, NULL, 0, 0, &zp);

1096         zio = arc_write(pio, os->os_spa, tx->tx_txg,
1097             os->os_rootbp, os->os_phys_buf, DMU_OS_IS_L2CACHEABLE(os),
1098             DMU_OS_IS_L2COMPRESSIBLE(os), &zp, dmu_objset_write_ready,
1099             NULL, dmu_objset_write_done, os, ZIO_PRIORITY_ASYNC_WRITE,
1100             ZIO_FLAG_MUSTSUCCEED, &zb);
```



```
1102         /*
1103          * Sync special dnodes - the parent IO for the sync is the root block
1104          */
1105         DMU_META_DNODE(os)->dn_zio = zio;
1106         dnode_sync(DMU_META_DNODE(os), tx);

1108         os->os_phys->os_flags = os->os_flags;

1110         if (DMU_USERUSED_DNODE(os) &&
1111             DMU_USERUSED_DNODE(os)->dn_type != DMU_OT_NONE) {
1112                 DMU_USERUSED_DNODE(os)->dn_zio = zio;
1113                 dnode_sync(DMU_USERUSED_DNODE(os), tx);
1114                 DMU_GROUPUSED_DNODE(os)->dn_zio = zio;
1115                 dnode_sync(DMU_GROUPUSED_DNODE(os), tx);
1116         }

1118         txgoff = tx->tx_txg & TXG_MASK;

1120         if (dmu_objset_userused_enabled(os)) {
1121                 newlist = &os->os_synced_dnodes;
1122                 /*
1123                  * We must create the list here because it uses the
1124                  * dn_dirty_link[] of this txg.
1125                  */
1126                 list_create(newlist, sizeof (dnode_t),
1127                     offsetof(dnode_t, dn_dirty_link[txgoff]));
1128         }

1130         dmu_objset_sync_dnodes(&os->os_free_dnodes[txgoff], newlist, tx);
1131         dmu_objset_sync_dnodes(&os->os_dirty_dnodes[txgoff], newlist, tx);

1133         list = &DMU_META_DNODE(os)->dn_dirty_records[txgoff];
1134         while (dr = list_head(list)) {
1135                 ASSERT0(dr->dr_dbuf->db_level);
1136                 list_remove(list, dr);
1137                 if (dr->dr_zio)
1138                         zio_nowait(dr->dr_zio);
1139         }
1140         /*
1141          * Free intent log blocks up to this tx.
1142          */
1143         zil_sync(os->os_zil, tx);
1144         os->os_phys->os_zil_header = os->os_zil_header;
1145         zio_nowait(zio);
1146 }

1148 boolean_t
1149 dmu_objset_is_dirty(objset_t *os, uint64_t txg)
1150 {
1151         return (!list_is_empty(&os->os_dirty_dnodes[txg & TXG_MASK]) ||
1152             !list_is_empty(&os->os_free_dnodes[txg & TXG_MASK]));
1153 }

1155 static objset_used_cb_t *used_cbs[DMU_OST_NUMTYPES];

1157 void
1158 dmu_objset_register_type(dmu_objset_type_t ost, objset_used_cb_t *cb)
1159 {
1160         used_cbs[ost] = cb;
1161 }

1163 boolean_t
1164 dmu_objset_userused_enabled(objset_t *os)
1165 {
1166         return (spa_version(os->os_spa) >= SPA_VERSION_USERSPACE &&
1167             used_cbs[os->os_phys->os_type] != NULL &&
```

```
1168             DMU_USERUSED_DNODE(os) != NULL);
1169 }
1171 static void
1172 do_userquota_update(objset_t *os, uint64_t used, uint64_t flags,
1173     uint64_t user, uint64_t group, boolean_t subtract, dmu_tx_t *tx)
1174 {
1175         if ((flags & DNODE_FLAG_USERUSED_ACCOUNTED)) {
1176                 int64_t delta = DNODE_SIZE + used;
1177                 if (subtract)
1178                         delta = -delta;
1179                 VERIFY3U(0, ==, zap_increment_int(os, DMU_USERUSED_OBJECT,
1180                     user, delta, tx));
1181                 VERIFY3U(0, ==, zap_increment_int(os, DMU_GROUPUSED_OBJECT,
1182                     group, delta, tx));
1183         }
1184 }
1186 void
1187 dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx)
1188 {
1189         dnode_t *dn;
1190         list_t *list = &os->os_synced_dnodes;
1192         ASSERT(list_head(list) == NULL || dmu_objset_userused_enabled(os));
1194         while (dn = list_head(list)) {
1195                 int flags;
1196                 ASSERT(!DMU_OBJECT_IS_SPECIAL(dn->dn_object));
1197                 ASSERT(dn->dn_phys->dn_type == DMU_OT_NONE ||
1198                     dn->dn_phys->dn_flags &
1199                     DNODE_FLAG_USERUSED_ACCOUNTED);
1201                 /* Allocate the user/groupused objects if necessary. */
1202                 if (DMU_USERUSED_DNODE(os)->dn_type == DMU_OT_NONE) {
1203                         VERIFY(0 == zap_create_claim(os,
1204                             DMU_USERUSED_OBJECT,
1205                             DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
1206                         VERIFY(0 == zap_create_claim(os,
1207                             DMU_GROUPUSED_OBJECT,
1208                             DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
1209                 }
1211                 /*
1212                  * We intentionally modify the zap object even if the
1213                  * net delta is zero.  Otherwise
1214                  * the block of the zap obj could be shared between
1215                  * datasets but need to be different between them after
1216                  * a bprewrite.
1217                  */
1219                 flags = dn->dn_id_flags;
1220                 ASSERT(flags);
1221                 if (flags & DN_ID_OLD_EXIST)  {
1222                         do_userquota_update(os, dn->dn_oldused, dn->dn_oldflags,
1223                             dn->dn_olduid, dn->dn_oldgid, B_TRUE, tx);
1224                 }
1225                 if (flags & DN_ID_NEW_EXIST) {
1226                         do_userquota_update(os, DN_USED_BYTES(dn->dn_phys),
1227                             dn->dn_phys->dn_flags,  dn->dn_newuid,
1228                             dn->dn_newgid, B_FALSE, tx);
1229                 }
1231                 mutex_enter(&dn->dn_mtx);
1232                 dn->dn_oldused = 0;
1233                 dn->dn_oldflags = 0;
```



```
1234                 if (dn->dn_id_flags & DN_ID_NEW_EXIST) {
1235                         dn->dn_olduid = dn->dn_newuid;
1236                         dn->dn_oldgid = dn->dn_newgid;
1237                         dn->dn_id_flags |= DN_ID_OLD_EXIST;
1238                         if (dn->dn_bonuslen == 0)
1239                                 dn->dn_id_flags |= DN_ID_CHKED_SPILL;
1240                         else
1241                                 dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1242                 }
1243                 dn->dn_id_flags &= ~(DN_ID_NEW_EXIST);
1244                 mutex_exit(&dn->dn_mtx);
1246                 list_remove(list, dn);
1247                 dnode_rele(dn, list);
1248         }
1249 }
1251 /*
1252  * Returns a pointer to data to find uid/gid from
1253  *
1254  * If a dirty record for transaction group that is syncing can't
1255  * be found then NULL is returned.  In the NULL case it is assumed
1256  * the uid/gid aren't changing.
1257  */
1258 static void *
1259 dmu_objset_userquota_find_data(dmu_buf_impl_t *db, dmu_tx_t *tx)
1260 {
1261         dbuf_dirty_record_t *dr, **drp;
1262         void *data;
1264         if (db->db_dirtycnt == 0)
1265                 return (db->db.db_data);  /* Nothing is changing */
1267         for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1268                 if (dr->dr_txg == tx->tx_txg)
1269                         break;
1271         if (dr == NULL) {
1272                 data = NULL;
1273         } else {
1274                 dnode_t *dn;
1276                 DB_DNODE_ENTER(dr->dr_dbuf);
1277                 dn = DB_DNODE(dr->dr_dbuf);
1279                 if (dn->dn_bonuslen == 0 &&
1280                     dr->dr_dbuf->db_blkid == DMU_SPILL_BLKID)
1281                         data = dr->dt.dl.dr_data->b_data;
1282                 else
1283                         data = dr->dt.dl.dr_data;
1285                 DB_DNODE_EXIT(dr->dr_dbuf);
1286         }
1288         return (data);
1289 }
1291 void
1292 dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx)
1293 {
1294         objset_t *os = dn->dn_objset;
1295         void *data = NULL;
1296         dmu_buf_impl_t *db = NULL;
1297         uint64_t *user = NULL;
1298         uint64_t *group = NULL;
1299         int flags = dn->dn_id_flags;
```

```
1300         int error;
1301         boolean_t have_spill = B_FALSE;

1303         if (!dmu_objset_userused_enabled(dn->dn_objset))
1304                 return;

1306         if (before && (flags & (DN_ID_CHKED_BONUS|DN_ID_OLD_EXIST|
1307             DN_ID_CHKED_SPILL)))
1308                 return;

1310         if (before && dn->dn_bonuslen != 0)
1311                 data = DN_BONUS(dn->dn_phys);
1312         else if (!before && dn->dn_bonuslen != 0) {
1313                 if (dn->dn_bonus) {
1314                         db = dn->dn_bonus;
1315                         mutex_enter(&db->db_mtx);
1316                         data = dmu_objset_userquota_find_data(db, tx);
1317                 } else {
1318                         data = DN_BONUS(dn->dn_phys);
1319                 }
1320         } else if (dn->dn_bonuslen == 0 && dn->dn_bonustype == DMU_OT_SA) {
1321                         int rf = 0;

1323                         if (RW_WRITE_HELD(&dn->dn_struct_rwlock))
1324                                 rf |= DB_RF_HAVESTRUCT;
1325                         error = dmu_spill_hold_by_dnode(dn,
1326                             rf | DB_RF_MUST_SUCCEED,
1327                             FTAG, (dmu_buf_t **)&db);
1328                         ASSERT(error == 0);
1329                         mutex_enter(&db->db_mtx);
1330                         data = (before) ? db->db.db_data :
1331                             dmu_objset_userquota_find_data(db, tx);
1332                         have_spill = B_TRUE;
1333         } else {
1334                 mutex_enter(&dn->dn_mtx);
1335                 dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1336                 mutex_exit(&dn->dn_mtx);
1337                 return;
1338         }

1340         if (before) {
1341                 ASSERT(data);
1342                 user = &dn->dn_olduid;
1343                 group = &dn->dn_oldgid;
1344         } else if (data) {
1345                 user = &dn->dn_newuid;
1346                 group = &dn->dn_newgid;
1347         }

1349         /*
1350          * Must always call the callback in case the object
1351          * type has changed and that type isn't an object type to track
1352          */
1353         error = used_cbs[os->os_phys->os_type](dn->dn_bonustype, data,
1354             user, group);

1356         /*
1357          * Preserve existing uid/gid when the callback can't determine
1358          * what the new uid/gid are and the callback returned EEXIST.
1359          * The EEXIST error tells us to just use the existing uid/gid.
1360          * If we don't know what the old values are then just assign
1361          * them to 0, since that is a new file  being created.
1362          */
1363         if (!before && data == NULL && error == EEXIST) {
1364                 if (flags & DN_ID_OLD_EXIST) {
1365                         dn->dn_newuid = dn->dn_olduid;
```



```
1366                         dn->dn_newgid = dn->dn_oldgid;
1367                 } else {
1368                         dn->dn_newuid = 0;
1369                         dn->dn_newgid = 0;
1370                 }
1371                 error = 0;
1372         }

1374         if (db)
1375                 mutex_exit(&db->db_mtx);

1377         mutex_enter(&dn->dn_mtx);
1378         if (error == 0 && before)
1379                 dn->dn_id_flags |= DN_ID_OLD_EXIST;
1380         if (error == 0 && !before)
1381                 dn->dn_id_flags |= DN_ID_NEW_EXIST;

1383         if (have_spill) {
1384                 dn->dn_id_flags |= DN_ID_CHKED_SPILL;
1385         } else {
1386                 dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1387         }
1388         mutex_exit(&dn->dn_mtx);
1389         if (have_spill)
1390                 dmu_buf_rele((dmu_buf_t *)db, FTAG);
1391 }

1393 boolean_t
1394 dmu_objset_userspace_present(objset_t *os)
1395 {
1396         return (os->os_phys->os_flags &
1397             OBJSET_FLAG_USERACCOUNTING_COMPLETE);
1398 }

1400 int
1401 dmu_objset_userspace_upgrade(objset_t *os)
1402 {
1403         uint64_t obj;
1404         int err = 0;

1406         if (dmu_objset_userspace_present(os))
1407                 return (0);
1408         if (!dmu_objset_userused_enabled(os))
1409                 return (SET_ERROR(ENOTSUP));
1410         if (dmu_objset_is_snapshot(os))
1411                 return (SET_ERROR(EINVAL));

1413         /*
1414          * We simply need to mark every object dirty, so that it will be
1415          * synced out and now accounted.  If this is called
1416          * concurrently, or if we already did some work before crashing,
1417          * that's fine, since we track each object's accounted state
1418          * independently.
1419          */

1421         for (obj = 0; err == 0; err = dmu_object_next(os, &obj, FALSE, 0)) {
1422                 dmu_tx_t *tx;
1423                 dmu_buf_t *db;
1424                 int objerr;

1426                 if (issig(JUSTLOOKING) && issig(FORREAL))
1427                         return (SET_ERROR(EINTR));

1429                 objerr = dmu_bonus_hold(os, obj, FTAG, &db);
1430                 if (objerr != 0)
1431                         continue;
```

```
1432                 tx = dmu_tx_create(os);
1433                 dmu_tx_hold_bonus(tx, obj);
1434                 objerr = dmu_tx_assign(tx, TXG_WAIT);
1435                 if (objerr != 0) {
1436                         dmu_tx_abort(tx);
1437                         continue;
1438                 }
1439                 dmu_buf_will_dirty(db, tx);
1440                 dmu_buf_rele(db, FTAG);
1441                 dmu_tx_commit(tx);
1442         }

1444         os->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
1445         txg_wait_synced(dmu_objset_pool(os), 0);
1446         return (0);
1447 }

1449 void
1450 dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
1451     uint64_t *usedobjsp, uint64_t *availobjsp)
1452 {
1453         dsl_dataset_space(os->os_dsl_dataset, refdbytesp, availbytesp,
1454             usedobjsp, availobjsp);
1455 }

1457 uint64_t
1458 dmu_objset_fsid_guid(objset_t *os)
1459 {
1460         return (dsl_dataset_fsid_guid(os->os_dsl_dataset));
1461 }

1463 void
1464 dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat)
1465 {
1466         stat->dds_type = os->os_phys->os_type;
1467         if (os->os_dsl_dataset)
1468                 dsl_dataset_fast_stat(os->os_dsl_dataset, stat);
1469 }

1471 void
1472 dmu_objset_stats(objset_t *os, nvlist_t *nv)
1473 {
1474         ASSERT(os->os_dsl_dataset ||
1475             os->os_phys->os_type == DMU_OST_META);

1477         if (os->os_dsl_dataset != NULL)
1478                 dsl_dataset_stats(os->os_dsl_dataset, nv);

1480         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_TYPE,
1481             os->os_phys->os_type);
1482         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERACCOUNTING,
1483             dmu_objset_userspace_present(os));
1484 }

1486 int
1487 dmu_objset_is_snapshot(objset_t *os)
1488 {
1489         if (os->os_dsl_dataset != NULL)
1490                 return (dsl_dataset_is_snapshot(os->os_dsl_dataset));
1491         else
1492                 return (B_FALSE);
1493 }

1495 int
1496 dmu_snapshot_realname(objset_t *os, char *name, char *real, int maxlen,
1497     boolean_t *conflict)
```



```
1498 {
1499         dsl_dataset_t *ds = os->os_dsl_dataset;
1500         uint64_t ignored;

1502         if (ds->ds_phys->ds_snapnames_zapobj == 0)
1503                 return (SET_ERROR(ENOENT));

1505         return (zap_lookup_norm(ds->ds_dir->dd_pool->dp_meta_objset,
1506             ds->ds_phys->ds_snapnames_zapobj, name, 8, 1, &ignored, MT_FIRST,
1507             real, maxlen, conflict));
1508 }

1510 int
1511 dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
1512     uint64_t *idp, uint64_t *offp, boolean_t *case_conflict)
1513 {
1514         dsl_dataset_t *ds = os->os_dsl_dataset;
1515         zap_cursor_t cursor;
1516         zap_attribute_t attr;

1518         ASSERT(dsl_pool_config_held(dmu_objset_pool(os)));

1520         if (ds->ds_phys->ds_snapnames_zapobj == 0)
1521                 return (SET_ERROR(ENOENT));

1523         zap_cursor_init_serialized(&cursor,
1524             ds->ds_dir->dd_pool->dp_meta_objset,
1525             ds->ds_phys->ds_snapnames_zapobj, *offp);

1527         if (zap_cursor_retrieve(&cursor, &attr) != 0) {
1528                 zap_cursor_fini(&cursor);
1529                 return (SET_ERROR(ENOENT));
1530         }

1532         if (strlen(attr.za_name) + 1 > namelen) {
1533                 zap_cursor_fini(&cursor);
1534                 return (SET_ERROR(ENAMETOOLONG));
1535         }

1537         (void) strcpy(name, attr.za_name);
1538         if (idp)
1539                 *idp = attr.za_first_integer;
1540         if (case_conflict)
1541                 *case_conflict = attr.za_normalization_conflict;
1542         zap_cursor_advance(&cursor);
1543         *offp = zap_cursor_serialize(&cursor);
1544         zap_cursor_fini(&cursor);

1546         return (0);
1547 }

1549 int
1550 dmu_dir_list_next(objset_t *os, int namelen, char *name,
1551     uint64_t *idp, uint64_t *offp)
1552 {
1553         dsl_dir_t *dd = os->os_dsl_dataset->ds_dir;
1554         zap_cursor_t cursor;
1555         zap_attribute_t attr;

1557         /* there is no next dir on a snapshot! */
1558         if (os->os_dsl_dataset->ds_object !=
1559             dd->dd_phys->dd_head_dataset_obj)
1560                 return (SET_ERROR(ENOENT));

1562         zap_cursor_init_serialized(&cursor,
1563             dd->dd_pool->dp_meta_objset,
```

```
1564             dd->dd_phys->dd_child_dir_zapobj, *offp);

1566         if (zap_cursor_retrieve(&cursor, &attr) != 0) {
1567                 zap_cursor_fini(&cursor);
1568                 return (SET_ERROR(ENOENT));
1569         }

1571         if (strlen(attr.za_name) + 1 > namelen) {
1572                 zap_cursor_fini(&cursor);
1573                 return (SET_ERROR(ENAMETOOLONG));
1574         }

1576         (void) strcpy(name, attr.za_name);
1577         if (idp)
1578                 *idp = attr.za_first_integer;
1579         zap_cursor_advance(&cursor);
1580         *offp = zap_cursor_serialize(&cursor);
1581         zap_cursor_fini(&cursor);

1583         return (0);
1584 }

1586 typedef struct dmu_objset_find_ctx {
1587         taskq_t         *dc_tq;
1588         dsl_pool_t      *dc_dp;
1589         uint64_t        dc_obj;
1590         int             (*dc_func)(dsl_pool_t *, dsl_dataset_t *, void *);
1591         void            *dc_arg;
1592         int             dc_flags;
1593         kmutex_t        *dc_error_lock;
1594         int             *dc_error;
1595 } dmu_objset_find_ctx_t;

1597 static void
1598 dmu_objset_find_dp_impl(void *arg)
  38 /*
  39  * Find objsets under and including ddobj, call func(ds) on each.
  40  */
  41 int
  42 dmu_objset_find_dp(dsl_pool_t *dp, uint64_t ddobj,
  43     int func(dsl_pool_t *, dsl_dataset_t *, void *), void *arg, int flags)
1599 {
1600         dmu_objset_find_ctx_t *dcp = arg;
1601         dsl_pool_t *dp = dcp->dc_dp;
1602         dmu_objset_find_ctx_t *child_dcp;
1603 #endif /* ! codereview */
1604         dsl_dir_t *dd;
1605         dsl_dataset_t *ds;
1606         zap_cursor_t zc;
1607         zap_attribute_t *attr;
1608         uint64_t thisobj;
1609         int err;

1611         dsl_pool_config_enter(dp, FTAG);

1613         /* don't process if there already was an error */
1614         if (*dcp->dc_error)
1615                 goto out;
  45         ASSERT(dsl_pool_config_held(dp));

1617         err = dsl_dir_hold_obj(dp, dcp->dc_obj, NULL, FTAG, &dd);
  47         err = dsl_dir_hold_obj(dp, ddobj, NULL, FTAG, &dd);
1618         if (err != 0)
1619                 goto fail;
  49                 return (err);
```



```
1621         /* Don't visit hidden ($MOS & $ORIGIN) objsets. */
1622         if (dd->dd_myname[0] == '$') {
1623                 dsl_dir_rele(dd, FTAG);
1624                 goto out;
  54                 return (0);
1625         }

1627         thisobj = dd->dd_phys->dd_head_dataset_obj;
1628         attr = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);

1630         /*
1631          * Iterate over all children.
1632          */
1633         if (dcp->dc_flags & DS_FIND_CHILDREN) {
  63         if (flags & DS_FIND_CHILDREN) {
1634                 for (zap_cursor_init(&zc, dp->dp_meta_objset,
1635                     dd->dd_phys->dd_child_dir_zapobj);
1636                     zap_cursor_retrieve(&zc, attr) == 0;
1637                     (void) zap_cursor_advance(&zc)) {
1638                         ASSERT3U(attr->za_integer_length, ==,
1639                             sizeof (uint64_t));
1640                         ASSERT3U(attr->za_num_integers, ==, 1);

1642                         child_dcp = kmem_alloc(sizeof(*child_dcp), KM_SLEEP);
1643                         *child_dcp = *dcp;
1644                         child_dcp->dc_obj = attr->za_first_integer;
1645                         taskq_dispatch(dcp->dc_tq, dmu_objset_find_dp_impl,
1646                             child_dcp, TQ_SLEEP);
  72                         err = dmu_objset_find_dp(dp, attr->za_first_integer,
  73                             func, arg, flags);
  74                         if (err != 0)
  75                                 break;
1647                 }
1648                 zap_cursor_fini(&zc);

  79                 if (err != 0) {
  80                         dsl_dir_rele(dd, FTAG);
  81                         kmem_free(attr, sizeof (zap_attribute_t));
  82                         return (err);
  83                 }
1649         }

1651         /*
1652          * Iterate over all snapshots.
1653          */
1654         if (dcp->dc_flags & DS_FIND_SNAPSHOTS) {
  89         if (flags & DS_FIND_SNAPSHOTS) {
1655                 dsl_dataset_t *ds;
1656                 err = dsl_dataset_hold_obj(dp, thisobj, FTAG, &ds);

1658                 if (err == 0) {
1659                         uint64_t snapobj = ds->ds_phys->ds_snapnames_zapobj;
1660                         dsl_dataset_rele(ds, FTAG);

1662                         for (zap_cursor_init(&zc, dp->dp_meta_objset, snapobj);
1663                             zap_cursor_retrieve(&zc, attr) == 0;
1664                             (void) zap_cursor_advance(&zc)) {
1665                                 ASSERT3U(attr->za_integer_length, ==,
1666                                     sizeof (uint64_t));
1667                                 ASSERT3U(attr->za_num_integers, ==, 1);

1669                                 err = dsl_dataset_hold_obj(dp,
1670                                     attr->za_first_integer, FTAG, &ds);
1671                                 if (err != 0)
1672                                         break;
1673                                 err = dcp->dc_func(dp, ds, dcp->dc_arg);
```

```
 108                                err = func(dp, ds, arg);
1674                                dsl_dataset_rele(ds, FTAG);
1675                                if (err != 0)
1676                                        break;
1677                        }
1678                        zap_cursor_fini(&zc);
1679                }
1680        }

1682        dsl_dir_rele(dd, FTAG);
1683        kmem_free(attr, sizeof (zap_attribute_t));

1685        if (err != 0)
1686                goto fail;
 121                return (err);

1688        /*
1689         * Apply to self.
1690         */
1691        err = dsl_dataset_hold_obj(dp, thisobj, FTAG, &ds);
1692        if (err != 0)
1693                goto fail;
1694        err = dcp->dc_func(dp, ds, dcp->dc_arg);
 128                return (err);
 129        err = func(dp, ds, arg);
1695        dsl_dataset_rele(ds, FTAG);

1697 fail:
1698        if (err) {
1699                mutex_enter(dcp->dc_error_lock);
1700                /* only keep first error */
1701                if (*dcp->dc_error == 0)
1702                        *dcp->dc_error = err;
1703                mutex_exit(dcp->dc_error_lock);
1704        }

1706 out:
1707        dsl_pool_config_exit(dp, FTAG);
1708        kmem_free(dcp, sizeof(*dcp));
1709 }

1711 /*
1712  * Find objsets under and including ddobj, call func(ds) on each.
1713  * The order for the enumeration is completely undefined.
1714  * func is called with dsl_pool_config held.
1715  */
1716 int
1717 dmu_objset_find_dp(dsl_pool_t *dp, uint64_t ddobj,
1718     int func(dsl_pool_t *, dsl_dataset_t *, void *), void *arg, int flags)
1719 {
1720        int error = 0;
1721        taskq_t *tq = NULL;
1722        int ntasks;
1723        dmu_objset_find_ctx_t *dcp;
1724        kmutex_t err_lock;

1726        ntasks = vdev_count_leaves(dp->dp_spa) * 4;
1727        tq = taskq_create("dmu_objset_find", ntasks, minclsyspri, ntasks,
1728            INT_MAX, 0);
1729        if (!tq)
1730                return (SET_ERROR(ENOMEM));

1732        mutex_init(&err_lock, NULL, MUTEX_DEFAULT, NULL);
1733        dcp = kmem_alloc(sizeof(*dcp), KM_SLEEP);
1734        dcp->dc_tq = tq;
1735        dcp->dc_dp = dp;
```



```
1736        dcp->dc_obj = ddobj;
1737        dcp->dc_func = func;
1738        dcp->dc_arg = arg;
1739        dcp->dc_flags = flags;
1740        dcp->dc_error_lock = &err_lock;
1741        dcp->dc_error = &error;
1742        /* dcp and dc_name will be freed by task */
1743        taskq_dispatch(tq, dmu_objset_find_dp_impl, dcp, TQ_SLEEP);

1745        taskq_wait(tq);
1746        taskq_destroy(tq);
1747        mutex_destroy(&err_lock);

1749        return (error);
 131        return (err);
1750 }
_____unchanged_portion_omitted_
```

```
**********************************************************
  176805 Fri Oct 31 10:14:51 2014
new/usr/src/uts/common/fs/zfs/spa.c
5269 zfs: zpool import slow
While importing a pool all objsets are enumerated twice, once to check
the zil log chains and once to claim them. On pools with many datasets
this process might take a substantial amount of time.
Speed up the process by parallelizing it utilizing a taskq. The number
of parallel tasks is limited to 4 times the number of leaf vdevs.
**********************************************************
______unchanged_portion_omitted_

1708 /*
1709  * Check for missing log devices
1710  */
1711 static boolean_t
1712 spa_check_logs(spa_t *spa)
1713 {
1714         boolean_t rv = B_FALSE;
1715         dsl_pool_t *dp = spa_get_dsl(spa);
1716 #endif /* ! codereview */

1718         switch (spa->spa_log_state) {
1719         case SPA_LOG_MISSING:
1720                 /* need to recheck in case slog has been restored */
1721         case SPA_LOG_UNKNOWN:
1722                 rv = (dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
1723                     zil_check_log_chain, NULL, DS_FIND_CHILDREN) != 0);
1715                 rv = (dmu_objset_find(spa->spa_name, zil_check_log_chain,
1716                     NULL, DS_FIND_CHILDREN) != 0);
1724                 if (rv)
1725                         spa_set_log_state(spa, SPA_LOG_MISSING);
1726                 break;
1727         }
1728         return (rv);
1729 }
______unchanged_portion_omitted_

2078 /*
2079  * Load an existing storage pool, using the pool's builtin spa_config as a
2080  * source of configuration information.
2081  */
2082 static int
2083 spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config,
2084     spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
2085     char **ereport)
2086 {
2087         int error = 0;
2088         nvlist_t *nvroot = NULL;
2089         nvlist_t *label;
2090         vdev_t *rvd;
2091         uberblock_t *ub = &spa->spa_uberblock;
2092         uint64_t children, config_cache_txg = spa->spa_config_txg;
2093         int orig_mode = spa->spa_mode;
2094         int parse;
2095         uint64_t obj;
2096         boolean_t missing_feat_write = B_FALSE;

2097         /*
2098          * If this is an untrusted config, access the pool in read-only mode.
2099          * This prevents things like resilvering recently removed devices.
2100          */
2101         if (!mosconfig)
2102                 spa->spa_mode = FREAD;

2104         ASSERT(MUTEX_HELD(&spa_namespace_lock));
```



```
2106         spa->spa_load_state = state;

2108         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot))
2109                 return (SET_ERROR(EINVAL));

2111         parse = (type == SPA_IMPORT_EXISTING ?
2112             VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);

2114         /*
2115          * Create "The Godfather" zio to hold all async IOs
2116          */
2117         spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
2118             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);

2120         /*
2121          * Parse the configuration into a vdev tree.  We explicitly set the
2122          * value that will be returned by spa_version() since parsing the
2123          * configuration requires knowing the version number.
2124          */
2125         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2126         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse);
2127         spa_config_exit(spa, SCL_ALL, FTAG);

2129         if (error != 0)
2130                 return (error);

2132         ASSERT(spa->spa_root_vdev == rvd);

2134         if (type != SPA_IMPORT_ASSEMBLE) {
2135                 ASSERT(spa_guid(spa) == pool_guid);
2136         }

2138         /*
2139          * Try to open all vdevs, loading each label in the process.
2140          */
2141         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2142         error = vdev_open(rvd);
2143         spa_config_exit(spa, SCL_ALL, FTAG);
2144         if (error != 0)
2145                 return (error);

2147         /*
2148          * We need to validate the vdev labels against the configuration that
2149          * we have in hand, which is dependent on the setting of mosconfig. If
2150          * mosconfig is true then we're validating the vdev labels based on
2151          * that config.  Otherwise, we're validating against the cached config
2152          * (zpool.cache) that was read when we loaded the zfs module, and then
2153          * later we will recursively call spa_load() and validate against
2154          * the vdev config.
2155          *
2156          * If we're assembling a new pool that's been split off from an
2157          * existing pool, the labels haven't yet been updated so we skip
2158          * validation for now.
2159          */
2160         if (type != SPA_IMPORT_ASSEMBLE) {
2161                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2162                 error = vdev_validate(rvd, mosconfig);
2163                 spa_config_exit(spa, SCL_ALL, FTAG);

2165                 if (error != 0)
2166                         return (error);

2168                 if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2169                         return (SET_ERROR(ENXIO));
2170         }
```

```
2172         /*
2173          * Find the best uberblock.
2174          */
2175         vdev_uberblock_load(rvd, ub, &label);

2177         /*
2178          * If we weren't able to find a single valid uberblock, return failure.
2179          */
2180         if (ub->ub_txg == 0) {
2181                 nvlist_free(label);
2182                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, ENXIO));
2183         }

2185         /*
2186          * If the pool has an unsupported version we can't open it.
2187          */
2188         if (!SPA_VERSION_IS_SUPPORTED(ub->ub_version)) {
2189                 nvlist_free(label);
2190                 return (spa_vdev_err(rvd, VDEV_AUX_VERSION_NEWER, ENOTSUP));
2191         }

2193         if (ub->ub_version >= SPA_VERSION_FEATURES) {
2194                 nvlist_t *features;

2196                 /*
2197                  * If we weren't able to find what's necessary for reading the
2198                  * MOS in the label, return failure.
2199                  */
2200                 if (label == NULL || nvlist_lookup_nvlist(label,
2201                     ZPOOL_CONFIG_FEATURES_FOR_READ, &features) != 0) {
2202                         nvlist_free(label);
2203                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2204                             ENXIO));
2205                 }

2207                 /*
2208                  * Update our in-core representation with the definitive values
2209                  * from the label.
2210                  */
2211                 nvlist_free(spa->spa_label_features);
2212                 VERIFY(nvlist_dup(features, &spa->spa_label_features, 0) == 0);
2213         }

2215         nvlist_free(label);

2217         /*
2218          * Look through entries in the label nvlist's features_for_read. If
2219          * there is a feature listed there which we don't understand then we
2220          * cannot open a pool.
2221          */
2222         if (ub->ub_version >= SPA_VERSION_FEATURES) {
2223                 nvlist_t *unsup_feat;

2225                 VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
2226                     0);

2228                 for (nvpair_t *nvp = nvlist_next_nvpair(spa->spa_label_features,
2229                     NULL); nvp != NULL;
2230                     nvp = nvlist_next_nvpair(spa->spa_label_features, nvp)) {
2231                         if (!zfeature_is_supported(nvpair_name(nvp))) {
2232                                 VERIFY(nvlist_add_string(unsup_feat,
2233                                     nvpair_name(nvp), "") == 0);
2234                         }
2235                 }
```



```
2237                 if (!nvlist_empty(unsup_feat)) {
2238                         VERIFY(nvlist_add_nvlist(spa->spa_load_info,
2239                             ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
2240                         nvlist_free(unsup_feat);
2241                         return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2242                             ENOTSUP));
2243                 }

2245                 nvlist_free(unsup_feat);
2246         }

2248         /*
2249          * If the vdev guid sum doesn't match the uberblock, we have an
2250          * incomplete configuration.  We first check to see if the pool
2251          * is aware of the complete config (i.e ZPOOL_CONFIG_VDEV_CHILDREN).
2252          * If it is, defer the vdev_guid_sum check till later so we
2253          * can handle missing vdevs.
2254          */
2255         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VDEV_CHILDREN,
2256             &children) != 0 && mosconfig && type != SPA_IMPORT_ASSEMBLE &&
2257             rvd->vdev_guid_sum != ub->ub_guid_sum)
2258                 return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));

2260         if (type != SPA_IMPORT_ASSEMBLE && spa->spa_config_splitting) {
2261                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2262                 spa_try_repair(spa, config);
2263                 spa_config_exit(spa, SCL_ALL, FTAG);
2264                 nvlist_free(spa->spa_config_splitting);
2265                 spa->spa_config_splitting = NULL;
2266         }

2268         /*
2269          * Initialize internal SPA structures.
2270          */
2271         spa->spa_state = POOL_STATE_ACTIVE;
2272         spa->spa_ubsync = spa->spa_uberblock;
2273         spa->spa_verify_min_txg = spa->spa_extreme_rewind ?
2274             TXG_INITIAL - 1 : spa_last_synced_txg(spa) - TXG_DEFER_SIZE - 1;
2275         spa->spa_first_txg = spa->spa_last_ubsync_txg ?
2276             spa->spa_last_ubsync_txg : spa_last_synced_txg(spa) + 1;
2277         spa->spa_claim_max_txg = spa->spa_first_txg;
2278         spa->spa_prev_software_version = ub->ub_software_version;

2280         error = dsl_pool_init(spa, spa->spa_first_txg, &spa->spa_dsl_pool);
2281         if (error)
2282                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2283         spa->spa_meta_objset = spa->spa_dsl_pool->dp_meta_objset;

2285         if (spa_dir_prop(spa, DMU_POOL_CONFIG, &spa->spa_config_object) != 0)
2286                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2288         if (spa_version(spa) >= SPA_VERSION_FEATURES) {
2289                 boolean_t missing_feat_read = B_FALSE;
2290                 nvlist_t *unsup_feat, *enabled_feat;

2292                 if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_READ,
2293                     &spa->spa_feat_for_read_obj) != 0) {
2294                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2295                 }

2297                 if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_WRITE,
2298                     &spa->spa_feat_for_write_obj) != 0) {
2299                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2300                 }

2302                 if (spa_dir_prop(spa, DMU_POOL_FEATURE_DESCRIPTIONS,
```

```
2303                    &spa->spa_feat_desc_obj) != 0) {
2304                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2305                }
2307                enabled_feat = fnvlist_alloc();
2308                unsup_feat = fnvlist_alloc();
2310                if (!spa_features_check(spa, B_FALSE,
2311                    unsup_feat, enabled_feat))
2312                        missing_feat_read = B_TRUE;
2314                if (spa_writeable(spa) || state == SPA_LOAD_TRYIMPORT) {
2315                        if (!spa_features_check(spa, B_TRUE,
2316                            unsup_feat, enabled_feat)) {
2317                                missing_feat_write = B_TRUE;
2318                        }
2319                }
2321                fnvlist_add_nvlist(spa->spa_load_info,
2322                    ZPOOL_CONFIG_ENABLED_FEAT, enabled_feat);
2324                if (!nvlist_empty(unsup_feat)) {
2325                        fnvlist_add_nvlist(spa->spa_load_info,
2326                            ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat);
2327                }
2329                fnvlist_free(enabled_feat);
2330                fnvlist_free(unsup_feat);
2332                if (!missing_feat_read) {
2333                        fnvlist_add_boolean(spa->spa_load_info,
2334                            ZPOOL_CONFIG_CAN_RDONLY);
2335                }
2337                /*
2338                 * If the state is SPA_LOAD_TRYIMPORT, our objective is
2339                 * twofold: to determine whether the pool is available for
2340                 * import in read-write mode and (if it is not) whether the
2341                 * pool is available for import in read-only mode. If the pool
2342                 * is available for import in read-write mode, it is displayed
2343                 * as available in userland; if it is not available for import
2344                 * in read-only mode, it is displayed as unavailable in
2345                 * userland. If the pool is available for import in read-only
2346                 * mode but not read-write mode, it is displayed as unavailable
2347                 * in userland with a special note that the pool is actually
2348                 * available for open in read-only mode.
2349                 *
2350                 * As a result, if the state is SPA_LOAD_TRYIMPORT and we are
2351                 * missing a feature for write, we must first determine whether
2352                 * the pool can be opened read-only before returning to
2353                 * userland in order to know whether to display the
2354                 * abovementioned note.
2355                 */
2356                if (missing_feat_read || (missing_feat_write &&
2357                    spa_writeable(spa))) {
2358                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2359                            ENOTSUP));
2360                }
2362                /*
2363                 * Load refcounts for ZFS features from disk into an in-memory
2364                 * cache during SPA initialization.
2365                 */
2366                for (spa_feature_t i = 0; i < SPA_FEATURES; i++) {
2367                        uint64_t refcount;
```



```
2369                        error = feature_get_refcount_from_disk(spa,
2370                            &spa_feature_table[i], &refcount);
2371                        if (error == 0) {
2372                                spa->spa_feat_refcount_cache[i] = refcount;
2373                        } else if (error == ENOTSUP) {
2374                                spa->spa_feat_refcount_cache[i] =
2375                                    SPA_FEATURE_DISABLED;
2376                        } else {
2377                                return (spa_vdev_err(rvd,
2378                                    VDEV_AUX_CORRUPT_DATA, EIO));
2379                        }
2380                }
2381        }
2383        if (spa_feature_is_active(spa, SPA_FEATURE_ENABLED_TXG)) {
2384                if (spa_dir_prop(spa, DMU_POOL_FEATURE_ENABLED_TXG,
2385                    &spa->spa_feat_enabled_txg_obj) != 0)
2386                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2387        }
2389        spa->spa_is_initializing = B_TRUE;
2390        error = dsl_pool_open(spa->spa_dsl_pool);
2391        spa->spa_is_initializing = B_FALSE;
2392        if (error != 0)
2393                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2395        if (!mosconfig) {
2396                uint64_t hostid;
2397                nvlist_t *policy = NULL, *nvconfig;
2399                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2400                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2402                if (!spa_is_root(spa) && nvlist_lookup_uint64(nvconfig,
2403                    ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
2404                        char *hostname;
2405                        unsigned long myhostid = 0;
2407                        VERIFY(nvlist_lookup_string(nvconfig,
2408                            ZPOOL_CONFIG_HOSTNAME, &hostname) == 0);
2410 #ifdef  _KERNEL
2411                        myhostid = zone_get_hostid(NULL);
2412 #else   /* _KERNEL */
2413                        /*
2414                         * We're emulating the system's hostid in userland, so
2415                         * we can't use zone_get_hostid().
2416                         */
2417                        (void) ddi_strtoul(hw_serial, NULL, 10, &myhostid);
2418 #endif  /* _KERNEL */
2419                        if (hostid != 0 && myhostid != 0 &&
2420                            hostid != myhostid) {
2421                                nvlist_free(nvconfig);
2422                                cmn_err(CE_WARN, "pool '%s' could not be "
2423                                    "loaded as it was last accessed by "
2424                                    "another system (host: %s hostid: 0x%lx). "
2425                                    "See: http://illumos.org/msg/ZFS-8000-EY",
2426                                    spa_name(spa), hostname,
2427                                    (unsigned long)hostid);
2428                                return (SET_ERROR(EBADF));
2429                        }
2430                }
2431                if (nvlist_lookup_nvlist(spa->spa_config,
2432                    ZPOOL_REWIND_POLICY, &policy) == 0)
2433                        VERIFY(nvlist_add_nvlist(nvconfig,
2434                            ZPOOL_REWIND_POLICY, policy) == 0);
```

```
2436                 spa_config_set(spa, nvconfig);
2437                 spa_unload(spa);
2438                 spa_deactivate(spa);
2439                 spa_activate(spa, orig_mode);

2441                 return (spa_load(spa, state, SPA_IMPORT_EXISTING, B_TRUE));
2442         }

2444         if (spa_dir_prop(spa, DMU_POOL_SYNC_BPOBJ, &obj) != 0)
2445                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2446         error = bpobj_open(&spa->spa_deferred_bpobj, spa->spa_meta_objset, obj);
2447         if (error != 0)
2448                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2450         /*
2451          * Load the bit that tells us to use the new accounting function
2452          * (raid-z deflation).  If we have an older pool, this will not
2453          * be present.
2454          */
2455         error = spa_dir_prop(spa, DMU_POOL_DEFLATE, &spa->spa_deflate);
2456         if (error != 0 && error != ENOENT)
2457                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2459         error = spa_dir_prop(spa, DMU_POOL_CREATION_VERSION,
2460             &spa->spa_creation_version);
2461         if (error != 0 && error != ENOENT)
2462                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2464         /*
2465          * Load the persistent error log.  If we have an older pool, this will
2466          * not be present.
2467          */
2468         error = spa_dir_prop(spa, DMU_POOL_ERRLOG_LAST, &spa->spa_errlog_last);
2469         if (error != 0 && error != ENOENT)
2470                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2472         error = spa_dir_prop(spa, DMU_POOL_ERRLOG_SCRUB,
2473             &spa->spa_errlog_scrub);
2474         if (error != 0 && error != ENOENT)
2475                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2477         /*
2478          * Load the history object.  If we have an older pool, this
2479          * will not be present.
2480          */
2481         error = spa_dir_prop(spa, DMU_POOL_HISTORY, &spa->spa_history);
2482         if (error != 0 && error != ENOENT)
2483                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2485         /*
2486          * If we're assembling the pool from the split-off vdevs of
2487          * an existing pool, we don't want to attach the spares & cache
2488          * devices.
2489          */

2491         /*
2492          * Load any hot spares for this pool.
2493          */
2494         error = spa_dir_prop(spa, DMU_POOL_SPARES, &spa->spa_spares.sav_object);
2495         if (error != 0 && error != ENOENT)
2496                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2497         if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2498                 ASSERT(spa_version(spa) >= SPA_VERSION_SPARES);
2499                 if (load_nvlist(spa, spa->spa_spares.sav_object,
2500                     &spa->spa_spares.sav_config) != 0)
```



```
2501                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2503                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2504                 spa_load_spares(spa);
2505                 spa_config_exit(spa, SCL_ALL, FTAG);
2506         } else if (error == 0) {
2507                 spa->spa_spares.sav_sync = B_TRUE;
2508         }

2510         /*
2511          * Load any level 2 ARC devices for this pool.
2512          */
2513         error = spa_dir_prop(spa, DMU_POOL_L2CACHE,
2514             &spa->spa_l2cache.sav_object);
2515         if (error != 0 && error != ENOENT)
2516                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2517         if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2518                 ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
2519                 if (load_nvlist(spa, spa->spa_l2cache.sav_object,
2520                     &spa->spa_l2cache.sav_config) != 0)
2521                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2523                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2524                 spa_load_l2cache(spa);
2525                 spa_config_exit(spa, SCL_ALL, FTAG);
2526         } else if (error == 0) {
2527                 spa->spa_l2cache.sav_sync = B_TRUE;
2528         }

2530         spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);

2532         error = spa_dir_prop(spa, DMU_POOL_PROPS, &spa->spa_pool_props_object);
2533         if (error && error != ENOENT)
2534                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2536         if (error == 0) {
2537                 uint64_t autoreplace;

2539                 spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
2540                 spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
2541                 spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
2542                 spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
2543                 spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
2544                 spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO,
2545                     &spa->spa_dedup_ditto);

2547                 spa->spa_autoreplace = (autoreplace != 0);
2548         }

2550         /*
2551          * If the 'autoreplace' property is set, then post a resource notifying
2552          * the ZFS DE that it should not issue any faults for unopenable
2553          * devices.  We also iterate over the vdevs, and post a sysevent for any
2554          * unopenable vdevs so that the normal autoreplace handler can take
2555          * over.
2556          */
2557         if (spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT) {
2558                 spa_check_removed(spa->spa_root_vdev);
2559                 /*
2560                  * For the import case, this is done in spa_import(), because
2561                  * at this point we're using the spare definitions from
2562                  * the MOS config, not necessarily from the userland config.
2563                  */
2564                 if (state != SPA_LOAD_IMPORT) {
2565                         spa_aux_check_removed(&spa->spa_spares);
2566                         spa_aux_check_removed(&spa->spa_l2cache);
```

```
2567                }
2568        }

2570        /*
2571         * Load the vdev state for all toplevel vdevs.
2572         */
2573        vdev_load(rvd);

2575        /*
2576         * Propagate the leaf DTLs we just loaded all the way up the tree.
2577         */
2578        spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2579        vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
2580        spa_config_exit(spa, SCL_ALL, FTAG);

2582        /*
2583         * Load the DDTs (dedup tables).
2584         */
2585        error = ddt_load(spa);
2586        if (error != 0)
2587                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2589        spa_update_dspace(spa);

2591        /*
2592         * Validate the config, using the MOS config to fill in any
2593         * information which might be missing.  If we fail to validate
2594         * the config then declare the pool unfit for use. If we're
2595         * assembling a pool from a split, the log is not transferred
2596         * over.
2597         */
2598        if (type != SPA_IMPORT_ASSEMBLE) {
2599                nvlist_t *nvconfig;

2601                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2602                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2604                if (!spa_config_valid(spa, nvconfig)) {
2605                        nvlist_free(nvconfig);
2606                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM,
2607                            ENXIO));
2608                }
2609                nvlist_free(nvconfig);

2611                /*
2612                 * Now that we've validated the config, check the state of the
2613                 * root vdev.  If it can't be opened, it indicates one or
2614                 * more toplevel vdevs are faulted.
2615                 */
2616                if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2617                        return (SET_ERROR(ENXIO));

2619                if (spa_check_logs(spa)) {
2620                        *ereport = FM_EREPORT_ZFS_LOG_REPLAY;
2621                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
2622                }
2623        }

2625        if (missing_feat_write) {
2626                ASSERT(state == SPA_LOAD_TRYIMPORT);

2628                /*
2629                 * At this point, we know that we can open the pool in
2630                 * read-only mode but not read-write mode. We now have enough
2631                 * information and can return to userland.
2632                 */
```



```
2633                return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
2634        }

2636        /*
2637         * We've successfully opened the pool, verify that we're ready
2638         * to start pushing transactions.
2639         */
2640        if (state != SPA_LOAD_TRYIMPORT) {
2641                if (error = spa_load_verify(spa))
2642                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2643                            error));
2644        }

2646        if (spa_writeable(spa) && (state == SPA_LOAD_RECOVER ||
2647            spa->spa_load_max_txg == UINT64_MAX)) {
2648                dmu_tx_t *tx;
2649                int need_update = B_FALSE;
2650                dsl_pool_t *dp = spa_get_dsl(spa);
2651 #endif /* ! codereview */

2653                ASSERT(state != SPA_LOAD_TRYIMPORT);

2655                /*
2656                 * Claim log blocks that haven't been committed yet.
2657                 * This must all happen in a single txg.
2658                 * Note: spa_claim_max_txg is updated by spa_claim_notify(),
2659                 * invoked from zil_claim_log_block()'s i/o done callback.
2660                 * Price of rollback is that we abandon the log.
2661                 */
2662                spa->spa_claiming = B_TRUE;

2664                tx = dmu_tx_create_assigned(dp, spa_first_txg(spa));
2665                (void) dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
2644                tx = dmu_tx_create_assigned(spa_get_dsl(spa),
2645                    spa_first_txg(spa));
2646                (void) dmu_objset_find(spa_name(spa),
2666                    zil_claim, tx, DS_FIND_CHILDREN);
2667                dmu_tx_commit(tx);

2669                spa->spa_claiming = B_FALSE;

2671                spa_set_log_state(spa, SPA_LOG_GOOD);
2672                spa->spa_sync_on = B_TRUE;
2673                txg_sync_start(spa->spa_dsl_pool);

2675                /*
2676                 * Wait for all claims to sync.  We sync up to the highest
2677                 * claimed log block birth time so that claimed log blocks
2678                 * don't appear to be from the future.  spa_claim_max_txg
2679                 * will have been set for us by either zil_check_log_chain()
2680                 * (invoked from spa_check_logs()) or zil_claim() above.
2681                 */
2682                txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_txg);

2684                /*
2685                 * If the config cache is stale, or we have uninitialized
2686                 * metaslabs (see spa_vdev_add()), then update the config.
2687                 *
2688                 * If this is a verbatim import, trust the current
2689                 * in-core spa_config and update the disk labels.
2690                 */
2691                if (config_cache_txg != spa->spa_config_txg ||
2692                    state == SPA_LOAD_IMPORT ||
2693                    state == SPA_LOAD_RECOVER ||
2694                    (spa->spa_import_flags & ZFS_IMPORT_VERBATIM))
2695                        need_update = B_TRUE;
```

```
2697                 for (int c = 0; c < rvd->vdev_children; c++)
2698                         if (rvd->vdev_child[c]->vdev_ms_array == 0)
2699                                 need_update = B_TRUE;

2701                 /*
2702                  * Update the config cache asychronously in case we're the
2703                  * root pool, in which case the config cache isn't writable yet.
2704                  */
2705                 if (need_update)
2706                         spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);

2708                 /*
2709                  * Check all DTLs to see if anything needs resilvering.
2710                  */
2711                 if (!dsl_scan_resilvering(spa->spa_dsl_pool) &&
2712                     vdev_resilver_needed(rvd, NULL, NULL))
2713                         spa_async_request(spa, SPA_ASYNC_RESILVER);

2715                 /*
2716                  * Log the fact that we booted up (so that we can detect if
2717                  * we rebooted in the middle of an operation).
2718                  */
2719                 spa_history_log_version(spa, "open");

2721                 /*
2722                  * Delete any inconsistent datasets.
2723                  */
2724                 (void) dmu_objset_find(spa_name(spa),
2725                     dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);

2727                 /*
2728                  * Clean up any stale temporary dataset userrefs.
2729                  */
2730                 dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
2731         }

2733         return (0);
2734 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   29556 Fri Oct 31 10:14:51 2014
new/usr/src/uts/common/fs/zfs/sys/dmu.h
5269 zfs: zpool import slow
While importing a pool all objsets are enumerated twice, once to check
the zil log chains and once to claim them. On pools with many datasets
this process might take a substantial amount of time.
Speed up the process by parallelizing it utilizing a taskq. The number
of parallel tasks is limited to 4 times the number of leaf vdevs.
**********************************************************
______unchanged_portion_omitted_

 235 void byteswap_uint64_array(void *buf, size_t size);
 236 void byteswap_uint32_array(void *buf, size_t size);
 237 void byteswap_uint16_array(void *buf, size_t size);
 238 void byteswap_uint8_array(void *buf, size_t size);
 239 void zap_byteswap(void *buf, size_t size);
 240 void zfs_oldacl_byteswap(void *buf, size_t size);
 241 void zfs_acl_byteswap(void *buf, size_t size);
 242 void zfs_znode_byteswap(void *buf, size_t size);

 244 #define DS_FIND_SNAPSHOTS       (1<<0)
 245 #define DS_FIND_CHILDREN        (1<<1)

 247 /*
 248  * The maximum number of bytes that can be accessed as part of one
 249  * operation, including metadata.
 250  */
 251 #define DMU_MAX_ACCESS (10<<20) /* 10MB */
 252 #define DMU_MAX_DELETEBLKCNT (20480) /* ~5MB of indirect blocks */

 254 #define DMU_USERUSED_OBJECT     (-1ULL)
 255 #define DMU_GROUPUSED_OBJECT    (-2ULL)

 257 /*
 258  * artificial blkids for bonus buffer and spill blocks
 259  */
 260 #define DMU_BONUS_BLKID         (-1ULL)
 261 #define DMU_SPILL_BLKID         (-2ULL)
 262 /*
 263  * Public routines to create, destroy, open, and close objsets.
 264  */
 265 int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
 266 int dmu_objset_own(const char *name, dmu_objset_type_t type,
 267     boolean_t readonly, void *tag, objset_t **osp);
 268 void dmu_objset_rele(objset_t *os, void *tag);
 269 void dmu_objset_rele_obj(objset_t *os, void *tag);
 270 #endif /* ! codereview */
 271 void dmu_objset_disown(objset_t *os, void *tag);
 272 int dmu_objset_open_ds(struct dsl_dataset *ds, objset_t **osp);

 274 void dmu_objset_evict_dbufs(objset_t *os);
 275 int dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
 276     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg);
 277 int dmu_objset_clone(const char *name, const char *origin);
 278 int dsl_destroy_snapshots_nvl(struct nvlist *snaps, boolean_t defer,
 279     struct nvlist *errlist);
 280 int dmu_objset_snapshot_one(const char *fsname, const char *snapname);
 281 int dmu_objset_snapshot_tmp(const char *, const char *, int);
 282 int dmu_objset_find(char *name, int func(const char *, void *), void *arg,
 283     int flags);
 284 void dmu_objset_byteswap(void *buf, size_t size);
 285 int dsl_dataset_rename_snapshot(const char *fsname,
 286     const char *oldsnapname, const char *newsnapname, boolean_t recursive);

 288 typedef struct dmu_buf {
```



```
 289         uint64_t db_object;             /* object that this buffer is part of */
 290         uint64_t db_offset;             /* byte offset in this object */
 291         uint64_t db_size;               /* size of buffer in bytes */
 292         void *db_data;                  /* data in buffer */
 293 } dmu_buf_t;

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

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

 324 /*
 325  * Allocate an object from this objset.  The range of object numbers
 326  * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
 327  *
 328  * The transaction must be assigned to a txg.  The newly allocated
 329  * object will be "held" in the transaction (ie. you can modify the
 330  * newly allocated object in this transaction).
 331  *
 332  * dmu_object_alloc() chooses an object and returns it in *objectp.
 333  *
 334  * dmu_object_claim() allocates a specific object number.  If that
 335  * number is already allocated, it fails and returns EEXIST.
 336  *
 337  * Return 0 on success, or ENOSPC or EEXIST as specified above.
 338  */
 339 uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
 340     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 341 int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 342     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 343 int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 344     int blocksize, dmu_object_type_t bonustype, int bonuslen);

 346 /*
 347  * Free an object from this objset.
 348  *
 349  * The object's data will be freed as well (ie. you don't need to call
 350  * dmu_free(object, 0, -1, tx)).
 351  *
 352  * The object need not be held in the transaction.
 353  *
 354  * If there are any holds on this object's buffers (via dmu_buf_hold()),
```

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

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

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

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

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

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

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

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



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

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

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

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

/*
 * Convenience functions.
 *
 * Canfail routines will return 0 on success, or an errno if there is a
 * nonrecoverable I/O error.
 */
#define DMU_READ_PREFETCH       0 /* prefetch */
#define DMU_READ_NO_PREFETCH    1 /* don't prefetch */
int dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        void *buf, uint32_t flags);
void dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        const void *buf, dmu_tx_t *tx);
void dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        dmu_tx_t *tx);
int dmu_read_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size);
int dmu_write_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size,
```

```
619     dmu_tx_t *tx);
620 int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
621     dmu_tx_t *tx);
622 int dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset,
623     uint64_t size, struct page *pp, dmu_tx_t *tx);
624 struct arc_buf *dmu_request_arcbuf(dmu_buf_t *handle, int size);
625 void dmu_return_arcbuf(struct arc_buf *buf);
626 void dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, struct arc_buf *buf,
627     dmu_tx_t *tx);
628 int dmu_xuio_init(struct xuio *uio, int niov);
629 void dmu_xuio_fini(struct xuio *uio);
630 int dmu_xuio_add(struct xuio *uio, struct arc_buf *abuf, offset_t off,
631     size_t n);
632 int dmu_xuio_cnt(struct xuio *uio);
633 struct arc_buf *dmu_xuio_arcbuf(struct xuio *uio, int i);
634 void dmu_xuio_clear(struct xuio *uio, int i);
635 void xuio_stat_wbuf_copied();
636 void xuio_stat_wbuf_nocopy();

638 extern int zfs_prefetch_disable;

640 /*
641  * Asynchronously try to read in the data.
642  */
643 void dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset,
644     uint64_t len);

646 typedef struct dmu_object_info {
647         /* All sizes are in bytes unless otherwise indicated. */
648         uint32_t doi_data_block_size;
649         uint32_t doi_metadata_block_size;
650         dmu_object_type_t doi_type;
651         dmu_object_type_t doi_bonus_type;
652         uint64_t doi_bonus_size;
653         uint8_t doi_indirection;                /* 2 = dnode->indirect->data */
654         uint8_t doi_checksum;
655         uint8_t doi_compress;
656         uint8_t doi_pad[5];
657         uint64_t doi_physical_blocks_512;       /* data + metadata, 512b blks */
658         uint64_t doi_max_offset;
659         uint64_t doi_fill_count;                /* number of non-empty blocks */
660 } dmu_object_info_t;

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

664 typedef struct dmu_object_type_info {
665         dmu_object_byteswap_t   ot_byteswap;
666         boolean_t               ot_metadata;
667         char                    *ot_name;
668 } dmu_object_type_info_t;

670 typedef struct dmu_object_byteswap_info {
671         arc_byteswap_func_t     *ob_func;
672         char                    *ob_name;
673 } dmu_object_byteswap_info_t;

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

678 /*
679  * Get information on a DMU object.
680  *
681  * Return 0 on success or ENOENT if object is not allocated.
682  *
683  * If doi is NULL, just indicates whether the object exists.
684  */
```



```
685 int dmu_object_info(objset_t *os, uint64_t object, dmu_object_info_t *doi);
686 /* Like dmu_object_info, but faster if you have a held dnode in hand. */
687 void dmu_object_info_from_dnode(struct dnode *dn, dmu_object_info_t *doi);
688 /* Like dmu_object_info, but faster if you have a held dbuf in hand. */
689 void dmu_object_info_from_db(dmu_buf_t *db, dmu_object_info_t *doi);
690 /*
691  * Like dmu_object_info_from_db, but faster still when you only care about
692  * the size.  This is specifically optimized for zfs_getattr().
693  */
694 void dmu_object_size_from_db(dmu_buf_t *db, uint32_t *blksize,
695     u_longlong_t *nblk512);

697 typedef struct dmu_objset_stats {
698         uint64_t dds_num_clones; /* number of clones of this */
699         uint64_t dds_creation_txg;
700         uint64_t dds_guid;
701         dmu_objset_type_t dds_type;
702         uint8_t dds_is_snapshot;
703         uint8_t dds_inconsistent;
704         char dds_origin[MAXNAMELEN];
705 } dmu_objset_stats_t;

707 /*
708  * Get stats on a dataset.
709  */
710 void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);

712 /*
713  * Add entries to the nvlist for all the objset's properties.  See
714  * zfs_prop_table[] and zfs(1m) for details on the properties.
715  */
716 void dmu_objset_stats(objset_t *os, struct nvlist *nv);

718 /*
719  * Get the space usage statistics for statvfs().
720  *
721  * refdbytes is the amount of space "referenced" by this objset.
722  * availbytes is the amount of space available to this objset, taking
723  * into account quotas & reservations, assuming that no other objsets
724  * use the space first.  These values correspond to the 'referenced' and
725  * 'available' properties, described in the zfs(1m) manpage.
726  *
727  * usedobjs and availobjs are the number of objects currently allocated,
728  * and available.
729  */
730 void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
731     uint64_t *usedobjsp, uint64_t *availobjsp);

733 /*
734  * The fsid_guid is a 56-bit ID that can change to avoid collisions.
735  * (Contrast with the ds_guid which is a 64-bit ID that will never
736  * change, so there is a small probability that it will collide.)
737  */
738 uint64_t dmu_objset_fsid_guid(objset_t *os);

740 /*
741  * Get the [cm]time for an objset's snapshot dir
742  */
743 timestruc_t dmu_objset_snap_cmtime(objset_t *os);

745 int dmu_objset_is_snapshot(objset_t *os);

747 extern struct spa *dmu_objset_spa(objset_t *os);
748 extern struct zilog *dmu_objset_zil(objset_t *os);
749 extern struct dsl_pool *dmu_objset_pool(objset_t *os);
750 extern struct dsl_dataset *dmu_objset_ds(objset_t *os);
```

```
751 extern void dmu_objset_name(objset_t *os, char *buf);
752 extern dmu_objset_type_t dmu_objset_type(objset_t *os);
753 extern uint64_t dmu_objset_id(objset_t *os);
754 extern zfs_sync_type_t dmu_objset_syncprop(objset_t *os);
755 extern zfs_logbias_op_t dmu_objset_logbias(objset_t *os);
756 extern int dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
757     uint64_t *id, uint64_t *offp, boolean_t *case_conflict);
758 extern int dmu_snapshot_realname(objset_t *os, char *name, char *real,
759     int maxlen, boolean_t *conflict);
760 extern int dmu_dir_list_next(objset_t *os, int namelen, char *name,
761     uint64_t *idp, uint64_t *offp);

763 typedef int objset_used_cb_t(dmu_object_type_t bonustype,
764     void *bonus, uint64_t *userp, uint64_t *groupp);
765 extern void dmu_objset_register_type(dmu_objset_type_t ost,
766     objset_used_cb_t *cb);
767 extern void dmu_objset_set_user(objset_t *os, void *user_ptr);
768 extern void *dmu_objset_get_user(objset_t *os);

770 /*
771  * Return the txg number for the given assigned transaction.
772  */
773 uint64_t dmu_tx_get_txg(dmu_tx_t *tx);

775 /*
776  * Synchronous write.
777  * If a parent zio is provided this function initiates a write on the
778  * provided buffer as a child of the parent zio.
779  * In the absence of a parent zio, the write is completed synchronously.
780  * At write completion, blk is filled with the bp of the written block.
781  * Note that while the data covered by this function will be on stable
782  * storage when the write completes this new data does not become a
783  * permanent part of the file until the associated transaction commits.
784  */

786 /*
787  * {zfs,zvol,ztest}_get_done() args
788  */
789 typedef struct zgd {
790         struct zilog    *zgd_zilog;
791         struct blkptr   *zgd_bp;
792         dmu_buf_t       *zgd_db;
793         struct rl       *zgd_rl;
794         void            *zgd_private;
795 } zgd_t;

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

800 /*
801  * Find the next hole or data block in file starting at *off
802  * Return found offset in *off. Return ESRCH for end of file.
803  */
804 int dmu_offset_next(objset_t *os, uint64_t object, boolean_t hole,
805     uint64_t *off);

807 /*
808  * Initial setup and final teardown.
809  */
810 extern void dmu_init(void);
811 extern void dmu_fini(void);

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



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

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

825 extern int zfs_mdcomp_disable;

827 #ifdef  __cplusplus
828 }
829 #endif

831 #endif  /* _SYS_DMU_H */
```

```
**********************************************************
    5875 Fri Oct 31 10:14:51 2014
new/usr/src/uts/common/fs/zfs/sys/dmu_objset.h
5269 zfs: zpool import slow
While importing a pool all objsets are enumerated twice, once to check
the zil log chains and once to claim them. On pools with many datasets
this process might take a substantial amount of time.
Speed up the process by parallelizing it utilizing a taskq. The number
of parallel tasks is limited to 4 times the number of leaf vdevs.
**********************************************************
______unchanged_portion_omitted_

#define DMU_META_OBJSET         0
#define DMU_META_DNODE_OBJECT   0
#define DMU_OBJECT_IS_SPECIAL(obj) ((int64_t)(obj) <= 0)
#define DMU_META_DNODE(os)      ((os)->os_meta_dnode.dnh_dnode)
#define DMU_USERUSED_DNODE(os)  ((os)->os_userused_dnode.dnh_dnode)
#define DMU_GROUPUSED_DNODE(os) ((os)->os_groupused_dnode.dnh_dnode)

#define DMU_OS_IS_L2CACHEABLE(os)                               \
        ((os)->os_secondary_cache == ZFS_CACHE_ALL ||           \
        (os)->os_secondary_cache == ZFS_CACHE_METADATA)

#define DMU_OS_IS_L2COMPRESSIBLE(os)    (zfs_mdcomp_disable == B_FALSE)

/* called from zpl */
int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
int dmu_objset_own(const char *name, dmu_objset_type_t type,
    boolean_t readonly, void *tag, objset_t **osp);
int dmu_objset_own_obj(dsl_pool_t *dp, uint64_t obj, dmu_objset_type_t type,
    boolean_t readonly, void *tag, objset_t **osp);
#endif /* ! codereview */
void dmu_objset_refresh_ownership(objset_t *os, void *tag);
void dmu_objset_rele(objset_t *os, void *tag);
void dmu_objset_disown(objset_t *os, void *tag);
int dmu_objset_from_ds(struct dsl_dataset *ds, objset_t **osp);

void dmu_objset_stats(objset_t *os, nvlist_t *nv);
void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);
void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
    uint64_t *usedobjsp, uint64_t *availobjsp);
uint64_t dmu_objset_fsid_guid(objset_t *os);
int dmu_objset_find_dp(struct dsl_pool *dp, uint64_t ddobj,
    int func(struct dsl_pool *, struct dsl_dataset *, void *),
    void *arg, int flags);
int dmu_objset_prefetch(const char *name, void *arg);
void dmu_objset_evict_dbufs(objset_t *os);
timestruc_t dmu_objset_snap_cmtime(objset_t *os);

/* called from dsl */
void dmu_objset_sync(objset_t *os, zio_t *zio, dmu_tx_t *tx);
boolean_t dmu_objset_is_dirty(objset_t *os, uint64_t txg);
objset_t *dmu_objset_create_impl(spa_t *spa, struct dsl_dataset *ds,
    blkptr_t *bp, dmu_objset_type_t type, dmu_tx_t *tx);
int dmu_objset_open_impl(spa_t *spa, struct dsl_dataset *ds, blkptr_t *bp,
    objset_t **osp);
void dmu_objset_evict(objset_t *os);
void dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx);
void dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx);
boolean_t dmu_objset_userused_enabled(objset_t *os);
int dmu_objset_userspace_upgrade(objset_t *os);
boolean_t dmu_objset_userspace_present(objset_t *os);
int dmu_fsname(const char *snapname, char *buf);

void dmu_objset_init(void);
void dmu_objset_fini(void);
```



```
#ifdef  __cplusplus
}
#endif

#endif /* _SYS_DMU_OBJSET_H */
```

```
**********************************************************
    5825 Fri Oct 31 10:14:52 2014
new/usr/src/uts/common/fs/zfs/sys/vdev.h
5269 zfs: zpool import slow
While importing a pool all objsets are enumerated twice, once to check
the zil log chains and once to claim them. On pools with many datasets
this process might take a substantial amount of time.
Speed up the process by parallelizing it utilizing a taskq. The number
of parallel tasks is limited to 4 times the number of leaf vdevs.
**********************************************************
______unchanged_portion_omitted_

  48 extern boolean_t zfs_nocacheflush;

  50 extern int vdev_open(vdev_t *);
  51 extern void vdev_open_children(vdev_t *);
  52 extern boolean_t vdev_uses_zvols(vdev_t *);
  53 extern int vdev_validate(vdev_t *, boolean_t);
  54 extern void vdev_close(vdev_t *);
  55 extern int vdev_create(vdev_t *, uint64_t txg, boolean_t isreplace);
  56 extern void vdev_reopen(vdev_t *);
  57 extern int vdev_validate_aux(vdev_t *vd);
  58 extern zio_t *vdev_probe(vdev_t *vd, zio_t *pio);

  60 extern boolean_t vdev_is_bootable(vdev_t *vd);
  61 extern vdev_t *vdev_lookup_top(spa_t *spa, uint64_t vdev);
  62 extern vdev_t *vdev_lookup_by_guid(vdev_t *vd, uint64_t guid);
  63 extern int vdev_count_leaves(spa_t *spa);
  64 #endif /* ! codereview */
  65 extern void vdev_dtl_dirty(vdev_t *vd, vdev_dtl_type_t d,
  66     uint64_t txg, uint64_t size);
  67 extern boolean_t vdev_dtl_contains(vdev_t *vd, vdev_dtl_type_t d,
  68     uint64_t txg, uint64_t size);
  69 extern boolean_t vdev_dtl_empty(vdev_t *vd, vdev_dtl_type_t d);
  70 extern void vdev_dtl_reassess(vdev_t *vd, uint64_t txg, uint64_t scrub_txg,
  71     int scrub_done);
  72 extern boolean_t vdev_dtl_required(vdev_t *vd);
  73 extern boolean_t vdev_resilver_needed(vdev_t *vd,
  74     uint64_t *minp, uint64_t *maxp);

  76 extern void vdev_hold(vdev_t *);
  77 extern void vdev_rele(vdev_t *);

  79 extern int vdev_metaslab_init(vdev_t *vd, uint64_t txg);
  80 extern void vdev_metaslab_fini(vdev_t *vd);
  81 extern void vdev_metaslab_set_size(vdev_t *);
  82 extern void vdev_expand(vdev_t *vd, uint64_t txg);
  83 extern void vdev_split(vdev_t *vd);
  84 extern void vdev_deadman(vdev_t *vd);


  87 extern void vdev_get_stats(vdev_t *vd, vdev_stat_t *vs);
  88 extern void vdev_clear_stats(vdev_t *vd);
  89 extern void vdev_stat_update(zio_t *zio, uint64_t psize);
  90 extern void vdev_scan_stat_init(vdev_t *vd);
  91 extern void vdev_propagate_state(vdev_t *vd);
  92 extern void vdev_set_state(vdev_t *vd, boolean_t isopen, vdev_state_t state,
  93     vdev_aux_t aux);

  95 extern void vdev_space_update(vdev_t *vd,
  96     int64_t alloc_delta, int64_t defer_delta, int64_t space_delta);

  98 extern uint64_t vdev_psize_to_asize(vdev_t *vd, uint64_t psize);

 100 extern int vdev_fault(spa_t *spa, uint64_t guid, vdev_aux_t aux);
 101 extern int vdev_degrade(spa_t *spa, uint64_t guid, vdev_aux_t aux);
```



```
 102 extern int vdev_online(spa_t *spa, uint64_t guid, uint64_t flags,
 103     vdev_state_t *);
 104 extern int vdev_offline(spa_t *spa, uint64_t guid, uint64_t flags);
 105 extern void vdev_clear(spa_t *spa, vdev_t *vd);

 107 extern boolean_t vdev_is_dead(vdev_t *vd);
 108 extern boolean_t vdev_readable(vdev_t *vd);
 109 extern boolean_t vdev_writeable(vdev_t *vd);
 110 extern boolean_t vdev_allocatable(vdev_t *vd);
 111 extern boolean_t vdev_accessible(vdev_t *vd, zio_t *zio);

 113 extern void vdev_cache_init(vdev_t *vd);
 114 extern void vdev_cache_fini(vdev_t *vd);
 115 extern boolean_t vdev_cache_read(zio_t *zio);
 116 extern void vdev_cache_write(zio_t *zio);
 117 extern void vdev_cache_purge(vdev_t *vd);

 119 extern void vdev_queue_init(vdev_t *vd);
 120 extern void vdev_queue_fini(vdev_t *vd);
 121 extern zio_t *vdev_queue_io(zio_t *zio);
 122 extern void vdev_queue_io_done(zio_t *zio);

 124 extern void vdev_config_dirty(vdev_t *vd);
 125 extern void vdev_config_clean(vdev_t *vd);
 126 extern int vdev_config_sync(vdev_t **svd, int svdcount, uint64_t txg,
 127     boolean_t);

 129 extern void vdev_state_dirty(vdev_t *vd);
 130 extern void vdev_state_clean(vdev_t *vd);

 132 typedef enum vdev_config_flag {
 133         VDEV_CONFIG_SPARE = 1 << 0,
 134         VDEV_CONFIG_L2CACHE = 1 << 1,
 135         VDEV_CONFIG_REMOVING = 1 << 2
 136 } vdev_config_flag_t;

 138 extern void vdev_top_config_generate(spa_t *spa, nvlist_t *config);
 139 extern nvlist_t *vdev_config_generate(spa_t *spa, vdev_t *vd,
 140     boolean_t getstats, vdev_config_flag_t flags);

 142 /*
 143  * Label routines
 144  */
 145 struct uberblock;
 146 extern uint64_t vdev_label_offset(uint64_t psize, int l, uint64_t offset);
 147 extern int vdev_label_number(uint64_t psize, uint64_t offset);
 148 extern nvlist_t *vdev_label_read_config(vdev_t *vd, uint64_t txg);
 149 extern void vdev_uberblock_load(vdev_t *, struct uberblock *, nvlist_t **);

 151 typedef enum {
 152         VDEV_LABEL_CREATE,      /* create/add a new device */
 153         VDEV_LABEL_REPLACE,     /* replace an existing device */
 154         VDEV_LABEL_SPARE,       /* add a new hot spare */
 155         VDEV_LABEL_REMOVE,      /* remove an existing device */
 156         VDEV_LABEL_L2CACHE,     /* add an L2ARC cache device */
 157         VDEV_LABEL_SPLIT        /* generating new label for split-off dev */
 158 } vdev_labeltype_t;

 160 extern int vdev_label_init(vdev_t *vd, uint64_t txg, vdev_labeltype_t reason);

 162 #ifdef  __cplusplus
 163 }
 164 #endif

 166 #endif  /* _SYS_VDEV_H */
```

```
**********************************************************
   15360 Fri Oct 31 10:14:52 2014
new/usr/src/uts/common/fs/zfs/sys/zil.h
5269 zfs: zpool import slow
While importing a pool all objsets are enumerated twice, once to check
the zil log chains and once to claim them. On pools with many datasets
this process might take a substantial amount of time.
Speed up the process by parallelizing it utilizing a taskq. The number
of parallel tasks is limited to 4 times the number of leaf vdevs.
**********************************************************
```

```
/*
 * 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.
 */

/* Portions Copyright 2010 Robert Milkowski */

#ifndef _SYS_ZIL_H
#define _SYS_ZIL_H

#include <sys/types.h>
#include <sys/spa.h>
#include <sys/zio.h>
#include <sys/dmu.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_dataset.h>
#endif /* ! codereview */

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Intent log format:
 *
 * Each objset has its own intent log.  The log header (zil_header_t)
 * for objset N's intent log is kept in the Nth object of the SPA's
 * intent_log objset.  The log header points to a chain of log blocks,
 * each of which contains log records (i.e., transactions) followed by
 * a log block trailer (zil_trailer_t).  The format of a log record
 * depends on the record (or transaction) type, but all records begin
 * with a common structure that defines the type, length, and txg.
 */

/*
 * Intent log header - this on disk structure holds fields to manage
```



```
 * the log.  All fields are 64 bit to easily handle cross architectures.
 */
typedef struct zil_header {
        uint64_t zh_claim_txg;  /* txg in which log blocks were claimed */
        uint64_t zh_replay_seq; /* highest replayed sequence number */
        blkptr_t zh_log;        /* log chain */
        uint64_t zh_claim_blk_seq; /* highest claimed block sequence number */
        uint64_t zh_flags;      /* header flags */
        uint64_t zh_claim_lr_seq; /* highest claimed lr sequence number */
        uint64_t zh_pad[3];
} zil_header_t;

/*
 * zh_flags bit settings
 */
#define ZIL_REPLAY_NEEDED       0x1     /* replay needed - internal only */
#define ZIL_CLAIM_LR_SEQ_VALID  0x2     /* zh_claim_lr_seq field is valid */

/*
 * Log block chaining.
 *
 * Log blocks are chained together. Originally they were chained at the
 * end of the block. For performance reasons the chain was moved to the
 * beginning of the block which allows writes for only the data being used.
 * The older position is supported for backwards compatability.
 *
 * The zio_eck_t contains a zec_cksum which for the intent log is
 * the sequence number of this log block. A seq of 0 is invalid.
 * The zec_cksum is checked by the SPA against the sequence
 * number passed in the blk_cksum field of the blkptr_t
 */
typedef struct zil_chain {
        uint64_t zc_pad;
        blkptr_t zc_next_blk;   /* next block in chain */
        uint64_t zc_nused;      /* bytes in log block used */
        zio_eck_t zc_eck;       /* block trailer */
} zil_chain_t;

#define ZIL_MIN_BLKSZ   4096ULL
#define ZIL_MAX_BLKSZ   SPA_MAXBLOCKSIZE

/*
 * The words of a log block checksum.
 */
#define ZIL_ZC_GUID_0   0
#define ZIL_ZC_GUID_1   1
#define ZIL_ZC_OBJSET   2
#define ZIL_ZC_SEQ      3

typedef enum zil_create {
        Z_FILE,
        Z_DIR,
        Z_XATTRDIR,
} zil_create_t;

/*
 * size of xvattr log section.
 * its composed of lr_attr_t + xvattr bitmap + 2 64 bit timestamps
 * for create time and a single 64 bit integer for all of the attributes,
 * and 4 64 bit integers (32 bytes) for the scanstamp.
 *
 */

#define ZIL_XVAT_SIZE(mapsize) \
        sizeof (lr_attr_t) + (sizeof (uint32_t) * (mapsize - 1)) + \
        (sizeof (uint64_t) * 7)
```

```
124 /*
125  * Size of ACL in log.  The ACE data is padded out to properly align
126  * on 8 byte boundary.
127  */

129 #define ZIL_ACE_LENGTH(x)       (roundup(x, sizeof (uint64_t)))

131 /*
132  * Intent log transaction types and record structures
133  */
134 #define TX_CREATE               1       /* Create file */
135 #define TX_MKDIR                2       /* Make directory */
136 #define TX_MKXATTR              3       /* Make XATTR directory */
137 #define TX_SYMLINK              4       /* Create symbolic link to a file */
138 #define TX_REMOVE               5       /* Remove file */
139 #define TX_RMDIR                6       /* Remove directory */
140 #define TX_LINK                 7       /* Create hard link to a file */
141 #define TX_RENAME               8       /* Rename a file */
142 #define TX_WRITE                9       /* File write */
143 #define TX_TRUNCATE             10      /* Truncate a file */
144 #define TX_SETATTR              11      /* Set file attributes */
145 #define TX_ACL_V0               12      /* Set old formatted ACL */
146 #define TX_ACL                  13      /* Set ACL */
147 #define TX_CREATE_ACL           14      /* create with ACL */
148 #define TX_CREATE_ATTR          15      /* create + attrs */
149 #define TX_CREATE_ACL_ATTR      16      /* create with ACL + attrs */
150 #define TX_MKDIR_ACL            17      /* mkdir with ACL */
151 #define TX_MKDIR_ATTR           18      /* mkdir with attr */
152 #define TX_MKDIR_ACL_ATTR       19      /* mkdir with ACL + attrs */
153 #define TX_WRITE2               20      /* dmu_sync EALREADY write */
154 #define TX_MAX_TYPE             21      /* Max transaction type */

156 /*
157  * The transactions for mkdir, symlink, remove, rmdir, link, and rename
158  * may have the following bit set, indicating the original request
159  * specified case-insensitive handling of names.
160  */
161 #define TX_CI   ((uint64_t)0x1 << 63) /* case-insensitive behavior requested */

163 /*
164  * Transactions for write, truncate, setattr, acl_v0, and acl can be logged
165  * out of order.  For convenience in the code, all such records must have
166  * lr_foid at the same offset.
167  */
168 #define TX_OOO(txtype)                  \
169         ((txtype) == TX_WRITE ||        \
170         (txtype) == TX_TRUNCATE ||      \
171         (txtype) == TX_SETATTR ||       \
172         (txtype) == TX_ACL_V0 ||        \
173         (txtype) == TX_ACL ||           \
174         (txtype) == TX_WRITE2)

176 /*
177  * Format of log records.
178  * The fields are carefully defined to allow them to be aligned
179  * and sized the same on sparc & intel architectures.
180  * Each log record has a common structure at the beginning.
181  *
182  * The log record on disk (lrc_seq) holds the sequence number of all log
183  * records which is used to ensure we don't replay the same record.
184  */
185 typedef struct {                        /* common log record header */
186         uint64_t        lrc_txtype;     /* intent log transaction type */
187         uint64_t        lrc_reclen;     /* transaction record length */
188         uint64_t        lrc_txg;        /* dmu transaction group number */
```



```
189         uint64_t        lrc_seq;        /* see comment above */
190 } lr_t;

192 /*
193  * Common start of all out-of-order record types (TX_OOO() above).
194  */
195 typedef struct {
196         lr_t            lr_common;      /* common portion of log record */
197         uint64_t        lr_foid;        /* object id */
198 } lr_ooo_t;

200 /*
201  * Handle option extended vattr attributes.
202  *
203  * Whenever new attributes are added the version number
204  * will need to be updated as will code in
205  * zfs_log.c and zfs_replay.c
206  */
207 typedef struct {
208         uint32_t        lr_attr_masksize; /* number of elements in array */
209         uint32_t        lr_attr_bitmap; /* First entry of array */
210         /* remainder of array and any additional fields */
211 } lr_attr_t;

213 /*
214  * log record for creates without optional ACL.
215  * This log record does support optional xvattr_t attributes.
216  */
217 typedef struct {
218         lr_t            lr_common;      /* common portion of log record */
219         uint64_t        lr_doid;        /* object id of directory */
220         uint64_t        lr_foid;        /* object id of created file object */
221         uint64_t        lr_mode;        /* mode of object */
222         uint64_t        lr_uid;         /* uid of object */
223         uint64_t        lr_gid;         /* gid of object */
224         uint64_t        lr_gen;         /* generation (txg of creation) */
225         uint64_t        lr_crtime[2];   /* creation time */
226         uint64_t        lr_rdev;        /* rdev of object to create */
227         /* name of object to create follows this */
228         /* for symlinks, link content follows name */
229         /* for creates with xvattr data, the name follows the xvattr info */
230 } lr_create_t;

232 /*
233  * FUID ACL record will be an array of ACEs from the original ACL.
234  * If this array includes ephemeral IDs, the record will also include
235  * an array of log-specific FUIDs to replace the ephemeral IDs.
236  * Only one copy of each unique domain will be present, so the log-specific
237  * FUIDs will use an index into a compressed domain table.  On replay this
238  * information will be used to construct real FUIDs (and bypass idmap,
239  * since it may not be available).
240  */

242 /*
243  * Log record for creates with optional ACL
244  * This log record is also used for recording any FUID
245  * information needed for replaying the create.  If the
246  * file doesn't have any actual ACEs then the lr_aclcnt
247  * would be zero.
248  *
249  * After lr_acl_flags, there are a lr_acl_bytes number of variable sized ace's.
250  * If create is also setting xvattr's, then acl data follows xvattr.
251  * If ACE FUIDs are needed then they will follow the xvattr_t.  Following
252  * the FUIDs will be the domain table information.  The FUIDs for the owner
253  * and group will be in lr_create.  Name follows ACL data.
254  */
```

```
255 typedef struct {
256         lr_create_t     lr_create;      /* common create portion */
257         uint64_t        lr_aclcnt;      /* number of ACEs in ACL */
258         uint64_t        lr_domcnt;      /* number of unique domains */
259         uint64_t        lr_fuidcnt;     /* number of real fuids */
260         uint64_t        lr_acl_bytes;   /* number of bytes in ACL */
261         uint64_t        lr_acl_flags;   /* ACL flags */
262 } lr_acl_create_t;

264 typedef struct {
265         lr_t            lr_common;      /* common portion of log record */
266         uint64_t        lr_doid;        /* obj id of directory */
267         /* name of object to remove follows this */
268 } lr_remove_t;

270 typedef struct {
271         lr_t            lr_common;      /* common portion of log record */
272         uint64_t        lr_doid;        /* obj id of directory */
273         uint64_t        lr_link_obj;    /* obj id of link */
274         /* name of object to link follows this */
275 } lr_link_t;

277 typedef struct {
278         lr_t            lr_common;      /* common portion of log record */
279         uint64_t        lr_sdoid;       /* obj id of source directory */
280         uint64_t        lr_tdoid;       /* obj id of target directory */
281         /* 2 strings: names of source and destination follow this */
282 } lr_rename_t;

284 typedef struct {
285         lr_t            lr_common;      /* common portion of log record */
286         uint64_t        lr_foid;        /* file object to write */
287         uint64_t        lr_offset;      /* offset to write to */
288         uint64_t        lr_length;      /* user data length to write */
289         uint64_t        lr_blkoff;      /* no longer used */
290         blkptr_t        lr_blkptr;      /* spa block pointer for replay */
291         /* write data will follow for small writes */
292 } lr_write_t;

294 typedef struct {
295         lr_t            lr_common;      /* common portion of log record */
296         uint64_t        lr_foid;        /* object id of file to truncate */
297         uint64_t        lr_offset;      /* offset to truncate from */
298         uint64_t        lr_length;      /* length to truncate */
299 } lr_truncate_t;

301 typedef struct {
302         lr_t            lr_common;      /* common portion of log record */
303         uint64_t        lr_foid;        /* file object to change attributes */
304         uint64_t        lr_mask;        /* mask of attributes to set */
305         uint64_t        lr_mode;        /* mode to set */
306         uint64_t        lr_uid;         /* uid to set */
307         uint64_t        lr_gid;         /* gid to set */
308         uint64_t        lr_size;        /* size to set */
309         uint64_t        lr_atime[2];    /* access time */
310         uint64_t        lr_mtime[2];    /* modification time */
311         /* optional attribute lr_attr_t may be here */
312 } lr_setattr_t;

314 typedef struct {
315         lr_t            lr_common;      /* common portion of log record */
316         uint64_t        lr_foid;        /* obj id of file */
317         uint64_t        lr_aclcnt;      /* number of acl entries */
318         /* lr_aclcnt number of ace_t entries follow this */
319 } lr_acl_v0_t;
```



```
321 typedef struct {
322         lr_t            lr_common;      /* common portion of log record */
323         uint64_t        lr_foid;        /* obj id of file */
324         uint64_t        lr_aclcnt;      /* number of ACEs in ACL */
325         uint64_t        lr_domcnt;      /* number of unique domains */
326         uint64_t        lr_fuidcnt;     /* number of real fuids */
327         uint64_t        lr_acl_bytes;   /* number of bytes in ACL */
328         uint64_t        lr_acl_flags;   /* ACL flags */
329         /* lr_acl_bytes number of variable sized ace's follows */
330 } lr_acl_t;

332 /*
333  * ZIL structure definitions, interface function prototype and globals.
334  */

336 /*
337  * Writes are handled in three different ways:
338  *
339  * WR_INDIRECT:
340  *    In this mode, if we need to commit the write later, then the block
341  *    is immediately written into the file system (using dmu_sync),
342  *    and a pointer to the block is put into the log record.
343  *    When the txg commits the block is linked in.
344  *    This saves additionally writing the data into the log record.
345  *    There are a few requirements for this to occur:
346  *      - write is greater than zfs/zvol_immediate_write_sz
347  *      - not using slogs (as slogs are assumed to always be faster
348  *        than writing into the main pool)
349  *      - the write occupies only one block
350  * WR_COPIED:
351  *    If we know we'll immediately be committing the
352  *    transaction (FSYNC or FDSYNC), the we allocate a larger
353  *    log record here for the data and copy the data in.
354  * WR_NEED_COPY:
355  *    Otherwise we don't allocate a buffer, and *if* we need to
356  *    flush the write later then a buffer is allocated and
357  *    we retrieve the data using the dmu.
358  */
359 typedef enum {
360         WR_INDIRECT,    /* indirect - a large write (dmu_sync() data */
361                         /* and put blkptr in log, rather than actual data) */
362         WR_COPIED,      /* immediate - data is copied into lr_write_t */
363         WR_NEED_COPY,   /* immediate - data needs to be copied if pushed */
364         WR_NUM_STATES   /* number of states */
365 } itx_wr_state_t;

367 typedef struct itx {
368         list_node_t     itx_node;       /* linkage on zl_itx_list */
369         void            *itx_private;   /* type-specific opaque data */
370         itx_wr_state_t  itx_wr_state;   /* write state */
371         uint8_t         itx_sync;       /* synchronous transaction */
372         uint64_t        itx_sod;        /* record size on disk */
373         uint64_t        itx_oid;        /* object id */
374         lr_t            itx_lr;         /* common part of log record */
375         /* followed by type-specific part of lr_xx_t and its immediate data */
376 } itx_t;

378 typedef int zil_parse_blk_func_t(zilog_t *zilog, blkptr_t *bp, void *arg,
379     uint64_t txg);
380 typedef int zil_parse_lr_func_t(zilog_t *zilog, lr_t *lr, void *arg,
381     uint64_t txg);
382 typedef int zil_replay_func_t();
383 typedef int zil_get_data_t(void *arg, lr_write_t *lr, char *dbuf, zio_t *zio);

385 extern int zil_parse(zilog_t *zilog, zil_parse_blk_func_t *parse_blk_func,
386     zil_parse_lr_func_t *parse_lr_func, void *arg, uint64_t txg);
```

```
388 extern void     zil_init(void);
389 extern void     zil_fini(void);

391 extern zilog_t  *zil_alloc(objset_t *os, zil_header_t *zh_phys);
392 extern void     zil_free(zilog_t *zilog);

394 extern zilog_t  *zil_open(objset_t *os, zil_get_data_t *get_data);
395 extern void     zil_close(zilog_t *zilog);

397 extern void     zil_replay(objset_t *os, void *arg,
398     zil_replay_func_t *replay_func[TX_MAX_TYPE]);
399 extern boolean_t zil_replaying(zilog_t *zilog, dmu_tx_t *tx);
400 extern void     zil_destroy(zilog_t *zilog, boolean_t keep_first);
401 extern void     zil_destroy_sync(zilog_t *zilog, dmu_tx_t *tx);
402 extern void     zil_rollback_destroy(zilog_t *zilog, dmu_tx_t *tx);

404 extern itx_t    *zil_itx_create(uint64_t txtype, size_t lrsize);
405 extern void     zil_itx_destroy(itx_t *itx);
406 extern void     zil_itx_assign(zilog_t *zilog, itx_t *itx, dmu_tx_t *tx);

408 extern void     zil_commit(zilog_t *zilog, uint64_t oid);

410 extern int      zil_vdev_offline(const char *osname, void *txarg);
411 extern int      zil_claim(dsl_pool_t *dp, dsl_dataset_t *ds, void *txarg);
412 extern int      zil_check_log_chain(dsl_pool_t *dp, dsl_dataset_t *ds,
413     void *tx);
 35 extern int      zil_claim(const char *osname, void *txarg);
 36 extern int      zil_check_log_chain(const char *osname, void *txarg);
414 extern void     zil_sync(zilog_t *zilog, dmu_tx_t *tx);
415 extern void     zil_clean(zilog_t *zilog, uint64_t synced_txg);

417 extern int      zil_suspend(const char *osname, void **cookiep);
418 extern void     zil_resume(void *cookie);

420 extern void     zil_add_block(zilog_t *zilog, const blkptr_t *bp);
421 extern int      zil_bp_tree_add(zilog_t *zilog, const blkptr_t *bp);

423 extern void     zil_set_sync(zilog_t *zilog, uint64_t syncval);

425 extern void     zil_set_logbias(zilog_t *zilog, uint64_t slogval);

427 extern int zil_replay_disable;

429 #ifdef  __cplusplus
430 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   89420 Fri Oct 31 10:14:52 2014
new/usr/src/uts/common/fs/zfs/vdev.c
5269 zfs: zpool import slow
While importing a pool all objsets are enumerated twice, once to check
the zil log chains and once to claim them. On pools with many datasets
this process might take a substantial amount of time.
Speed up the process by parallelizing it utilizing a taskq. The number
of parallel tasks is limited to 4 times the number of leaf vdevs.
**********************************************************
______unchanged_portion_omitted_
```

```
175 static int
176 vdev_count_leaves_impl(vdev_t *vd)
177 {
178         vdev_t *mvd;
179         int n = 0;
180
181         if (vd->vdev_children == 0)
182                 return (1);
183
184         for (int c = 0; c < vd->vdev_children; c++)
185                 n += vdev_count_leaves_impl(vd->vdev_child[c]);
186
187         return (n);
188 }
189
190 int
191 vdev_count_leaves(spa_t *spa)
192 {
193         return (vdev_count_leaves_impl(spa->spa_root_vdev));
194 }
195
196 #endif /* ! codereview */
197 void
198 vdev_add_child(vdev_t *pvd, vdev_t *cvd)
199 {
200         size_t oldsize, newsize;
201         uint64_t id = cvd->vdev_id;
202         vdev_t **newchild;
203
204         ASSERT(spa_config_held(cvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);
205         ASSERT(cvd->vdev_parent == NULL);
206
207         cvd->vdev_parent = pvd;
208
209         if (pvd == NULL)
210                 return;
211
212         ASSERT(id >= pvd->vdev_children || pvd->vdev_child[id] == NULL);
213
214         oldsize = pvd->vdev_children * sizeof (vdev_t *);
215         pvd->vdev_children = MAX(pvd->vdev_children, id + 1);
216         newsize = pvd->vdev_children * sizeof (vdev_t *);
217
218         newchild = kmem_zalloc(newsize, KM_SLEEP);
219         if (pvd->vdev_child != NULL) {
220                 bcopy(pvd->vdev_child, newchild, oldsize);
221                 kmem_free(pvd->vdev_child, oldsize);
222         }
223
224         pvd->vdev_child = newchild;
225         pvd->vdev_child[id] = cvd;
226
227         cvd->vdev_top = (pvd->vdev_top ? pvd->vdev_top: cvd);
228         ASSERT(cvd->vdev_top->vdev_parent->vdev_parent == NULL);
```



```
230         /*
231          * Walk up all ancestors to update guid sum.
232          */
233         for (; pvd != NULL; pvd = pvd->vdev_parent)
234                 pvd->vdev_guid_sum += cvd->vdev_guid_sum;
235 }
236
237 void
238 vdev_remove_child(vdev_t *pvd, vdev_t *cvd)
239 {
240         int c;
241         uint_t id = cvd->vdev_id;
242
243         ASSERT(cvd->vdev_parent == pvd);
244
245         if (pvd == NULL)
246                 return;
247
248         ASSERT(id < pvd->vdev_children);
249         ASSERT(pvd->vdev_child[id] == cvd);
250
251         pvd->vdev_child[id] = NULL;
252         cvd->vdev_parent = NULL;
253
254         for (c = 0; c < pvd->vdev_children; c++)
255                 if (pvd->vdev_child[c])
256                         break;
257
258         if (c == pvd->vdev_children) {
259                 kmem_free(pvd->vdev_child, c * sizeof (vdev_t *));
260                 pvd->vdev_child = NULL;
261                 pvd->vdev_children = 0;
262         }
263
264         /*
265          * Walk up all ancestors to update guid sum.
266          */
267         for (; pvd != NULL; pvd = pvd->vdev_parent)
268                 pvd->vdev_guid_sum -= cvd->vdev_guid_sum;
269 }
270
271 /*
272  * Remove any holes in the child array.
273  */
274 void
275 vdev_compact_children(vdev_t *pvd)
276 {
277         vdev_t **newchild, *cvd;
278         int oldc = pvd->vdev_children;
279         int newc;
280
281         ASSERT(spa_config_held(pvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);
282
283         for (int c = newc = 0; c < oldc; c++)
284                 if (pvd->vdev_child[c])
285                         newc++;
286
287         newchild = kmem_alloc(newc * sizeof (vdev_t *), KM_SLEEP);
288
289         for (int c = newc = 0; c < oldc; c++) {
290                 if ((cvd = pvd->vdev_child[c]) != NULL) {
291                         newchild[newc] = cvd;
292                         cvd->vdev_id = newc++;
293                 }
294         }
```

```
296         kmem_free(pvd->vdev_child, oldc * sizeof (vdev_t *));
297         pvd->vdev_child = newchild;
298         pvd->vdev_children = newc;
299 }

301 /*
302  * Allocate and minimally initialize a vdev_t.
303  */
304 vdev_t *
305 vdev_alloc_common(spa_t *spa, uint_t id, uint64_t guid, vdev_ops_t *ops)
306 {
307         vdev_t *vd;

309         vd = kmem_zalloc(sizeof (vdev_t), KM_SLEEP);

311         if (spa->spa_root_vdev == NULL) {
312                 ASSERT(ops == &vdev_root_ops);
313                 spa->spa_root_vdev = vd;
314                 spa->spa_load_guid = spa_generate_guid(NULL);
315         }

317         if (guid == 0 && ops != &vdev_hole_ops) {
318                 if (spa->spa_root_vdev == vd) {
319                         /*
320                          * The root vdev's guid will also be the pool guid,
321                          * which must be unique among all pools.
322                          */
323                         guid = spa_generate_guid(NULL);
324                 } else {
325                         /*
326                          * Any other vdev's guid must be unique within the pool.
327                          */
328                         guid = spa_generate_guid(spa);
329                 }
330                 ASSERT(!spa_guid_exists(spa_guid(spa), guid));
331         }

333         vd->vdev_spa = spa;
334         vd->vdev_id = id;
335         vd->vdev_guid = guid;
336         vd->vdev_guid_sum = guid;
337         vd->vdev_ops = ops;
338         vd->vdev_state = VDEV_STATE_CLOSED;
339         vd->vdev_ishole = (ops == &vdev_hole_ops);

341         mutex_init(&vd->vdev_dtl_lock, NULL, MUTEX_DEFAULT, NULL);
342         mutex_init(&vd->vdev_stat_lock, NULL, MUTEX_DEFAULT, NULL);
343         mutex_init(&vd->vdev_probe_lock, NULL, MUTEX_DEFAULT, NULL);
344         for (int t = 0; t < DTL_TYPES; t++) {
345                 vd->vdev_dtl[t] = range_tree_create(NULL, NULL,
346                     &vd->vdev_dtl_lock);
347         }
348         txg_list_create(&vd->vdev_ms_list,
349             offsetof(struct metaslab, ms_txg_node));
350         txg_list_create(&vd->vdev_dtl_list,
351             offsetof(struct vdev, vdev_dtl_node));
352         vd->vdev_stat.vs_timestamp = gethrtime();
353         vdev_queue_init(vd);
354         vdev_cache_init(vd);

356         return (vd);
357 }

359 /*
360  * Allocate a new vdev.  The 'alloctype' is used to control whether we are
```



```
361  * creating a new vdev or loading an existing one - the behavior is slightly
362  * different for each case.
363  */
364 int
365 vdev_alloc(spa_t *spa, vdev_t **vdp, nvlist_t *nv, vdev_t *parent, uint_t id,
366     int alloctype)
367 {
368         vdev_ops_t *ops;
369         char *type;
370         uint64_t guid = 0, islog, nparity;
371         vdev_t *vd;

373         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

375         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_TYPE, &type) != 0)
376                 return (SET_ERROR(EINVAL));

378         if ((ops = vdev_getops(type)) == NULL)
379                 return (SET_ERROR(EINVAL));

381         /*
382          * If this is a load, get the vdev guid from the nvlist.
383          * Otherwise, vdev_alloc_common() will generate one for us.
384          */
385         if (alloctype == VDEV_ALLOC_LOAD) {
386                 uint64_t label_id;

388                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ID, &label_id) ||
389                     label_id != id)
390                         return (SET_ERROR(EINVAL));

392                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
393                         return (SET_ERROR(EINVAL));
394         } else if (alloctype == VDEV_ALLOC_SPARE) {
395                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
396                         return (SET_ERROR(EINVAL));
397         } else if (alloctype == VDEV_ALLOC_L2CACHE) {
398                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
399                         return (SET_ERROR(EINVAL));
400         } else if (alloctype == VDEV_ALLOC_ROOTPOOL) {
401                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
402                         return (SET_ERROR(EINVAL));
403         }

405         /*
406          * The first allocated vdev must be of type 'root'.
407          */
408         if (ops != &vdev_root_ops && spa->spa_root_vdev == NULL)
409                 return (SET_ERROR(EINVAL));

411         /*
412          * Determine whether we're a log vdev.
413          */
414         islog = 0;
415         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_IS_LOG, &islog);
416         if (islog && spa_version(spa) < SPA_VERSION_SLOGS)
417                 return (SET_ERROR(ENOTSUP));

419         if (ops == &vdev_hole_ops && spa_version(spa) < SPA_VERSION_HOLES)
420                 return (SET_ERROR(ENOTSUP));

422         /*
423          * Set the nparity property for RAID-Z vdevs.
424          */
425         nparity = -1ULL;
426         if (ops == &vdev_raidz_ops) {
```

```
427                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_NPARITY,
428                     &nparity) == 0) {
429                         if (nparity == 0 || nparity > VDEV_RAIDZ_MAXPARITY)
430                                 return (SET_ERROR(EINVAL));
431                         /*
432                          * Previous versions could only support 1 or 2 parity
433                          * device.
434                          */
435                         if (nparity > 1 &&
436                             spa_version(spa) < SPA_VERSION_RAIDZ2)
437                                 return (SET_ERROR(ENOTSUP));
438                         if (nparity > 2 &&
439                             spa_version(spa) < SPA_VERSION_RAIDZ3)
440                                 return (SET_ERROR(ENOTSUP));
441                 } else {
442                         /*
443                          * We require the parity to be specified for SPAs that
444                          * support multiple parity levels.
445                          */
446                         if (spa_version(spa) >= SPA_VERSION_RAIDZ2)
447                                 return (SET_ERROR(EINVAL));
448                         /*
449                          * Otherwise, we default to 1 parity device for RAID-Z.
450                          */
451                         nparity = 1;
452                 }
453         } else {
454                 nparity = 0;
455         }
456         ASSERT(nparity != -1ULL);

458         vd = vdev_alloc_common(spa, id, guid, ops);

460         vd->vdev_islog = islog;
461         vd->vdev_nparity = nparity;

463         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PATH, &vd->vdev_path) == 0)
464                 vd->vdev_path = spa_strdup(vd->vdev_path);
465         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_DEVID, &vd->vdev_devid) == 0)
466                 vd->vdev_devid = spa_strdup(vd->vdev_devid);
467         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PHYS_PATH,
468             &vd->vdev_physpath) == 0)
469                 vd->vdev_physpath = spa_strdup(vd->vdev_physpath);
470         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_FRU, &vd->vdev_fru) == 0)
471                 vd->vdev_fru = spa_strdup(vd->vdev_fru);

473         /*
474          * Set the whole_disk property.  If it's not specified, leave the value
475          * as -1.
476          */
477         if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_WHOLE_DISK,
478             &vd->vdev_wholedisk) != 0)
479                 vd->vdev_wholedisk = -1ULL;

481         /*
482          * Look for the 'not present' flag.  This will only be set if the device
483          * was not present at the time of import.
484          */
485         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_NOT_PRESENT,
486             &vd->vdev_not_present);

488         /*
489          * Get the alignment requirement.
490          */
491         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ASHIFT, &vd->vdev_ashift);
```



```
493         /*
494          * Retrieve the vdev creation time.
495          */
496         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_CREATE_TXG,
497             &vd->vdev_crtxg);

499         /*
500          * If we're a top-level vdev, try to load the allocation parameters.
501          */
502         if (parent && !parent->vdev_parent &&
503             (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_SPLIT)) {
504                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_ARRAY,
505                     &vd->vdev_ms_array);
506                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_SHIFT,
507                     &vd->vdev_ms_shift);
508                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ASIZE,
509                     &vd->vdev_asize);
510                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_REMOVING,
511                     &vd->vdev_removing);
512         }

514         if (parent && !parent->vdev_parent && alloctype != VDEV_ALLOC_ATTACH) {
515                 ASSERT(alloctype == VDEV_ALLOC_LOAD ||
516                     alloctype == VDEV_ALLOC_ADD ||
517                     alloctype == VDEV_ALLOC_SPLIT ||
518                     alloctype == VDEV_ALLOC_ROOTPOOL);
519                 vd->vdev_mg = metaslab_group_create(islog ?
520                     spa_log_class(spa) : spa_normal_class(spa), vd);
521         }

523         /*
524          * If we're a leaf vdev, try to load the DTL object and other state.
525          */
526         if (vd->vdev_ops->vdev_op_leaf &&
527             (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_L2CACHE ||
528             alloctype == VDEV_ALLOC_ROOTPOOL)) {
529                 if (alloctype == VDEV_ALLOC_LOAD) {
530                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_DTL,
531                             &vd->vdev_dtl_object);
532                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_UNSPARE,
533                             &vd->vdev_unspare);
534                 }

536                 if (alloctype == VDEV_ALLOC_ROOTPOOL) {
537                         uint64_t spare = 0;

539                         if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_IS_SPARE,
540                             &spare) == 0 && spare)
541                                 spa_spare_add(vd);
542                 }

544                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_OFFLINE,
545                     &vd->vdev_offline);

547                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_RESILVER_TXG,
548                     &vd->vdev_resilver_txg);

550                 /*
551                  * When importing a pool, we want to ignore the persistent fault
552                  * state, as the diagnosis made on another system may not be
553                  * valid in the current context.  Local vdevs will
554                  * remain in the faulted state.
555                  */
556                 if (spa_load_state(spa) == SPA_LOAD_OPEN) {
557                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_FAULTED,
558                             &vd->vdev_faulted);
```

```
559                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_DEGRADED,
560                             &vd->vdev_degraded);
561                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_REMOVED,
562                             &vd->vdev_removed);
563
564                         if (vd->vdev_faulted || vd->vdev_degraded) {
565                                 char *aux;
566
567                                 vd->vdev_label_aux =
568                                     VDEV_AUX_ERR_EXCEEDED;
569                                 if (nvlist_lookup_string(nv,
570                                     ZPOOL_CONFIG_AUX_STATE, &aux) == 0 &&
571                                     strcmp(aux, "external") == 0)
572                                         vd->vdev_label_aux = VDEV_AUX_EXTERNAL;
573                         }
574                 }
575         }
576
577         /*
578          * Add ourselves to the parent's list of children.
579          */
580         vdev_add_child(parent, vd);
581
582         *vdp = vd;
583
584         return (0);
585 }
586
587 void
588 vdev_free(vdev_t *vd)
589 {
590         spa_t *spa = vd->vdev_spa;
591
592         /*
593          * vdev_free() implies closing the vdev first.  This is simpler than
594          * trying to ensure complicated semantics for all callers.
595          */
596         vdev_close(vd);
597
598         ASSERT(!list_link_active(&vd->vdev_config_dirty_node));
599         ASSERT(!list_link_active(&vd->vdev_state_dirty_node));
600
601         /*
602          * Free all children.
603          */
604         for (int c = 0; c < vd->vdev_children; c++)
605                 vdev_free(vd->vdev_child[c]);
606
607         ASSERT(vd->vdev_child == NULL);
608         ASSERT(vd->vdev_guid_sum == vd->vdev_guid);
609
610         /*
611          * Discard allocation state.
612          */
613         if (vd->vdev_mg != NULL) {
614                 vdev_metaslab_fini(vd);
615                 metaslab_group_destroy(vd->vdev_mg);
616         }
617
618         ASSERT0(vd->vdev_stat.vs_space);
619         ASSERT0(vd->vdev_stat.vs_dspace);
620         ASSERT0(vd->vdev_stat.vs_alloc);
621
622         /*
623          * Remove this vdev from its parent's child list.
624          */
```



```
625         vdev_remove_child(vd->vdev_parent, vd);
626
627         ASSERT(vd->vdev_parent == NULL);
628
629         /*
630          * Clean up vdev structure.
631          */
632         vdev_queue_fini(vd);
633         vdev_cache_fini(vd);
634
635         if (vd->vdev_path)
636                 spa_strfree(vd->vdev_path);
637         if (vd->vdev_devid)
638                 spa_strfree(vd->vdev_devid);
639         if (vd->vdev_physpath)
640                 spa_strfree(vd->vdev_physpath);
641         if (vd->vdev_fru)
642                 spa_strfree(vd->vdev_fru);
643
644         if (vd->vdev_isspare)
645                 spa_spare_remove(vd);
646         if (vd->vdev_isl2cache)
647                 spa_l2cache_remove(vd);
648
649         txg_list_destroy(&vd->vdev_ms_list);
650         txg_list_destroy(&vd->vdev_dtl_list);
651
652         mutex_enter(&vd->vdev_dtl_lock);
653         space_map_close(vd->vdev_dtl_sm);
654         for (int t = 0; t < DTL_TYPES; t++) {
655                 range_tree_vacate(vd->vdev_dtl[t], NULL, NULL);
656                 range_tree_destroy(vd->vdev_dtl[t]);
657         }
658         mutex_exit(&vd->vdev_dtl_lock);
659
660         mutex_destroy(&vd->vdev_dtl_lock);
661         mutex_destroy(&vd->vdev_stat_lock);
662         mutex_destroy(&vd->vdev_probe_lock);
663
664         if (vd == spa->spa_root_vdev)
665                 spa->spa_root_vdev = NULL;
666
667         kmem_free(vd, sizeof (vdev_t));
668 }
669
670 /*
671  * Transfer top-level vdev state from svd to tvd.
672  */
673 static void
674 vdev_top_transfer(vdev_t *svd, vdev_t *tvd)
675 {
676         spa_t *spa = svd->vdev_spa;
677         metaslab_t *msp;
678         vdev_t *vd;
679         int t;
680
681         ASSERT(tvd == tvd->vdev_top);
682
683         tvd->vdev_ms_array = svd->vdev_ms_array;
684         tvd->vdev_ms_shift = svd->vdev_ms_shift;
685         tvd->vdev_ms_count = svd->vdev_ms_count;
686
687         svd->vdev_ms_array = 0;
688         svd->vdev_ms_shift = 0;
689         svd->vdev_ms_count = 0;
```

```
691         if (tvd->vdev_mg)
692                 ASSERT3P(tvd->vdev_mg, ==, svd->vdev_mg);
693         tvd->vdev_mg = svd->vdev_mg;
694         tvd->vdev_ms = svd->vdev_ms;
695
696         svd->vdev_mg = NULL;
697         svd->vdev_ms = NULL;
698
699         if (tvd->vdev_mg != NULL)
700                 tvd->vdev_mg->mg_vd = tvd;
701
702         tvd->vdev_stat.vs_alloc = svd->vdev_stat.vs_alloc;
703         tvd->vdev_stat.vs_space = svd->vdev_stat.vs_space;
704         tvd->vdev_stat.vs_dspace = svd->vdev_stat.vs_dspace;
705
706         svd->vdev_stat.vs_alloc = 0;
707         svd->vdev_stat.vs_space = 0;
708         svd->vdev_stat.vs_dspace = 0;
709
710         for (t = 0; t < TXG_SIZE; t++) {
711                 while ((msp = txg_list_remove(&svd->vdev_ms_list, t)) != NULL)
712                         (void) txg_list_add(&tvd->vdev_ms_list, msp, t);
713                 while ((vd = txg_list_remove(&svd->vdev_dtl_list, t)) != NULL)
714                         (void) txg_list_add(&tvd->vdev_dtl_list, vd, t);
715                 if (txg_list_remove_this(&spa->spa_vdev_txg_list, svd, t))
716                         (void) txg_list_add(&spa->spa_vdev_txg_list, tvd, t);
717         }
718
719         if (list_link_active(&svd->vdev_config_dirty_node)) {
720                 vdev_config_clean(svd);
721                 vdev_config_dirty(tvd);
722         }
723
724         if (list_link_active(&svd->vdev_state_dirty_node)) {
725                 vdev_state_clean(svd);
726                 vdev_state_dirty(tvd);
727         }
728
729         tvd->vdev_deflate_ratio = svd->vdev_deflate_ratio;
730         svd->vdev_deflate_ratio = 0;
731
732         tvd->vdev_islog = svd->vdev_islog;
733         svd->vdev_islog = 0;
734 }
735
736 static void
737 vdev_top_update(vdev_t *tvd, vdev_t *vd)
738 {
739         if (vd == NULL)
740                 return;
741
742         vd->vdev_top = tvd;
743
744         for (int c = 0; c < vd->vdev_children; c++)
745                 vdev_top_update(tvd, vd->vdev_child[c]);
746 }
747
748 /*
749  * Add a mirror/replacing vdev above an existing vdev.
750  */
751 vdev_t *
752 vdev_add_parent(vdev_t *cvd, vdev_ops_t *ops)
753 {
754         spa_t *spa = cvd->vdev_spa;
755         vdev_t *pvd = cvd->vdev_parent;
756         vdev_t *mvd;
```



```
757
758         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
759
760         mvd = vdev_alloc_common(spa, cvd->vdev_id, 0, ops);
761
762         mvd->vdev_asize = cvd->vdev_asize;
763         mvd->vdev_min_asize = cvd->vdev_min_asize;
764         mvd->vdev_max_asize = cvd->vdev_max_asize;
765         mvd->vdev_ashift = cvd->vdev_ashift;
766         mvd->vdev_state = cvd->vdev_state;
767         mvd->vdev_crtxg = cvd->vdev_crtxg;
768
769         vdev_remove_child(pvd, cvd);
770         vdev_add_child(pvd, mvd);
771         cvd->vdev_id = mvd->vdev_children;
772         vdev_add_child(mvd, cvd);
773         vdev_top_update(cvd->vdev_top, cvd->vdev_top);
774
775         if (mvd == mvd->vdev_top)
776                 vdev_top_transfer(cvd, mvd);
777
778         return (mvd);
779 }
780
781 /*
782  * Remove a 1-way mirror/replacing vdev from the tree.
783  */
784 void
785 vdev_remove_parent(vdev_t *cvd)
786 {
787         vdev_t *mvd = cvd->vdev_parent;
788         vdev_t *pvd = mvd->vdev_parent;
789
790         ASSERT(spa_config_held(cvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);
791
792         ASSERT(mvd->vdev_children == 1);
793         ASSERT(mvd->vdev_ops == &vdev_mirror_ops ||
794             mvd->vdev_ops == &vdev_replacing_ops ||
795             mvd->vdev_ops == &vdev_spare_ops);
796         cvd->vdev_ashift = mvd->vdev_ashift;
797
798         vdev_remove_child(mvd, cvd);
799         vdev_remove_child(pvd, mvd);
800
801         /*
802          * If cvd will replace mvd as a top-level vdev, preserve mvd's guid.
803          * Otherwise, we could have detached an offline device, and when we
804          * go to import the pool we'll think we have two top-level vdevs,
805          * instead of a different version of the same top-level vdev.
806          */
807         if (mvd->vdev_top == mvd) {
808                 uint64_t guid_delta = mvd->vdev_guid - cvd->vdev_guid;
809                 cvd->vdev_orig_guid = cvd->vdev_guid;
810                 cvd->vdev_guid += guid_delta;
811                 cvd->vdev_guid_sum += guid_delta;
812         }
813         cvd->vdev_id = mvd->vdev_id;
814         vdev_add_child(pvd, cvd);
815         vdev_top_update(cvd->vdev_top, cvd->vdev_top);
816
817         if (cvd == cvd->vdev_top)
818                 vdev_top_transfer(mvd, cvd);
819
820         ASSERT(mvd->vdev_children == 0);
821         vdev_free(mvd);
822 }
```

```
824 int
825 vdev_metaslab_init(vdev_t *vd, uint64_t txg)
826 {
827         spa_t *spa = vd->vdev_spa;
828         objset_t *mos = spa->spa_meta_objset;
829         uint64_t m;
830         uint64_t oldc = vd->vdev_ms_count;
831         uint64_t newc = vd->vdev_asize >> vd->vdev_ms_shift;
832         metaslab_t **mspp;
833         int error;
834
835         ASSERT(txg == 0 || spa_config_held(spa, SCL_ALLOC, RW_WRITER));
836
837         /*
838          * This vdev is not being allocated from yet or is a hole.
839          */
840         if (vd->vdev_ms_shift == 0)
841                 return (0);
842
843         ASSERT(!vd->vdev_ishole);
844
845         /*
846          * Compute the raidz-deflation ratio.  Note, we hard-code
847          * in 128k (1 << 17) because it is the current "typical" blocksize.
848          * Even if SPA_MAXBLOCKSIZE changes, this algorithm must never change,
849          * or we will inconsistently account for existing bp's.
850          */
851         vd->vdev_deflate_ratio = (1 << 17) /
852             (vdev_psize_to_asize(vd, 1 << 17) >> SPA_MINBLOCKSHIFT);
853
854         ASSERT(oldc <= newc);
855
856         mspp = kmem_zalloc(newc * sizeof (*mspp), KM_SLEEP);
857
858         if (oldc != 0) {
859                 bcopy(vd->vdev_ms, mspp, oldc * sizeof (*mspp));
860                 kmem_free(vd->vdev_ms, oldc * sizeof (*mspp));
861         }
862
863         vd->vdev_ms = mspp;
864         vd->vdev_ms_count = newc;
865
866         for (m = oldc; m < newc; m++) {
867                 uint64_t object = 0;
868
869                 if (txg == 0) {
870                         error = dmu_read(mos, vd->vdev_ms_array,
871                             m * sizeof (uint64_t), sizeof (uint64_t), &object,
872                             DMU_READ_PREFETCH);
873                         if (error)
874                                 return (error);
875                 }
876                 vd->vdev_ms[m] = metaslab_init(vd->vdev_mg, m, object, txg);
877         }
878
879         if (txg == 0)
880                 spa_config_enter(spa, SCL_ALLOC, FTAG, RW_WRITER);
881
882         /*
883          * If the vdev is being removed we don't activate
884          * the metaslabs since we want to ensure that no new
885          * allocations are performed on this device.
886          */
887         if (oldc == 0 && !vd->vdev_removing)
888                 metaslab_group_activate(vd->vdev_mg);
```



```
890         if (txg == 0)
891                 spa_config_exit(spa, SCL_ALLOC, FTAG);
892
893         return (0);
894 }
895
896 void
897 vdev_metaslab_fini(vdev_t *vd)
898 {
899         uint64_t m;
900         uint64_t count = vd->vdev_ms_count;
901
902         if (vd->vdev_ms != NULL) {
903                 metaslab_group_passivate(vd->vdev_mg);
904                 for (m = 0; m < count; m++) {
905                         metaslab_t *msp = vd->vdev_ms[m];
906
907                         if (msp != NULL)
908                                 metaslab_fini(msp);
909                 }
910                 kmem_free(vd->vdev_ms, count * sizeof (metaslab_t *));
911                 vd->vdev_ms = NULL;
912         }
913 }
914
915 typedef struct vdev_probe_stats {
916         boolean_t       vps_readable;
917         boolean_t       vps_writeable;
918         int             vps_flags;
919 } vdev_probe_stats_t;
920
921 static void
922 vdev_probe_done(zio_t *zio)
923 {
924         spa_t *spa = zio->io_spa;
925         vdev_t *vd = zio->io_vd;
926         vdev_probe_stats_t *vps = zio->io_private;
927
928         ASSERT(vd->vdev_probe_zio != NULL);
929
930         if (zio->io_type == ZIO_TYPE_READ) {
931                 if (zio->io_error == 0)
932                         vps->vps_readable = 1;
933                 if (zio->io_error == 0 && spa_writeable(spa)) {
934                         zio_nowait(zio_write_phys(vd->vdev_probe_zio, vd,
935                             zio->io_offset, zio->io_size, zio->io_data,
936                             ZIO_CHECKSUM_OFF, vdev_probe_done, vps,
937                             ZIO_PRIORITY_SYNC_WRITE, vps->vps_flags, B_TRUE));
938                 } else {
939                         zio_buf_free(zio->io_data, zio->io_size);
940                 }
941         } else if (zio->io_type == ZIO_TYPE_WRITE) {
942                 if (zio->io_error == 0)
943                         vps->vps_writeable = 1;
944                 zio_buf_free(zio->io_data, zio->io_size);
945         } else if (zio->io_type == ZIO_TYPE_NULL) {
946                 zio_t *pio;
947
948                 vd->vdev_cant_read |= !vps->vps_readable;
949                 vd->vdev_cant_write |= !vps->vps_writeable;
950
951                 if (vdev_readable(vd) &&
952                     (vdev_writeable(vd) || !spa_writeable(spa))) {
953                         zio->io_error = 0;
954                 } else {
```

```
955                         ASSERT(zio->io_error != 0);
956                         zfs_ereport_post(FM_EREPORT_ZFS_PROBE_FAILURE,
957                             spa, vd, NULL, 0, 0);
958                         zio->io_error = SET_ERROR(ENXIO);
959                 }
960
961                 mutex_enter(&vd->vdev_probe_lock);
962                 ASSERT(vd->vdev_probe_zio == zio);
963                 vd->vdev_probe_zio = NULL;
964                 mutex_exit(&vd->vdev_probe_lock);
965
966                 while ((pio = zio_walk_parents(zio)) != NULL)
967                         if (!vdev_accessible(vd, pio))
968                                 pio->io_error = SET_ERROR(ENXIO);
969
970                 kmem_free(vps, sizeof (*vps));
971         }
972 }
973
974 /*
975  * Determine whether this device is accessible.
976  *
977  * Read and write to several known locations: the pad regions of each
978  * vdev label but the first, which we leave alone in case it contains
979  * a VTOC.
980  */
981 zio_t *
982 vdev_probe(vdev_t *vd, zio_t *zio)
983 {
984         spa_t *spa = vd->vdev_spa;
985         vdev_probe_stats_t *vps = NULL;
986         zio_t *pio;
987
988         ASSERT(vd->vdev_ops->vdev_op_leaf);
989
990         /*
991          * Don't probe the probe.
992          */
993         if (zio && (zio->io_flags & ZIO_FLAG_PROBE))
994                 return (NULL);
995
996         /*
997          * To prevent 'probe storms' when a device fails, we create
998          * just one probe i/o at a time.  All zios that want to probe
999          * this vdev will become parents of the probe io.
1000         */
1001        mutex_enter(&vd->vdev_probe_lock);
1002
1003        if ((pio = vd->vdev_probe_zio) == NULL) {
1004                vps = kmem_zalloc(sizeof (*vps), KM_SLEEP);
1005
1006                vps->vps_flags = ZIO_FLAG_CANFAIL | ZIO_FLAG_PROBE |
1007                    ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_AGGREGATE |
1008                    ZIO_FLAG_TRYHARD;
1009
1010                if (spa_config_held(spa, SCL_ZIO, RW_WRITER)) {
1011                        /*
1012                         * vdev_cant_read and vdev_cant_write can only
1013                         * transition from TRUE to FALSE when we have the
1014                         * SCL_ZIO lock as writer; otherwise they can only
1015                         * transition from FALSE to TRUE.  This ensures that
1016                         * any zio looking at these values can assume that
1017                         * failures persist for the life of the I/O.  That's
1018                         * important because when a device has intermittent
1019                         * connectivity problems, we want to ensure that
1020                         * they're ascribed to the device (ENXIO) and not
```



```
1021                         * the zio (EIO).
1022                         *
1023                         * Since we hold SCL_ZIO as writer here, clear both
1024                         * values so the probe can reevaluate from first
1025                         * principles.
1026                         */
1027                        vps->vps_flags |= ZIO_FLAG_CONFIG_WRITER;
1028                        vd->vdev_cant_read = B_FALSE;
1029                        vd->vdev_cant_write = B_FALSE;
1030                }
1031
1032                vd->vdev_probe_zio = pio = zio_null(NULL, spa, vd,
1033                    vdev_probe_done, vps,
1034                    vps->vps_flags | ZIO_FLAG_DONT_PROPAGATE);
1035
1036                /*
1037                 * We can't change the vdev state in this context, so we
1038                 * kick off an async task to do it on our behalf.
1039                 */
1040                if (zio != NULL) {
1041                        vd->vdev_probe_wanted = B_TRUE;
1042                        spa_async_request(spa, SPA_ASYNC_PROBE);
1043                }
1044        }
1045
1046        if (zio != NULL)
1047                zio_add_child(zio, pio);
1048
1049        mutex_exit(&vd->vdev_probe_lock);
1050
1051        if (vps == NULL) {
1052                ASSERT(zio != NULL);
1053                return (NULL);
1054        }
1055
1056        for (int l = 1; l < VDEV_LABELS; l++) {
1057                zio_nowait(zio_read_phys(pio, vd,
1058                    vdev_label_offset(vd->vdev_psize, l,
1059                    offsetof(vdev_label_t, vl_pad2)),
1060                    VDEV_PAD_SIZE, zio_buf_alloc(VDEV_PAD_SIZE),
1061                    ZIO_CHECKSUM_OFF, vdev_probe_done, vps,
1062                    ZIO_PRIORITY_SYNC_READ, vps->vps_flags, B_TRUE));
1063        }
1064
1065        if (zio == NULL)
1066                return (pio);
1067
1068        zio_nowait(pio);
1069        return (NULL);
1070 }
1071
1072 static void
1073 vdev_open_child(void *arg)
1074 {
1075         vdev_t *vd = arg;
1076
1077         vd->vdev_open_thread = curthread;
1078         vd->vdev_open_error = vdev_open(vd);
1079         vd->vdev_open_thread = NULL;
1080 }
1081
1082 boolean_t
1083 vdev_uses_zvols(vdev_t *vd)
1084 {
1085         if (vd->vdev_path && strncmp(vd->vdev_path, ZVOL_DIR,
1086             strlen(ZVOL_DIR)) == 0)
```

```
1087                return (B_TRUE);
1088        for (int c = 0; c < vd->vdev_children; c++)
1089                if (vdev_uses_zvols(vd->vdev_child[c]))
1090                        return (B_TRUE);
1091        return (B_FALSE);
1092 }

1094 void
1095 vdev_open_children(vdev_t *vd)
1096 {
1097        taskq_t *tq;
1098        int children = vd->vdev_children;

1100        /*
1101         * in order to handle pools on top of zvols, do the opens
1102         * in a single thread so that the same thread holds the
1103         * spa_namespace_lock
1104         */
1105        if (vdev_uses_zvols(vd)) {
1106                for (int c = 0; c < children; c++)
1107                        vd->vdev_child[c]->vdev_open_error =
1108                            vdev_open(vd->vdev_child[c]);
1109                return;
1110        }
1111        tq = taskq_create("vdev_open", children, minclsyspri,
1112            children, children, TASKQ_PREPOPULATE);

1114        for (int c = 0; c < children; c++)
1115                VERIFY(taskq_dispatch(tq, vdev_open_child, vd->vdev_child[c],
1116                    TQ_SLEEP) != NULL);

1118        taskq_destroy(tq);
1119 }

1121 /*
1122  * Prepare a virtual device for access.
1123  */
1124 int
1125 vdev_open(vdev_t *vd)
1126 {
1127        spa_t *spa = vd->vdev_spa;
1128        int error;
1129        uint64_t osize = 0;
1130        uint64_t max_osize = 0;
1131        uint64_t asize, max_asize, psize;
1132        uint64_t ashift = 0;

1134        ASSERT(vd->vdev_open_thread == curthread ||
1135            spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
1136        ASSERT(vd->vdev_state == VDEV_STATE_CLOSED ||
1137            vd->vdev_state == VDEV_STATE_CANT_OPEN ||
1138            vd->vdev_state == VDEV_STATE_OFFLINE);

1140        vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
1141        vd->vdev_cant_read = B_FALSE;
1142        vd->vdev_cant_write = B_FALSE;
1143        vd->vdev_min_asize = vdev_get_min_asize(vd);

1145        /*
1146         * If this vdev is not removed, check its fault status.  If it's
1147         * faulted, bail out of the open.
1148         */
1149        if (!vd->vdev_removed && vd->vdev_faulted) {
1150                ASSERT(vd->vdev_children == 0);
1151                ASSERT(vd->vdev_label_aux == VDEV_AUX_ERR_EXCEEDED ||
1152                    vd->vdev_label_aux == VDEV_AUX_EXTERNAL);
```



```
1153                vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
1154                    vd->vdev_label_aux);
1155                return (SET_ERROR(ENXIO));
1156        } else if (vd->vdev_offline) {
1157                ASSERT(vd->vdev_children == 0);
1158                vdev_set_state(vd, B_TRUE, VDEV_STATE_OFFLINE, VDEV_AUX_NONE);
1159                return (SET_ERROR(ENXIO));
1160        }

1162        error = vd->vdev_ops->vdev_op_open(vd, &osize, &max_osize, &ashift);

1164        /*
1165         * Reset the vdev_reopening flag so that we actually close
1166         * the vdev on error.
1167         */
1168        vd->vdev_reopening = B_FALSE;
1169        if (zio_injection_enabled && error == 0)
1170                error = zio_handle_device_injection(vd, NULL, ENXIO);

1172        if (error) {
1173                if (vd->vdev_removed &&
1174                    vd->vdev_stat.vs_aux != VDEV_AUX_OPEN_FAILED)
1175                        vd->vdev_removed = B_FALSE;

1177                vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1178                    vd->vdev_stat.vs_aux);
1179                return (error);
1180        }

1182        vd->vdev_removed = B_FALSE;

1184        /*
1185         * Recheck the faulted flag now that we have confirmed that
1186         * the vdev is accessible.  If we're faulted, bail.
1187         */
1188        if (vd->vdev_faulted) {
1189                ASSERT(vd->vdev_children == 0);
1190                ASSERT(vd->vdev_label_aux == VDEV_AUX_ERR_EXCEEDED ||
1191                    vd->vdev_label_aux == VDEV_AUX_EXTERNAL);
1192                vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
1193                    vd->vdev_label_aux);
1194                return (SET_ERROR(ENXIO));
1195        }

1197        if (vd->vdev_degraded) {
1198                ASSERT(vd->vdev_children == 0);
1199                vdev_set_state(vd, B_TRUE, VDEV_STATE_DEGRADED,
1200                    VDEV_AUX_ERR_EXCEEDED);
1201        } else {
1202                vdev_set_state(vd, B_TRUE, VDEV_STATE_HEALTHY, 0);
1203        }

1205        /*
1206         * For hole or missing vdevs we just return success.
1207         */
1208        if (vd->vdev_ishole || vd->vdev_ops == &vdev_missing_ops)
1209                return (0);

1211        for (int c = 0; c < vd->vdev_children; c++) {
1212                if (vd->vdev_child[c]->vdev_state != VDEV_STATE_HEALTHY) {
1213                        vdev_set_state(vd, B_TRUE, VDEV_STATE_DEGRADED,
1214                            VDEV_AUX_NONE);
1215                        break;
1216                }
1217        }
```

```
1219         osize = P2ALIGN(osize, (uint64_t)sizeof (vdev_label_t));
1220         max_osize = P2ALIGN(max_osize, (uint64_t)sizeof (vdev_label_t));

1222         if (vd->vdev_children == 0) {
1223                 if (osize < SPA_MINDEVSIZE) {
1224                         vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1225                             VDEV_AUX_TOO_SMALL);
1226                         return (SET_ERROR(EOVERFLOW));
1227                 }
1228                 psize = osize;
1229                 asize = osize - (VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE);
1230                 max_asize = max_osize - (VDEV_LABEL_START_SIZE +
1231                     VDEV_LABEL_END_SIZE);
1232         } else {
1233                 if (vd->vdev_parent != NULL && osize < SPA_MINDEVSIZE -
1234                     (VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE)) {
1235                         vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1236                             VDEV_AUX_TOO_SMALL);
1237                         return (SET_ERROR(EOVERFLOW));
1238                 }
1239                 psize = 0;
1240                 asize = osize;
1241                 max_asize = max_osize;
1242         }

1244         vd->vdev_psize = psize;

1246         /*
1247          * Make sure the allocatable size hasn't shrunk.
1248          */
1249         if (asize < vd->vdev_min_asize) {
1250                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1251                     VDEV_AUX_BAD_LABEL);
1252                 return (SET_ERROR(EINVAL));
1253         }

1255         if (vd->vdev_asize == 0) {
1256                 /*
1257                  * This is the first-ever open, so use the computed values.
1258                  * For testing purposes, a higher ashift can be requested.
1259                  */
1260                 vd->vdev_asize = asize;
1261                 vd->vdev_max_asize = max_asize;
1262                 vd->vdev_ashift = MAX(ashift, vd->vdev_ashift);
1263         } else {
1264                 /*
1265                  * Detect if the alignment requirement has increased.
1266                  * We don't want to make the pool unavailable, just
1267                  * issue a warning instead.
1268                  */
1269                 if (ashift > vd->vdev_top->vdev_ashift &&
1270                     vd->vdev_ops->vdev_op_leaf) {
1271                         cmn_err(CE_WARN,
1272                             "Disk, '%s', has a block alignment that is "
1273                             "larger than the pool's alignment\n",
1274                             vd->vdev_path);
1275                 }
1276                 vd->vdev_max_asize = max_asize;
1277         }

1279         /*
1280          * If all children are healthy and the asize has increased,
1281          * then we've experienced dynamic LUN growth.  If automatic
1282          * expansion is enabled then use the additional space.
1283          */
1284         if (vd->vdev_state == VDEV_STATE_HEALTHY && asize > vd->vdev_asize &&
```



```
1285             (vd->vdev_expanding || spa->spa_autoexpand))
1286                 vd->vdev_asize = asize;

1288         vdev_set_min_asize(vd);

1290         /*
1291          * Ensure we can issue some IO before declaring the
1292          * vdev open for business.
1293          */
1294         if (vd->vdev_ops->vdev_op_leaf &&
1295             (error = zio_wait(vdev_probe(vd, NULL))) != 0) {
1296                 vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
1297                     VDEV_AUX_ERR_EXCEEDED);
1298                 return (error);
1299         }

1301         /*
1302          * If a leaf vdev has a DTL, and seems healthy, then kick off a
1303          * resilver.  But don't do this if we are doing a reopen for a scrub,
1304          * since this would just restart the scrub we are already doing.
1305          */
1306         if (vd->vdev_ops->vdev_op_leaf && !spa->spa_scrub_reopen &&
1307             vdev_resilver_needed(vd, NULL, NULL))
1308                 spa_async_request(spa, SPA_ASYNC_RESILVER);

1310         return (0);
1311 }

1313 /*
1314  * Called once the vdevs are all opened, this routine validates the label
1315  * contents.  This needs to be done before vdev_load() so that we don't
1316  * inadvertently do repair I/Os to the wrong device.
1317  *
1318  * If 'strict' is false ignore the spa guid check. This is necessary because
1319  * if the machine crashed during a re-guid the new guid might have been written
1320  * to all of the vdev labels, but not the cached config. The strict check
1321  * will be performed when the pool is opened again using the mos config.
1322  *
1323  * This function will only return failure if one of the vdevs indicates that it
1324  * has since been destroyed or exported.  This is only possible if
1325  * /etc/zfs/zpool.cache was readonly at the time.  Otherwise, the vdev state
1326  * will be updated but the function will return 0.
1327  */
1328 int
1329 vdev_validate(vdev_t *vd, boolean_t strict)
1330 {
1331         spa_t *spa = vd->vdev_spa;
1332         nvlist_t *label;
1333         uint64_t guid = 0, top_guid;
1334         uint64_t state;

1336         for (int c = 0; c < vd->vdev_children; c++)
1337                 if (vdev_validate(vd->vdev_child[c], strict) != 0)
1338                         return (SET_ERROR(EBADF));

1340         /*
1341          * If the device has already failed, or was marked offline, don't do
1342          * any further validation.  Otherwise, label I/O will fail and we will
1343          * overwrite the previous state.
1344          */
1345         if (vd->vdev_ops->vdev_op_leaf && vdev_readable(vd)) {
1346                 uint64_t aux_guid = 0;
1347                 nvlist_t *nvl;
1348                 uint64_t txg = spa_last_synced_txg(spa) != 0 ?
1349                     spa_last_synced_txg(spa) : -1ULL;
```

```
1351                if ((label = vdev_label_read_config(vd, txg)) == NULL) {
1352                        vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1353                            VDEV_AUX_BAD_LABEL);
1354                        return (0);
1355                }

1357                /*
1358                 * Determine if this vdev has been split off into another
1359                 * pool.  If so, then refuse to open it.
1360                 */
1361                if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_SPLIT_GUID,
1362                    &aux_guid) == 0 && aux_guid == spa_guid(spa)) {
1363                        vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1364                            VDEV_AUX_SPLIT_POOL);
1365                        nvlist_free(label);
1366                        return (0);
1367                }

1369                if (strict && (nvlist_lookup_uint64(label,
1370                    ZPOOL_CONFIG_POOL_GUID, &guid) != 0 ||
1371                    guid != spa_guid(spa))) {
1372                        vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1373                            VDEV_AUX_CORRUPT_DATA);
1374                        nvlist_free(label);
1375                        return (0);
1376                }

1378                if (nvlist_lookup_nvlist(label, ZPOOL_CONFIG_VDEV_TREE, &nvl)
1379                    != 0 || nvlist_lookup_uint64(nvl, ZPOOL_CONFIG_ORIG_GUID,
1380                    &aux_guid) != 0)
1381                        aux_guid = 0;

1383                /*
1384                 * If this vdev just became a top-level vdev because its
1385                 * sibling was detached, it will have adopted the parent's
1386                 * vdev guid -- but the label may or may not be on disk yet.
1387                 * Fortunately, either version of the label will have the
1388                 * same top guid, so if we're a top-level vdev, we can
1389                 * safely compare to that instead.
1390                 *
1391                 * If we split this vdev off instead, then we also check the
1392                 * original pool's guid.  We don't want to consider the vdev
1393                 * corrupt if it is partway through a split operation.
1394                 */
1395                if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_GUID,
1396                    &guid) != 0 ||
1397                    nvlist_lookup_uint64(label, ZPOOL_CONFIG_TOP_GUID,
1398                    &top_guid) != 0 ||
1399                    ((vd->vdev_guid != guid && vd->vdev_guid != aux_guid) &&
1400                    (vd->vdev_guid != top_guid || vd != vd->vdev_top))) {
1401                        vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1402                            VDEV_AUX_CORRUPT_DATA);
1403                        nvlist_free(label);
1404                        return (0);
1405                }

1407                if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_STATE,
1408                    &state) != 0) {
1409                        vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1410                            VDEV_AUX_CORRUPT_DATA);
1411                        nvlist_free(label);
1412                        return (0);
1413                }

1415                nvlist_free(label);
```



```
1417                /*
1418                 * If this is a verbatim import, no need to check the
1419                 * state of the pool.
1420                 */
1421                if (!(spa->spa_import_flags & ZFS_IMPORT_VERBATIM) &&
1422                    spa_load_state(spa) == SPA_LOAD_OPEN &&
1423                    state != POOL_STATE_ACTIVE)
1424                        return (SET_ERROR(EBADF));

1426                /*
1427                 * If we were able to open and validate a vdev that was
1428                 * previously marked permanently unavailable, clear that state
1429                 * now.
1430                 */
1431                if (vd->vdev_not_present)
1432                        vd->vdev_not_present = 0;
1433        }

1435        return (0);
1436 }

1438 /*
1439  * Close a virtual device.
1440  */
1441 void
1442 vdev_close(vdev_t *vd)
1443 {
1444        spa_t *spa = vd->vdev_spa;
1445        vdev_t *pvd = vd->vdev_parent;

1447        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

1449        /*
1450         * If our parent is reopening, then we are as well, unless we are
1451         * going offline.
1452         */
1453        if (pvd != NULL && pvd->vdev_reopening)
1454                vd->vdev_reopening = (pvd->vdev_reopening && !vd->vdev_offline);

1456        vd->vdev_ops->vdev_op_close(vd);

1458        vdev_cache_purge(vd);

1460        /*
1461         * We record the previous state before we close it, so that if we are
1462         * doing a reopen(), we don't generate FMA ereports if we notice that
1463         * it's still faulted.
1464         */
1465        vd->vdev_prevstate = vd->vdev_state;

1467        if (vd->vdev_offline)
1468                vd->vdev_state = VDEV_STATE_OFFLINE;
1469        else
1470                vd->vdev_state = VDEV_STATE_CLOSED;
1471        vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
1472 }

1474 void
1475 vdev_hold(vdev_t *vd)
1476 {
1477        spa_t *spa = vd->vdev_spa;

1479        ASSERT(spa_is_root(spa));
1480        if (spa->spa_state == POOL_STATE_UNINITIALIZED)
1481                return;
```

```
1483         for (int c = 0; c < vd->vdev_children; c++)
1484                 vdev_hold(vd->vdev_child[c]);
1485
1486         if (vd->vdev_ops->vdev_op_leaf)
1487                 vd->vdev_ops->vdev_op_hold(vd);
1488 }
1489
1490 void
1491 vdev_rele(vdev_t *vd)
1492 {
1493         spa_t *spa = vd->vdev_spa;
1494
1495         ASSERT(spa_is_root(spa));
1496         for (int c = 0; c < vd->vdev_children; c++)
1497                 vdev_rele(vd->vdev_child[c]);
1498
1499         if (vd->vdev_ops->vdev_op_leaf)
1500                 vd->vdev_ops->vdev_op_rele(vd);
1501 }
1502
1503 /*
1504  * Reopen all interior vdevs and any unopened leaves.  We don't actually
1505  * reopen leaf vdevs which had previously been opened as they might deadlock
1506  * on the spa_config_lock.  Instead we only obtain the leaf's physical size.
1507  * If the leaf has never been opened then open it, as usual.
1508  */
1509 void
1510 vdev_reopen(vdev_t *vd)
1511 {
1512         spa_t *spa = vd->vdev_spa;
1513
1514         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
1515
1516         /* set the reopening flag unless we're taking the vdev offline */
1517         vd->vdev_reopening = !vd->vdev_offline;
1518         vdev_close(vd);
1519         (void) vdev_open(vd);
1520
1521         /*
1522          * Call vdev_validate() here to make sure we have the same device.
1523          * Otherwise, a device with an invalid label could be successfully
1524          * opened in response to vdev_reopen().
1525          */
1526         if (vd->vdev_aux) {
1527                 (void) vdev_validate_aux(vd);
1528                 if (vdev_readable(vd) && vdev_writeable(vd) &&
1529                     vd->vdev_aux == &spa->spa_l2cache &&
1530                     !l2arc_vdev_present(vd))
1531                         l2arc_add_vdev(spa, vd);
1532         } else {
1533                 (void) vdev_validate(vd, B_TRUE);
1534         }
1535
1536         /*
1537          * Reassess parent vdev's health.
1538          */
1539         vdev_propagate_state(vd);
1540 }
1541
1542 int
1543 vdev_create(vdev_t *vd, uint64_t txg, boolean_t isreplacing)
1544 {
1545         int error;
1546
1547         /*
1548          * Normally, partial opens (e.g. of a mirror) are allowed.
```



```
1549          * For a create, however, we want to fail the request if
1550          * there are any components we can't open.
1551          */
1552         error = vdev_open(vd);
1553
1554         if (error || vd->vdev_state != VDEV_STATE_HEALTHY) {
1555                 vdev_close(vd);
1556                 return (error ? error : ENXIO);
1557         }
1558
1559         /*
1560          * Recursively load DTLs and initialize all labels.
1561          */
1562         if ((error = vdev_dtl_load(vd)) != 0 ||
1563             (error = vdev_label_init(vd, txg, isreplacing ?
1564             VDEV_LABEL_REPLACE : VDEV_LABEL_CREATE)) != 0) {
1565                 vdev_close(vd);
1566                 return (error);
1567         }
1568
1569         return (0);
1570 }
1571
1572 void
1573 vdev_metaslab_set_size(vdev_t *vd)
1574 {
1575         /*
1576          * Aim for roughly 200 metaslabs per vdev.
1577          */
1578         vd->vdev_ms_shift = highbit64(vd->vdev_asize / 200);
1579         vd->vdev_ms_shift = MAX(vd->vdev_ms_shift, SPA_MAXBLOCKSHIFT);
1580 }
1581
1582 void
1583 vdev_dirty(vdev_t *vd, int flags, void *arg, uint64_t txg)
1584 {
1585         ASSERT(vd == vd->vdev_top);
1586         ASSERT(!vd->vdev_ishole);
1587         ASSERT(ISP2(flags));
1588         ASSERT(spa_writeable(vd->vdev_spa));
1589
1590         if (flags & VDD_METASLAB)
1591                 (void) txg_list_add(&vd->vdev_ms_list, arg, txg);
1592
1593         if (flags & VDD_DTL)
1594                 (void) txg_list_add(&vd->vdev_dtl_list, arg, txg);
1595
1596         (void) txg_list_add(&vd->vdev_spa->spa_vdev_txg_list, vd, txg);
1597 }
1598
1599 void
1600 vdev_dirty_leaves(vdev_t *vd, int flags, uint64_t txg)
1601 {
1602         for (int c = 0; c < vd->vdev_children; c++)
1603                 vdev_dirty_leaves(vd->vdev_child[c], flags, txg);
1604
1605         if (vd->vdev_ops->vdev_op_leaf)
1606                 vdev_dirty(vd->vdev_top, flags, vd, txg);
1607 }
1608
1609 /*
1610  * DTLs.
1611  *
1612  * A vdev's DTL (dirty time log) is the set of transaction groups for which
1613  * the vdev has less than perfect replication.  There are four kinds of DTL:
1614  *
```

```
1615  * DTL_MISSING: txgs for which the vdev has no valid copies of the data
1616  *
1617  * DTL_PARTIAL: txgs for which data is available, but not fully replicated
1618  *
1619  * DTL_SCRUB: the txgs that could not be repaired by the last scrub; upon
1620  *      scrub completion, DTL_SCRUB replaces DTL_MISSING in the range of
1621  *      txgs that was scrubbed.
1622  *
1623  * DTL_OUTAGE: txgs which cannot currently be read, whether due to
1624  *      persistent errors or just some device being offline.
1625  *      Unlike the other three, the DTL_OUTAGE map is not generally
1626  *      maintained; it's only computed when needed, typically to
1627  *      determine whether a device can be detached.
1628  *
1629  * For leaf vdevs, DTL_MISSING and DTL_PARTIAL are identical: the device
1630  * either has the data or it doesn't.
1631  *
1632  * For interior vdevs such as mirror and RAID-Z the picture is more complex.
1633  * A vdev's DTL_PARTIAL is the union of its children's DTL_PARTIALs, because
1634  * if any child is less than fully replicated, then so is its parent.
1635  * A vdev's DTL_MISSING is a modified union of its children's DTL_MISSINGs,
1636  * comprising only those txgs which appear in 'maxfaults' or more children;
1637  * those are the txgs we don't have enough replication to read.  For example,
1638  * double-parity RAID-Z can tolerate up to two missing devices (maxfaults == 2);
1639  * thus, its DTL_MISSING consists of the set of txgs that appear in more than
1640  * two child DTL_MISSING maps.
1641  *
1642  * It should be clear from the above that to compute the DTLs and outage maps
1643  * for all vdevs, it suffices to know just the leaf vdevs' DTL_MISSING maps.
1644  * Therefore, that is all we keep on disk.  When loading the pool, or after
1645  * a configuration change, we generate all other DTLs from first principles.
1646  */
1647 void
1648 vdev_dtl_dirty(vdev_t *vd, vdev_dtl_type_t t, uint64_t txg, uint64_t size)
1649 {
1650         range_tree_t *rt = vd->vdev_dtl[t];

1652         ASSERT(t < DTL_TYPES);
1653         ASSERT(vd != vd->vdev_spa->spa_root_vdev);
1654         ASSERT(spa_writeable(vd->vdev_spa));

1656         mutex_enter(rt->rt_lock);
1657         if (!range_tree_contains(rt, txg, size))
1658                 range_tree_add(rt, txg, size);
1659         mutex_exit(rt->rt_lock);
1660 }

1662 boolean_t
1663 vdev_dtl_contains(vdev_t *vd, vdev_dtl_type_t t, uint64_t txg, uint64_t size)
1664 {
1665         range_tree_t *rt = vd->vdev_dtl[t];
1666         boolean_t dirty = B_FALSE;

1668         ASSERT(t < DTL_TYPES);
1669         ASSERT(vd != vd->vdev_spa->spa_root_vdev);

1671         mutex_enter(rt->rt_lock);
1672         if (range_tree_space(rt) != 0)
1673                 dirty = range_tree_contains(rt, txg, size);
1674         mutex_exit(rt->rt_lock);

1676         return (dirty);
1677 }

1679 boolean_t
1680 vdev_dtl_empty(vdev_t *vd, vdev_dtl_type_t t)
```



```
1681 {
1682         range_tree_t *rt = vd->vdev_dtl[t];
1683         boolean_t empty;

1685         mutex_enter(rt->rt_lock);
1686         empty = (range_tree_space(rt) == 0);
1687         mutex_exit(rt->rt_lock);

1689         return (empty);
1690 }

1692 /*
1693  * Returns the lowest txg in the DTL range.
1694  */
1695 static uint64_t
1696 vdev_dtl_min(vdev_t *vd)
1697 {
1698         range_seg_t *rs;

1700         ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
1701         ASSERT3U(range_tree_space(vd->vdev_dtl[DTL_MISSING]), !=, 0);
1702         ASSERT0(vd->vdev_children);

1704         rs = avl_first(&vd->vdev_dtl[DTL_MISSING]->rt_root);
1705         return (rs->rs_start - 1);
1706 }

1708 /*
1709  * Returns the highest txg in the DTL.
1710  */
1711 static uint64_t
1712 vdev_dtl_max(vdev_t *vd)
1713 {
1714         range_seg_t *rs;

1716         ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
1717         ASSERT3U(range_tree_space(vd->vdev_dtl[DTL_MISSING]), !=, 0);
1718         ASSERT0(vd->vdev_children);

1720         rs = avl_last(&vd->vdev_dtl[DTL_MISSING]->rt_root);
1721         return (rs->rs_end);
1722 }

1724 /*
1725  * Determine if a resilvering vdev should remove any DTL entries from
1726  * its range. If the vdev was resilvering for the entire duration of the
1727  * scan then it should excise that range from its DTLs. Otherwise, this
1728  * vdev is considered partially resilvered and should leave its DTL
1729  * entries intact. The comment in vdev_dtl_reassess() describes how we
1730  * excise the DTLs.
1731  */
1732 static boolean_t
1733 vdev_dtl_should_excise(vdev_t *vd)
1734 {
1735         spa_t *spa = vd->vdev_spa;
1736         dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;

1738         ASSERT0(scn->scn_phys.scn_errors);
1739         ASSERT0(vd->vdev_children);

1741         if (vd->vdev_resilver_txg == 0 ||
1742             range_tree_space(vd->vdev_dtl[DTL_MISSING]) == 0)
1743                 return (B_TRUE);

1745         /*
1746          * When a resilver is initiated the scan will assign the scn_max_txg
```

```
1747          * value to the highest txg value that exists in all DTLs. If this
1748          * device's max DTL is not part of this scan (i.e. it is not in
1749          * the range (scn_min_txg, scn_max_txg] then it is not eligible
1750          * for excision.
1751          */
1752         if (vdev_dtl_max(vd) <= scn->scn_phys.scn_max_txg) {
1753                 ASSERT3U(scn->scn_phys.scn_min_txg, <=, vdev_dtl_min(vd));
1754                 ASSERT3U(scn->scn_phys.scn_min_txg, <, vd->vdev_resilver_txg);
1755                 ASSERT3U(vd->vdev_resilver_txg, <=, scn->scn_phys.scn_max_txg);
1756                 return (B_TRUE);
1757         }
1758         return (B_FALSE);
1759 }
1761 /*
1762  * Reassess DTLs after a config change or scrub completion.
1763  */
1764 void
1765 vdev_dtl_reassess(vdev_t *vd, uint64_t txg, uint64_t scrub_txg, int scrub_done)
1766 {
1767         spa_t *spa = vd->vdev_spa;
1768         avl_tree_t reftree;
1769         int minref;
1771         ASSERT(spa_config_held(spa, SCL_ALL, RW_READER) != 0);
1773         for (int c = 0; c < vd->vdev_children; c++)
1774                 vdev_dtl_reassess(vd->vdev_child[c], txg,
1775                     scrub_txg, scrub_done);
1777         if (vd == spa->spa_root_vdev || vd->vdev_ishole || vd->vdev_aux)
1778                 return;
1780         if (vd->vdev_ops->vdev_op_leaf) {
1781                 dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;
1783                 mutex_enter(&vd->vdev_dtl_lock);
1785                 /*
1786                  * If we've completed a scan cleanly then determine
1787                  * if this vdev should remove any DTLs. We only want to
1788                  * excise regions on vdevs that were available during
1789                  * the entire duration of this scan.
1790                  */
1791                 if (scrub_txg != 0 &&
1792                     (spa->spa_scrub_started ||
1793                     (scn != NULL && scn->scn_phys.scn_errors == 0)) &&
1794                     vdev_dtl_should_excise(vd)) {
1795                         /*
1796                          * We completed a scrub up to scrub_txg.  If we
1797                          * did it without rebooting, then the scrub dtl
1798                          * will be valid, so excise the old region and
1799                          * fold in the scrub dtl.  Otherwise, leave the
1800                          * dtl as-is if there was an error.
1801                          *
1802                          * There's little trick here: to excise the beginning
1803                          * of the DTL_MISSING map, we put it into a reference
1804                          * tree and then add a segment with refcnt -1 that
1805                          * covers the range [0, scrub_txg).  This means
1806                          * that each txg in that range has refcnt -1 or 0.
1807                          * We then add DTL_SCRUB with a refcnt of 2, so that
1808                          * entries in the range [0, scrub_txg) will have a
1809                          * positive refcnt -- either 1 or 2.  We then convert
1810                          * the reference tree into the new DTL_MISSING map.
1811                          */
1812                         space_reftree_create(&reftree);
```



```
1813                         space_reftree_add_map(&reftree,
1814                             vd->vdev_dtl[DTL_MISSING], 1);
1815                         space_reftree_add_seg(&reftree, 0, scrub_txg, -1);
1816                         space_reftree_add_map(&reftree,
1817                             vd->vdev_dtl[DTL_SCRUB], 2);
1818                         space_reftree_generate_map(&reftree,
1819                             vd->vdev_dtl[DTL_MISSING], 1);
1820                         space_reftree_destroy(&reftree);
1821                 }
1822                 range_tree_vacate(vd->vdev_dtl[DTL_PARTIAL], NULL, NULL);
1823                 range_tree_walk(vd->vdev_dtl[DTL_MISSING],
1824                     range_tree_add, vd->vdev_dtl[DTL_PARTIAL]);
1825                 if (scrub_done)
1826                         range_tree_vacate(vd->vdev_dtl[DTL_SCRUB], NULL, NULL);
1827                 range_tree_vacate(vd->vdev_dtl[DTL_OUTAGE], NULL, NULL);
1828                 if (!vdev_readable(vd))
1829                         range_tree_add(vd->vdev_dtl[DTL_OUTAGE], 0, -1ULL);
1830                 else
1831                         range_tree_walk(vd->vdev_dtl[DTL_MISSING],
1832                             range_tree_add, vd->vdev_dtl[DTL_OUTAGE]);
1834                 /*
1835                  * If the vdev was resilvering and no longer has any
1836                  * DTLs then reset its resilvering flag.
1837                  */
1838                 if (vd->vdev_resilver_txg != 0 &&
1839                     range_tree_space(vd->vdev_dtl[DTL_MISSING]) == 0 &&
1840                     range_tree_space(vd->vdev_dtl[DTL_OUTAGE]) == 0)
1841                         vd->vdev_resilver_txg = 0;
1843                 mutex_exit(&vd->vdev_dtl_lock);
1845                 if (txg != 0)
1846                         vdev_dirty(vd->vdev_top, VDD_DTL, vd, txg);
1847                 return;
1848         }
1850         mutex_enter(&vd->vdev_dtl_lock);
1851         for (int t = 0; t < DTL_TYPES; t++) {
1852                 /* account for child's outage in parent's missing map */
1853                 int s = (t == DTL_MISSING) ? DTL_OUTAGE: t;
1854                 if (t == DTL_SCRUB)
1855                         continue;                       /* leaf vdevs only */
1856                 if (t == DTL_PARTIAL)
1857                         minref = 1;                     /* i.e. non-zero */
1858                 else if (vd->vdev_nparity != 0)
1859                         minref = vd->vdev_nparity + 1;  /* RAID-Z */
1860                 else
1861                         minref = vd->vdev_children;     /* any kind of mirror */
1862                 space_reftree_create(&reftree);
1863                 for (int c = 0; c < vd->vdev_children; c++) {
1864                         vdev_t *cvd = vd->vdev_child[c];
1865                         mutex_enter(&cvd->vdev_dtl_lock);
1866                         space_reftree_add_map(&reftree, cvd->vdev_dtl[s], 1);
1867                         mutex_exit(&cvd->vdev_dtl_lock);
1868                 }
1869                 space_reftree_generate_map(&reftree, vd->vdev_dtl[t], minref);
1870                 space_reftree_destroy(&reftree);
1871         }
1872         mutex_exit(&vd->vdev_dtl_lock);
1873 }
1875 int
1876 vdev_dtl_load(vdev_t *vd)
1877 {
1878         spa_t *spa = vd->vdev_spa;
```

```
1879         objset_t *mos = spa->spa_meta_objset;
1880         int error = 0;

1882         if (vd->vdev_ops->vdev_op_leaf && vd->vdev_dtl_object != 0) {
1883                 ASSERT(!vd->vdev_ishole);

1885                 error = space_map_open(&vd->vdev_dtl_sm, mos,
1886                     vd->vdev_dtl_object, 0, -1ULL, 0, &vd->vdev_dtl_lock);
1887                 if (error)
1888                         return (error);
1889                 ASSERT(vd->vdev_dtl_sm != NULL);

1891                 mutex_enter(&vd->vdev_dtl_lock);

1893                 /*
1894                  * Now that we've opened the space_map we need to update
1895                  * the in-core DTL.
1896                  */
1897                 space_map_update(vd->vdev_dtl_sm);

1899                 error = space_map_load(vd->vdev_dtl_sm,
1900                     vd->vdev_dtl[DTL_MISSING], SM_ALLOC);
1901                 mutex_exit(&vd->vdev_dtl_lock);

1903                 return (error);
1904         }

1906         for (int c = 0; c < vd->vdev_children; c++) {
1907                 error = vdev_dtl_load(vd->vdev_child[c]);
1908                 if (error != 0)
1909                         break;
1910         }

1912         return (error);
1913 }

1915 void
1916 vdev_dtl_sync(vdev_t *vd, uint64_t txg)
1917 {
1918         spa_t *spa = vd->vdev_spa;
1919         range_tree_t *rt = vd->vdev_dtl[DTL_MISSING];
1920         objset_t *mos = spa->spa_meta_objset;
1921         range_tree_t *rtsync;
1922         kmutex_t rtlock;
1923         dmu_tx_t *tx;
1924         uint64_t object = space_map_object(vd->vdev_dtl_sm);

1926         ASSERT(!vd->vdev_ishole);
1927         ASSERT(vd->vdev_ops->vdev_op_leaf);

1929         tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);

1931         if (vd->vdev_detached || vd->vdev_top->vdev_removing) {
1932                 mutex_enter(&vd->vdev_dtl_lock);
1933                 space_map_free(vd->vdev_dtl_sm, tx);
1934                 space_map_close(vd->vdev_dtl_sm);
1935                 vd->vdev_dtl_sm = NULL;
1936                 mutex_exit(&vd->vdev_dtl_lock);
1937                 dmu_tx_commit(tx);
1938                 return;
1939         }

1941         if (vd->vdev_dtl_sm == NULL) {
1942                 uint64_t new_object;

1944                 new_object = space_map_alloc(mos, tx);
```



```
1945                 VERIFY3U(new_object, !=, 0);

1947                 VERIFY0(space_map_open(&vd->vdev_dtl_sm, mos, new_object,
1948                     0, -1ULL, 0, &vd->vdev_dtl_lock));
1949                 ASSERT(vd->vdev_dtl_sm != NULL);
1950         }

1952         mutex_init(&rtlock, NULL, MUTEX_DEFAULT, NULL);

1954         rtsync = range_tree_create(NULL, NULL, &rtlock);

1956         mutex_enter(&rtlock);

1958         mutex_enter(&vd->vdev_dtl_lock);
1959         range_tree_walk(rt, range_tree_add, rtsync);
1960         mutex_exit(&vd->vdev_dtl_lock);

1962         space_map_truncate(vd->vdev_dtl_sm, tx);
1963         space_map_write(vd->vdev_dtl_sm, rtsync, SM_ALLOC, tx);
1964         range_tree_vacate(rtsync, NULL, NULL);

1966         range_tree_destroy(rtsync);

1968         mutex_exit(&rtlock);
1969         mutex_destroy(&rtlock);

1971         /*
1972          * If the object for the space map has changed then dirty
1973          * the top level so that we update the config.
1974          */
1975         if (object != space_map_object(vd->vdev_dtl_sm)) {
1976                 zfs_dbgmsg("txg %llu, spa %s, DTL old object %llu, "
1977                     "new object %llu", txg, spa_name(spa), object,
1978                     space_map_object(vd->vdev_dtl_sm));
1979                 vdev_config_dirty(vd->vdev_top);
1980         }

1982         dmu_tx_commit(tx);

1984         mutex_enter(&vd->vdev_dtl_lock);
1985         space_map_update(vd->vdev_dtl_sm);
1986         mutex_exit(&vd->vdev_dtl_lock);
1987 }

1989 /*
1990  * Determine whether the specified vdev can be offlined/detached/removed
1991  * without losing data.
1992  */
1993 boolean_t
1994 vdev_dtl_required(vdev_t *vd)
1995 {
1996         spa_t *spa = vd->vdev_spa;
1997         vdev_t *tvd = vd->vdev_top;
1998         uint8_t cant_read = vd->vdev_cant_read;
1999         boolean_t required;

2001         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

2003         if (vd == spa->spa_root_vdev || vd == tvd)
2004                 return (B_TRUE);

2006         /*
2007          * Temporarily mark the device as unreadable, and then determine
2008          * whether this results in any DTL outages in the top-level vdev.
2009          * If not, we can safely offline/detach/remove the device.
2010          */
```

```
2011         vd->vdev_cant_read = B_TRUE;
2012         vdev_dtl_reassess(tvd, 0, 0, B_FALSE);
2013         required = !vdev_dtl_empty(tvd, DTL_OUTAGE);
2014         vd->vdev_cant_read = cant_read;
2015         vdev_dtl_reassess(tvd, 0, 0, B_FALSE);
2016 
2017         if (!required && zio_injection_enabled)
2018                 required = !!zio_handle_device_injection(vd, NULL, ECHILD);
2019 
2020         return (required);
2021 }
2022 
2023 /*
2024  * Determine if resilver is needed, and if so the txg range.
2025  */
2026 boolean_t
2027 vdev_resilver_needed(vdev_t *vd, uint64_t *minp, uint64_t *maxp)
2028 {
2029         boolean_t needed = B_FALSE;
2030         uint64_t thismin = UINT64_MAX;
2031         uint64_t thismax = 0;
2032 
2033         if (vd->vdev_children == 0) {
2034                 mutex_enter(&vd->vdev_dtl_lock);
2035                 if (range_tree_space(vd->vdev_dtl[DTL_MISSING]) != 0 &&
2036                     vdev_writeable(vd)) {
2037 
2038                         thismin = vdev_dtl_min(vd);
2039                         thismax = vdev_dtl_max(vd);
2040                         needed = B_TRUE;
2041                 }
2042                 mutex_exit(&vd->vdev_dtl_lock);
2043         } else {
2044                 for (int c = 0; c < vd->vdev_children; c++) {
2045                         vdev_t *cvd = vd->vdev_child[c];
2046                         uint64_t cmin, cmax;
2047 
2048                         if (vdev_resilver_needed(cvd, &cmin, &cmax)) {
2049                                 thismin = MIN(thismin, cmin);
2050                                 thismax = MAX(thismax, cmax);
2051                                 needed = B_TRUE;
2052                         }
2053                 }
2054         }
2055 
2056         if (needed && minp) {
2057                 *minp = thismin;
2058                 *maxp = thismax;
2059         }
2060         return (needed);
2061 }
2062 
2063 void
2064 vdev_load(vdev_t *vd)
2065 {
2066         /*
2067          * Recursively load all children.
2068          */
2069         for (int c = 0; c < vd->vdev_children; c++)
2070                 vdev_load(vd->vdev_child[c]);
2071 
2072         /*
2073          * If this is a top-level vdev, initialize its metaslabs.
2074          */
2075         if (vd == vd->vdev_top && !vd->vdev_ishole &&
2076             (vd->vdev_ashift == 0 || vd->vdev_asize == 0 ||
```



```
2077                 vdev_metaslab_init(vd, 0) != 0))
2078                     vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
2079                         VDEV_AUX_CORRUPT_DATA);
2080 
2081         /*
2082          * If this is a leaf vdev, load its DTL.
2083          */
2084         if (vd->vdev_ops->vdev_op_leaf && vdev_dtl_load(vd) != 0)
2085                 vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
2086                     VDEV_AUX_CORRUPT_DATA);
2087 }
2088 
2089 /*
2090  * The special vdev case is used for hot spares and l2cache devices.  Its
2091  * sole purpose it to set the vdev state for the associated vdev.  To do this,
2092  * we make sure that we can open the underlying device, then try to read the
2093  * label, and make sure that the label is sane and that it hasn't been
2094  * repurposed to another pool.
2095  */
2096 int
2097 vdev_validate_aux(vdev_t *vd)
2098 {
2099         nvlist_t *label;
2100         uint64_t guid, version;
2101         uint64_t state;
2102 
2103         if (!vdev_readable(vd))
2104                 return (0);
2105 
2106         if ((label = vdev_label_read_config(vd, -1ULL)) == NULL) {
2107                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
2108                     VDEV_AUX_CORRUPT_DATA);
2109                 return (-1);
2110         }
2111 
2112         if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_VERSION, &version) != 0 ||
2113             !SPA_VERSION_IS_SUPPORTED(version) ||
2114             nvlist_lookup_uint64(label, ZPOOL_CONFIG_GUID, &guid) != 0 ||
2115             guid != vd->vdev_guid ||
2116             nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_STATE, &state) != 0) {
2117                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
2118                     VDEV_AUX_CORRUPT_DATA);
2119                 nvlist_free(label);
2120                 return (-1);
2121         }
2122 
2123         /*
2124          * We don't actually check the pool state here.  If it's in fact in
2125          * use by another pool, we update this fact on the fly when requested.
2126          */
2127         nvlist_free(label);
2128         return (0);
2129 }
2130 
2131 void
2132 vdev_remove(vdev_t *vd, uint64_t txg)
2133 {
2134         spa_t *spa = vd->vdev_spa;
2135         objset_t *mos = spa->spa_meta_objset;
2136         dmu_tx_t *tx;
2137 
2138         tx = dmu_tx_create_assigned(spa_get_dsl(spa), txg);
2139 
2140         if (vd->vdev_ms != NULL) {
2141                 for (int m = 0; m < vd->vdev_ms_count; m++) {
2142                         metaslab_t *msp = vd->vdev_ms[m];
```

```
2144                        if (msp == NULL || msp->ms_sm == NULL)
2145                                continue;

2147                        mutex_enter(&msp->ms_lock);
2148                        VERIFY0(space_map_allocated(msp->ms_sm));
2149                        space_map_free(msp->ms_sm, tx);
2150                        space_map_close(msp->ms_sm);
2151                        msp->ms_sm = NULL;
2152                        mutex_exit(&msp->ms_lock);
2153                }
2154        }

2156        if (vd->vdev_ms_array) {
2157                (void) dmu_object_free(mos, vd->vdev_ms_array, tx);
2158                vd->vdev_ms_array = 0;
2159        }
2160        dmu_tx_commit(tx);
2161 }

2163 void
2164 vdev_sync_done(vdev_t *vd, uint64_t txg)
2165 {
2166        metaslab_t *msp;
2167        boolean_t reassess = !txg_list_empty(&vd->vdev_ms_list, TXG_CLEAN(txg));

2169        ASSERT(!vd->vdev_ishole);

2171        while (msp = txg_list_remove(&vd->vdev_ms_list, TXG_CLEAN(txg)))
2172                metaslab_sync_done(msp, txg);

2174        if (reassess)
2175                metaslab_sync_reassess(vd->vdev_mg);
2176 }

2178 void
2179 vdev_sync(vdev_t *vd, uint64_t txg)
2180 {
2181        spa_t *spa = vd->vdev_spa;
2182        vdev_t *lvd;
2183        metaslab_t *msp;
2184        dmu_tx_t *tx;

2186        ASSERT(!vd->vdev_ishole);

2188        if (vd->vdev_ms_array == 0 && vd->vdev_ms_shift != 0) {
2189                ASSERT(vd == vd->vdev_top);
2190                tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);
2191                vd->vdev_ms_array = dmu_object_alloc(spa->spa_meta_objset,
2192                    DMU_OT_OBJECT_ARRAY, 0, DMU_OT_NONE, 0, tx);
2193                ASSERT(vd->vdev_ms_array != 0);
2194                vdev_config_dirty(vd);
2195                dmu_tx_commit(tx);
2196        }

2198        /*
2199         * Remove the metadata associated with this vdev once it's empty.
2200         */
2201        if (vd->vdev_stat.vs_alloc == 0 && vd->vdev_removing)
2202                vdev_remove(vd, txg);

2204        while ((msp = txg_list_remove(&vd->vdev_ms_list, txg)) != NULL) {
2205                metaslab_sync(msp, txg);
2206                (void) txg_list_add(&vd->vdev_ms_list, msp, TXG_CLEAN(txg));
2207        }
```



```
2209        while ((lvd = txg_list_remove(&vd->vdev_dtl_list, txg)) != NULL)
2210                vdev_dtl_sync(lvd, txg);

2212        (void) txg_list_add(&spa->spa_vdev_txg_list, vd, TXG_CLEAN(txg));
2213 }

2215 uint64_t
2216 vdev_psize_to_asize(vdev_t *vd, uint64_t psize)
2217 {
2218        return (vd->vdev_ops->vdev_op_asize(vd, psize));
2219 }

2221 /*
2222  * Mark the given vdev faulted.  A faulted vdev behaves as if the device could
2223  * not be opened, and no I/O is attempted.
2224  */
2225 int
2226 vdev_fault(spa_t *spa, uint64_t guid, vdev_aux_t aux)
2227 {
2228        vdev_t *vd, *tvd;

2230        spa_vdev_state_enter(spa, SCL_NONE);

2232        if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2233                return (spa_vdev_state_exit(spa, NULL, ENODEV));

2235        if (!vd->vdev_ops->vdev_op_leaf)
2236                return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2238        tvd = vd->vdev_top;

2240        /*
2241         * We don't directly use the aux state here, but if we do a
2242         * vdev_reopen(), we need this value to be present to remember why we
2243         * were faulted.
2244         */
2245        vd->vdev_label_aux = aux;

2247        /*
2248         * Faulted state takes precedence over degraded.
2249         */
2250        vd->vdev_delayed_close = B_FALSE;
2251        vd->vdev_faulted = 1ULL;
2252        vd->vdev_degraded = 0ULL;
2253        vdev_set_state(vd, B_FALSE, VDEV_STATE_FAULTED, aux);

2255        /*
2256         * If this device has the only valid copy of the data, then
2257         * back off and simply mark the vdev as degraded instead.
2258         */
2259        if (!tvd->vdev_islog && vd->vdev_aux == NULL && vdev_dtl_required(vd)) {
2260                vd->vdev_degraded = 1ULL;
2261                vd->vdev_faulted = 0ULL;

2263                /*
2264                 * If we reopen the device and it's not dead, only then do we
2265                 * mark it degraded.
2266                 */
2267                vdev_reopen(tvd);

2269                if (vdev_readable(vd))
2270                        vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED, aux);
2271        }

2273        return (spa_vdev_state_exit(spa, vd, 0));
2274 }
```

```
2276 /*
2277  * Mark the given vdev degraded.  A degraded vdev is purely an indication to the
2278  * user that something is wrong.  The vdev continues to operate as normal as far
2279  * as I/O is concerned.
2280  */
2281 int
2282 vdev_degrade(spa_t *spa, uint64_t guid, vdev_aux_t aux)
2283 {
2284         vdev_t *vd;

2286         spa_vdev_state_enter(spa, SCL_NONE);

2288         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2289                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2291         if (!vd->vdev_ops->vdev_op_leaf)
2292                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2294         /*
2295          * If the vdev is already faulted, then don't do anything.
2296          */
2297         if (vd->vdev_faulted || vd->vdev_degraded)
2298                 return (spa_vdev_state_exit(spa, NULL, 0));

2300         vd->vdev_degraded = 1ULL;
2301         if (!vdev_is_dead(vd))
2302                 vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED,
2303                     aux);

2305         return (spa_vdev_state_exit(spa, vd, 0));
2306 }

2308 /*
2309  * Online the given vdev.
2310  *
2311  * If 'ZFS_ONLINE_UNSPARE' is set, it implies two things.  First, any attached
2312  * spare device should be detached when the device finishes resilvering.
2313  * Second, the online should be treated like a 'test' online case, so no FMA
2314  * events are generated if the device fails to open.
2315  */
2316 int
2317 vdev_online(spa_t *spa, uint64_t guid, uint64_t flags, vdev_state_t *newstate)
2318 {
2319         vdev_t *vd, *tvd, *pvd, *rvd = spa->spa_root_vdev;

2321         spa_vdev_state_enter(spa, SCL_NONE);

2323         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2324                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2326         if (!vd->vdev_ops->vdev_op_leaf)
2327                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2329         tvd = vd->vdev_top;
2330         vd->vdev_offline = B_FALSE;
2331         vd->vdev_tmpoffline = B_FALSE;
2332         vd->vdev_checkremove = !!(flags & ZFS_ONLINE_CHECKREMOVE);
2333         vd->vdev_forcefault = !!(flags & ZFS_ONLINE_FORCEFAULT);

2335         /* XXX - L2ARC 1.0 does not support expansion */
2336         if (!vd->vdev_aux) {
2337                 for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2338                         pvd->vdev_expanding = !!(flags & ZFS_ONLINE_EXPAND);
2339         }
```



```
2341         vdev_reopen(tvd);
2342         vd->vdev_checkremove = vd->vdev_forcefault = B_FALSE;

2344         if (!vd->vdev_aux) {
2345                 for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2346                         pvd->vdev_expanding = B_FALSE;
2347         }

2349         if (newstate)
2350                 *newstate = vd->vdev_state;
2351         if ((flags & ZFS_ONLINE_UNSPARE) &&
2352             !vdev_is_dead(vd) && vd->vdev_parent &&
2353             vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
2354             vd->vdev_parent->vdev_child[0] == vd)
2355                 vd->vdev_unspare = B_TRUE;

2357         if ((flags & ZFS_ONLINE_EXPAND) || spa->spa_autoexpand) {

2359                 /* XXX - L2ARC 1.0 does not support expansion */
2360                 if (vd->vdev_aux)
2361                         return (spa_vdev_state_exit(spa, vd, ENOTSUP));
2362                 spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
2363         }
2364         return (spa_vdev_state_exit(spa, vd, 0));
2365 }

2367 static int
2368 vdev_offline_locked(spa_t *spa, uint64_t guid, uint64_t flags)
2369 {
2370         vdev_t *vd, *tvd;
2371         int error = 0;
2372         uint64_t generation;
2373         metaslab_group_t *mg;

2375 top:
2376         spa_vdev_state_enter(spa, SCL_ALLOC);

2378         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2379                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2381         if (!vd->vdev_ops->vdev_op_leaf)
2382                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2384         tvd = vd->vdev_top;
2385         mg = tvd->vdev_mg;
2386         generation = spa->spa_config_generation + 1;

2388         /*
2389          * If the device isn't already offline, try to offline it.
2390          */
2391         if (!vd->vdev_offline) {
2392                 /*
2393                  * If this device has the only valid copy of some data,
2394                  * don't allow it to be offlined. Log devices are always
2395                  * expendable.
2396                  */
2397                 if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
2398                     vdev_dtl_required(vd))
2399                         return (spa_vdev_state_exit(spa, NULL, EBUSY));

2401                 /*
2402                  * If the top-level is a slog and it has had allocations
2403                  * then proceed.  We check that the vdev's metaslab group
2404                  * is not NULL since it's possible that we may have just
2405                  * added this vdev but not yet initialized its metaslabs.
2406                  */
```

```
2407                 if (tvd->vdev_islog && mg != NULL) {
2408                         /*
2409                          * Prevent any future allocations.
2410                          */
2411                         metaslab_group_passivate(mg);
2412                         (void) spa_vdev_state_exit(spa, vd, 0);

2414                         error = spa_offline_log(spa);

2416                         spa_vdev_state_enter(spa, SCL_ALLOC);

2418                         /*
2419                          * Check to see if the config has changed.
2420                          */
2421                         if (error || generation != spa->spa_config_generation) {
2422                                 metaslab_group_activate(mg);
2423                                 if (error)
2424                                         return (spa_vdev_state_exit(spa,
2425                                             vd, error));
2426                                 (void) spa_vdev_state_exit(spa, vd, 0);
2427                                 goto top;
2428                         }
2429                         ASSERT0(tvd->vdev_stat.vs_alloc);
2430                 }

2432                 /*
2433                  * Offline this device and reopen its top-level vdev.
2434                  * If the top-level vdev is a log device then just offline
2435                  * it. Otherwise, if this action results in the top-level
2436                  * vdev becoming unusable, undo it and fail the request.
2437                  */
2438                 vd->vdev_offline = B_TRUE;
2439                 vdev_reopen(tvd);

2441                 if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
2442                     vdev_is_dead(tvd)) {
2443                         vd->vdev_offline = B_FALSE;
2444                         vdev_reopen(tvd);
2445                         return (spa_vdev_state_exit(spa, NULL, EBUSY));
2446                 }

2448                 /*
2449                  * Add the device back into the metaslab rotor so that
2450                  * once we online the device it's open for business.
2451                  */
2452                 if (tvd->vdev_islog && mg != NULL)
2453                         metaslab_group_activate(mg);
2454         }

2456         vd->vdev_tmpoffline = !!(flags & ZFS_OFFLINE_TEMPORARY);

2458         return (spa_vdev_state_exit(spa, vd, 0));
2459 }

2461 int
2462 vdev_offline(spa_t *spa, uint64_t guid, uint64_t flags)
2463 {
2464         int error;

2466         mutex_enter(&spa->spa_vdev_top_lock);
2467         error = vdev_offline_locked(spa, guid, flags);
2468         mutex_exit(&spa->spa_vdev_top_lock);

2470         return (error);
2471 }
```



```
2473 /*
2474  * Clear the error counts associated with this vdev.  Unlike vdev_online() and
2475  * vdev_offline(), we assume the spa config is locked.  We also clear all
2476  * children.  If 'vd' is NULL, then the user wants to clear all vdevs.
2477  */
2478 void
2479 vdev_clear(spa_t *spa, vdev_t *vd)
2480 {
2481         vdev_t *rvd = spa->spa_root_vdev;

2483         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

2485         if (vd == NULL)
2486                 vd = rvd;

2488         vd->vdev_stat.vs_read_errors = 0;
2489         vd->vdev_stat.vs_write_errors = 0;
2490         vd->vdev_stat.vs_checksum_errors = 0;

2492         for (int c = 0; c < vd->vdev_children; c++)
2493                 vdev_clear(spa, vd->vdev_child[c]);

2495         /*
2496          * If we're in the FAULTED state or have experienced failed I/O, then
2497          * clear the persistent state and attempt to reopen the device.  We
2498          * also mark the vdev config dirty, so that the new faulted state is
2499          * written out to disk.
2500          */
2501         if (vd->vdev_faulted || vd->vdev_degraded ||
2502             !vdev_readable(vd) || !vdev_writeable(vd)) {

2504                 /*
2505                  * When reopening in reponse to a clear event, it may be due to
2506                  * a fmadm repair request.  In this case, if the device is
2507                  * still broken, we want to still post the ereport again.
2508                  */
2509                 vd->vdev_forcefault = B_TRUE;

2511                 vd->vdev_faulted = vd->vdev_degraded = 0ULL;
2512                 vd->vdev_cant_read = B_FALSE;
2513                 vd->vdev_cant_write = B_FALSE;

2515                 vdev_reopen(vd == rvd ? rvd : vd->vdev_top);

2517                 vd->vdev_forcefault = B_FALSE;

2519                 if (vd != rvd && vdev_writeable(vd->vdev_top))
2520                         vdev_state_dirty(vd->vdev_top);

2522                 if (vd->vdev_aux == NULL && !vdev_is_dead(vd))
2523                         spa_async_request(spa, SPA_ASYNC_RESILVER);

2525                 spa_event_notify(spa, vd, ESC_ZFS_VDEV_CLEAR);
2526         }

2528         /*
2529          * When clearing a FMA-diagnosed fault, we always want to
2530          * unspare the device, as we assume that the original spare was
2531          * done in response to the FMA fault.
2532          */
2533         if (!vdev_is_dead(vd) && vd->vdev_parent != NULL &&
2534             vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
2535             vd->vdev_parent->vdev_child[0] == vd)
2536                 vd->vdev_unspare = B_TRUE;
2537 }
```

```
2539 boolean_t
2540 vdev_is_dead(vdev_t *vd)
2541 {
2542         /*
2543          * Holes and missing devices are always considered "dead".
2544          * This simplifies the code since we don't have to check for
2545          * these types of devices in the various code paths.
2546          * Instead we rely on the fact that we skip over dead devices
2547          * before issuing I/O to them.
2548          */
2549         return (vd->vdev_state < VDEV_STATE_DEGRADED || vd->vdev_ishole ||
2550             vd->vdev_ops == &vdev_missing_ops);
2551 }

2553 boolean_t
2554 vdev_readable(vdev_t *vd)
2555 {
2556         return (!vdev_is_dead(vd) && !vd->vdev_cant_read);
2557 }

2559 boolean_t
2560 vdev_writeable(vdev_t *vd)
2561 {
2562         return (!vdev_is_dead(vd) && !vd->vdev_cant_write);
2563 }

2565 boolean_t
2566 vdev_allocatable(vdev_t *vd)
2567 {
2568         uint64_t state = vd->vdev_state;

2570         /*
2571          * We currently allow allocations from vdevs which may be in the
2572          * process of reopening (i.e. VDEV_STATE_CLOSED). If the device
2573          * fails to reopen then we'll catch it later when we're holding
2574          * the proper locks.  Note that we have to get the vdev state
2575          * in a local variable because although it changes atomically,
2576          * we're asking two separate questions about it.
2577          */
2578         return (!(state < VDEV_STATE_DEGRADED && state != VDEV_STATE_CLOSED) &&
2579             !vd->vdev_cant_write && !vd->vdev_ishole);
2580 }

2582 boolean_t
2583 vdev_accessible(vdev_t *vd, zio_t *zio)
2584 {
2585         ASSERT(zio->io_vd == vd);

2587         if (vdev_is_dead(vd) || vd->vdev_remove_wanted)
2588                 return (B_FALSE);

2590         if (zio->io_type == ZIO_TYPE_READ)
2591                 return (!vd->vdev_cant_read);

2593         if (zio->io_type == ZIO_TYPE_WRITE)
2594                 return (!vd->vdev_cant_write);

2596         return (B_TRUE);
2597 }

2599 /*
2600  * Get statistics for the given vdev.
2601  */
2602 void
2603 vdev_get_stats(vdev_t *vd, vdev_stat_t *vs)
2604 {
```



```
2605         vdev_t *rvd = vd->vdev_spa->spa_root_vdev;

2607         mutex_enter(&vd->vdev_stat_lock);
2608         bcopy(&vd->vdev_stat, vs, sizeof (*vs));
2609         vs->vs_timestamp = gethrtime() - vs->vs_timestamp;
2610         vs->vs_state = vd->vdev_state;
2611         vs->vs_rsize = vdev_get_min_asize(vd);
2612         if (vd->vdev_ops->vdev_op_leaf)
2613                 vs->vs_rsize += VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE;
2614         vs->vs_esize = vd->vdev_max_asize - vd->vdev_asize;
2615         mutex_exit(&vd->vdev_stat_lock);

2617         /*
2618          * If we're getting stats on the root vdev, aggregate the I/O counts
2619          * over all top-level vdevs (i.e. the direct children of the root).
2620          */
2621         if (vd == rvd) {
2622                 for (int c = 0; c < rvd->vdev_children; c++) {
2623                         vdev_t *cvd = rvd->vdev_child[c];
2624                         vdev_stat_t *cvs = &cvd->vdev_stat;

2626                         mutex_enter(&vd->vdev_stat_lock);
2627                         for (int t = 0; t < ZIO_TYPES; t++) {
2628                                 vs->vs_ops[t] += cvs->vs_ops[t];
2629                                 vs->vs_bytes[t] += cvs->vs_bytes[t];
2630                         }
2631                         cvs->vs_scan_removing = cvd->vdev_removing;
2632                         mutex_exit(&vd->vdev_stat_lock);
2633                 }
2634         }
2635 }

2637 void
2638 vdev_clear_stats(vdev_t *vd)
2639 {
2640         mutex_enter(&vd->vdev_stat_lock);
2641         vd->vdev_stat.vs_space = 0;
2642         vd->vdev_stat.vs_dspace = 0;
2643         vd->vdev_stat.vs_alloc = 0;
2644         mutex_exit(&vd->vdev_stat_lock);
2645 }

2647 void
2648 vdev_scan_stat_init(vdev_t *vd)
2649 {
2650         vdev_stat_t *vs = &vd->vdev_stat;

2652         for (int c = 0; c < vd->vdev_children; c++)
2653                 vdev_scan_stat_init(vd->vdev_child[c]);

2655         mutex_enter(&vd->vdev_stat_lock);
2656         vs->vs_scan_processed = 0;
2657         mutex_exit(&vd->vdev_stat_lock);
2658 }

2660 void
2661 vdev_stat_update(zio_t *zio, uint64_t psize)
2662 {
2663         spa_t *spa = zio->io_spa;
2664         vdev_t *rvd = spa->spa_root_vdev;
2665         vdev_t *vd = zio->io_vd ? zio->io_vd : rvd;
2666         vdev_t *pvd;
2667         uint64_t txg = zio->io_txg;
2668         vdev_stat_t *vs = &vd->vdev_stat;
2669         zio_type_t type = zio->io_type;
2670         int flags = zio->io_flags;
```

```
2672         /*
2673          * If this i/o is a gang leader, it didn't do any actual work.
2674          */
2675         if (zio->io_gang_tree)
2676                 return;

2678         if (zio->io_error == 0) {
2679                 /*
2680                  * If this is a root i/o, don't count it -- we've already
2681                  * counted the top-level vdevs, and vdev_get_stats() will
2682                  * aggregate them when asked.  This reduces contention on
2683                  * the root vdev_stat_lock and implicitly handles blocks
2684                  * that compress away to holes, for which there is no i/o.
2685                  * (Holes never create vdev children, so all the counters
2686                  * remain zero, which is what we want.)
2687                  *
2688                  * Note: this only applies to successful i/o (io_error == 0)
2689                  * because unlike i/o counts, errors are not additive.
2690                  * When reading a ditto block, for example, failure of
2691                  * one top-level vdev does not imply a root-level error.
2692                  */
2693                 if (vd == rvd)
2694                         return;

2696                 ASSERT(vd == zio->io_vd);

2698                 if (flags & ZIO_FLAG_IO_BYPASS)
2699                         return;

2701                 mutex_enter(&vd->vdev_stat_lock);

2703                 if (flags & ZIO_FLAG_IO_REPAIR) {
2704                         if (flags & ZIO_FLAG_SCAN_THREAD) {
2705                                 dsl_scan_phys_t *scn_phys =
2706                                     &spa->spa_dsl_pool->dp_scan->scn_phys;
2707                                 uint64_t *processed = &scn_phys->scn_processed;

2709                                 /* XXX cleanup? */
2710                                 if (vd->vdev_ops->vdev_op_leaf)
2711                                         atomic_add_64(processed, psize);
2712                                 vs->vs_scan_processed += psize;
2713                         }

2715                         if (flags & ZIO_FLAG_SELF_HEAL)
2716                                 vs->vs_self_healed += psize;
2717                 }

2719                 vs->vs_ops[type]++;
2720                 vs->vs_bytes[type] += psize;

2722                 mutex_exit(&vd->vdev_stat_lock);
2723                 return;
2724         }

2726         if (flags & ZIO_FLAG_SPECULATIVE)
2727                 return;

2729         /*
2730          * If this is an I/O error that is going to be retried, then ignore the
2731          * error.  Otherwise, the user may interpret B_FAILFAST I/O errors as
2732          * hard errors, when in reality they can happen for any number of
2733          * innocuous reasons (bus resets, MPxIO link failure, etc).
2734          */
2735         if (zio->io_error == EIO &&
2736             !(zio->io_flags & ZIO_FLAG_IO_RETRY))
```



```
2737                 return;

2739         /*
2740          * Intent logs writes won't propagate their error to the root
2741          * I/O so don't mark these types of failures as pool-level
2742          * errors.
2743          */
2744         if (zio->io_vd == NULL && (zio->io_flags & ZIO_FLAG_DONT_PROPAGATE))
2745                 return;

2747         mutex_enter(&vd->vdev_stat_lock);
2748         if (type == ZIO_TYPE_READ && !vdev_is_dead(vd)) {
2749                 if (zio->io_error == ECKSUM)
2750                         vs->vs_checksum_errors++;
2751                 else
2752                         vs->vs_read_errors++;
2753         }
2754         if (type == ZIO_TYPE_WRITE && !vdev_is_dead(vd))
2755                 vs->vs_write_errors++;
2756         mutex_exit(&vd->vdev_stat_lock);

2758         if (type == ZIO_TYPE_WRITE && txg != 0 &&
2759             (!(flags & ZIO_FLAG_IO_REPAIR) ||
2760             (flags & ZIO_FLAG_SCAN_THREAD) ||
2761             spa->spa_claiming)) {
2762                 /*
2763                  * This is either a normal write (not a repair), or it's
2764                  * a repair induced by the scrub thread, or it's a repair
2765                  * made by zil_claim() during spa_load() in the first txg.
2766                  * In the normal case, we commit the DTL change in the same
2767                  * txg as the block was born.  In the scrub-induced repair
2768                  * case, we know that scrubs run in first-pass syncing context,
2769                  * so we commit the DTL change in spa_syncing_txg(spa).
2770                  * In the zil_claim() case, we commit in spa_first_txg(spa).
2771                  *
2772                  * We currently do not make DTL entries for failed spontaneous
2773                  * self-healing writes triggered by normal (non-scrubbing)
2774                  * reads, because we have no transactional context in which to
2775                  * do so -- and it's not clear that it'd be desirable anyway.
2776                  */
2777                 if (vd->vdev_ops->vdev_op_leaf) {
2778                         uint64_t commit_txg = txg;
2779                         if (flags & ZIO_FLAG_SCAN_THREAD) {
2780                                 ASSERT(flags & ZIO_FLAG_IO_REPAIR);
2781                                 ASSERT(spa_sync_pass(spa) == 1);
2782                                 vdev_dtl_dirty(vd, DTL_SCRUB, txg, 1);
2783                                 commit_txg = spa_syncing_txg(spa);
2784                         } else if (spa->spa_claiming) {
2785                                 ASSERT(flags & ZIO_FLAG_IO_REPAIR);
2786                                 commit_txg = spa_first_txg(spa);
2787                         }
2788                         ASSERT(commit_txg >= spa_syncing_txg(spa));
2789                         if (vdev_dtl_contains(vd, DTL_MISSING, txg, 1))
2790                                 return;
2791                         for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2792                                 vdev_dtl_dirty(pvd, DTL_PARTIAL, txg, 1);
2793                         vdev_dirty(vd->vdev_top, VDD_DTL, vd, commit_txg);
2794                 }
2795                 if (vd != rvd)
2796                         vdev_dtl_dirty(vd, DTL_MISSING, txg, 1);
2797         }
2798 }

2800 /*
2801  * Update the in-core space usage stats for this vdev, its metaslab class,
2802  * and the root vdev.
```

```
2803  */
2804 void
2805 vdev_space_update(vdev_t *vd, int64_t alloc_delta, int64_t defer_delta,
2806     int64_t space_delta)
2807 {
2808         int64_t dspace_delta = space_delta;
2809         spa_t *spa = vd->vdev_spa;
2810         vdev_t *rvd = spa->spa_root_vdev;
2811         metaslab_group_t *mg = vd->vdev_mg;
2812         metaslab_class_t *mc = mg ? mg->mg_class : NULL;

2814         ASSERT(vd == vd->vdev_top);

2816         /*
2817          * Apply the inverse of the psize-to-asize (ie. RAID-Z) space-expansion
2818          * factor.  We must calculate this here and not at the root vdev
2819          * because the root vdev's psize-to-asize is simply the max of its
2820          * childrens', thus not accurate enough for us.
2821          */
2822         ASSERT((dspace_delta & (SPA_MINBLOCKSIZE-1)) == 0);
2823         ASSERT(vd->vdev_deflate_ratio != 0 || vd->vdev_isl2cache);
2824         dspace_delta = (dspace_delta >> SPA_MINBLOCKSHIFT) *
2825             vd->vdev_deflate_ratio;

2827         mutex_enter(&vd->vdev_stat_lock);
2828         vd->vdev_stat.vs_alloc += alloc_delta;
2829         vd->vdev_stat.vs_space += space_delta;
2830         vd->vdev_stat.vs_dspace += dspace_delta;
2831         mutex_exit(&vd->vdev_stat_lock);

2833         if (mc == spa_normal_class(spa)) {
2834                 mutex_enter(&rvd->vdev_stat_lock);
2835                 rvd->vdev_stat.vs_alloc += alloc_delta;
2836                 rvd->vdev_stat.vs_space += space_delta;
2837                 rvd->vdev_stat.vs_dspace += dspace_delta;
2838                 mutex_exit(&rvd->vdev_stat_lock);
2839         }

2841         if (mc != NULL) {
2842                 ASSERT(rvd == vd->vdev_parent);
2843                 ASSERT(vd->vdev_ms_count != 0);

2845                 metaslab_class_space_update(mc,
2846                     alloc_delta, defer_delta, space_delta, dspace_delta);
2847         }
2848 }

2850 /*
2851  * Mark a top-level vdev's config as dirty, placing it on the dirty list
2852  * so that it will be written out next time the vdev configuration is synced.
2853  * If the root vdev is specified (vdev_top == NULL), dirty all top-level vdevs.
2854  */
2855 void
2856 vdev_config_dirty(vdev_t *vd)
2857 {
2858         spa_t *spa = vd->vdev_spa;
2859         vdev_t *rvd = spa->spa_root_vdev;
2860         int c;

2862         ASSERT(spa_writeable(spa));

2864         /*
2865          * If this is an aux vdev (as with l2cache and spare devices), then we
2866          * update the vdev config manually and set the sync flag.
2867          */
2868         if (vd->vdev_aux != NULL) {
```



```
2869                 spa_aux_vdev_t *sav = vd->vdev_aux;
2870                 nvlist_t **aux;
2871                 uint_t naux;

2873                 for (c = 0; c < sav->sav_count; c++) {
2874                         if (sav->sav_vdevs[c] == vd)
2875                                 break;
2876                 }

2878                 if (c == sav->sav_count) {
2879                         /*
2880                          * We're being removed.  There's nothing more to do.
2881                          */
2882                         ASSERT(sav->sav_sync == B_TRUE);
2883                         return;
2884                 }

2886                 sav->sav_sync = B_TRUE;

2888                 if (nvlist_lookup_nvlist_array(sav->sav_config,
2889                     ZPOOL_CONFIG_L2CACHE, &aux, &naux) != 0) {
2890                         VERIFY(nvlist_lookup_nvlist_array(sav->sav_config,
2891                             ZPOOL_CONFIG_SPARES, &aux, &naux) == 0);
2892                 }

2894                 ASSERT(c < naux);

2896                 /*
2897                  * Setting the nvlist in the middle if the array is a little
2898                  * sketchy, but it will work.
2899                  */
2900                 nvlist_free(aux[c]);
2901                 aux[c] = vdev_config_generate(spa, vd, B_TRUE, 0);

2903                 return;
2904         }

2906         /*
2907          * The dirty list is protected by the SCL_CONFIG lock.  The caller
2908          * must either hold SCL_CONFIG as writer, or must be the sync thread
2909          * (which holds SCL_CONFIG as reader).  There's only one sync thread,
2910          * so this is sufficient to ensure mutual exclusion.
2911          */
2912         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_WRITER) ||
2913             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2914             spa_config_held(spa, SCL_CONFIG, RW_READER)));

2916         if (vd == rvd) {
2917                 for (c = 0; c < rvd->vdev_children; c++)
2918                         vdev_config_dirty(rvd->vdev_child[c]);
2919         } else {
2920                 ASSERT(vd == vd->vdev_top);

2922                 if (!list_link_active(&vd->vdev_config_dirty_node) &&
2923                     !vd->vdev_ishole)
2924                         list_insert_head(&spa->spa_config_dirty_list, vd);
2925         }
2926 }

2928 void
2929 vdev_config_clean(vdev_t *vd)
2930 {
2931         spa_t *spa = vd->vdev_spa;

2933         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_WRITER) ||
2934             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
```

```
2935             spa_config_held(spa, SCL_CONFIG, RW_READER)));

2937         ASSERT(list_link_active(&vd->vdev_config_dirty_node));
2938         list_remove(&spa->spa_config_dirty_list, vd);
2939 }

2941 /*
2942  * Mark a top-level vdev's state as dirty, so that the next pass of
2943  * spa_sync() can convert this into vdev_config_dirty().  We distinguish
2944  * the state changes from larger config changes because they require
2945  * much less locking, and are often needed for administrative actions.
2946  */
2947 void
2948 vdev_state_dirty(vdev_t *vd)
2949 {
2950         spa_t *spa = vd->vdev_spa;

2952         ASSERT(spa_writeable(spa));
2953         ASSERT(vd == vd->vdev_top);

2955         /*
2956          * The state list is protected by the SCL_STATE lock.  The caller
2957          * must either hold SCL_STATE as writer, or must be the sync thread
2958          * (which holds SCL_STATE as reader).  There's only one sync thread,
2959          * so this is sufficient to ensure mutual exclusion.
2960          */
2961         ASSERT(spa_config_held(spa, SCL_STATE, RW_WRITER) ||
2962             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2963             spa_config_held(spa, SCL_STATE, RW_READER)));

2965         if (!list_link_active(&vd->vdev_state_dirty_node) && !vd->vdev_ishole)
2966                 list_insert_head(&spa->spa_state_dirty_list, vd);
2967 }

2969 void
2970 vdev_state_clean(vdev_t *vd)
2971 {
2972         spa_t *spa = vd->vdev_spa;

2974         ASSERT(spa_config_held(spa, SCL_STATE, RW_WRITER) ||
2975             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2976             spa_config_held(spa, SCL_STATE, RW_READER)));

2978         ASSERT(list_link_active(&vd->vdev_state_dirty_node));
2979         list_remove(&spa->spa_state_dirty_list, vd);
2980 }

2982 /*
2983  * Propagate vdev state up from children to parent.
2984  */
2985 void
2986 vdev_propagate_state(vdev_t *vd)
2987 {
2988         spa_t *spa = vd->vdev_spa;
2989         vdev_t *rvd = spa->spa_root_vdev;
2990         int degraded = 0, faulted = 0;
2991         int corrupted = 0;
2992         vdev_t *child;

2994         if (vd->vdev_children > 0) {
2995                 for (int c = 0; c < vd->vdev_children; c++) {
2996                         child = vd->vdev_child[c];

2998                         /*
2999                          * Don't factor holes into the decision.
3000                          */
```



```
3001                         if (child->vdev_ishole)
3002                                 continue;

3004                         if (!vdev_readable(child) ||
3005                             (!vdev_writeable(child) && spa_writeable(spa))) {
3006                                 /*
3007                                  * Root special: if there is a top-level log
3008                                  * device, treat the root vdev as if it were
3009                                  * degraded.
3010                                  */
3011                                 if (child->vdev_islog && vd == rvd)
3012                                         degraded++;
3013                                 else
3014                                         faulted++;
3015                         } else if (child->vdev_state <= VDEV_STATE_DEGRADED) {
3016                                 degraded++;
3017                         }

3019                         if (child->vdev_stat.vs_aux == VDEV_AUX_CORRUPT_DATA)
3020                                 corrupted++;
3021                 }

3023                 vd->vdev_ops->vdev_op_state_change(vd, faulted, degraded);

3025                 /*
3026                  * Root special: if there is a top-level vdev that cannot be
3027                  * opened due to corrupted metadata, then propagate the root
3028                  * vdev's aux state as 'corrupt' rather than 'insufficient
3029                  * replicas'.
3030                  */
3031                 if (corrupted && vd == rvd &&
3032                     rvd->vdev_state == VDEV_STATE_CANT_OPEN)
3033                         vdev_set_state(rvd, B_FALSE, VDEV_STATE_CANT_OPEN,
3034                             VDEV_AUX_CORRUPT_DATA);
3035         }

3037         if (vd->vdev_parent)
3038                 vdev_propagate_state(vd->vdev_parent);
3039 }

3041 /*
3042  * Set a vdev's state.  If this is during an open, we don't update the parent
3043  * state, because we're in the process of opening children depth-first.
3044  * Otherwise, we propagate the change to the parent.
3045  *
3046  * If this routine places a device in a faulted state, an appropriate ereport is
3047  * generated.
3048  */
3049 void
3050 vdev_set_state(vdev_t *vd, boolean_t isopen, vdev_state_t state, vdev_aux_t aux)
3051 {
3052         uint64_t save_state;
3053         spa_t *spa = vd->vdev_spa;

3055         if (state == vd->vdev_state) {
3056                 vd->vdev_stat.vs_aux = aux;
3057                 return;
3058         }

3060         save_state = vd->vdev_state;

3062         vd->vdev_state = state;
3063         vd->vdev_stat.vs_aux = aux;

3065         /*
3066          * If we are setting the vdev state to anything but an open state, then
```

```
3067          * always close the underlying device unless the device has requested
3068          * a delayed close (i.e. we're about to remove or fault the device).
3069          * Otherwise, we keep accessible but invalid devices open forever.
3070          * We don't call vdev_close() itself, because that implies some extra
3071          * checks (offline, etc) that we don't want here.  This is limited to
3072          * leaf devices, because otherwise closing the device will affect other
3073          * children.
3074          */
3075         if (!vd->vdev_delayed_close && vdev_is_dead(vd) &&
3076             vd->vdev_ops->vdev_op_leaf)
3077                 vd->vdev_ops->vdev_op_close(vd);

3079         /*
3080          * If we have brought this vdev back into service, we need
3081          * to notify fmd so that it can gracefully repair any outstanding
3082          * cases due to a missing device.  We do this in all cases, even those
3083          * that probably don't correlate to a repaired fault.  This is sure to
3084          * catch all cases, and we let the zfs-retire agent sort it out.  If
3085          * this is a transient state it's OK, as the retire agent will
3086          * double-check the state of the vdev before repairing it.
3087          */
3088         if (state == VDEV_STATE_HEALTHY && vd->vdev_ops->vdev_op_leaf &&
3089             vd->vdev_prevstate != state)
3090                 zfs_post_state_change(spa, vd);

3092         if (vd->vdev_removed &&
3093             state == VDEV_STATE_CANT_OPEN &&
3094             (aux == VDEV_AUX_OPEN_FAILED || vd->vdev_checkremove)) {
3095                 /*
3096                  * If the previous state is set to VDEV_STATE_REMOVED, then this
3097                  * device was previously marked removed and someone attempted to
3098                  * reopen it.  If this failed due to a nonexistent device, then
3099                  * keep the device in the REMOVED state.  We also let this be if
3100                  * it is one of our special test online cases, which is only
3101                  * attempting to online the device and shouldn't generate an FMA
3102                  * fault.
3103                  */
3104                 vd->vdev_state = VDEV_STATE_REMOVED;
3105                 vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
3106         } else if (state == VDEV_STATE_REMOVED) {
3107                 vd->vdev_removed = B_TRUE;
3108         } else if (state == VDEV_STATE_CANT_OPEN) {
3109                 /*
3110                  * If we fail to open a vdev during an import or recovery, we
3111                  * mark it as "not available", which signifies that it was
3112                  * never there to begin with.  Failure to open such a device
3113                  * is not considered an error.
3114                  */
3115                 if ((spa_load_state(spa) == SPA_LOAD_IMPORT ||
3116                     spa_load_state(spa) == SPA_LOAD_RECOVER) &&
3117                     vd->vdev_ops->vdev_op_leaf)
3118                         vd->vdev_not_present = 1;

3120                 /*
3121                  * Post the appropriate ereport.  If the 'prevstate' field is
3122                  * set to something other than VDEV_STATE_UNKNOWN, it indicates
3123                  * that this is part of a vdev_reopen().  In this case, we don't
3124                  * want to post the ereport if the device was already in the
3125                  * CANT_OPEN state beforehand.
3126                  *
3127                  * If the 'checkremove' flag is set, then this is an attempt to
3128                  * online the device in response to an insertion event.  If we
3129                  * hit this case, then we have detected an insertion event for a
3130                  * faulted or offline device that wasn't in the removed state.
3131                  * In this scenario, we don't post an ereport because we are
3132                  * about to replace the device, or attempt an online with
```



```
3133                  * vdev_forcefault, which will generate the fault for us.
3134                  */
3135                 if ((vd->vdev_prevstate != state || vd->vdev_forcefault) &&
3136                     !vd->vdev_not_present && !vd->vdev_checkremove &&
3137                     vd != spa->spa_root_vdev) {
3138                         const char *class;

3140                         switch (aux) {
3141                         case VDEV_AUX_OPEN_FAILED:
3142                                 class = FM_EREPORT_ZFS_DEVICE_OPEN_FAILED;
3143                                 break;
3144                         case VDEV_AUX_CORRUPT_DATA:
3145                                 class = FM_EREPORT_ZFS_DEVICE_CORRUPT_DATA;
3146                                 break;
3147                         case VDEV_AUX_NO_REPLICAS:
3148                                 class = FM_EREPORT_ZFS_DEVICE_NO_REPLICAS;
3149                                 break;
3150                         case VDEV_AUX_BAD_GUID_SUM:
3151                                 class = FM_EREPORT_ZFS_DEVICE_BAD_GUID_SUM;
3152                                 break;
3153                         case VDEV_AUX_TOO_SMALL:
3154                                 class = FM_EREPORT_ZFS_DEVICE_TOO_SMALL;
3155                                 break;
3156                         case VDEV_AUX_BAD_LABEL:
3157                                 class = FM_EREPORT_ZFS_DEVICE_BAD_LABEL;
3158                                 break;
3159                         default:
3160                                 class = FM_EREPORT_ZFS_DEVICE_UNKNOWN;
3161                         }

3163                         zfs_ereport_post(class, spa, vd, NULL, save_state, 0);
3164                 }

3166                 /* Erase any notion of persistent removed state */
3167                 vd->vdev_removed = B_FALSE;
3168         } else {
3169                 vd->vdev_removed = B_FALSE;
3170         }

3172         if (!isopen && vd->vdev_parent)
3173                 vdev_propagate_state(vd->vdev_parent);
3174 }

3176 /*
3177  * Check the vdev configuration to ensure that it's capable of supporting
3178  * a root pool. Currently, we do not support RAID-Z or partial configuration.
3179  * In addition, only a single top-level vdev is allowed and none of the leaves
3180  * can be wholedisks.
3181  */
3182 boolean_t
3183 vdev_is_bootable(vdev_t *vd)
3184 {
3185         if (!vd->vdev_ops->vdev_op_leaf) {
3186                 char *vdev_type = vd->vdev_ops->vdev_op_type;

3188                 if (strcmp(vdev_type, VDEV_TYPE_ROOT) == 0 &&
3189                     vd->vdev_children > 1) {
3190                         return (B_FALSE);
3191                 } else if (strcmp(vdev_type, VDEV_TYPE_RAIDZ) == 0 ||
3192                     strcmp(vdev_type, VDEV_TYPE_MISSING) == 0) {
3193                         return (B_FALSE);
3194                 }
3195         } else if (vd->vdev_wholedisk == 1) {
3196                 return (B_FALSE);
3197         }
```

```
3199         for (int c = 0; c < vd->vdev_children; c++) {
3200                 if (!vdev_is_bootable(vd->vdev_child[c]))
3201                         return (B_FALSE);
3202         }
3203         return (B_TRUE);
3204 }

3206 /*
3207  * Load the state from the original vdev tree (ovd) which
3208  * we've retrieved from the MOS config object. If the original
3209  * vdev was offline or faulted then we transfer that state to the
3210  * device in the current vdev tree (nvd).
3211  */
3212 void
3213 vdev_load_log_state(vdev_t *nvd, vdev_t *ovd)
3214 {
3215         spa_t *spa = nvd->vdev_spa;

3217         ASSERT(nvd->vdev_top->vdev_islog);
3218         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
3219         ASSERT3U(nvd->vdev_guid, ==, ovd->vdev_guid);

3221         for (int c = 0; c < nvd->vdev_children; c++)
3222                 vdev_load_log_state(nvd->vdev_child[c], ovd->vdev_child[c]);

3224         if (nvd->vdev_ops->vdev_op_leaf) {
3225                 /*
3226                  * Restore the persistent vdev state
3227                  */
3228                 nvd->vdev_offline = ovd->vdev_offline;
3229                 nvd->vdev_faulted = ovd->vdev_faulted;
3230                 nvd->vdev_degraded = ovd->vdev_degraded;
3231                 nvd->vdev_removed = ovd->vdev_removed;
3232         }
3233 }

3235 /*
3236  * Determine if a log device has valid content.  If the vdev was
3237  * removed or faulted in the MOS config then we know that
3238  * the content on the log device has already been written to the pool.
3239  */
3240 boolean_t
3241 vdev_log_state_valid(vdev_t *vd)
3242 {
3243         if (vd->vdev_ops->vdev_op_leaf && !vd->vdev_faulted &&
3244             !vd->vdev_removed)
3245                 return (B_TRUE);

3247         for (int c = 0; c < vd->vdev_children; c++)
3248                 if (vdev_log_state_valid(vd->vdev_child[c]))
3249                         return (B_TRUE);

3251         return (B_FALSE);
3252 }

3254 /*
3255  * Expand a vdev if possible.
3256  */
3257 void
3258 vdev_expand(vdev_t *vd, uint64_t txg)
3259 {
3260         ASSERT(vd->vdev_top == vd);
3261         ASSERT(spa_config_held(vd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);

3263         if ((vd->vdev_asize >> vd->vdev_ms_shift) > vd->vdev_ms_count) {
3264                 VERIFY(vdev_metaslab_init(vd, txg) == 0);
```



```
3265                 vdev_config_dirty(vd);
3266         }
3267 }

3269 /*
3270  * Split a vdev.
3271  */
3272 void
3273 vdev_split(vdev_t *vd)
3274 {
3275         vdev_t *cvd, *pvd = vd->vdev_parent;

3277         vdev_remove_child(pvd, vd);
3278         vdev_compact_children(pvd);

3280         cvd = pvd->vdev_child[0];
3281         if (pvd->vdev_children == 1) {
3282                 vdev_remove_parent(cvd);
3283                 cvd->vdev_splitting = B_TRUE;
3284         }
3285         vdev_propagate_state(cvd);
3286 }

3288 void
3289 vdev_deadman(vdev_t *vd)
3290 {
3291         for (int c = 0; c < vd->vdev_children; c++) {
3292                 vdev_t *cvd = vd->vdev_child[c];

3294                 vdev_deadman(cvd);
3295         }

3297         if (vd->vdev_ops->vdev_op_leaf) {
3298                 vdev_queue_t *vq = &vd->vdev_queue;

3300                 mutex_enter(&vq->vq_lock);
3301                 if (avl_numnodes(&vq->vq_active_tree) > 0) {
3302                         spa_t *spa = vd->vdev_spa;
3303                         zio_t *fio;
3304                         uint64_t delta;

3306                         /*
3307                          * Look at the head of all the pending queues,
3308                          * if any I/O has been outstanding for longer than
3309                          * the spa_deadman_synctime we panic the system.
3310                          */
3311                         fio = avl_first(&vq->vq_active_tree);
3312                         delta = gethrtime() - fio->io_timestamp;
3313                         if (delta > spa_deadman_synctime(spa)) {
3314                                 zfs_dbgmsg("SLOW IO: zio timestamp %lluns, "
3315                                     "delta %lluns, last io %lluns",
3316                                     fio->io_timestamp, delta,
3317                                     vq->vq_io_complete_ts);
3318                                 fm_panic("I/O to pool '%s' appears to be "
3319                                     "hung.", spa_name(spa));
3320                         }
3321                 }
3322                 mutex_exit(&vq->vq_lock);
3323         }
3324 }
```

```
**********************************************************
   57829 Fri Oct 31 10:14:52 2014
new/usr/src/uts/common/fs/zfs/zil.c
5269 zfs: zpool import slow
While importing a pool all objsets are enumerated twice, once to check
the zil log chains and once to claim them. On pools with many datasets
this process might take a substantial amount of time.
Speed up the process by parallelizing it utilizing a taskq. The number
of parallel tasks is limited to 4 times the number of leaf vdevs.
**********************************************************
______unchanged_portion_omitted_
```

```
626 int
627 zil_claim(dsl_pool_t *dp, dsl_dataset_t *ds, void *txarg)
627 zil_claim(const char *osname, void *txarg)
628 {
629         dmu_tx_t *tx = txarg;
630         uint64_t first_txg = dmu_tx_get_txg(tx);
631         zilog_t *zilog;
632         zil_header_t *zh;
633         objset_t *os;
634         int error;

636         error = dmu_objset_own_obj(dp, ds->ds_object,
637             DMU_OST_ANY, B_FALSE, FTAG, &os);
636         error = dmu_objset_own(osname, DMU_OST_ANY, B_FALSE, FTAG, &os);
638         if (error != 0) {
639                 cmn_err(CE_WARN, "can't open objset %llu, error %d",
640                     (unsigned long long)ds->ds_object, error);
638                 cmn_err(CE_WARN, "can't open objset for %s", osname);
641                 return (0);
642         }

644         zilog = dmu_objset_zil(os);
645         zh = zil_header_in_syncing_context(zilog);

647         if (spa_get_log_state(zilog->zl_spa) == SPA_LOG_CLEAR) {
648                 if (!BP_IS_HOLE(&zh->zh_log))
649                         zio_free_zil(zilog->zl_spa, first_txg, &zh->zh_log);
650                 BP_ZERO(&zh->zh_log);
651                 dsl_dataset_dirty(dmu_objset_ds(os), tx);
652                 dmu_objset_disown(os, FTAG);
653                 return (0);
654         }

656         /*
657          * Claim all log blocks if we haven't already done so, and remember
658          * the highest claimed sequence number.  This ensures that if we can
659          * read only part of the log now (e.g. due to a missing device),
660          * but we can read the entire log later, we will not try to replay
661          * or destroy beyond the last block we successfully claimed.
662          */
663         ASSERT3U(zh->zh_claim_txg, <=, first_txg);
664         if (zh->zh_claim_txg == 0 && !BP_IS_HOLE(&zh->zh_log)) {
665                 (void) zil_parse(zilog, zil_claim_log_block,
666                     zil_claim_log_record, tx, first_txg);
667                 zh->zh_claim_txg = first_txg;
668                 zh->zh_claim_blk_seq = zilog->zl_parse_blk_seq;
669                 zh->zh_claim_lr_seq = zilog->zl_parse_lr_seq;
670                 if (zilog->zl_parse_lr_count || zilog->zl_parse_blk_count > 1)
671                         zh->zh_flags |= ZIL_REPLAY_NEEDED;
672                 zh->zh_flags |= ZIL_CLAIM_LR_SEQ_VALID;
673                 dsl_dataset_dirty(dmu_objset_ds(os), tx);
674         }

676         ASSERT3U(first_txg, ==, (spa_last_synced_txg(zilog->zl_spa) + 1));
```



```
677         dmu_objset_disown(os, FTAG);
678         return (0);
679 }

681 /*
682  * Check the log by walking the log chain.
683  * Checksum errors are ok as they indicate the end of the chain.
684  * Any other error (no device or read failure) returns an error.
685  */
686 int
687 zil_check_log_chain(dsl_pool_t *dp, dsl_dataset_t *ds, void *tx)
685 zil_check_log_chain(const char *osname, void *tx)
688 {
689         zilog_t *zilog;
690         objset_t *os;
691         blkptr_t *bp;
692         int error;

694         ASSERT(tx == NULL);

696         error = dmu_objset_from_ds(ds, &os);
694         error = dmu_objset_hold(osname, FTAG, &os);
697         if (error != 0) {
698                 cmn_err(CE_WARN, "can't open objset %llu, error %d",
699                     (unsigned long long)ds->ds_object, error);
696                 cmn_err(CE_WARN, "can't open objset for %s", osname);
700                 return (0);
701         }

703         zilog = dmu_objset_zil(os);
704         bp = (blkptr_t *)&zilog->zl_header->zh_log;

706         /*
707          * Check the first block and determine if it's on a log device
708          * which may have been removed or faulted prior to loading this
709          * pool.  If so, there's no point in checking the rest of the log
710          * as its content should have already been synced to the pool.
711          */
712         if (!BP_IS_HOLE(bp)) {
713                 vdev_t *vd;
714                 boolean_t valid = B_TRUE;

716                 spa_config_enter(os->os_spa, SCL_STATE, FTAG, RW_READER);
717                 vd = vdev_lookup_top(os->os_spa, DVA_GET_VDEV(&bp->blk_dva[0]));
718                 if (vd->vdev_islog && vdev_is_dead(vd))
719                         valid = vdev_log_state_valid(vd);
720                 spa_config_exit(os->os_spa, SCL_STATE, FTAG);

722                 if (!valid)
719                 if (!valid) {
720                         dmu_objset_rele(os, FTAG);
723                         return (0);
724         }
723         }

726         /*
727          * Because tx == NULL, zil_claim_log_block() will not actually claim
728          * any blocks, but just determine whether it is possible to do so.
729          * In addition to checking the log chain, zil_claim_log_block()
730          * will invoke zio_claim() with a done func of spa_claim_notify(),
731          * which will update spa_max_claim_txg.  See spa_load() for details.
732          */
733         error = zil_parse(zilog, zil_claim_log_block, zil_claim_log_record, tx,
734             zilog->zl_header->zh_claim_txg ? -1ULL : spa_first_txg(os->os_spa));

735         dmu_objset_rele(os, FTAG);
```

```
 736         return ((error == ECKSUM || error == ENOENT) ? 0 : error);
 737 }
```

unchanged_portion_omitted_