```
**********************************************************
   48886 Thu Oct 16 19:15:50 2014
new/usr/src/uts/common/fs/zfs/dmu_objset.c
zpool import speedup
**********************************************************
```

```
/*
 * 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.
 */

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

zfs_logbias_op_t
dmu_objset_logbias(objset_t *os)
```

```
128 {
129         return (os->os_logbias);
130 }

132 static void
133 checksum_changed_cb(void *arg, uint64_t newval)
134 {
135         objset_t *os = arg;

137         /*
138          * Inheritance should have been done by now.
139          */
140         ASSERT(newval != ZIO_CHECKSUM_INHERIT);

142         os->os_checksum = zio_checksum_select(newval, ZIO_CHECKSUM_ON_VALUE);
143 }

145 static void
146 compression_changed_cb(void *arg, uint64_t newval)
147 {
148         objset_t *os = arg;

150         /*
151          * Inheritance and range checking should have been done by now.
152          */
153         ASSERT(newval != ZIO_COMPRESS_INHERIT);

155         os->os_compress = zio_compress_select(newval, ZIO_COMPRESS_ON_VALUE);
156 }

158 static void
159 copies_changed_cb(void *arg, uint64_t newval)
160 {
161         objset_t *os = arg;

163         /*
164          * Inheritance and range checking should have been done by now.
165          */
166         ASSERT(newval > 0);
167         ASSERT(newval <= spa_max_replication(os->os_spa));

169         os->os_copies = newval;
170 }

172 static void
173 dedup_changed_cb(void *arg, uint64_t newval)
174 {
175         objset_t *os = arg;
176         spa_t *spa = os->os_spa;
177         enum zio_checksum checksum;

179         /*
180          * Inheritance should have been done by now.
181          */
182         ASSERT(newval != ZIO_CHECKSUM_INHERIT);

184         checksum = zio_checksum_dedup_select(spa, newval, ZIO_CHECKSUM_OFF);

186         os->os_dedup_checksum = checksum & ZIO_CHECKSUM_MASK;
187         os->os_dedup_verify = !!(checksum & ZIO_CHECKSUM_VERIFY);
188 }

190 static void
191 primary_cache_changed_cb(void *arg, uint64_t newval)
192 {
193         objset_t *os = arg;
```



```
195         /*
196          * Inheritance and range checking should have been done by now.
197          */
198         ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
199             newval == ZFS_CACHE_METADATA);

201         os->os_primary_cache = newval;
202 }

204 static void
205 secondary_cache_changed_cb(void *arg, uint64_t newval)
206 {
207         objset_t *os = arg;

209         /*
210          * Inheritance and range checking should have been done by now.
211          */
212         ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
213             newval == ZFS_CACHE_METADATA);

215         os->os_secondary_cache = newval;
216 }

218 static void
219 sync_changed_cb(void *arg, uint64_t newval)
220 {
221         objset_t *os = arg;

223         /*
224          * Inheritance and range checking should have been done by now.
225          */
226         ASSERT(newval == ZFS_SYNC_STANDARD || newval == ZFS_SYNC_ALWAYS ||
227             newval == ZFS_SYNC_DISABLED);

229         os->os_sync = newval;
230         if (os->os_zil)
231                 zil_set_sync(os->os_zil, newval);
232 }

234 static void
235 redundant_metadata_changed_cb(void *arg, uint64_t newval)
236 {
237         objset_t *os = arg;

239         /*
240          * Inheritance and range checking should have been done by now.
241          */
242         ASSERT(newval == ZFS_REDUNDANT_METADATA_ALL ||
243             newval == ZFS_REDUNDANT_METADATA_MOST);

245         os->os_redundant_metadata = newval;
246 }

248 static void
249 logbias_changed_cb(void *arg, uint64_t newval)
250 {
251         objset_t *os = arg;

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

```
260 void
261 dmu_objset_byteswap(void *buf, size_t size)
262 {
263         objset_phys_t *osp = buf;

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

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

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

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

295                 if (DMU_OS_IS_L2CACHEABLE(os))
296                         aflags |= ARC_L2CACHE;
297                 if (DMU_OS_IS_L2COMPRESSIBLE(os))
298                         aflags |= ARC_L2COMPRESS;

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

312                 /* Increase the blocksize if we are permitted. */
313                 if (spa_version(spa) >= SPA_VERSION_USERSPACE &&
314                     arc_buf_size(os->os_phys_buf) < sizeof (objset_phys_t)) {
315                         arc_buf_t *buf = arc_buf_alloc(spa,
316                             sizeof (objset_phys_t), &os->os_phys_buf,
317                             ARC_BUFC_METADATA);
318                         bzero(buf->b_data, sizeof (objset_phys_t));
319                         bcopy(os->os_phys_buf->b_data, buf->b_data,
320                             arc_buf_size(os->os_phys_buf));
321                         (void) arc_buf_remove_ref(os->os_phys_buf,
322                             &os->os_phys_buf);
323                         os->os_phys_buf = buf;
324                 }
```



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

337         /*
338          * Note: the changed_cb will be called once before the register
339          * func returns, thus changing the checksum/compression from the
340          * default (fletcher2/off).  Snapshots don't need to know about
341          * checksum/compression/copies.
342          */
343         if (ds != NULL) {
344                 err = dsl_prop_register(ds,
345                     zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
346                     primary_cache_changed_cb, os);
347                 if (err == 0) {
348                         err = dsl_prop_register(ds,
349                             zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
350                             secondary_cache_changed_cb, os);
351                 }
352                 if (!dsl_dataset_is_snapshot(ds)) {
353                         if (err == 0) {
354                                 err = dsl_prop_register(ds,
355                                     zfs_prop_to_name(ZFS_PROP_CHECKSUM),
356                                     checksum_changed_cb, os);
357                         }
358                         if (err == 0) {
359                                 err = dsl_prop_register(ds,
360                                     zfs_prop_to_name(ZFS_PROP_COMPRESSION),
361                                     compression_changed_cb, os);
362                         }
363                         if (err == 0) {
364                                 err = dsl_prop_register(ds,
365                                     zfs_prop_to_name(ZFS_PROP_COPIES),
366                                     copies_changed_cb, os);
367                         }
368                         if (err == 0) {
369                                 err = dsl_prop_register(ds,
370                                     zfs_prop_to_name(ZFS_PROP_DEDUP),
371                                     dedup_changed_cb, os);
372                         }
373                         if (err == 0) {
374                                 err = dsl_prop_register(ds,
375                                     zfs_prop_to_name(ZFS_PROP_LOGBIAS),
376                                     logbias_changed_cb, os);
377                         }
378                         if (err == 0) {
379                                 err = dsl_prop_register(ds,
380                                     zfs_prop_to_name(ZFS_PROP_SYNC),
381                                     sync_changed_cb, os);
382                         }
383                         if (err == 0) {
384                                 err = dsl_prop_register(ds,
385                                     zfs_prop_to_name(
386                                     ZFS_PROP_REDUNDANT_METADATA),
387                                     redundant_metadata_changed_cb, os);
388                         }
389                 }
390                 if (err != 0) {
391                         VERIFY(arc_buf_remove_ref(os->os_phys_buf,
```

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

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

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

424         mutex_init(&os->os_lock, NULL, MUTEX_DEFAULT, NULL);
425         mutex_init(&os->os_obj_lock, NULL, MUTEX_DEFAULT, NULL);
426         mutex_init(&os->os_user_ptr_lock, NULL, MUTEX_DEFAULT, NULL);

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

440         *osp = os;
441         return (0);
442 }

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

449         mutex_enter(&ds->ds_opening_lock);
450         if (ds->ds_objset == NULL) {
451                 objset_t *os;
452                 err = dmu_objset_open_impl(dsl_dataset_get_spa(ds),
453                     ds, dsl_dataset_get_blkptr(ds), &os);

455                 if (err == 0) {
456                         mutex_enter(&ds->ds_lock);
457                         ASSERT(ds->ds_objset == NULL);
```



```
458                         ds->ds_objset = os;
459                         mutex_exit(&ds->ds_lock);
460                 }
461         }
462         *osp = ds->ds_objset;
463         mutex_exit(&ds->ds_opening_lock);
464         return (err);
465 }

467 /*
468  * Holds the pool while the objset is held.  Therefore only one objset
469  * can be held at a time.
470  */
471 static int
472 dmu_objset_hold_impl(const char *name, void *tag, objset_t **osp, int lock)
 50 int
 51 dmu_objset_hold(const char *name, void *tag, objset_t **osp)
473 {
474         dsl_pool_t *dp;
475         dsl_dataset_t *ds;
476         int err;

478         err = dsl_pool_hold_lock(name, tag, &dp, lock);
 57         err = dsl_pool_hold(name, tag, &dp);
479         if (err != 0)
480                 return (err);
481         err = dsl_dataset_hold(dp, name, tag, &ds);
482         if (err != 0) {
483                 dsl_pool_rele(dp, tag);
484                 return (err);
485         }

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

493         return (err);
494 }

496 int
497 dmu_objset_hold(const char *name, void *tag, objset_t **osp)
498 {
499         return (dmu_objset_hold_impl(name, tag, osp, 1));
500 }

502 int
503 dmu_objset_hold_nolock(const char *name, void *tag, objset_t **osp)
504 {
505         return (dmu_objset_hold_impl(name, tag, osp, 0));
506 }

508 #endif /* ! codereview */
509 /*
510  * dsl_pool must not be held when this is called.
511  * Upon successful return, there will be a longhold on the dataset,
512  * and the dsl_pool will not be held.
513  */
514 static int
515 dmu_objset_own_impl(const char *name, dmu_objset_type_t type,
516     boolean_t readonly, void *tag, objset_t **osp, int lock)
 75 int
 76 dmu_objset_own(const char *name, dmu_objset_type_t type,
 77     boolean_t readonly, void *tag, objset_t **osp)
517 {
```

```
518         dsl_pool_t *dp;
519         dsl_dataset_t *ds;
520         int err;

522         err = dsl_pool_hold_lock(name, FTAG, &dp, lock);
 83         err = dsl_pool_hold(name, FTAG, &dp);
523         if (err != 0)
524                 return (err);
525         err = dsl_dataset_own(dp, name, tag, &ds);
526         if (err != 0) {
527                 dsl_pool_rele(dp, FTAG);
528                 return (err);
529         }

531         err = dmu_objset_from_ds(ds, osp);
532         dsl_pool_rele(dp, FTAG);
533         if (err != 0) {
534                 dsl_dataset_disown(ds, tag);
535         } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
536                 dsl_dataset_disown(ds, tag);
537                 return (SET_ERROR(EINVAL));
538         } else if (!readonly && dsl_dataset_is_snapshot(ds)) {
539                 dsl_dataset_disown(ds, tag);
540                 return (SET_ERROR(EROFS));
541         }
542         return (err);
543 }

545 int
546 dmu_objset_own(const char *name, dmu_objset_type_t type,
547     boolean_t readonly, void *tag, objset_t **osp)
548 {
549         return (dmu_objset_own_impl(name, type, readonly, tag, osp, 1));
550 }

552 int
553 dmu_objset_own_nolock(const char *name, dmu_objset_type_t type,
554     boolean_t readonly, void *tag, objset_t **osp)
555 {
556         return (dmu_objset_own_impl(name, type, readonly, tag, osp, 0));
557 }

559 #endif /* ! codereview */
560 void
561 dmu_objset_rele(objset_t *os, void *tag)
562 {
563         dsl_pool_t *dp = dmu_objset_pool(os);
564         dsl_dataset_rele(os->os_dsl_dataset, tag);
565         dsl_pool_rele(dp, tag);
566 }

568 /*
569  * When we are called, os MUST refer to an objset associated with a dataset
570  * that is owned by 'tag'; that is, is held and long held by 'tag' and ds_owner
571  * == tag.  We will then release and reacquire ownership of the dataset while
572  * holding the pool config_rwlock to avoid intervening namespace or ownership
573  * changes may occur.
574  *
575  * This exists solely to accommodate zfs_ioc_userspace_upgrade()'s desire to
576  * release the hold on its dataset and acquire a new one on the dataset of the
577  * same name so that it can be partially torn down and reconstructed.
578  */
579 void
580 dmu_objset_refresh_ownership(objset_t *os, void *tag)
581 {
582         dsl_pool_t *dp;
```



```
583         dsl_dataset_t *ds, *newds;
584         char name[MAXNAMELEN];

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

591         dsl_dataset_name(ds, name);
592         dp = dmu_objset_pool(os);
593         dsl_pool_config_enter(dp, FTAG);
594         dmu_objset_disown(os, tag);
595         VERIFY0(dsl_dataset_own(dp, name, tag, &newds));
596         VERIFY3P(newds, ==, os->os_dsl_dataset);
597         dsl_pool_config_exit(dp, FTAG);
598 }

600 void
601 dmu_objset_disown(objset_t *os, void *tag)
602 {
603         dsl_dataset_disown(os->os_dsl_dataset, tag);
604 }

606 void
607 dmu_objset_evict_dbufs(objset_t *os)
608 {
609         dnode_t *dn;

611         mutex_enter(&os->os_lock);

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

617         /*
618          * Find the first dnode with holds.  We have to do this dance
619          * because dnode_add_ref() only works if you already have a
620          * hold.  If there are no holds then it has no dbufs so OK to
621          * skip.
622          */
623         for (dn = list_head(&os->os_dnodes);
624             dn && !dnode_add_ref(dn, FTAG);
625             dn = list_next(&os->os_dnodes, dn))
626                 continue;

628         while (dn) {
629                 dnode_t *next_dn = dn;

631                 do {
632                         next_dn = list_next(&os->os_dnodes, next_dn);
633                 } while (next_dn && !dnode_add_ref(next_dn, FTAG));

635                 mutex_exit(&os->os_lock);
636                 dnode_evict_dbufs(dn);
637                 dnode_rele(dn, FTAG);
638                 mutex_enter(&os->os_lock);
639                 dn = next_dn;
640         }
641         mutex_exit(&os->os_lock);
642 }

644 void
645 dmu_objset_evict(objset_t *os)
646 {
647         dsl_dataset_t *ds = os->os_dsl_dataset;
```

```
649         for (int t = 0; t < TXG_SIZE; t++)
650                 ASSERT(!dmu_objset_is_dirty(os, t));
651
652         if (ds) {
653                 if (!dsl_dataset_is_snapshot(ds)) {
654                         VERIFY0(dsl_prop_unregister(ds,
655                             zfs_prop_to_name(ZFS_PROP_CHECKSUM),
656                             checksum_changed_cb, os));
657                         VERIFY0(dsl_prop_unregister(ds,
658                             zfs_prop_to_name(ZFS_PROP_COMPRESSION),
659                             compression_changed_cb, os));
660                         VERIFY0(dsl_prop_unregister(ds,
661                             zfs_prop_to_name(ZFS_PROP_COPIES),
662                             copies_changed_cb, os));
663                         VERIFY0(dsl_prop_unregister(ds,
664                             zfs_prop_to_name(ZFS_PROP_DEDUP),
665                             dedup_changed_cb, os));
666                         VERIFY0(dsl_prop_unregister(ds,
667                             zfs_prop_to_name(ZFS_PROP_LOGBIAS),
668                             logbias_changed_cb, os));
669                         VERIFY0(dsl_prop_unregister(ds,
670                             zfs_prop_to_name(ZFS_PROP_SYNC),
671                             sync_changed_cb, os));
672                         VERIFY0(dsl_prop_unregister(ds,
673                             zfs_prop_to_name(ZFS_PROP_REDUNDANT_METADATA),
674                             redundant_metadata_changed_cb, os));
675                 }
676                 VERIFY0(dsl_prop_unregister(ds,
677                     zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
678                     primary_cache_changed_cb, os));
679                 VERIFY0(dsl_prop_unregister(ds,
680                     zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
681                     secondary_cache_changed_cb, os));
682         }
683
684         if (os->os_sa)
685                 sa_tear_down(os);
686
687         dmu_objset_evict_dbufs(os);
688
689         dnode_special_close(&os->os_meta_dnode);
690         if (DMU_USERUSED_DNODE(os)) {
691                 dnode_special_close(&os->os_userused_dnode);
692                 dnode_special_close(&os->os_groupused_dnode);
693         }
694         zil_free(os->os_zil);
695
696         ASSERT3P(list_head(&os->os_dnodes), ==, NULL);
697
698         VERIFY(arc_buf_remove_ref(os->os_phys_buf, &os->os_phys_buf));
699
700         /*
701          * This is a barrier to prevent the objset from going away in
702          * dnode_move() until we can safely ensure that the objset is still in
703          * use. We consider the objset valid before the barrier and invalid
704          * after the barrier.
705          */
706         rw_enter(&os_lock, RW_READER);
707         rw_exit(&os_lock);
708
709         mutex_destroy(&os->os_lock);
710         mutex_destroy(&os->os_obj_lock);
711         mutex_destroy(&os->os_user_ptr_lock);
712         kmem_free(os, sizeof (objset_t));
713 }
```



```
715 timestruc_t
716 dmu_objset_snap_cmtime(objset_t *os)
717 {
718         return (dsl_dir_snap_cmtime(os->os_dsl_dataset->ds_dir));
719 }
720
721 /* called from dsl for meta-objset */
722 objset_t *
723 dmu_objset_create_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
724     dmu_objset_type_t type, dmu_tx_t *tx)
725 {
726         objset_t *os;
727         dnode_t *mdn;
728
729         ASSERT(dmu_tx_is_syncing(tx));
730
731         if (ds != NULL)
732                 VERIFY0(dmu_objset_from_ds(ds, &os));
733         else
734                 VERIFY0(dmu_objset_open_impl(spa, NULL, bp, &os));
735
736         mdn = DMU_META_DNODE(os);
737
738         dnode_allocate(mdn, DMU_OT_DNODE, 1 << DNODE_BLOCK_SHIFT,
739             DN_MAX_INDBLKSHIFT, DMU_OT_NONE, 0, tx);
740
741         /*
742          * We don't want to have to increase the meta-dnode's nlevels
743          * later, because then we could do it in quescing context while
744          * we are also accessing it in open context.
745          *
746          * This precaution is not necessary for the MOS (ds == NULL),
747          * because the MOS is only updated in syncing context.
748          * This is most fortunate: the MOS is the only objset that
749          * needs to be synced multiple times as spa_sync() iterates
750          * to convergence, so minimizing its dn_nlevels matters.
751          */
752         if (ds != NULL) {
753                 int levels = 1;
754
755                 /*
756                  * Determine the number of levels necessary for the meta-dnode
757                  * to contain DN_MAX_OBJECT dnodes.
758                  */
759                 while ((uint64_t)mdn->dn_nblkptr << (mdn->dn_datablkshift +
760                     (levels - 1) * (mdn->dn_indblkshift - SPA_BLKPTRSHIFT)) <
761                     DN_MAX_OBJECT * sizeof (dnode_phys_t))
762                         levels++;
763
764                 mdn->dn_next_nlevels[tx->tx_txg & TXG_MASK] =
765                     mdn->dn_nlevels = levels;
766         }
767
768         ASSERT(type != DMU_OST_NONE);
769         ASSERT(type != DMU_OST_ANY);
770         ASSERT(type < DMU_OST_NUMTYPES);
771         os->os_phys->os_type = type;
772         if (dmu_objset_userused_enabled(os)) {
773                 os->os_phys->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
774                 os->os_flags = os->os_phys->os_flags;
775         }
776
777         dsl_dataset_dirty(ds, tx);
778
779         return (os);
780 }
```

```
782 typedef struct dmu_objset_create_arg {
783         const char *doca_name;
784         cred_t *doca_cred;
785         void (*doca_userfunc)(objset_t *os, void *arg,
786             cred_t *cr, dmu_tx_t *tx);
787         void *doca_userarg;
788         dmu_objset_type_t doca_type;
789         uint64_t doca_flags;
790 } dmu_objset_create_arg_t;

792 /*ARGSUSED*/
793 static int
794 dmu_objset_create_check(void *arg, dmu_tx_t *tx)
795 {
796         dmu_objset_create_arg_t *doca = arg;
797         dsl_pool_t *dp = dmu_tx_pool(tx);
798         dsl_dir_t *pdd;
799         const char *tail;
800         int error;

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

805         error = dsl_dir_hold(dp, doca->doca_name, FTAG, &pdd, &tail);
806         if (error != 0)
807                 return (error);
808         if (tail == NULL) {
809                 dsl_dir_rele(pdd, FTAG);
810                 return (SET_ERROR(EEXIST));
811         }
812         error = dsl_fs_ss_limit_check(pdd, 1, ZFS_PROP_FILESYSTEM_LIMIT, NULL,
813             doca->doca_cred);
814         dsl_dir_rele(pdd, FTAG);

816         return (error);
817 }

819 static void
820 dmu_objset_create_sync(void *arg, dmu_tx_t *tx)
821 {
822         dmu_objset_create_arg_t *doca = arg;
823         dsl_pool_t *dp = dmu_tx_pool(tx);
824         dsl_dir_t *pdd;
825         const char *tail;
826         dsl_dataset_t *ds;
827         uint64_t obj;
828         blkptr_t *bp;
829         objset_t *os;

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

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

836         VERIFY0(dsl_dataset_hold_obj(pdd->dd_pool, obj, FTAG, &ds));
837         bp = dsl_dataset_get_blkptr(ds);
838         os = dmu_objset_create_impl(pdd->dd_pool->dp_spa,
839             ds, bp, doca->doca_type, tx);

841         if (doca->doca_userfunc != NULL) {
842                 doca->doca_userfunc(os, doca->doca_userarg,
843                     doca->doca_cred, tx);
844         }

846         spa_history_log_internal_ds(ds, "create", tx, "");
```



```
847         dsl_dataset_rele(ds, FTAG);
848         dsl_dir_rele(pdd, FTAG);
849 }

851 int
852 dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
853     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg)
854 {
855         dmu_objset_create_arg_t doca;

857         doca.doca_name = name;
858         doca.doca_cred = CRED();
859         doca.doca_flags = flags;
860         doca.doca_userfunc = func;
861         doca.doca_userarg = arg;
862         doca.doca_type = type;

864         return (dsl_sync_task(name,
865             dmu_objset_create_check, dmu_objset_create_sync, &doca, 5));
866 }

868 typedef struct dmu_objset_clone_arg {
869         const char *doca_clone;
870         const char *doca_origin;
871         cred_t *doca_cred;
872 } dmu_objset_clone_arg_t;

874 /*ARGSUSED*/
875 static int
876 dmu_objset_clone_check(void *arg, dmu_tx_t *tx)
877 {
878         dmu_objset_clone_arg_t *doca = arg;
879         dsl_dir_t *pdd;
880         const char *tail;
881         int error;
882         dsl_dataset_t *origin;
883         dsl_pool_t *dp = dmu_tx_pool(tx);

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

888         error = dsl_dir_hold(dp, doca->doca_clone, FTAG, &pdd, &tail);
889         if (error != 0)
890                 return (error);
891         if (tail == NULL) {
892                 dsl_dir_rele(pdd, FTAG);
893                 return (SET_ERROR(EEXIST));
894         }
895         /* You can't clone across pools. */
896         if (pdd->dd_pool != dp) {
897                 dsl_dir_rele(pdd, FTAG);
898                 return (SET_ERROR(EXDEV));
899         }
900         error = dsl_fs_ss_limit_check(pdd, 1, ZFS_PROP_FILESYSTEM_LIMIT, NULL,
901             doca->doca_cred);
902         if (error != 0) {
903                 dsl_dir_rele(pdd, FTAG);
904                 return (SET_ERROR(EDQUOT));
905         }
906         dsl_dir_rele(pdd, FTAG);

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

912         /* You can't clone across pools. */
```

```
913         if (origin->ds_dir->dd_pool != dp) {
914                 dsl_dataset_rele(origin, FTAG);
915                 return (SET_ERROR(EXDEV));
916         }
917
918         /* You can only clone snapshots, not the head datasets. */
919         if (!dsl_dataset_is_snapshot(origin)) {
920                 dsl_dataset_rele(origin, FTAG);
921                 return (SET_ERROR(EINVAL));
922         }
923         dsl_dataset_rele(origin, FTAG);
924
925         return (0);
926 }
927
928 static void
929 dmu_objset_clone_sync(void *arg, dmu_tx_t *tx)
930 {
931         dmu_objset_clone_arg_t *doca = arg;
932         dsl_pool_t *dp = dmu_tx_pool(tx);
933         dsl_dir_t *pdd;
934         const char *tail;
935         dsl_dataset_t *origin, *ds;
936         uint64_t obj;
937         char namebuf[MAXNAMELEN];
938
939         VERIFY0(dsl_dir_hold(dp, doca->doca_clone, FTAG, &pdd, &tail));
940         VERIFY0(dsl_dataset_hold(dp, doca->doca_origin, FTAG, &origin));
941
942         obj = dsl_dataset_create_sync(pdd, tail, origin, 0,
943             doca->doca_cred, tx);
944
945         VERIFY0(dsl_dataset_hold_obj(pdd->dd_pool, obj, FTAG, &ds));
946         dsl_dataset_name(origin, namebuf);
947         spa_history_log_internal_ds(ds, "clone", tx,
948             "origin=%s (%llu)", namebuf, origin->ds_object);
949         dsl_dataset_rele(ds, FTAG);
950         dsl_dataset_rele(origin, FTAG);
951         dsl_dir_rele(pdd, FTAG);
952 }
953
954 int
955 dmu_objset_clone(const char *clone, const char *origin)
956 {
957         dmu_objset_clone_arg_t doca;
958
959         doca.doca_clone = clone;
960         doca.doca_origin = origin;
961         doca.doca_cred = CRED();
962
963         return (dsl_sync_task(clone,
964             dmu_objset_clone_check, dmu_objset_clone_sync, &doca, 5));
965 }
966
967 int
968 dmu_objset_snapshot_one(const char *fsname, const char *snapname)
969 {
970         int err;
971         char *longsnap = kmem_asprintf("%s@%s", fsname, snapname);
972         nvlist_t *snaps = fnvlist_alloc();
973
974         fnvlist_add_boolean(snaps, longsnap);
975         strfree(longsnap);
976         err = dsl_dataset_snapshot(snaps, NULL, NULL);
977         fnvlist_free(snaps);
978         return (err);
```



```
979 }
980
981 static void
982 dmu_objset_sync_dnodes(list_t *list, list_t *newlist, dmu_tx_t *tx)
983 {
984         dnode_t *dn;
985
986         while (dn = list_head(list)) {
987                 ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
988                 ASSERT(dn->dn_dbuf->db_data_pending);
989                 /*
990                  * Initialize dn_zio outside dnode_sync() because the
991                  * meta-dnode needs to set it ouside dnode_sync().
992                  */
993                 dn->dn_zio = dn->dn_dbuf->db_data_pending->dr_zio;
994                 ASSERT(dn->dn_zio);
995
996                 ASSERT3U(dn->dn_nlevels, <=, DN_MAX_LEVELS);
997                 list_remove(list, dn);
998
999                 if (newlist) {
1000                         (void) dnode_add_ref(dn, newlist);
1001                         list_insert_tail(newlist, dn);
1002                 }
1003
1004                 dnode_sync(dn, tx);
1005         }
1006 }
1007
1008 /* ARGSUSED */
1009 static void
1010 dmu_objset_write_ready(zio_t *zio, arc_buf_t *abuf, void *arg)
1011 {
1012         blkptr_t *bp = zio->io_bp;
1013         objset_t *os = arg;
1014         dnode_phys_t *dnp = &os->os_phys->os_meta_dnode;
1015
1016         ASSERT(!BP_IS_EMBEDDED(bp));
1017         ASSERT3P(bp, ==, os->os_rootbp);
1018         ASSERT3U(BP_GET_TYPE(bp), ==, DMU_OT_OBJSET);
1019         ASSERT0(BP_GET_LEVEL(bp));
1020
1021         /*
1022          * Update rootbp fill count: it should be the number of objects
1023          * allocated in the object set (not counting the "special"
1024          * objects that are stored in the objset_phys_t -- the meta
1025          * dnode and user/group accounting objects).
1026          */
1027         bp->blk_fill = 0;
1028         for (int i = 0; i < dnp->dn_nblkptr; i++)
1029                 bp->blk_fill += BP_GET_FILL(&dnp->dn_blkptr[i]);
1030 }
1031
1032 /* ARGSUSED */
1033 static void
1034 dmu_objset_write_done(zio_t *zio, arc_buf_t *abuf, void *arg)
1035 {
1036         blkptr_t *bp = zio->io_bp;
1037         blkptr_t *bp_orig = &zio->io_bp_orig;
1038         objset_t *os = arg;
1039
1040         if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {
1041                 ASSERT(BP_EQUAL(bp, bp_orig));
1042         } else {
1043                 dsl_dataset_t *ds = os->os_dsl_dataset;
1044                 dmu_tx_t *tx = os->os_synctx;
```

```
1046                 (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
1047                 dsl_dataset_block_born(ds, bp, tx);
1048         }
1049 }

1051 /* called from dsl */
1052 void
1053 dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
1054 {
1055         int txgoff;
1056         zbookmark_t zb;
1057         zio_prop_t zp;
1058         zio_t *zio;
1059         list_t *list;
1060         list_t *newlist = NULL;
1061         dbuf_dirty_record_t *dr;

1063         dprintf_ds(os->os_dsl_dataset, "txg=%llu\n", tx->tx_txg);

1065         ASSERT(dmu_tx_is_syncing(tx));
1066         /* XXX the write_done callback should really give us the tx... */
1067         os->os_synctx = tx;

1069         if (os->os_dsl_dataset == NULL) {
1070                 /*
1071                  * This is the MOS.  If we have upgraded,
1072                  * spa_max_replication() could change, so reset
1073                  * os_copies here.
1074                  */
1075                 os->os_copies = spa_max_replication(os->os_spa);
1076         }

1078         /*
1079          * Create the root block IO
1080          */
1081         SET_BOOKMARK(&zb, os->os_dsl_dataset ?
1082             os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
1083             ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
1084         arc_release(os->os_phys_buf, &os->os_phys_buf);

1086         dmu_write_policy(os, NULL, 0, 0, &zp);

1088         zio = arc_write(pio, os->os_spa, tx->tx_txg,
1089             os->os_rootbp, os->os_phys_buf, DMU_OS_IS_L2CACHEABLE(os),
1090             DMU_OS_IS_L2COMPRESSIBLE(os), &zp, dmu_objset_write_ready,
1091             NULL, dmu_objset_write_done, os, ZIO_PRIORITY_ASYNC_WRITE,
1092             ZIO_FLAG_MUSTSUCCEED, &zb);

1094         /*
1095          * Sync special dnodes - the parent IO for the sync is the root block
1096          */
1097         DMU_META_DNODE(os)->dn_zio = zio;
1098         dnode_sync(DMU_META_DNODE(os), tx);

1100         os->os_phys->os_flags = os->os_flags;

1102         if (DMU_USERUSED_DNODE(os) &&
1103             DMU_USERUSED_DNODE(os)->dn_type != DMU_OT_NONE) {
1104                 DMU_USERUSED_DNODE(os)->dn_zio = zio;
1105                 dnode_sync(DMU_USERUSED_DNODE(os), tx);
1106                 DMU_GROUPUSED_DNODE(os)->dn_zio = zio;
1107                 dnode_sync(DMU_GROUPUSED_DNODE(os), tx);
1108         }

1110         txgoff = tx->tx_txg & TXG_MASK;
```



```
1112         if (dmu_objset_userused_enabled(os)) {
1113                 newlist = &os->os_synced_dnodes;
1114                 /*
1115                  * We must create the list here because it uses the
1116                  * dn_dirty_link[] of this txg.
1117                  */
1118                 list_create(newlist, sizeof (dnode_t),
1119                     offsetof(dnode_t, dn_dirty_link[txgoff]));
1120         }

1122         dmu_objset_sync_dnodes(&os->os_free_dnodes[txgoff], newlist, tx);
1123         dmu_objset_sync_dnodes(&os->os_dirty_dnodes[txgoff], newlist, tx);

1125         list = &DMU_META_DNODE(os)->dn_dirty_records[txgoff];
1126         while (dr = list_head(list)) {
1127                 ASSERT0(dr->dr_dbuf->db_level);
1128                 list_remove(list, dr);
1129                 if (dr->dr_zio)
1130                         zio_nowait(dr->dr_zio);
1131         }
1132         /*
1133          * Free intent log blocks up to this tx.
1134          */
1135         zil_sync(os->os_zil, tx);
1136         os->os_phys->os_zil_header = os->os_zil_header;
1137         zio_nowait(zio);
1138 }

1140 boolean_t
1141 dmu_objset_is_dirty(objset_t *os, uint64_t txg)
1142 {
1143         return (!list_is_empty(&os->os_dirty_dnodes[txg & TXG_MASK]) ||
1144             !list_is_empty(&os->os_free_dnodes[txg & TXG_MASK]));
1145 }

1147 static objset_used_cb_t *used_cbs[DMU_OST_NUMTYPES];

1149 void
1150 dmu_objset_register_type(dmu_objset_type_t ost, objset_used_cb_t *cb)
1151 {
1152         used_cbs[ost] = cb;
1153 }

1155 boolean_t
1156 dmu_objset_userused_enabled(objset_t *os)
1157 {
1158         return (spa_version(os->os_spa) >= SPA_VERSION_USERSPACE &&
1159             used_cbs[os->os_phys->os_type] != NULL &&
1160             DMU_USERUSED_DNODE(os) != NULL);
1161 }

1163 static void
1164 do_userquota_update(objset_t *os, uint64_t used, uint64_t flags,
1165     uint64_t user, uint64_t group, boolean_t subtract, dmu_tx_t *tx)
1166 {
1167         if ((flags & DNODE_FLAG_USERUSED_ACCOUNTED)) {
1168                 int64_t delta = DNODE_SIZE + used;
1169                 if (subtract)
1170                         delta = -delta;
1171                 VERIFY3U(0, ==, zap_increment_int(os, DMU_USERUSED_OBJECT,
1172                     user, delta, tx));
1173                 VERIFY3U(0, ==, zap_increment_int(os, DMU_GROUPUSED_OBJECT,
1174                     group, delta, tx));
1175         }
1176 }
```

```
1178 void
1179 dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx)
1180 {
1181         dnode_t *dn;
1182         list_t *list = &os->os_synced_dnodes;

1184         ASSERT(list_head(list) == NULL || dmu_objset_userused_enabled(os));

1186         while (dn = list_head(list)) {
1187                 int flags;
1188                 ASSERT(!DMU_OBJECT_IS_SPECIAL(dn->dn_object));
1189                 ASSERT(dn->dn_phys->dn_type == DMU_OT_NONE ||
1190                     dn->dn_phys->dn_flags &
1191                     DNODE_FLAG_USERUSED_ACCOUNTED);

1193                 /* Allocate the user/groupused objects if necessary. */
1194                 if (DMU_USERUSED_DNODE(os)->dn_type == DMU_OT_NONE) {
1195                         VERIFY(0 == zap_create_claim(os,
1196                             DMU_USERUSED_OBJECT,
1197                             DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
1198                         VERIFY(0 == zap_create_claim(os,
1199                             DMU_GROUPUSED_OBJECT,
1200                             DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
1201                 }

1203                 /*
1204                  * We intentionally modify the zap object even if the
1205                  * net delta is zero.  Otherwise
1206                  * the block of the zap obj could be shared between
1207                  * datasets but need to be different between them after
1208                  * a bprewrite.
1209                  */

1211                 flags = dn->dn_id_flags;
1212                 ASSERT(flags);
1213                 if (flags & DN_ID_OLD_EXIST)  {
1214                         do_userquota_update(os, dn->dn_oldused, dn->dn_oldflags,
1215                             dn->dn_olduid, dn->dn_oldgid, B_TRUE, tx);
1216                 }
1217                 if (flags & DN_ID_NEW_EXIST) {
1218                         do_userquota_update(os, DN_USED_BYTES(dn->dn_phys),
1219                             dn->dn_phys->dn_flags,  dn->dn_newuid,
1220                             dn->dn_newgid, B_FALSE, tx);
1221                 }

1223                 mutex_enter(&dn->dn_mtx);
1224                 dn->dn_oldused = 0;
1225                 dn->dn_oldflags = 0;
1226                 if (dn->dn_id_flags & DN_ID_NEW_EXIST) {
1227                         dn->dn_olduid = dn->dn_newuid;
1228                         dn->dn_oldgid = dn->dn_newgid;
1229                         dn->dn_id_flags |= DN_ID_OLD_EXIST;
1230                         if (dn->dn_bonuslen == 0)
1231                                 dn->dn_id_flags |= DN_ID_CHKED_SPILL;
1232                         else
1233                                 dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1234                 }
1235                 dn->dn_id_flags &= ~(DN_ID_NEW_EXIST);
1236                 mutex_exit(&dn->dn_mtx);

1238                 list_remove(list, dn);
1239                 dnode_rele(dn, list);
1240         }
1241 }
```



```
1243 /*
1244  * Returns a pointer to data to find uid/gid from
1245  *
1246  * If a dirty record for transaction group that is syncing can't
1247  * be found then NULL is returned.  In the NULL case it is assumed
1248  * the uid/gid aren't changing.
1249  */
1250 static void *
1251 dmu_objset_userquota_find_data(dmu_buf_impl_t *db, dmu_tx_t *tx)
1252 {
1253         dbuf_dirty_record_t *dr, **drp;
1254         void *data;

1256         if (db->db_dirtycnt == 0)
1257                 return (db->db.db_data);  /* Nothing is changing */

1259         for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1260                 if (dr->dr_txg == tx->tx_txg)
1261                         break;

1263         if (dr == NULL) {
1264                 data = NULL;
1265         } else {
1266                 dnode_t *dn;

1268                 DB_DNODE_ENTER(dr->dr_dbuf);
1269                 dn = DB_DNODE(dr->dr_dbuf);

1271                 if (dn->dn_bonuslen == 0 &&
1272                     dr->dr_dbuf->db_blkid == DMU_SPILL_BLKID)
1273                         data = dr->dt.dl.dr_data->b_data;
1274                 else
1275                         data = dr->dt.dl.dr_data;

1277                 DB_DNODE_EXIT(dr->dr_dbuf);
1278         }

1280         return (data);
1281 }

1283 void
1284 dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx)
1285 {
1286         objset_t *os = dn->dn_objset;
1287         void *data = NULL;
1288         dmu_buf_impl_t *db = NULL;
1289         uint64_t *user = NULL;
1290         uint64_t *group = NULL;
1291         int flags = dn->dn_id_flags;
1292         int error;
1293         boolean_t have_spill = B_FALSE;

1295         if (!dmu_objset_userused_enabled(dn->dn_objset))
1296                 return;

1298         if (before && (flags & (DN_ID_CHKED_BONUS|DN_ID_OLD_EXIST|
1299             DN_ID_CHKED_SPILL)))
1300                 return;

1302         if (before && dn->dn_bonuslen != 0)
1303                 data = DN_BONUS(dn->dn_phys);
1304         else if (!before && dn->dn_bonuslen != 0) {
1305                 if (dn->dn_bonus) {
1306                         db = dn->dn_bonus;
1307                         mutex_enter(&db->db_mtx);
1308                         data = dmu_objset_userquota_find_data(db, tx);
```

```
1309                 } else {
1310                         data = DN_BONUS(dn->dn_phys);
1311                 }
1312         } else if (dn->dn_bonuslen == 0 && dn->dn_bonustype == DMU_OT_SA) {
1313                         int rf = 0;
1314
1315                         if (RW_WRITE_HELD(&dn->dn_struct_rwlock))
1316                                 rf |= DB_RF_HAVESTRUCT;
1317                         error = dmu_spill_hold_by_dnode(dn,
1318                             rf | DB_RF_MUST_SUCCEED,
1319                             FTAG, (dmu_buf_t **)&db);
1320                         ASSERT(error == 0);
1321                         mutex_enter(&db->db_mtx);
1322                         data = (before) ? db->db.db_data :
1323                             dmu_objset_userquota_find_data(db, tx);
1324                         have_spill = B_TRUE;
1325         } else {
1326                 mutex_enter(&dn->dn_mtx);
1327                 dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1328                 mutex_exit(&dn->dn_mtx);
1329                 return;
1330         }
1331
1332         if (before) {
1333                 ASSERT(data);
1334                 user = &dn->dn_olduid;
1335                 group = &dn->dn_oldgid;
1336         } else if (data) {
1337                 user = &dn->dn_newuid;
1338                 group = &dn->dn_newgid;
1339         }
1340
1341         /*
1342          * Must always call the callback in case the object
1343          * type has changed and that type isn't an object type to track
1344          */
1345         error = used_cbs[os->os_phys->os_type](dn->dn_bonustype, data,
1346             user, group);
1347
1348         /*
1349          * Preserve existing uid/gid when the callback can't determine
1350          * what the new uid/gid are and the callback returned EEXIST.
1351          * The EEXIST error tells us to just use the existing uid/gid.
1352          * If we don't know what the old values are then just assign
1353          * them to 0, since that is a new file  being created.
1354          */
1355         if (!before && data == NULL && error == EEXIST) {
1356                 if (flags & DN_ID_OLD_EXIST) {
1357                         dn->dn_newuid = dn->dn_olduid;
1358                         dn->dn_newgid = dn->dn_oldgid;
1359                 } else {
1360                         dn->dn_newuid = 0;
1361                         dn->dn_newgid = 0;
1362                 }
1363                 error = 0;
1364         }
1365
1366         if (db)
1367                 mutex_exit(&db->db_mtx);
1368
1369         mutex_enter(&dn->dn_mtx);
1370         if (error == 0 && before)
1371                 dn->dn_id_flags |= DN_ID_OLD_EXIST;
1372         if (error == 0 && !before)
1373                 dn->dn_id_flags |= DN_ID_NEW_EXIST;
```



```
1375         if (have_spill) {
1376                 dn->dn_id_flags |= DN_ID_CHKED_SPILL;
1377         } else {
1378                 dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1379         }
1380         mutex_exit(&dn->dn_mtx);
1381         if (have_spill)
1382                 dmu_buf_rele((dmu_buf_t *)db, FTAG);
1383 }
1384
1385 boolean_t
1386 dmu_objset_userspace_present(objset_t *os)
1387 {
1388         return (os->os_phys->os_flags &
1389             OBJSET_FLAG_USERACCOUNTING_COMPLETE);
1390 }
1391
1392 int
1393 dmu_objset_userspace_upgrade(objset_t *os)
1394 {
1395         uint64_t obj;
1396         int err = 0;
1397
1398         if (dmu_objset_userspace_present(os))
1399                 return (0);
1400         if (!dmu_objset_userused_enabled(os))
1401                 return (SET_ERROR(ENOTSUP));
1402         if (dmu_objset_is_snapshot(os))
1403                 return (SET_ERROR(EINVAL));
1404
1405         /*
1406          * We simply need to mark every object dirty, so that it will be
1407          * synced out and now accounted.  If this is called
1408          * concurrently, or if we already did some work before crashing,
1409          * that's fine, since we track each object's accounted state
1410          * independently.
1411          */
1412
1413         for (obj = 0; err == 0; err = dmu_object_next(os, &obj, FALSE, 0)) {
1414                 dmu_tx_t *tx;
1415                 dmu_buf_t *db;
1416                 int objerr;
1417
1418                 if (issig(JUSTLOOKING) && issig(FORREAL))
1419                         return (SET_ERROR(EINTR));
1420
1421                 objerr = dmu_bonus_hold(os, obj, FTAG, &db);
1422                 if (objerr != 0)
1423                         continue;
1424                 tx = dmu_tx_create(os);
1425                 dmu_tx_hold_bonus(tx, obj);
1426                 objerr = dmu_tx_assign(tx, TXG_WAIT);
1427                 if (objerr != 0) {
1428                         dmu_tx_abort(tx);
1429                         continue;
1430                 }
1431                 dmu_buf_will_dirty(db, tx);
1432                 dmu_buf_rele(db, FTAG);
1433                 dmu_tx_commit(tx);
1434         }
1435
1436         os->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
1437         txg_wait_synced(dmu_objset_pool(os), 0);
1438         return (0);
1439 }
```

```
1441 void
1442 dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
1443     uint64_t *usedobjsp, uint64_t *availobjsp)
1444 {
1445         dsl_dataset_space(os->os_dsl_dataset, refdbytesp, availbytesp,
1446             usedobjsp, availobjsp);
1447 }

1449 uint64_t
1450 dmu_objset_fsid_guid(objset_t *os)
1451 {
1452         return (dsl_dataset_fsid_guid(os->os_dsl_dataset));
1453 }

1455 void
1456 dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat)
1457 {
1458         stat->dds_type = os->os_phys->os_type;
1459         if (os->os_dsl_dataset)
1460                 dsl_dataset_fast_stat(os->os_dsl_dataset, stat);
1461 }

1463 void
1464 dmu_objset_stats(objset_t *os, nvlist_t *nv)
1465 {
1466         ASSERT(os->os_dsl_dataset ||
1467             os->os_phys->os_type == DMU_OST_META);

1469         if (os->os_dsl_dataset != NULL)
1470                 dsl_dataset_stats(os->os_dsl_dataset, nv);

1472         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_TYPE,
1473             os->os_phys->os_type);
1474         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERACCOUNTING,
1475             dmu_objset_userspace_present(os));
1476 }

1478 int
1479 dmu_objset_is_snapshot(objset_t *os)
1480 {
1481         if (os->os_dsl_dataset != NULL)
1482                 return (dsl_dataset_is_snapshot(os->os_dsl_dataset));
1483         else
1484                 return (B_FALSE);
1485 }

1487 int
1488 dmu_snapshot_realname(objset_t *os, char *name, char *real, int maxlen,
1489     boolean_t *conflict)
1490 {
1491         dsl_dataset_t *ds = os->os_dsl_dataset;
1492         uint64_t ignored;

1494         if (ds->ds_phys->ds_snapnames_zapobj == 0)
1495                 return (SET_ERROR(ENOENT));

1497         return (zap_lookup_norm(ds->ds_dir->dd_pool->dp_meta_objset,
1498             ds->ds_phys->ds_snapnames_zapobj, name, 8, 1, &ignored, MT_FIRST,
1499             real, maxlen, conflict));
1500 }

1502 int
1503 dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
1504     uint64_t *idp, uint64_t *offp, boolean_t *case_conflict)
1505 {
1506         dsl_dataset_t *ds = os->os_dsl_dataset;
```



```
1507         zap_cursor_t cursor;
1508         zap_attribute_t attr;

1510         ASSERT(dsl_pool_config_held(dmu_objset_pool(os)));

1512         if (ds->ds_phys->ds_snapnames_zapobj == 0)
1513                 return (SET_ERROR(ENOENT));

1515         zap_cursor_init_serialized(&cursor,
1516             ds->ds_dir->dd_pool->dp_meta_objset,
1517             ds->ds_phys->ds_snapnames_zapobj, *offp);

1519         if (zap_cursor_retrieve(&cursor, &attr) != 0) {
1520                 zap_cursor_fini(&cursor);
1521                 return (SET_ERROR(ENOENT));
1522         }

1524         if (strlen(attr.za_name) + 1 > namelen) {
1525                 zap_cursor_fini(&cursor);
1526                 return (SET_ERROR(ENAMETOOLONG));
1527         }

1529         (void) strcpy(name, attr.za_name);
1530         if (idp)
1531                 *idp = attr.za_first_integer;
1532         if (case_conflict)
1533                 *case_conflict = attr.za_normalization_conflict;
1534         zap_cursor_advance(&cursor);
1535         *offp = zap_cursor_serialize(&cursor);
1536         zap_cursor_fini(&cursor);

1538         return (0);
1539 }

1541 int
1542 dmu_dir_list_next(objset_t *os, int namelen, char *name,
1543     uint64_t *idp, uint64_t *offp)
1544 {
1545         dsl_dir_t *dd = os->os_dsl_dataset->ds_dir;
1546         zap_cursor_t cursor;
1547         zap_attribute_t attr;

1549         /* there is no next dir on a snapshot! */
1550         if (os->os_dsl_dataset->ds_object !=
1551             dd->dd_phys->dd_head_dataset_obj)
1552                 return (SET_ERROR(ENOENT));

1554         zap_cursor_init_serialized(&cursor,
1555             dd->dd_pool->dp_meta_objset,
1556             dd->dd_phys->dd_child_dir_zapobj, *offp);

1558         if (zap_cursor_retrieve(&cursor, &attr) != 0) {
1559                 zap_cursor_fini(&cursor);
1560                 return (SET_ERROR(ENOENT));
1561         }

1563         if (strlen(attr.za_name) + 1 > namelen) {
1564                 zap_cursor_fini(&cursor);
1565                 return (SET_ERROR(ENAMETOOLONG));
1566         }

1568         (void) strcpy(name, attr.za_name);
1569         if (idp)
1570                 *idp = attr.za_first_integer;
1571         zap_cursor_advance(&cursor);
1572         *offp = zap_cursor_serialize(&cursor);
```

```
1573         zap_cursor_fini(&cursor);

1575         return (0);
1576 }

1578 /*
1579  * Find objsets under and including ddobj, call func(ds) on each.
1580  */
1581 int
1582 dmu_objset_find_dp(dsl_pool_t *dp, uint64_t ddobj,
1583     int func(dsl_pool_t *, dsl_dataset_t *, void *), void *arg, int flags)
1584 {
1585         dsl_dir_t *dd;
1586         dsl_dataset_t *ds;
1587         zap_cursor_t zc;
1588         zap_attribute_t *attr;
1589         uint64_t thisobj;
1590         int err;

1592         ASSERT(dsl_pool_config_held(dp));

1594         err = dsl_dir_hold_obj(dp, ddobj, NULL, FTAG, &dd);
1595         if (err != 0)
1596                 return (err);

1598         /* Don't visit hidden ($MOS & $ORIGIN) objsets. */
1599         if (dd->dd_myname[0] == '$') {
1600                 dsl_dir_rele(dd, FTAG);
1601                 return (0);
1602         }

1604         thisobj = dd->dd_phys->dd_head_dataset_obj;
1605         attr = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);

1607         /*
1608          * Iterate over all children.
1609          */
1610         if (flags & DS_FIND_CHILDREN) {
1611                 for (zap_cursor_init(&zc, dp->dp_meta_objset,
1612                     dd->dd_phys->dd_child_dir_zapobj);
1613                     zap_cursor_retrieve(&zc, attr) == 0;
1614                     (void) zap_cursor_advance(&zc)) {
1615                         ASSERT3U(attr->za_integer_length, ==,
1616                             sizeof (uint64_t));
1617                         ASSERT3U(attr->za_num_integers, ==, 1);

1619                         err = dmu_objset_find_dp(dp, attr->za_first_integer,
1620                             func, arg, flags);
1621                         if (err != 0)
1622                                 break;
1623                 }
1624                 zap_cursor_fini(&zc);

1626                 if (err != 0) {
1627                         dsl_dir_rele(dd, FTAG);
1628                         kmem_free(attr, sizeof (zap_attribute_t));
1629                         return (err);
1630                 }
1631         }

1633         /*
1634          * Iterate over all snapshots.
1635          */
1636         if (flags & DS_FIND_SNAPSHOTS) {
1637                 dsl_dataset_t *ds;
1638                 err = dsl_dataset_hold_obj(dp, thisobj, FTAG, &ds);
```



```
1640                 if (err == 0) {
1641                         uint64_t snapobj = ds->ds_phys->ds_snapnames_zapobj;
1642                         dsl_dataset_rele(ds, FTAG);

1644                         for (zap_cursor_init(&zc, dp->dp_meta_objset, snapobj);
1645                             zap_cursor_retrieve(&zc, attr) == 0;
1646                             (void) zap_cursor_advance(&zc)) {
1647                                 ASSERT3U(attr->za_integer_length, ==,
1648                                     sizeof (uint64_t));
1649                                 ASSERT3U(attr->za_num_integers, ==, 1);

1651                                 err = dsl_dataset_hold_obj(dp,
1652                                     attr->za_first_integer, FTAG, &ds);
1653                                 if (err != 0)
1654                                         break;
1655                                 err = func(dp, ds, arg);
1656                                 dsl_dataset_rele(ds, FTAG);
1657                                 if (err != 0)
1658                                         break;
1659                         }
1660                         zap_cursor_fini(&zc);
1661                 }
1662         }

1664         dsl_dir_rele(dd, FTAG);
1665         kmem_free(attr, sizeof (zap_attribute_t));

1667         if (err != 0)
1668                 return (err);

1670         /*
1671          * Apply to self.
1672          */
1673         err = dsl_dataset_hold_obj(dp, thisobj, FTAG, &ds);
1674         if (err != 0)
1675                 return (err);
1676         err = func(dp, ds, arg);
1677         dsl_dataset_rele(ds, FTAG);
1678         return (err);
1679 }

1681 /*
1682  * Find all objsets under name, and for each, call 'func(child_name, arg)'.
1683  * The dp_config_rwlock must not be held when this is called, and it
1684  * will not be held when the callback is called.
1685  * Therefore this function should only be used when the pool is not changing
1686  * (e.g. in syncing context), or the callback can deal with the possible races.
1687  */
1688 static int
1689 dmu_objset_find_impl(spa_t *spa, const char *name,
1690     int func(const char *, void *), void *arg, int flags)
1691 {
1692         dsl_dir_t *dd;
1693         dsl_pool_t *dp = spa_get_dsl(spa);
1694         dsl_dataset_t *ds;
1695         zap_cursor_t zc;
1696         zap_attribute_t *attr;
1697         char *child;
1698         uint64_t thisobj;
1699         int err;

1701         dsl_pool_config_enter(dp, FTAG);

1703         err = dsl_dir_hold(dp, name, FTAG, &dd, NULL);
1704         if (err != 0) {
```

```
1705                dsl_pool_config_exit(dp, FTAG);
1706                return (err);
1707        }

1709        /* Don't visit hidden ($MOS & $ORIGIN) objsets. */
1710        if (dd->dd_myname[0] == '$') {
1711                dsl_dir_rele(dd, FTAG);
1712                dsl_pool_config_exit(dp, FTAG);
1713                return (0);
1714        }

1716        thisobj = dd->dd_phys->dd_head_dataset_obj;
1717        attr = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);

1719        /*
1720         * Iterate over all children.
1721         */
1722        if (flags & DS_FIND_CHILDREN) {
1723                for (zap_cursor_init(&zc, dp->dp_meta_objset,
1724                    dd->dd_phys->dd_child_dir_zapobj);
1725                    zap_cursor_retrieve(&zc, attr) == 0;
1726                    (void) zap_cursor_advance(&zc)) {
1727                        ASSERT3U(attr->za_integer_length, ==,
1728                            sizeof (uint64_t));
1729                        ASSERT3U(attr->za_num_integers, ==, 1);

1731                        child = kmem_asprintf("%s/%s", name, attr->za_name);
1732                        dsl_pool_config_exit(dp, FTAG);
1733                        err = dmu_objset_find_impl(spa, child,
1734                            func, arg, flags);
1735                        dsl_pool_config_enter(dp, FTAG);
1736                        strfree(child);
1737                        if (err != 0)
1738                                break;
1739                }
1740                zap_cursor_fini(&zc);

1742                if (err != 0) {
1743                        dsl_dir_rele(dd, FTAG);
1744                        dsl_pool_config_exit(dp, FTAG);
1745                        kmem_free(attr, sizeof (zap_attribute_t));
1746                        return (err);
1747                }
1748        }

1750        /*
1751         * Iterate over all snapshots.
1752         */
1753        if (flags & DS_FIND_SNAPSHOTS) {
1754                err = dsl_dataset_hold_obj(dp, thisobj, FTAG, &ds);

1756                if (err == 0) {
1757                        uint64_t snapobj = ds->ds_phys->ds_snapnames_zapobj;
1758                        dsl_dataset_rele(ds, FTAG);

1760                        for (zap_cursor_init(&zc, dp->dp_meta_objset, snapobj);
1761                            zap_cursor_retrieve(&zc, attr) == 0;
1762                            (void) zap_cursor_advance(&zc)) {
1763                                ASSERT3U(attr->za_integer_length, ==,
1764                                    sizeof (uint64_t));
1765                                ASSERT3U(attr->za_num_integers, ==, 1);

1767                                child = kmem_asprintf("%s@%s",
1768                                    name, attr->za_name);
1769                                dsl_pool_config_exit(dp, FTAG);
1770                                err = func(child, arg);
```



```
1771                                dsl_pool_config_enter(dp, FTAG);
1772                                strfree(child);
1773                                if (err != 0)
1774                                        break;
1775                        }
1776                        zap_cursor_fini(&zc);
1777                }
1778        }

1780        dsl_dir_rele(dd, FTAG);
1781        kmem_free(attr, sizeof (zap_attribute_t));
1782        dsl_pool_config_exit(dp, FTAG);

1784        if (err != 0)
1785                return (err);

1787        /* Apply to self. */
1788        return (func(name, arg));
1789 }

1791 /*
1792  * See comment above dmu_objset_find_impl().
1793  */
1794 int
1795 dmu_objset_find(char *name, int func(const char *, void *), void *arg,
1796     int flags)
1797 {
1798        spa_t *spa;
1799        int error;

1801        error = spa_open(name, &spa, FTAG);
1802        if (error != 0)
1803                return (error);
1804        error = dmu_objset_find_impl(spa, name, func, arg, flags);
1805        spa_close(spa, FTAG);

1807        return (error);
1808 }

1810 typedef struct dmu_objset_find_ctx {
1811        taskq_t         *dc_tq;
1812        spa_t           *dc_spa;
1813        char            *dc_name;
1814        int             (*dc_func)(const char *, void *);
1815        void            *dc_arg;
1816        int             dc_flags;
1817        kmutex_t        *dc_error_lock;
1818        int             *dc_error;
1819 } dmu_objset_find_ctx_t;

1821 static void
1822 dmu_objset_find_parallel_impl(void *arg)
1823 {
1824        dmu_objset_find_ctx_t *dcp = arg;
1825        dsl_dir_t *dd;
1826        dsl_pool_t *dp = spa_get_dsl(dcp->dc_spa);
1827        dsl_dataset_t *ds;
1828        zap_cursor_t zc;
1829        zap_attribute_t *attr;
1830        char *child;
1831        dmu_objset_find_ctx_t *child_dcp;
1832        uint64_t thisobj;
1833        int err;

1835        /* don't process if there already was an error */
1836        if (*dcp->dc_error)
```

```
1837                goto out;

1839        dsl_pool_config_enter(dp, FTAG);

1841        err = dsl_dir_hold(dp, dcp->dc_name, FTAG, &dd, NULL);
1842        if (err != 0) {
1843                dsl_pool_config_exit(dp, FTAG);
1844                goto fail;
1845        }

1847        /* Don't visit hidden ($MOS & $ORIGIN) objsets. */
1848        if (dd->dd_myname[0] == '$') {
1849                dsl_dir_rele(dd, FTAG);
1850                dsl_pool_config_exit(dp, FTAG);
1851                goto out;
1852        }

1854        thisobj = dd->dd_phys->dd_head_dataset_obj;
1855        attr = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);

1857        /*
1858         * Iterate over all children.
1859         */
1860        if (dcp->dc_flags & DS_FIND_CHILDREN) {
1861                for (zap_cursor_init(&zc, dp->dp_meta_objset,
1862                    dd->dd_phys->dd_child_dir_zapobj);
1863                    zap_cursor_retrieve(&zc, attr) == 0;
1864                    (void) zap_cursor_advance(&zc)) {
1865                        ASSERT3U(attr->za_integer_length, ==,
1866                            sizeof (uint64_t));
1867                        ASSERT3U(attr->za_num_integers, ==, 1);

1869                        child = kmem_asprintf("%s/%s", dcp->dc_name,
1870                            attr->za_name);
1871                        dsl_pool_config_exit(dp, FTAG);
1872                        child_dcp = kmem_alloc(sizeof(*child_dcp), KM_SLEEP);
1873                        *child_dcp = *dcp;
1874                        child_dcp->dc_name = child;
1875                        taskq_dispatch(dcp->dc_tq,
1876                            dmu_objset_find_parallel_impl, child_dcp, TQ_SLEEP);
1877                        dsl_pool_config_enter(dp, FTAG);
1878                }
1879                zap_cursor_fini(&zc);
1880        }

1882        dsl_dir_rele(dd, FTAG);
1883        kmem_free(attr, sizeof (zap_attribute_t));
1884        dsl_pool_config_exit(dp, FTAG);

1886        err = dcp->dc_func(dcp->dc_name, dcp->dc_arg);

1888 fail:
1889        if (err) {
1890                mutex_enter(dcp->dc_error_lock);
1891                /* only keep first error */
1892                if (*dcp->dc_error == 0)
1893                        *dcp->dc_error = err;
1894                mutex_exit(dcp->dc_error_lock);
1895        }

1897 out:
1898        strfree(dcp->dc_name);
1899        kmem_free(dcp, sizeof(*dcp));
1900 }

1902 int
```



```
1903 dmu_objset_find_parallel(char *name, int func(const char *, void *), void *arg,
1904     int flags)
1905 {
1906        spa_t *spa;
1907        int error;
1908        taskq_t *tq = NULL;
1909        int ntasks;
1910        dmu_objset_find_ctx_t *dcp;
1911        kmutex_t err_lock;

1913        error = spa_open(name, &spa, FTAG);
1914        if (error != 0)
1915                return (error);

1917        ntasks = vdev_count_leaves(spa) * 4;
1918        tq = taskq_create("dmu_objset_find", ntasks, minclsyspri, ntasks,
1919            INT_MAX, 0);
1920        if (!tq) {
1921                spa_close(spa, FTAG);
1922                return (dmu_objset_find(name, func, arg, flags));
1923        }

1925        mutex_init(&err_lock, NULL, MUTEX_DEFAULT, NULL);
1926        dcp = kmem_alloc(sizeof(*dcp), KM_SLEEP);
1927        dcp->dc_tq = tq;
1928        dcp->dc_spa = spa;
1929        dcp->dc_name = strdup(name);
1930        dcp->dc_func = func;
1931        dcp->dc_arg = arg;
1932        dcp->dc_flags = flags;
1933        dcp->dc_error_lock = &err_lock;
1934        dcp->dc_error = &error;
1935        /* dcp and dc_name will be freed by task */
1936        taskq_dispatch(tq, dmu_objset_find_parallel_impl, dcp, TQ_SLEEP);

1938        taskq_wait(tq);
1939        taskq_destroy(tq);
1940        mutex_destroy(&err_lock);

1942        spa_close(spa, FTAG);

1944 #endif /* ! codereview */
1945        return (error);
1946 }

1948 void
1949 dmu_objset_set_user(objset_t *os, void *user_ptr)
1950 {
1951        ASSERT(MUTEX_HELD(&os->os_user_ptr_lock));
1952        os->os_user_ptr = user_ptr;
1953 }

1955 void *
1956 dmu_objset_get_user(objset_t *os)
1957 {
1958        ASSERT(MUTEX_HELD(&os->os_user_ptr_lock));
1959        return (os->os_user_ptr);
1960 }

1962 /*
1963  * Determine name of filesystem, given name of snapshot.
1964  * buf must be at least MAXNAMELEN bytes
1965  */
1966 int
1967 dmu_fsname(const char *snapname, char *buf)
1968 {
```

```
1969         char *atp = strchr(snapname, '@');
1970         if (atp == NULL)
1971                 return (SET_ERROR(EINVAL));
1972         if (atp - snapname >= MAXNAMELEN)
1973                 return (SET_ERROR(ENAMETOOLONG));
1974         (void) strlcpy(buf, snapname, atp - snapname + 1);
1975         return (0);
1976 }
```

```
**********************************************************
   30992 Thu Oct 16 19:15:50 2014
new/usr/src/uts/common/fs/zfs/dsl_pool.c
zpool import speedup
**********************************************************
______unchanged_portion_omitted_

 954 /*
 955  * DSL Pool Configuration Lock
 956  *
 957  * The dp_config_rwlock protects against changes to DSL state (e.g. dataset
 958  * creation / destruction / rename / property setting).  It must be held for
 959  * read to hold a dataset or dsl_dir.  I.e. you must call
 960  * dsl_pool_config_enter() or dsl_pool_hold() before calling
 961  * dsl_{dataset,dir}_hold{_obj}.  In most circumstances, the dp_config_rwlock
 962  * must be held continuously until all datasets and dsl_dirs are released.
 963  *
 964  * The only exception to this rule is that if a "long hold" is placed on
 965  * a dataset, then the dp_config_rwlock may be dropped while the dataset
 966  * is still held.  The long hold will prevent the dataset from being
 967  * destroyed -- the destroy will fail with EBUSY.  A long hold can be
 968  * obtained by calling dsl_dataset_long_hold(), or by "owning" a dataset
 969  * (by calling dsl_{dataset,objset}_{try}own{_obj}).
 970  *
 971  * Legitimate long-holders (including owners) should be long-running, cancelable
 972  * tasks that should cause "zfs destroy" to fail.  This includes DMU
 973  * consumers (i.e. a ZPL filesystem being mounted or ZVOL being open),
 974  * "zfs send", and "zfs diff".  There are several other long-holders whose
 975  * uses are suboptimal (e.g. "zfs promote", and zil_suspend()).
 976  *
 977  * The usual formula for long-holding would be:
 978  * dsl_pool_hold()
 979  * dsl_dataset_hold()
 980  * ... perform checks ...
 981  * dsl_dataset_long_hold()
 982  * dsl_pool_rele()
 983  * ... perform long-running task ...
 984  * dsl_dataset_long_rele()
 985  * dsl_dataset_rele()
 986  *
 987  * Note that when the long hold is released, the dataset is still held but
 988  * the pool is not held.  The dataset may change arbitrarily during this time
 989  * (e.g. it could be destroyed).  Therefore you shouldn't do anything to the
 990  * dataset except release it.
 991  *
 992  * User-initiated operations (e.g. ioctls, zfs_ioc_*()) are either read-only
 993  * or modifying operations.
 994  *
 995  * Modifying operations should generally use dsl_sync_task().  The synctask
 996  * infrastructure enforces proper locking strategy with respect to the
 997  * dp_config_rwlock.  See the comment above dsl_sync_task() for details.
 998  *
 999  * Read-only operations will manually hold the pool, then the dataset, obtain
1000  * information from the dataset, then release the pool and dataset.
1001  * dmu_objset_{hold,rele}() are convenience routines that also do the pool
1002  * hold/rele.
1003  */

1005 int
1006 dsl_pool_hold_lock(const char *name, void *tag, dsl_pool_t **dp, int lock)
1006 dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp)
1007 {
1008         spa_t *spa;
1009         int error;

1011         error = spa_open_lock(name, &spa, tag, lock);
```



```
1011         error = spa_open(name, &spa, tag);
1012         if (error == 0) {
1013                 *dp = spa_get_dsl(spa);
1014                 dsl_pool_config_enter(*dp, tag);
1015         }
1016         return (error);
1017 }

1019 int
1020 dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp)
1021 {
1022         return (dsl_pool_hold_lock(name, tag, dp, 1));
1023 }

1025 #endif /* ! codereview */
1026 void
1027 dsl_pool_rele(dsl_pool_t *dp, void *tag)
1028 {
1029         dsl_pool_config_exit(dp, tag);
1030         spa_close(dp->dp_spa, tag);
1031 }

1033 void
1034 dsl_pool_config_enter(dsl_pool_t *dp, void *tag)
1035 {
1036         /*
1037          * We use a "reentrant" reader-writer lock, but not reentrantly.
1038          *
1039          * The rrwlock can (with the track_all flag) track all reading threads,
1040          * which is very useful for debugging which code path failed to release
1041          * the lock, and for verifying that the *current* thread does hold
1042          * the lock.
1043          *
1044          * (Unlike a rwlock, which knows that N threads hold it for
1045          * read, but not *which* threads, so rw_held(RW_READER) returns TRUE
1046          * if any thread holds it for read, even if this thread doesn't).
1047          */
1048         ASSERT(!rrw_held(&dp->dp_config_rwlock, RW_READER));
1049         rrw_enter(&dp->dp_config_rwlock, RW_READER, tag);
1050 }

1052 void
1053 dsl_pool_config_exit(dsl_pool_t *dp, void *tag)
1054 {
1055         rrw_exit(&dp->dp_config_rwlock, tag);
1056 }

1058 boolean_t
1059 dsl_pool_config_held(dsl_pool_t *dp)
1060 {
1061         return (RRW_LOCK_HELD(&dp->dp_config_rwlock));
1062 }
```

```
**********************************************************
  176914 Thu Oct 16 19:15:51 2014
new/usr/src/uts/common/fs/zfs/spa.c
zpool import speedup
**********************************************************
______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;

1716         switch (spa->spa_log_state) {
1717         case SPA_LOG_MISSING:
1718                 /* need to recheck in case slog has been restored */
1719         case SPA_LOG_UNKNOWN:
1720                 rv = (dmu_objset_find_parallel(spa->spa_name,
1721                     zil_check_log_chain, NULL, DS_FIND_CHILDREN) != 0);
1720                 rv = (dmu_objset_find(spa->spa_name, zil_check_log_chain,
1721                     NULL, DS_FIND_CHILDREN) != 0);
1722                 if (rv)
1723                         spa_set_log_state(spa, SPA_LOG_MISSING);
1724                 break;
1725         }
1726         return (rv);
1727 }
______unchanged_portion_omitted_

2076 /*
2077  * Load an existing storage pool, using the pool's builtin spa_config as a
2078  * source of configuration information.
2079  */
2080 static int
2081 spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config,
2082     spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
2083     char **ereport)
2084 {
2085         int error = 0;
2086         nvlist_t *nvroot = NULL;
2087         nvlist_t *label;
2088         vdev_t *rvd;
2089         uberblock_t *ub = &spa->spa_uberblock;
2090         uint64_t children, config_cache_txg = spa->spa_config_txg;
2091         int orig_mode = spa->spa_mode;
2092         int parse;
2093         uint64_t obj;
2094         boolean_t missing_feat_write = B_FALSE;

2096         /*
2097          * If this is an untrusted config, access the pool in read-only mode.
2098          * This prevents things like resilvering recently removed devices.
2099          */
2100         if (!mosconfig)
2101                 spa->spa_mode = FREAD;

2103         ASSERT(MUTEX_HELD(&spa_namespace_lock));

2105         spa->spa_load_state = state;

2107         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot))
2108                 return (SET_ERROR(EINVAL));

2110         parse = (type == SPA_IMPORT_EXISTING ?
```



```
2111             VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);

2113         /*
2114          * Create "The Godfather" zio to hold all async IOs
2115          */
2116         spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
2117             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);

2119         /*
2120          * Parse the configuration into a vdev tree.  We explicitly set the
2121          * value that will be returned by spa_version() since parsing the
2122          * configuration requires knowing the version number.
2123          */
2124         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2125         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse);
2126         spa_config_exit(spa, SCL_ALL, FTAG);

2128         if (error != 0)
2129                 return (error);

2131         ASSERT(spa->spa_root_vdev == rvd);

2133         if (type != SPA_IMPORT_ASSEMBLE) {
2134                 ASSERT(spa_guid(spa) == pool_guid);
2135         }

2137         /*
2138          * Try to open all vdevs, loading each label in the process.
2139          */
2140         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2141         error = vdev_open(rvd);
2142         spa_config_exit(spa, SCL_ALL, FTAG);
2143         if (error != 0)
2144                 return (error);

2146         /*
2147          * We need to validate the vdev labels against the configuration that
2148          * we have in hand, which is dependent on the setting of mosconfig. If
2149          * mosconfig is true then we're validating the vdev labels based on
2150          * that config.  Otherwise, we're validating against the cached config
2151          * (zpool.cache) that was read when we loaded the zfs module, and then
2152          * later we will recursively call spa_load() and validate against
2153          * the vdev config.
2154          *
2155          * If we're assembling a new pool that's been split off from an
2156          * existing pool, the labels haven't yet been updated so we skip
2157          * validation for now.
2158          */
2159         if (type != SPA_IMPORT_ASSEMBLE) {
2160                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2161                 error = vdev_validate(rvd, mosconfig);
2162                 spa_config_exit(spa, SCL_ALL, FTAG);

2164                 if (error != 0)
2165                         return (error);

2167                 if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2168                         return (SET_ERROR(ENXIO));
2169         }

2171         /*
2172          * Find the best uberblock.
2173          */
2174         vdev_uberblock_load(rvd, ub, &label);

2176         /*
```

```
2177          * If we weren't able to find a single valid uberblock, return failure.
2178          */
2179         if (ub->ub_txg == 0) {
2180                 nvlist_free(label);
2181                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, ENXIO));
2182         }
2183
2184         /*
2185          * If the pool has an unsupported version we can't open it.
2186          */
2187         if (!SPA_VERSION_IS_SUPPORTED(ub->ub_version)) {
2188                 nvlist_free(label);
2189                 return (spa_vdev_err(rvd, VDEV_AUX_VERSION_NEWER, ENOTSUP));
2190         }
2191
2192         if (ub->ub_version >= SPA_VERSION_FEATURES) {
2193                 nvlist_t *features;
2194
2195                 /*
2196                  * If we weren't able to find what's necessary for reading the
2197                  * MOS in the label, return failure.
2198                  */
2199                 if (label == NULL || nvlist_lookup_nvlist(label,
2200                     ZPOOL_CONFIG_FEATURES_FOR_READ, &features) != 0) {
2201                         nvlist_free(label);
2202                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2203                             ENXIO));
2204                 }
2205
2206                 /*
2207                  * Update our in-core representation with the definitive values
2208                  * from the label.
2209                  */
2210                 nvlist_free(spa->spa_label_features);
2211                 VERIFY(nvlist_dup(features, &spa->spa_label_features, 0) == 0);
2212         }
2213
2214         nvlist_free(label);
2215
2216         /*
2217          * Look through entries in the label nvlist's features_for_read. If
2218          * there is a feature listed there which we don't understand then we
2219          * cannot open a pool.
2220          */
2221         if (ub->ub_version >= SPA_VERSION_FEATURES) {
2222                 nvlist_t *unsup_feat;
2223
2224                 VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
2225                     0);
2226
2227                 for (nvpair_t *nvp = nvlist_next_nvpair(spa->spa_label_features,
2228                     NULL); nvp != NULL;
2229                     nvp = nvlist_next_nvpair(spa->spa_label_features, nvp)) {
2230                         if (!zfeature_is_supported(nvpair_name(nvp))) {
2231                                 VERIFY(nvlist_add_string(unsup_feat,
2232                                     nvpair_name(nvp), "") == 0);
2233                         }
2234                 }
2235
2236                 if (!nvlist_empty(unsup_feat)) {
2237                         VERIFY(nvlist_add_nvlist(spa->spa_load_info,
2238                             ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
2239                         nvlist_free(unsup_feat);
2240                         return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2241                             ENOTSUP));
2242                 }
```



```
2244                 nvlist_free(unsup_feat);
2245         }
2246
2247         /*
2248          * If the vdev guid sum doesn't match the uberblock, we have an
2249          * incomplete configuration.  We first check to see if the pool
2250          * is aware of the complete config (i.e ZPOOL_CONFIG_VDEV_CHILDREN).
2251          * If it is, defer the vdev_guid_sum check till later so we
2252          * can handle missing vdevs.
2253          */
2254         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VDEV_CHILDREN,
2255             &children) != 0 && mosconfig && type != SPA_IMPORT_ASSEMBLE &&
2256             rvd->vdev_guid_sum != ub->ub_guid_sum)
2257                 return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));
2258
2259         if (type != SPA_IMPORT_ASSEMBLE && spa->spa_config_splitting) {
2260                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2261                 spa_try_repair(spa, config);
2262                 spa_config_exit(spa, SCL_ALL, FTAG);
2263                 nvlist_free(spa->spa_config_splitting);
2264                 spa->spa_config_splitting = NULL;
2265         }
2266
2267         /*
2268          * Initialize internal SPA structures.
2269          */
2270         spa->spa_state = POOL_STATE_ACTIVE;
2271         spa->spa_ubsync = spa->spa_uberblock;
2272         spa->spa_verify_min_txg = spa->spa_extreme_rewind ?
2273             TXG_INITIAL - 1 : spa_last_synced_txg(spa) - TXG_DEFER_SIZE - 1;
2274         spa->spa_first_txg = spa->spa_last_ubsync_txg ?
2275             spa->spa_last_ubsync_txg : spa_last_synced_txg(spa) + 1;
2276         spa->spa_claim_max_txg = spa->spa_first_txg;
2277         spa->spa_prev_software_version = ub->ub_software_version;
2278
2279         error = dsl_pool_init(spa, spa->spa_first_txg, &spa->spa_dsl_pool);
2280         if (error)
2281                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2282         spa->spa_meta_objset = spa->spa_dsl_pool->dp_meta_objset;
2283
2284         if (spa_dir_prop(spa, DMU_POOL_CONFIG, &spa->spa_config_object) != 0)
2285                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2286
2287         if (spa_version(spa) >= SPA_VERSION_FEATURES) {
2288                 boolean_t missing_feat_read = B_FALSE;
2289                 nvlist_t *unsup_feat, *enabled_feat;
2290
2291                 if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_READ,
2292                     &spa->spa_feat_for_read_obj) != 0) {
2293                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2294                 }
2295
2296                 if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_WRITE,
2297                     &spa->spa_feat_for_write_obj) != 0) {
2298                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2299                 }
2300
2301                 if (spa_dir_prop(spa, DMU_POOL_FEATURE_DESCRIPTIONS,
2302                     &spa->spa_feat_desc_obj) != 0) {
2303                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2304                 }
2305
2306                 enabled_feat = fnvlist_alloc();
2307                 unsup_feat = fnvlist_alloc();
```

```
2309                if (!spa_features_check(spa, B_FALSE,
2310                    unsup_feat, enabled_feat))
2311                        missing_feat_read = B_TRUE;

2313                if (spa_writeable(spa) || state == SPA_LOAD_TRYIMPORT) {
2314                        if (!spa_features_check(spa, B_TRUE,
2315                            unsup_feat, enabled_feat)) {
2316                                missing_feat_write = B_TRUE;
2317                        }
2318                }

2320                fnvlist_add_nvlist(spa->spa_load_info,
2321                    ZPOOL_CONFIG_ENABLED_FEAT, enabled_feat);

2323                if (!nvlist_empty(unsup_feat)) {
2324                        fnvlist_add_nvlist(spa->spa_load_info,
2325                            ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat);
2326                }

2328                fnvlist_free(enabled_feat);
2329                fnvlist_free(unsup_feat);

2331                if (!missing_feat_read) {
2332                        fnvlist_add_boolean(spa->spa_load_info,
2333                            ZPOOL_CONFIG_CAN_RDONLY);
2334                }

2336                /*
2337                 * If the state is SPA_LOAD_TRYIMPORT, our objective is
2338                 * twofold: to determine whether the pool is available for
2339                 * import in read-write mode and (if it is not) whether the
2340                 * pool is available for import in read-only mode. If the pool
2341                 * is available for import in read-write mode, it is displayed
2342                 * as available in userland; if it is not available for import
2343                 * in read-only mode, it is displayed as unavailable in
2344                 * userland. If the pool is available for import in read-only
2345                 * mode but not read-write mode, it is displayed as unavailable
2346                 * in userland with a special note that the pool is actually
2347                 * available for open in read-only mode.
2348                 *
2349                 * As a result, if the state is SPA_LOAD_TRYIMPORT and we are
2350                 * missing a feature for write, we must first determine whether
2351                 * the pool can be opened read-only before returning to
2352                 * userland in order to know whether to display the
2353                 * abovementioned note.
2354                 */
2355                if (missing_feat_read || (missing_feat_write &&
2356                    spa_writeable(spa))) {
2357                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2358                            ENOTSUP));
2359                }

2361                /*
2362                 * Load refcounts for ZFS features from disk into an in-memory
2363                 * cache during SPA initialization.
2364                 */
2365                for (spa_feature_t i = 0; i < SPA_FEATURES; i++) {
2366                        uint64_t refcount;

2368                        error = feature_get_refcount_from_disk(spa,
2369                            &spa_feature_table[i], &refcount);
2370                        if (error == 0) {
2371                                spa->spa_feat_refcount_cache[i] = refcount;
2372                        } else if (error == ENOTSUP) {
2373                                spa->spa_feat_refcount_cache[i] =
2374                                    SPA_FEATURE_DISABLED;
```



```
2375                        } else {
2376                                return (spa_vdev_err(rvd,
2377                                    VDEV_AUX_CORRUPT_DATA, EIO));
2378                        }
2379                }
2380        }

2382        if (spa_feature_is_active(spa, SPA_FEATURE_ENABLED_TXG)) {
2383                if (spa_dir_prop(spa, DMU_POOL_FEATURE_ENABLED_TXG,
2384                    &spa->spa_feat_enabled_txg_obj) != 0)
2385                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2386        }

2388        spa->spa_is_initializing = B_TRUE;
2389        error = dsl_pool_open(spa->spa_dsl_pool);
2390        spa->spa_is_initializing = B_FALSE;
2391        if (error != 0)
2392                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2394        if (!mosconfig) {
2395                uint64_t hostid;
2396                nvlist_t *policy = NULL, *nvconfig;

2398                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2399                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2401                if (!spa_is_root(spa) && nvlist_lookup_uint64(nvconfig,
2402                    ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
2403                        char *hostname;
2404                        unsigned long myhostid = 0;

2406                        VERIFY(nvlist_lookup_string(nvconfig,
2407                            ZPOOL_CONFIG_HOSTNAME, &hostname) == 0);

2409 #ifdef  _KERNEL
2410                        myhostid = zone_get_hostid(NULL);
2411 #else   /* _KERNEL */
2412                        /*
2413                         * We're emulating the system's hostid in userland, so
2414                         * we can't use zone_get_hostid().
2415                         */
2416                        (void) ddi_strtoul(hw_serial, NULL, 10, &myhostid);
2417 #endif  /* _KERNEL */
2418                        if (hostid != 0 && myhostid != 0 &&
2419                            hostid != myhostid) {
2420                                nvlist_free(nvconfig);
2421                                cmn_err(CE_WARN, "pool '%s' could not be "
2422                                    "loaded as it was last accessed by "
2423                                    "another system (host: %s hostid: 0x%lx). "
2424                                    "See: http://illumos.org/msg/ZFS-8000-EY",
2425                                    spa_name(spa), hostname,
2426                                    (unsigned long)hostid);
2427                                return (SET_ERROR(EBADF));
2428                        }
2429                }
2430                if (nvlist_lookup_nvlist(spa->spa_config,
2431                    ZPOOL_REWIND_POLICY, &policy) == 0)
2432                        VERIFY(nvlist_add_nvlist(nvconfig,
2433                            ZPOOL_REWIND_POLICY, policy) == 0);

2435                spa_config_set(spa, nvconfig);
2436                spa_unload(spa);
2437                spa_deactivate(spa);
2438                spa_activate(spa, orig_mode);

2440                return (spa_load(spa, state, SPA_IMPORT_EXISTING, B_TRUE));
```

```
2441        }

2443        if (spa_dir_prop(spa, DMU_POOL_SYNC_BPOBJ, &obj) != 0)
2444                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2445        error = bpobj_open(&spa->spa_deferred_bpobj, spa->spa_meta_objset, obj);
2446        if (error != 0)
2447                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2449        /*
2450         * Load the bit that tells us to use the new accounting function
2451         * (raid-z deflation).  If we have an older pool, this will not
2452         * be present.
2453         */
2454        error = spa_dir_prop(spa, DMU_POOL_DEFLATE, &spa->spa_deflate);
2455        if (error != 0 && error != ENOENT)
2456                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2458        error = spa_dir_prop(spa, DMU_POOL_CREATION_VERSION,
2459            &spa->spa_creation_version);
2460        if (error != 0 && error != ENOENT)
2461                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2463        /*
2464         * Load the persistent error log.  If we have an older pool, this will
2465         * not be present.
2466         */
2467        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_LAST, &spa->spa_errlog_last);
2468        if (error != 0 && error != ENOENT)
2469                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2471        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_SCRUB,
2472            &spa->spa_errlog_scrub);
2473        if (error != 0 && error != ENOENT)
2474                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2476        /*
2477         * Load the history object.  If we have an older pool, this
2478         * will not be present.
2479         */
2480        error = spa_dir_prop(spa, DMU_POOL_HISTORY, &spa->spa_history);
2481        if (error != 0 && error != ENOENT)
2482                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2484        /*
2485         * If we're assembling the pool from the split-off vdevs of
2486         * an existing pool, we don't want to attach the spares & cache
2487         * devices.
2488         */

2490        /*
2491         * Load any hot spares for this pool.
2492         */
2493        error = spa_dir_prop(spa, DMU_POOL_SPARES, &spa->spa_spares.sav_object);
2494        if (error != 0 && error != ENOENT)
2495                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2496        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2497                ASSERT(spa_version(spa) >= SPA_VERSION_SPARES);
2498                if (load_nvlist(spa, spa->spa_spares.sav_object,
2499                    &spa->spa_spares.sav_config) != 0)
2500                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2502                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2503                spa_load_spares(spa);
2504                spa_config_exit(spa, SCL_ALL, FTAG);
2505        } else if (error == 0) {
2506                spa->spa_spares.sav_sync = B_TRUE;
```



```
2507        }

2509        /*
2510         * Load any level 2 ARC devices for this pool.
2511         */
2512        error = spa_dir_prop(spa, DMU_POOL_L2CACHE,
2513            &spa->spa_l2cache.sav_object);
2514        if (error != 0 && error != ENOENT)
2515                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2516        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2517                ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
2518                if (load_nvlist(spa, spa->spa_l2cache.sav_object,
2519                    &spa->spa_l2cache.sav_config) != 0)
2520                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2522                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2523                spa_load_l2cache(spa);
2524                spa_config_exit(spa, SCL_ALL, FTAG);
2525        } else if (error == 0) {
2526                spa->spa_l2cache.sav_sync = B_TRUE;
2527        }

2529        spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);

2531        error = spa_dir_prop(spa, DMU_POOL_PROPS, &spa->spa_pool_props_object);
2532        if (error && error != ENOENT)
2533                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2535        if (error == 0) {
2536                uint64_t autoreplace;

2538                spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
2539                spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
2540                spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
2541                spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
2542                spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
2543                spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO,
2544                    &spa->spa_dedup_ditto);

2546                spa->spa_autoreplace = (autoreplace != 0);
2547        }

2549        /*
2550         * If the 'autoreplace' property is set, then post a resource notifying
2551         * the ZFS DE that it should not issue any faults for unopenable
2552         * devices.  We also iterate over the vdevs, and post a sysevent for any
2553         * unopenable vdevs so that the normal autoreplace handler can take
2554         * over.
2555         */
2556        if (spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT) {
2557                spa_check_removed(spa->spa_root_vdev);
2558                /*
2559                 * For the import case, this is done in spa_import(), because
2560                 * at this point we're using the spare definitions from
2561                 * the MOS config, not necessarily from the userland config.
2562                 */
2563                if (state != SPA_LOAD_IMPORT) {
2564                        spa_aux_check_removed(&spa->spa_spares);
2565                        spa_aux_check_removed(&spa->spa_l2cache);
2566                }
2567        }

2569        /*
2570         * Load the vdev state for all toplevel vdevs.
2571         */
2572        vdev_load(rvd);
```

```
2574         /*
2575          * Propagate the leaf DTLs we just loaded all the way up the tree.
2576          */
2577         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2578         vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
2579         spa_config_exit(spa, SCL_ALL, FTAG);

2581         /*
2582          * Load the DDTs (dedup tables).
2583          */
2584         error = ddt_load(spa);
2585         if (error != 0)
2586                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2588         spa_update_dspace(spa);

2590         /*
2591          * Validate the config, using the MOS config to fill in any
2592          * information which might be missing.  If we fail to validate
2593          * the config then declare the pool unfit for use. If we're
2594          * assembling a pool from a split, the log is not transferred
2595          * over.
2596          */
2597         if (type != SPA_IMPORT_ASSEMBLE) {
2598                 nvlist_t *nvconfig;

2600                 if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2601                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2603                 if (!spa_config_valid(spa, nvconfig)) {
2604                         nvlist_free(nvconfig);
2605                         return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM,
2606                             ENXIO));
2607                 }
2608                 nvlist_free(nvconfig);

2610                 /*
2611                  * Now that we've validated the config, check the state of the
2612                  * root vdev.  If it can't be opened, it indicates one or
2613                  * more toplevel vdevs are faulted.
2614                  */
2615                 if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2616                         return (SET_ERROR(ENXIO));

2618                 if (spa_check_logs(spa)) {
2619                         *ereport = FM_EREPORT_ZFS_LOG_REPLAY;
2620                         return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
2621                 }
2622         }

2624         if (missing_feat_write) {
2625                 ASSERT(state == SPA_LOAD_TRYIMPORT);

2627                 /*
2628                  * At this point, we know that we can open the pool in
2629                  * read-only mode but not read-write mode. We now have enough
2630                  * information and can return to userland.
2631                  */
2632                 return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
2633         }

2635         /*
2636          * We've successfully opened the pool, verify that we're ready
2637          * to start pushing transactions.
2638          */
```



```
2639         if (state != SPA_LOAD_TRYIMPORT) {
2640                 if (error = spa_load_verify(spa))
2641                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2642                             error));
2643         }

2645         if (spa_writeable(spa) && (state == SPA_LOAD_RECOVER ||
2646             spa->spa_load_max_txg == UINT64_MAX)) {
2647                 dmu_tx_t *tx;
2648                 int need_update = B_FALSE;

2650                 ASSERT(state != SPA_LOAD_TRYIMPORT);

2652                 /*
2653                  * Claim log blocks that haven't been committed yet.
2654                  * This must all happen in a single txg.
2655                  * Note: spa_claim_max_txg is updated by spa_claim_notify(),
2656                  * invoked from zil_claim_log_block()'s i/o done callback.
2657                  * Price of rollback is that we abandon the log.
2658                  */
2659                 spa->spa_claiming = B_TRUE;

2661                 tx = dmu_tx_create_assigned(spa_get_dsl(spa),
2662                     spa_first_txg(spa));
2663                 (void) dmu_objset_find_parallel(spa_name(spa),
2663                 (void) dmu_objset_find(spa_name(spa),
2664                     zil_claim, tx, DS_FIND_CHILDREN);
2665                 dmu_tx_commit(tx);

2667                 spa->spa_claiming = B_FALSE;

2669                 spa_set_log_state(spa, SPA_LOG_GOOD);
2670                 spa->spa_sync_on = B_TRUE;
2671                 txg_sync_start(spa->spa_dsl_pool);

2673                 /*
2674                  * Wait for all claims to sync.  We sync up to the highest
2675                  * claimed log block birth time so that claimed log blocks
2676                  * don't appear to be from the future.  spa_claim_max_txg
2677                  * will have been set for us by either zil_check_log_chain()
2678                  * (invoked from spa_check_logs()) or zil_claim() above.
2679                  */
2680                 txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_txg);

2682                 /*
2683                  * If the config cache is stale, or we have uninitialized
2684                  * metaslabs (see spa_vdev_add()), then update the config.
2685                  *
2686                  * If this is a verbatim import, trust the current
2687                  * in-core spa_config and update the disk labels.
2688                  */
2689                 if (config_cache_txg != spa->spa_config_txg ||
2690                     state == SPA_LOAD_IMPORT ||
2691                     state == SPA_LOAD_RECOVER ||
2692                     (spa->spa_import_flags & ZFS_IMPORT_VERBATIM))
2693                         need_update = B_TRUE;

2695                 for (int c = 0; c < rvd->vdev_children; c++)
2696                         if (rvd->vdev_child[c]->vdev_ms_array == 0)
2697                                 need_update = B_TRUE;

2699                 /*
2700                  * Update the config cache asychronously in case we're the
2701                  * root pool, in which case the config cache isn't writable yet.
2702                  */
2703                 if (need_update)
```

```
2704                        spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);

2706                /*
2707                 * Check all DTLs to see if anything needs resilvering.
2708                 */
2709                if (!dsl_scan_resilvering(spa->spa_dsl_pool) &&
2710                    vdev_resilver_needed(rvd, NULL, NULL))
2711                        spa_async_request(spa, SPA_ASYNC_RESILVER);

2713                /*
2714                 * Log the fact that we booted up (so that we can detect if
2715                 * we rebooted in the middle of an operation).
2716                 */
2717                spa_history_log_version(spa, "open");

2719                /*
2720                 * Delete any inconsistent datasets.
2721                 */
2722                (void) dmu_objset_find(spa_name(spa),
2723                    dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);

2725                /*
2726                 * Clean up any stale temporary dataset userrefs.
2727                 */
2728                dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
2729        }

2731        return (0);
2732 }
```

*unchanged_portion_omitted_*

```
2841 /*
2842  * Pool Open/Import
2843  *
2844  * The import case is identical to an open except that the configuration is sent
2845  * down from userland, instead of grabbed from the configuration cache.  For the
2846  * case of an open, the pool configuration will exist in the
2847  * POOL_STATE_UNINITIALIZED state.
2848  *
2849  * The stats information (gen/count/ustats) is used to gather vdev statistics at
2850  * the same time open the pool, without having to keep around the spa_t in some
2851  * ambiguous state.
2852  */
2853 static int
2854 spa_open_common(const char *pool, spa_t **spapp, void *tag, nvlist_t *nvpolicy,
2855     nvlist_t **config, int lock)
2855     nvlist_t **config)
2856 {
2857        spa_t *spa;
2858        spa_load_state_t state = SPA_LOAD_OPEN;
2859        int error;
2860        int locked = B_FALSE;

2862        *spapp = NULL;

2864        /*
2865         * As disgusting as this is, we need to support recursive calls to this
2866         * function because dsl_dir_open() is called during spa_load(), and ends
2867         * up calling spa_open() again.  The real fix is to figure out how to
2868         * avoid dsl_dir_open() calling this in the first place.
2869         */
2870        if (lock && (mutex_owner(&spa_namespace_lock) != curthread)) {
2870        if (mutex_owner(&spa_namespace_lock) != curthread) {
2871                mutex_enter(&spa_namespace_lock);
2872                locked = B_TRUE;
2873        }
```



```
2875        if ((spa = spa_lookup(pool)) == NULL) {
2876                if (locked)
2877                        mutex_exit(&spa_namespace_lock);
2878                return (SET_ERROR(ENOENT));
2879        }

2881        if (spa->spa_state == POOL_STATE_UNINITIALIZED) {
2882                zpool_rewind_policy_t policy;

2884                zpool_get_rewind_policy(nvpolicy ? nvpolicy : spa->spa_config,
2885                    &policy);
2886                if (policy.zrp_request & ZPOOL_DO_REWIND)
2887                        state = SPA_LOAD_RECOVER;

2889                spa_activate(spa, spa_mode_global);

2891                if (state != SPA_LOAD_RECOVER)
2892                        spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;

2894                error = spa_load_best(spa, state, B_FALSE, policy.zrp_txg,
2895                    policy.zrp_request);

2897                if (error == EBADF) {
2898                        /*
2899                         * If vdev_validate() returns failure (indicated by
2900                         * EBADF), it indicates that one of the vdevs indicates
2901                         * that the pool has been exported or destroyed.  If
2902                         * this is the case, the config cache is out of sync and
2903                         * we should remove the pool from the namespace.
2904                         */
2905                        spa_unload(spa);
2906                        spa_deactivate(spa);
2907                        spa_config_sync(spa, B_TRUE, B_TRUE);
2908                        spa_remove(spa);
2909                        if (locked)
2910                                mutex_exit(&spa_namespace_lock);
2911                        return (SET_ERROR(ENOENT));
2912                }

2914                if (error) {
2915                        /*
2916                         * We can't open the pool, but we still have useful
2917                         * information: the state of each vdev after the
2918                         * attempted vdev_open().  Return this to the user.
2919                         */
2920                        if (config != NULL && spa->spa_config) {
2921                                VERIFY(nvlist_dup(spa->spa_config, config,
2922                                    KM_SLEEP) == 0);
2923                                VERIFY(nvlist_add_nvlist(*config,
2924                                    ZPOOL_CONFIG_LOAD_INFO,
2925                                    spa->spa_load_info) == 0);
2926                        }
2927                        spa_unload(spa);
2928                        spa_deactivate(spa);
2929                        spa->spa_last_open_failed = error;
2930                        if (locked)
2931                                mutex_exit(&spa_namespace_lock);
2932                        *spapp = NULL;
2933                        return (error);
2934                }
2935        }

2937        spa_open_ref(spa, tag);

2939        if (config != NULL)
```

```
2940                 *config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);

2942         /*
2943          * If we've recovered the pool, pass back any information we
2944          * gathered while doing the load.
2945          */
2946         if (state == SPA_LOAD_RECOVER) {
2947                 VERIFY(nvlist_add_nvlist(*config, ZPOOL_CONFIG_LOAD_INFO,
2948                     spa->spa_load_info) == 0);
2949         }

2951         if (locked) {
2952                 spa->spa_last_open_failed = 0;
2953                 spa->spa_last_ubsync_txg = 0;
2954                 spa->spa_load_txg = 0;
2955                 mutex_exit(&spa_namespace_lock);
2956         }

2958         *spapp = spa;

2960         return (0);
2961 }

2963 int
2964 spa_open_rewind(const char *name, spa_t **spapp, void *tag, nvlist_t *policy,
2965     nvlist_t **config)
2966 {
2967         return (spa_open_common(name, spapp, tag, policy, config, 1));
2967         return (spa_open_common(name, spapp, tag, policy, config));
2968 }

2970 int
2971 spa_open(const char *name, spa_t **spapp, void *tag)
2972 {
2973         return (spa_open_common(name, spapp, tag, NULL, NULL, 1));
2974 }

2976 int
2977 spa_open_lock(const char *name, spa_t **spapp, void *tag, int lock)
2978 {
2979         return (spa_open_common(name, spapp, tag, NULL, NULL, lock));
2973         return (spa_open_common(name, spapp, tag, NULL, NULL));
2980 }
```

_____*unchanged_portion_omitted_*

```
3155 int
3156 spa_get_stats(const char *name, nvlist_t **config,
3157     char *altroot, size_t buflen)
3158 {
3159         int error;
3160         spa_t *spa;

3162         *config = NULL;
3163         error = spa_open_common(name, &spa, FTAG, NULL, config, 1);
3157         error = spa_open_common(name, &spa, FTAG, NULL, config);

3165         if (spa != NULL) {
3166                 /*
3167                  * This still leaves a window of inconsistency where the spares
3168                  * or l2cache devices could change and the config would be
3169                  * self-inconsistent.
3170                  */
3171                 spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);

3173                 if (*config != NULL) {
3174                         uint64_t loadtimes[2];
```



```
3176                         loadtimes[0] = spa->spa_loaded_ts.tv_sec;
3177                         loadtimes[1] = spa->spa_loaded_ts.tv_nsec;
3178                         VERIFY(nvlist_add_uint64_array(*config,
3179                             ZPOOL_CONFIG_LOADED_TIME, loadtimes, 2) == 0);

3181                         VERIFY(nvlist_add_uint64(*config,
3182                             ZPOOL_CONFIG_ERRCOUNT,
3183                             spa_get_errlog_size(spa)) == 0);

3185                         if (spa_suspended(spa))
3186                                 VERIFY(nvlist_add_uint64(*config,
3187                                     ZPOOL_CONFIG_SUSPENDED,
3188                                     spa->spa_failmode) == 0);

3190                         spa_add_spares(spa, *config);
3191                         spa_add_l2cache(spa, *config);
3192                         spa_add_feature_stats(spa, *config);
3193                 }
3194         }

3196         /*
3197          * We want to get the alternate root even for faulted pools, so we cheat
3198          * and call spa_lookup() directly.
3199          */
3200         if (altroot) {
3201                 if (spa == NULL) {
3202                         mutex_enter(&spa_namespace_lock);
3203                         spa = spa_lookup(name);
3204                         if (spa)
3205                                 spa_altroot(spa, altroot, buflen);
3206                         else
3207                                 altroot[0] = '\0';
3208                         spa = NULL;
3209                         mutex_exit(&spa_namespace_lock);
3210                 } else {
3211                         spa_altroot(spa, altroot, buflen);
3212                 }
3213         }

3215         if (spa != NULL) {
3216                 spa_config_exit(spa, SCL_CONFIG, FTAG);
3217                 spa_close(spa, FTAG);
3218         }

3220         return (error);
3221 }
```

_____*unchanged_portion_omitted_*

****
48795 Thu Oct 16 19:15:51 2014
new/usr/src/uts/common/fs/zfs/spa_misc.c
zpool import speedup
****
_____***unchanged_portion_omitted_***

```
 448 /*
 449  * ==========================================================================
 450  * SPA namespace functions
 451  * ==========================================================================
 452  */

 454 /*
 455  * Lookup the named spa_t in the AVL tree.  The spa_namespace_lock must be held.
 456  * Returns NULL if no matching spa_t is found.
 457  */
 458 spa_t *
 459 spa_lookup(const char *name)
 460 {
 461         spa_t *search;
 461         static spa_t search;    /* spa_t is large; don't allocate on stack */
 462         spa_t *spa;
 463         avl_index_t where;
 464         char *cp;

 466         search = kmem_alloc(sizeof(*search), KM_SLEEP);
 466         ASSERT(MUTEX_HELD(&spa_namespace_lock));

 468         (void) strlcpy(search->spa_name, name, sizeof (search->spa_name));
 468         (void) strlcpy(search.spa_name, name, sizeof (search.spa_name));

 470         /*
 471          * If it's a full dataset name, figure out the pool name and
 472          * just use that.
 473          */
 474         cp = strpbrk(search->spa_name, "/@#");
 474         cp = strpbrk(search.spa_name, "/@#");
 475         if (cp != NULL)
 476                 *cp = '\0';

 478         spa = avl_find(&spa_namespace_avl, search, &where);
 479         kmem_free(search, sizeof(*search));
 478         spa = avl_find(&spa_namespace_avl, &search, &where);

 481         return (spa);
 482 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   29678 Thu Oct 16 19:15:51 2014
new/usr/src/uts/common/fs/zfs/sys/dmu.h
zpool import speedup
**********************************************************
______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_hold_nolock(const char *name, void *tag, objset_t **osp);
 267 #endif /* ! codereview */
 268 int dmu_objset_own(const char *name, dmu_objset_type_t type,
 269     boolean_t readonly, void *tag, objset_t **osp);
 270 void dmu_objset_rele(objset_t *os, void *tag);
 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 int dmu_objset_find_parallel(char *name, int func(const char *, void *),
 285     void *arg, int flags);
 286 #endif /* ! codereview */
 287 void dmu_objset_byteswap(void *buf, size_t size);
 288 int dsl_dataset_rename_snapshot(const char *fsname,
 289     const char *oldsnapname, const char *newsnapname, boolean_t recursive);

 291 typedef struct dmu_buf {
 292         uint64_t db_object;             /* object that this buffer is part of */
 293         uint64_t db_offset;             /* byte offset in this object */
```



```
 294         uint64_t db_size;               /* size of buffer in bytes */
 295         void *db_data;                  /* data in buffer */
 296 } dmu_buf_t;

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

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

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

 349 /*
 350  * Free an object from this objset.
 351  *
 352  * The object's data will be freed as well (ie. you don't need to call
 353  * dmu_free(object, 0, -1, tx)).
 354  *
 355  * The object need not be held in the transaction.
 356  *
 357  * If there are any holds on this object's buffers (via dmu_buf_hold()),
 358  * or tx holds on the object (via dmu_tx_hold_object()), you can not
 359  * 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,
    dmu_tx_t *tx);
int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
```

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

641 extern int zfs_prefetch_disable;

643 /*
644  * Asynchronously try to read in the data.
645  */
646 void dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset,
647     uint64_t len);

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

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

667 typedef struct dmu_object_type_info {
668         dmu_object_byteswap_t   ot_byteswap;
669         boolean_t               ot_metadata;
670         char                    *ot_name;
671 } dmu_object_type_info_t;

673 typedef struct dmu_object_byteswap_info {
674         arc_byteswap_func_t     *ob_func;
675         char                    *ob_name;
676 } dmu_object_byteswap_info_t;

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

681 /*
682  * Get information on a DMU object.
683  *
684  * Return 0 on success or ENOENT if object is not allocated.
685  *
686  * If doi is NULL, just indicates whether the object exists.
687  */
688 int dmu_object_info(objset_t *os, uint64_t object, dmu_object_info_t *doi);
689 /* Like dmu_object_info, but faster if you have a held dnode in hand. */
```



```
690 void dmu_object_info_from_dnode(struct dnode *dn, dmu_object_info_t *doi);
691 /* Like dmu_object_info, but faster if you have a held dbuf in hand. */
692 void dmu_object_info_from_db(dmu_buf_t *db, dmu_object_info_t *doi);
693 /*
694  * Like dmu_object_info_from_db, but faster still when you only care about
695  * the size.  This is specifically optimized for zfs_getattr().
696  */
697 void dmu_object_size_from_db(dmu_buf_t *db, uint32_t *blksize,
698     u_longlong_t *nblk512);

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

710 /*
711  * Get stats on a dataset.
712  */
713 void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);

715 /*
716  * Add entries to the nvlist for all the objset's properties.  See
717  * zfs_prop_table[] and zfs(1m) for details on the properties.
718  */
719 void dmu_objset_stats(objset_t *os, struct nvlist *nv);

721 /*
722  * Get the space usage statistics for statvfs().
723  *
724  * refdbytes is the amount of space "referenced" by this objset.
725  * availbytes is the amount of space available to this objset, taking
726  * into account quotas & reservations, assuming that no other objsets
727  * use the space first.  These values correspond to the 'referenced' and
728  * 'available' properties, described in the zfs(1m) manpage.
729  *
730  * usedobjs and availobjs are the number of objects currently allocated,
731  * and available.
732  */
733 void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
734     uint64_t *usedobjsp, uint64_t *availobjsp);

736 /*
737  * The fsid_guid is a 56-bit ID that can change to avoid collisions.
738  * (Contrast with the ds_guid which is a 64-bit ID that will never
739  * change, so there is a small probability that it will collide.)
740  */
741 uint64_t dmu_objset_fsid_guid(objset_t *os);

743 /*
744  * Get the [cm]time for an objset's snapshot dir
745  */
746 timestruc_t dmu_objset_snap_cmtime(objset_t *os);

748 int dmu_objset_is_snapshot(objset_t *os);

750 extern struct spa *dmu_objset_spa(objset_t *os);
751 extern struct zilog *dmu_objset_zil(objset_t *os);
752 extern struct dsl_pool *dmu_objset_pool(objset_t *os);
753 extern struct dsl_dataset *dmu_objset_ds(objset_t *os);
754 extern void dmu_objset_name(objset_t *os, char *buf);
755 extern dmu_objset_type_t dmu_objset_type(objset_t *os);
```

```
extern uint64_t dmu_objset_id(objset_t *os);
extern zfs_sync_type_t dmu_objset_syncprop(objset_t *os);
extern zfs_logbias_op_t dmu_objset_logbias(objset_t *os);
extern int dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
    uint64_t *id, uint64_t *offp, boolean_t *case_conflict);
extern int dmu_snapshot_realname(objset_t *os, char *name, char *real,
    int maxlen, boolean_t *conflict);
extern int dmu_dir_list_next(objset_t *os, int namelen, char *name,
    uint64_t *idp, uint64_t *offp);

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

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

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

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

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

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

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

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

int dmu_diff(const char *tosnap_name, const char *fromsnap_name,
```



```
    struct vnode *vp, offset_t *offp);

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

extern int zfs_mdcomp_disable;

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_DMU_H */
```

```
**********************************************************
    5939 Thu Oct 16 19:15:51 2014
new/usr/src/uts/common/fs/zfs/sys/dmu_objset.h
zpool import speedup
**********************************************************
______unchanged_portion_omitted_

 123 #define DMU_META_OBJSET         0
 124 #define DMU_META_DNODE_OBJECT   0
 125 #define DMU_OBJECT_IS_SPECIAL(obj) ((int64_t)(obj) <= 0)
 126 #define DMU_META_DNODE(os)      ((os)->os_meta_dnode.dnh_dnode)
 127 #define DMU_USERUSED_DNODE(os)  ((os)->os_userused_dnode.dnh_dnode)
 128 #define DMU_GROUPUSED_DNODE(os) ((os)->os_groupused_dnode.dnh_dnode)

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

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

 136 /* called from zpl */
 137 int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
 138 int dmu_objset_hold_nolock(const char *name, void *tag, objset_t **osp);
 139 #endif /* ! codereview */
 140 int dmu_objset_own(const char *name, dmu_objset_type_t type,
 141     boolean_t readonly, void *tag, objset_t **osp);
 142 int dmu_objset_own_nolock(const char *name, dmu_objset_type_t type,
 143     boolean_t readonly, void *tag, objset_t **osp);
 144 #endif /* ! codereview */
 145 void dmu_objset_refresh_ownership(objset_t *os, void *tag);
 146 void dmu_objset_rele(objset_t *os, void *tag);
 147 void dmu_objset_disown(objset_t *os, void *tag);
 148 int dmu_objset_from_ds(struct dsl_dataset *ds, objset_t **osp);

 150 void dmu_objset_stats(objset_t *os, nvlist_t *nv);
 151 void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);
 152 void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
 153     uint64_t *usedobjsp, uint64_t *availobjsp);
 154 uint64_t dmu_objset_fsid_guid(objset_t *os);
 155 int dmu_objset_find_dp(struct dsl_pool *dp, uint64_t ddobj,
 156     int func(struct dsl_pool *, struct dsl_dataset *, void *),
 157     void *arg, int flags);
 158 int dmu_objset_prefetch(const char *name, void *arg);
 159 void dmu_objset_evict_dbufs(objset_t *os);
 160 timestruc_t dmu_objset_snap_cmtime(objset_t *os);

 162 /* called from dsl */
 163 void dmu_objset_sync(objset_t *os, zio_t *zio, dmu_tx_t *tx);
 164 boolean_t dmu_objset_is_dirty(objset_t *os, uint64_t txg);
 165 objset_t *dmu_objset_create_impl(spa_t *spa, struct dsl_dataset *ds,
 166     blkptr_t *bp, dmu_objset_type_t type, dmu_tx_t *tx);
 167 int dmu_objset_open_impl(spa_t *spa, struct dsl_dataset *ds, blkptr_t *bp,
 168     objset_t **osp);
 169 void dmu_objset_evict(objset_t *os);
 170 void dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx);
 171 void dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx);
 172 boolean_t dmu_objset_userused_enabled(objset_t *os);
 173 int dmu_objset_userspace_upgrade(objset_t *os);
 174 boolean_t dmu_objset_userspace_present(objset_t *os);
 175 int dmu_fsname(const char *snapname, char *buf);

 177 void dmu_objset_init(void);
 178 void dmu_objset_fini(void);

 180 #ifdef  __cplusplus
 181 }
```



```
 182 #endif

 184 #endif /* _SYS_DMU_OBJSET_H */
```

****
5558 Thu Oct 16 19:15:51 2014
new/usr/src/uts/common/fs/zfs/sys/dsl_pool.h
zpool import speedup
****
_____***unchanged_portion_omitted_***

```
 135 int dsl_pool_init(spa_t *spa, uint64_t txg, dsl_pool_t **dpp);
 136 int dsl_pool_open(dsl_pool_t *dp);
 137 void dsl_pool_close(dsl_pool_t *dp);
 138 dsl_pool_t *dsl_pool_create(spa_t *spa, nvlist_t *zplprops, uint64_t txg);
 139 void dsl_pool_sync(dsl_pool_t *dp, uint64_t txg);
 140 void dsl_pool_sync_done(dsl_pool_t *dp, uint64_t txg);
 141 int dsl_pool_sync_context(dsl_pool_t *dp);
 142 uint64_t dsl_pool_adjustedsize(dsl_pool_t *dp, boolean_t netfree);
 143 uint64_t dsl_pool_adjustedfree(dsl_pool_t *dp, boolean_t netfree);
 144 void dsl_pool_dirty_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
 145 void dsl_pool_undirty_space(dsl_pool_t *dp, int64_t space, uint64_t txg);
 146 void dsl_free(dsl_pool_t *dp, uint64_t txg, const blkptr_t *bpp);
 147 void dsl_free_sync(zio_t *pio, dsl_pool_t *dp, uint64_t txg,
 148     const blkptr_t *bpp);
 149 void dsl_pool_create_origin(dsl_pool_t *dp, dmu_tx_t *tx);
 150 void dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx);
 151 void dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx);
 152 void dsl_pool_mos_diduse_space(dsl_pool_t *dp,
 153     int64_t used, int64_t comp, int64_t uncomp);
 154 void dsl_pool_config_enter(dsl_pool_t *dp, void *tag);
 155 void dsl_pool_config_exit(dsl_pool_t *dp, void *tag);
 156 boolean_t dsl_pool_config_held(dsl_pool_t *dp);
 157 boolean_t dsl_pool_need_dirty_delay(dsl_pool_t *dp);

 159 taskq_t *dsl_pool_vnrele_taskq(dsl_pool_t *dp);

 161 int dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj,
 162     const char *tag, uint64_t now, dmu_tx_t *tx);
 163 int dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj,
 164     const char *tag, dmu_tx_t *tx);
 165 void dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp);
 166 int dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **);
 167 int dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp);
 168 int dsl_pool_hold_lock(const char *name, void *tag, dsl_pool_t **dp, int lock);
 169 #endif /* ! codereview */
 170 void dsl_pool_rele(dsl_pool_t *dp, void *tag);
 171 void dsl_pool_rele_spa(dsl_pool_t *dp, void *tag);
 172 #endif /* ! codereview */

 174 #ifdef  __cplusplus
 175 }
 176 #endif

 178 #endif /* _SYS_DSL_POOL_H */
```

```
**********************************************************
   32150 Thu Oct 16 19:15:52 2014
new/usr/src/uts/common/fs/zfs/sys/spa.h
zpool import speedup
**********************************************************
______unchanged_portion_omitted_

 571 /* state manipulation functions */
 572 extern int spa_open(const char *pool, spa_t **, void *tag);
 573 extern int spa_open_lock(const char *pool, spa_t **, void *tag, int lock);
 574 #endif /* ! codereview */
 575 extern int spa_open_rewind(const char *pool, spa_t **, void *tag,
 576     nvlist_t *policy, nvlist_t **config);
 577 extern int spa_get_stats(const char *pool, nvlist_t **config, char *altroot,
 578     size_t buflen);
 579 extern int spa_create(const char *pool, nvlist_t *config, nvlist_t *props,
 580     nvlist_t *zplprops);
 581 extern int spa_import_rootpool(char *devpath, char *devid);
 582 extern int spa_import(const char *pool, nvlist_t *config, nvlist_t *props,
 583     uint64_t flags);
 584 extern nvlist_t *spa_tryimport(nvlist_t *tryconfig);
 585 extern int spa_destroy(char *pool);
 586 extern int spa_export(char *pool, nvlist_t **oldconfig, boolean_t force,
 587     boolean_t hardforce);
 588 extern int spa_reset(char *pool);
 589 extern void spa_async_request(spa_t *spa, int flag);
 590 extern void spa_async_unrequest(spa_t *spa, int flag);
 591 extern void spa_async_suspend(spa_t *spa);
 592 extern void spa_async_resume(spa_t *spa);
 593 extern spa_t *spa_inject_addref(char *pool);
 594 extern void spa_inject_delref(spa_t *spa);
 595 extern void spa_scan_stat_init(spa_t *spa);
 596 extern int spa_scan_get_stats(spa_t *spa, pool_scan_stat_t *ps);

 598 #define SPA_ASYNC_CONFIG_UPDATE 0x01
 599 #define SPA_ASYNC_REMOVE        0x02
 600 #define SPA_ASYNC_PROBE         0x04
 601 #define SPA_ASYNC_RESILVER_DONE 0x08
 602 #define SPA_ASYNC_RESILVER      0x10
 603 #define SPA_ASYNC_AUTOEXPAND    0x20
 604 #define SPA_ASYNC_REMOVE_DONE   0x40
 605 #define SPA_ASYNC_REMOVE_STOP   0x80

 607 /*
 608  * Controls the behavior of spa_vdev_remove().
 609  */
 610 #define SPA_REMOVE_UNSPARE      0x01
 611 #define SPA_REMOVE_DONE         0x02

 613 /* device manipulation */
 614 extern int spa_vdev_add(spa_t *spa, nvlist_t *nvroot);
 615 extern int spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot,
 616     int replacing);
 617 extern int spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid,
 618     int replace_done);
 619 extern int spa_vdev_remove(spa_t *spa, uint64_t guid, boolean_t unspare);
 620 extern boolean_t spa_vdev_remove_active(spa_t *spa);
 621 extern int spa_vdev_setpath(spa_t *spa, uint64_t guid, const char *newpath);
 622 extern int spa_vdev_setfru(spa_t *spa, uint64_t guid, const char *newfru);
 623 extern int spa_vdev_split_mirror(spa_t *spa, char *newname, nvlist_t *config,
 624     nvlist_t *props, boolean_t exp);

 626 /* spare state (which is global across all pools) */
 627 extern void spa_spare_add(vdev_t *vd);
 628 extern void spa_spare_remove(vdev_t *vd);
 629 extern boolean_t spa_spare_exists(uint64_t guid, uint64_t *pool, int *refcnt);
```



```
 630 extern void spa_spare_activate(vdev_t *vd);

 632 /* L2ARC state (which is global across all pools) */
 633 extern void spa_l2cache_add(vdev_t *vd);
 634 extern void spa_l2cache_remove(vdev_t *vd);
 635 extern boolean_t spa_l2cache_exists(uint64_t guid, uint64_t *pool);
 636 extern void spa_l2cache_activate(vdev_t *vd);
 637 extern void spa_l2cache_drop(spa_t *spa);

 639 /* scanning */
 640 extern int spa_scan(spa_t *spa, pool_scan_func_t func);
 641 extern int spa_scan_stop(spa_t *spa);

 643 /* spa syncing */
 644 extern void spa_sync(spa_t *spa, uint64_t txg); /* only for DMU use */
 645 extern void spa_sync_allpools(void);

 647 /* spa namespace global mutex */
 648 extern kmutex_t spa_namespace_lock;

 650 /*
 651  * SPA configuration functions in spa_config.c
 652  */

 654 #define SPA_CONFIG_UPDATE_POOL  0
 655 #define SPA_CONFIG_UPDATE_VDEVS 1

 657 extern void spa_config_sync(spa_t *, boolean_t, boolean_t);
 658 extern void spa_config_load(void);
 659 extern nvlist_t *spa_all_configs(uint64_t *);
 660 extern void spa_config_set(spa_t *spa, nvlist_t *config);
 661 extern nvlist_t *spa_config_generate(spa_t *spa, vdev_t *vd, uint64_t txg,
 662     int getstats);
 663 extern void spa_config_update(spa_t *spa, int what);

 665 /*
 666  * Miscellaneous SPA routines in spa_misc.c
 667  */

 669 /* Namespace manipulation */
 670 extern spa_t *spa_lookup(const char *name);
 671 extern spa_t *spa_add(const char *name, nvlist_t *config, const char *altroot);
 672 extern void spa_remove(spa_t *spa);
 673 extern spa_t *spa_next(spa_t *prev);

 675 /* Refcount functions */
 676 extern void spa_open_ref(spa_t *spa, void *tag);
 677 extern void spa_close(spa_t *spa, void *tag);
 678 extern boolean_t spa_refcount_zero(spa_t *spa);

 680 #define SCL_NONE        0x00
 681 #define SCL_CONFIG      0x01
 682 #define SCL_STATE       0x02
 683 #define SCL_L2ARC       0x04            /* hack until L2ARC 2.0 */
 684 #define SCL_ALLOC       0x08
 685 #define SCL_ZIO         0x10
 686 #define SCL_FREE        0x20
 687 #define SCL_VDEV        0x40
 688 #define SCL_LOCKS       7
 689 #define SCL_ALL         ((1 << SCL_LOCKS) - 1)
 690 #define SCL_STATE_ALL   (SCL_STATE | SCL_L2ARC | SCL_ZIO)

 692 /* Pool configuration locks */
 693 extern int spa_config_tryenter(spa_t *spa, int locks, void *tag, krw_t rw);
 694 extern void spa_config_enter(spa_t *spa, int locks, void *tag, krw_t rw);
 695 extern void spa_config_exit(spa_t *spa, int locks, void *tag);
```

```
696 extern int spa_config_held(spa_t *spa, int locks, krw_t rw);

698 /* Pool vdev add/remove lock */
699 extern uint64_t spa_vdev_enter(spa_t *spa);
700 extern uint64_t spa_vdev_config_enter(spa_t *spa);
701 extern void spa_vdev_config_exit(spa_t *spa, vdev_t *vd, uint64_t txg,
702     int error, char *tag);
703 extern int spa_vdev_exit(spa_t *spa, vdev_t *vd, uint64_t txg, int error);

705 /* Pool vdev state change lock */
706 extern void spa_vdev_state_enter(spa_t *spa, int oplock);
707 extern int spa_vdev_state_exit(spa_t *spa, vdev_t *vd, int error);

709 /* Log state */
710 typedef enum spa_log_state {
711         SPA_LOG_UNKNOWN = 0,    /* unknown log state */
712         SPA_LOG_MISSING,        /* missing log(s) */
713         SPA_LOG_CLEAR,          /* clear the log(s) */
714         SPA_LOG_GOOD,           /* log(s) are good */
715 } spa_log_state_t;

717 extern spa_log_state_t spa_get_log_state(spa_t *spa);
718 extern void spa_set_log_state(spa_t *spa, spa_log_state_t state);
719 extern int spa_offline_log(spa_t *spa);

721 /* Log claim callback */
722 extern void spa_claim_notify(zio_t *zio);

724 /* Accessor functions */
725 extern boolean_t spa_shutting_down(spa_t *spa);
726 extern struct dsl_pool *spa_get_dsl(spa_t *spa);
727 extern boolean_t spa_is_initializing(spa_t *spa);
728 extern blkptr_t *spa_get_rootblkptr(spa_t *spa);
729 extern void spa_set_rootblkptr(spa_t *spa, const blkptr_t *bp);
730 extern void spa_altroot(spa_t *, char *, size_t);
731 extern int spa_sync_pass(spa_t *spa);
732 extern char *spa_name(spa_t *spa);
733 extern uint64_t spa_guid(spa_t *spa);
734 extern uint64_t spa_load_guid(spa_t *spa);
735 extern uint64_t spa_last_synced_txg(spa_t *spa);
736 extern uint64_t spa_first_txg(spa_t *spa);
737 extern uint64_t spa_syncing_txg(spa_t *spa);
738 extern uint64_t spa_version(spa_t *spa);
739 extern pool_state_t spa_state(spa_t *spa);
740 extern spa_load_state_t spa_load_state(spa_t *spa);
741 extern uint64_t spa_freeze_txg(spa_t *spa);
742 extern uint64_t spa_get_asize(spa_t *spa, uint64_t lsize);
743 extern uint64_t spa_get_dspace(spa_t *spa);
744 extern void spa_update_dspace(spa_t *spa);
745 extern uint64_t spa_version(spa_t *spa);
746 extern boolean_t spa_deflate(spa_t *spa);
747 extern metaslab_class_t *spa_normal_class(spa_t *spa);
748 extern metaslab_class_t *spa_log_class(spa_t *spa);
749 extern int spa_max_replication(spa_t *spa);
750 extern int spa_prev_software_version(spa_t *spa);
751 extern int spa_busy(void);
752 extern uint8_t spa_get_failmode(spa_t *spa);
753 extern boolean_t spa_suspended(spa_t *spa);
754 extern uint64_t spa_bootfs(spa_t *spa);
755 extern uint64_t spa_delegation(spa_t *spa);
756 extern objset_t *spa_meta_objset(spa_t *spa);
757 extern uint64_t spa_deadman_synctime(spa_t *spa);

759 /* Miscellaneous support routines */
760 extern void spa_activate_mos_feature(spa_t *spa, const char *feature,
761     dmu_tx_t *tx);
```



```
762 extern void spa_deactivate_mos_feature(spa_t *spa, const char *feature);
763 extern int spa_rename(const char *oldname, const char *newname);
764 extern spa_t *spa_by_guid(uint64_t pool_guid, uint64_t device_guid);
765 extern boolean_t spa_guid_exists(uint64_t pool_guid, uint64_t device_guid);
766 extern char *spa_strdup(const char *);
767 extern void spa_strfree(char *);
768 extern uint64_t spa_get_random(uint64_t range);
769 extern uint64_t spa_generate_guid(spa_t *spa);
770 extern void snprintf_blkptr(char *buf, size_t buflen, const blkptr_t *bp);
771 extern void spa_freeze(spa_t *spa);
772 extern int spa_change_guid(spa_t *spa);
773 extern void spa_upgrade(spa_t *spa, uint64_t version);
774 extern void spa_evict_all(void);
775 extern vdev_t *spa_lookup_by_guid(spa_t *spa, uint64_t guid,
776     boolean_t l2cache);
777 extern boolean_t spa_has_spare(spa_t *, uint64_t guid);
778 extern uint64_t dva_get_dsize_sync(spa_t *spa, const dva_t *dva);
779 extern uint64_t bp_get_dsize_sync(spa_t *spa, const blkptr_t *bp);
780 extern uint64_t bp_get_dsize(spa_t *spa, const blkptr_t *bp);
781 extern boolean_t spa_has_slogs(spa_t *spa);
782 extern boolean_t spa_is_root(spa_t *spa);
783 extern boolean_t spa_writeable(spa_t *spa);

785 extern int spa_mode(spa_t *spa);
786 extern uint64_t strtonum(const char *str, char **nptr);

788 extern char *spa_his_ievent_table[];

790 extern void spa_history_create_obj(spa_t *spa, dmu_tx_t *tx);
791 extern int spa_history_get(spa_t *spa, uint64_t *offset, uint64_t *len_read,
792     char *his_buf);
793 extern int spa_history_log(spa_t *spa, const char *his_buf);
794 extern int spa_history_log_nvl(spa_t *spa, nvlist_t *nvl);
795 extern void spa_history_log_version(spa_t *spa, const char *operation);
796 extern void spa_history_log_internal(spa_t *spa, const char *operation,
797     dmu_tx_t *tx, const char *fmt, ...);
798 extern void spa_history_log_internal_ds(struct dsl_dataset *ds, const char *op,
799     dmu_tx_t *tx, const char *fmt, ...);
800 extern void spa_history_log_internal_dd(dsl_dir_t *dd, const char *operation,
801     dmu_tx_t *tx, const char *fmt, ...);

803 /* error handling */
804 struct zbookmark;
805 extern void spa_log_error(spa_t *spa, zio_t *zio);
806 extern void zfs_ereport_post(const char *class, spa_t *spa, vdev_t *vd,
807     zio_t *zio, uint64_t stateoroffset, uint64_t length);
808 extern void zfs_post_remove(spa_t *spa, vdev_t *vd);
809 extern void zfs_post_state_change(spa_t *spa, vdev_t *vd);
810 extern void zfs_post_autoreplace(spa_t *spa, vdev_t *vd);
811 extern uint64_t spa_get_errlog_size(spa_t *spa);
812 extern int spa_get_errlog(spa_t *spa, void *uaddr, size_t *count);
813 extern void spa_errlog_rotate(spa_t *spa);
814 extern void spa_errlog_drain(spa_t *spa);
815 extern void spa_errlog_sync(spa_t *spa, uint64_t txg);
816 extern void spa_get_errlists(spa_t *spa, avl_tree_t *last, avl_tree_t *scrub);

818 /* vdev cache */
819 extern void vdev_cache_stat_init(void);
820 extern void vdev_cache_stat_fini(void);

822 /* Initialization and termination */
823 extern void spa_init(int flags);
824 extern void spa_fini(void);
825 extern void spa_boot_init();

827 /* properties */
```

```
828 extern int spa_prop_set(spa_t *spa, nvlist_t *nvp);
829 extern int spa_prop_get(spa_t *spa, nvlist_t **nvp);
830 extern void spa_prop_clear_bootfs(spa_t *spa, uint64_t obj, dmu_tx_t *tx);
831 extern void spa_configfile_set(spa_t *, nvlist_t *, boolean_t);

833 /* asynchronous event notification */
834 extern void spa_event_notify(spa_t *spa, vdev_t *vdev, const char *name);

836 #ifdef ZFS_DEBUG
837 #define dprintf_bp(bp, fmt, ...) do {                           \
838         if (zfs_flags & ZFS_DEBUG_DPRINTF) {                    \
839         char *__blkbuf = kmem_alloc(BP_SPRINTF_LEN, KM_SLEEP);  \
840         snprintf_blkptr(__blkbuf, BP_SPRINTF_LEN, (bp));        \
841         dprintf(fmt " %s\n", __VA_ARGS__, __blkbuf);            \
842         kmem_free(__blkbuf, BP_SPRINTF_LEN);                    \
843         } \
844 _NOTE(CONSTCOND) } while (0)
845 #else
846 #define dprintf_bp(bp, fmt, ...)
847 #endif

849 extern boolean_t spa_debug_enabled(spa_t *spa);
850 #define spa_dbgmsg(spa, ...)                    \
851 {                                               \
852         if (spa_debug_enabled(spa))             \
853                 zfs_dbgmsg(__VA_ARGS__);        \
854 }

856 extern int spa_mode_global;                     /* mode, e.g. FREAD | FWRITE */

858 #ifdef  __cplusplus
859 }
860 #endif

862 #endif  /* _SYS_SPA_H */
```

```
**********************************************************
    5825 Thu Oct 16 19:15:52 2014
new/usr/src/uts/common/fs/zfs/sys/vdev.h
zpool import speedup
**********************************************************
______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 psise, 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 */
```

```
**********************************************************
   89413 Thu Oct 16 19:15:52 2014
new/usr/src/uts/common/fs/zfs/vdev.c
zpool import speedup
**********************************************************
______unchanged_portion_omitted_
```

```
175 int
176 vdev_count_leaves_impl(vdev_t *vd)
177 {
178         vdev_t *mvd;
179         int n = 0;

181         if (vd->vdev_children == 0)
182                 return (1);

184         for (int c = 0; c < vd->vdev_children; c++)
185                 n += vdev_count_leaves_impl(vd->vdev_child[c]);

187         return (n);
188 }

190 int
191 vdev_count_leaves(spa_t *spa)
192 {
193         return (vdev_count_leaves_impl(spa->spa_root_vdev));
194 }

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;

204         ASSERT(spa_config_held(cvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);
205         ASSERT(cvd->vdev_parent == NULL);

207         cvd->vdev_parent = pvd;

209         if (pvd == NULL)
210                 return;

212         ASSERT(id >= pvd->vdev_children || pvd->vdev_child[id] == NULL);

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 *);

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         }

224         pvd->vdev_child = newchild;
225         pvd->vdev_child[id] = cvd;

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 }

237 void
238 vdev_remove_child(vdev_t *pvd, vdev_t *cvd)
239 {
240         int c;
241         uint_t id = cvd->vdev_id;

243         ASSERT(cvd->vdev_parent == pvd);

245         if (pvd == NULL)
246                 return;

248         ASSERT(id < pvd->vdev_children);
249         ASSERT(pvd->vdev_child[id] == cvd);

251         pvd->vdev_child[id] = NULL;
252         cvd->vdev_parent = NULL;

254         for (c = 0; c < pvd->vdev_children; c++)
255                 if (pvd->vdev_child[c])
256                         break;

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         }

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 }

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;

281         ASSERT(spa_config_held(pvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);

283         for (int c = newc = 0; c < oldc; c++)
284                 if (pvd->vdev_child[c])
285                         newc++;

287         newchild = kmem_alloc(newc * sizeof (vdev_t *), KM_SLEEP);

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                         /*
```

```
                         * Previous versions could only support 1 or 2 parity
                         * device.
                         */
                        if (nparity > 1 &&
                            spa_version(spa) < SPA_VERSION_RAIDZ2)
                                return (SET_ERROR(ENOTSUP));
                        if (nparity > 2 &&
                            spa_version(spa) < SPA_VERSION_RAIDZ3)
                                return (SET_ERROR(ENOTSUP));
                } else {
                        /*
                         * We require the parity to be specified for SPAs that
                         * support multiple parity levels.
                         */
                        if (spa_version(spa) >= SPA_VERSION_RAIDZ2)
                                return (SET_ERROR(EINVAL));
                        /*
                         * Otherwise, we default to 1 parity device for RAID-Z.
                         */
                        nparity = 1;
                }
        } else {
                nparity = 0;
        }
        ASSERT(nparity != -1ULL);

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

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

        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PATH, &vd->vdev_path) == 0)
                vd->vdev_path = spa_strdup(vd->vdev_path);
        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_DEVID, &vd->vdev_devid) == 0)
                vd->vdev_devid = spa_strdup(vd->vdev_devid);
        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PHYS_PATH,
            &vd->vdev_physpath) == 0)
                vd->vdev_physpath = spa_strdup(vd->vdev_physpath);
        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_FRU, &vd->vdev_fru) == 0)
                vd->vdev_fru = spa_strdup(vd->vdev_fru);

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

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

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

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



```
        /*
         * If we're a top-level vdev, try to load the allocation parameters.
         */
        if (parent && !parent->vdev_parent &&
            (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_SPLIT)) {
                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_ARRAY,
                    &vd->vdev_ms_array);
                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_SHIFT,
                    &vd->vdev_ms_shift);
                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ASIZE,
                    &vd->vdev_asize);
                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_REMOVING,
                    &vd->vdev_removing);
        }

        if (parent && !parent->vdev_parent && alloctype != VDEV_ALLOC_ATTACH) {
                ASSERT(alloctype == VDEV_ALLOC_LOAD ||
                    alloctype == VDEV_ALLOC_ADD ||
                    alloctype == VDEV_ALLOC_SPLIT ||
                    alloctype == VDEV_ALLOC_ROOTPOOL);
                vd->vdev_mg = metaslab_group_create(islog ?
                    spa_log_class(spa) : spa_normal_class(spa), vd);
        }

        /*
         * If we're a leaf vdev, try to load the DTL object and other state.
         */
        if (vd->vdev_ops->vdev_op_leaf &&
            (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_L2CACHE ||
            alloctype == VDEV_ALLOC_ROOTPOOL)) {
                if (alloctype == VDEV_ALLOC_LOAD) {
                        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_DTL,
                            &vd->vdev_dtl_object);
                        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_UNSPARE,
                            &vd->vdev_unspare);
                }

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

                        if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_IS_SPARE,
                            &spare) == 0 && spare)
                                spa_spare_add(vd);
                }

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

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

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

```
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;
690
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;
```

```
696         svd->vdev_mg = NULL;
697         svd->vdev_ms = NULL;
699         if (tvd->vdev_mg != NULL)
700                 tvd->vdev_mg->mg_vd = tvd;
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;
706         svd->vdev_stat.vs_alloc = 0;
707         svd->vdev_stat.vs_space = 0;
708         svd->vdev_stat.vs_dspace = 0;
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         }
719         if (list_link_active(&svd->vdev_config_dirty_node)) {
720                 vdev_config_clean(svd);
721                 vdev_config_dirty(tvd);
722         }
724         if (list_link_active(&svd->vdev_state_dirty_node)) {
725                 vdev_state_clean(svd);
726                 vdev_state_dirty(tvd);
727         }
729         tvd->vdev_deflate_ratio = svd->vdev_deflate_ratio;
730         svd->vdev_deflate_ratio = 0;
732         tvd->vdev_islog = svd->vdev_islog;
733         svd->vdev_islog = 0;
734 }
736 static void
737 vdev_top_update(vdev_t *tvd, vdev_t *vd)
738 {
739         if (vd == NULL)
740                 return;
742         vd->vdev_top = tvd;
744         for (int c = 0; c < vd->vdev_children; c++)
745                 vdev_top_update(tvd, vd->vdev_child[c]);
746 }
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;
758         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
760         mvd = vdev_alloc_common(spa, cvd->vdev_id, 0, ops);
```



```
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;
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);
775         if (mvd == mvd->vdev_top)
776                 vdev_top_transfer(cvd, mvd);
778         return (mvd);
779 }
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;
790         ASSERT(spa_config_held(cvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);
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;
798         vdev_remove_child(mvd, cvd);
799         vdev_remove_child(pvd, mvd);
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);
817         if (cvd == cvd->vdev_top)
818                 vdev_top_transfer(mvd, cvd);
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;

835         ASSERT(txg == 0 || spa_config_held(spa, SCL_ALLOC, RW_WRITER));

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);

843         ASSERT(!vd->vdev_ishole);

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);

854         ASSERT(oldc <= newc);

856         mspp = kmem_zalloc(newc * sizeof (*mspp), KM_SLEEP);

858         if (oldc != 0) {
859                 bcopy(vd->vdev_ms, mspp, oldc * sizeof (*mspp));
860                 kmem_free(vd->vdev_ms, oldc * sizeof (*mspp));
861         }

863         vd->vdev_ms = mspp;
864         vd->vdev_ms_count = newc;

866         for (m = oldc; m < newc; m++) {
867                 uint64_t object = 0;

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         }

879         if (txg == 0)
880                 spa_config_enter(spa, SCL_ALLOC, FTAG, RW_WRITER);

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);

893         return (0);
```



```
894 }

896 void
897 vdev_metaslab_fini(vdev_t *vd)
898 {
899         uint64_t m;
900         uint64_t count = vd->vdev_ms_count;

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];

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 }

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;

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;

928         ASSERT(vd->vdev_probe_zio != NULL);

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;

948                 vd->vdev_cant_read |= !vps->vps_readable;
949                 vd->vdev_cant_write |= !vps->vps_writeable;

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                 }
```

```
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);

966                 while ((pio = zio_walk_parents(zio)) != NULL)
967                         if (!vdev_accessible(vd, pio))
968                                 pio->io_error = SET_ERROR(ENXIO);

970                 kmem_free(vps, sizeof (*vps));
971         }
972 }

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;

988         ASSERT(vd->vdev_ops->vdev_op_leaf);

990         /*
991          * Don't probe the probe.
992          */
993         if (zio && (zio->io_flags & ZIO_FLAG_PROBE))
994                 return (NULL);

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);

1003        if ((pio = vd->vdev_probe_zio) == NULL) {
1004                vps = kmem_zalloc(sizeof (*vps), KM_SLEEP);

1006                vps->vps_flags = ZIO_FLAG_CANFAIL | ZIO_FLAG_PROBE |
1007                    ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_AGGREGATE |
1008                    ZIO_FLAG_TRYHARD;

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                }

1032                vd->vdev_probe_zio = pio = zio_null(NULL, spa, vd,
1033                    vdev_probe_done, vps,
1034                    vps->vps_flags | ZIO_FLAG_DONT_PROPAGATE);

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        }

1046        if (zio != NULL)
1047                zio_add_child(zio, pio);

1049        mutex_exit(&vd->vdev_probe_lock);

1051        if (vps == NULL) {
1052                ASSERT(zio != NULL);
1053                return (NULL);
1054        }

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        }

1065        if (zio == NULL)
1066                return (pio);

1068        zio_nowait(pio);
1069        return (NULL);
1070 }

1072 static void
1073 vdev_open_child(void *arg)
1074 {
1075         vdev_t *vd = arg;

1077         vd->vdev_open_thread = curthread;
1078         vd->vdev_open_error = vdev_open(vd);
1079         vd->vdev_open_thread = NULL;
1080 }

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 }
1093
1094 void
1095 vdev_open_children(vdev_t *vd)
1096 {
1097         taskq_t *tq;
1098         int children = vd->vdev_children;
1099
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);
1113
1114         for (int c = 0; c < children; c++)
1115                 VERIFY(taskq_dispatch(tq, vdev_open_child, vd->vdev_child[c],
1116                     TQ_SLEEP) != NULL);
1117
1118         taskq_destroy(tq);
1119 }
1120
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;
1133
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);
1139
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);
1144
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         }
1161
1162         error = vd->vdev_ops->vdev_op_open(vd, &osize, &max_osize, &ashift);
1163
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);
1171
1172         if (error) {
1173                 if (vd->vdev_removed &&
1174                     vd->vdev_stat.vs_aux != VDEV_AUX_OPEN_FAILED)
1175                         vd->vdev_removed = B_FALSE;
1176
1177                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1178                     vd->vdev_stat.vs_aux);
1179                 return (error);
1180         }
1181
1182         vd->vdev_removed = B_FALSE;
1183
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         }
1196
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         }
1204
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);
1210
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         }
1218
1219         osize = P2ALIGN(osize, (uint64_t)sizeof (vdev_label_t));
1220         max_osize = P2ALIGN(max_osize, (uint64_t)sizeof (vdev_label_t));
1221
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]);

1486         if (vd->vdev_ops->vdev_op_leaf)
1487                 vd->vdev_ops->vdev_op_hold(vd);
```

```
1488 }
1490 void
1491 vdev_rele(vdev_t *vd)
1492 {
1493         spa_t *spa = vd->vdev_spa;
1495         ASSERT(spa_is_root(spa));
1496         for (int c = 0; c < vd->vdev_children; c++)
1497                 vdev_rele(vd->vdev_child[c]);
1499         if (vd->vdev_ops->vdev_op_leaf)
1500                 vd->vdev_ops->vdev_op_rele(vd);
1501 }
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;
1514         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
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);
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         }
1536         /*
1537          * Reassess parent vdev's health.
1538          */
1539         vdev_propagate_state(vd);
1540 }
1542 int
1543 vdev_create(vdev_t *vd, uint64_t txg, boolean_t isreplacing)
1544 {
1545         int error;
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);
```



```
1554         if (error || vd->vdev_state != VDEV_STATE_HEALTHY) {
1555                 vdev_close(vd);
1556                 return (error ? error : ENXIO);
1557         }
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         }
1569         return (0);
1570 }
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 }
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));
1590         if (flags & VDD_METASLAB)
1591                 (void) txg_list_add(&vd->vdev_ms_list, arg, txg);
1593         if (flags & VDD_DTL)
1594                 (void) txg_list_add(&vd->vdev_dtl_list, arg, txg);
1596         (void) txg_list_add(&vd->vdev_spa->spa_vdev_txg_list, vd, txg);
1597 }
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);
1605         if (vd->vdev_ops->vdev_op_leaf)
1606                 vdev_dirty(vd->vdev_top, flags, vd, txg);
1607 }
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
```

```
 *      scrub completion, DTL_SCRUB replaces DTL_MISSING in the range of
 *      txgs that was scrubbed.
 *
 * DTL_OUTAGE: txgs which cannot currently be read, whether due to
 *      persistent errors or just some device being offline.
 *      Unlike the other three, the DTL_OUTAGE map is not generally
 *      maintained; it's only computed when needed, typically to
 *      determine whether a device can be detached.
 *
 * For leaf vdevs, DTL_MISSING and DTL_PARTIAL are identical: the device
 * either has the data or it doesn't.
 *
 * For interior vdevs such as mirror and RAID-Z the picture is more complex.
 * A vdev's DTL_PARTIAL is the union of its children's DTL_PARTIALs, because
 * if any child is less than fully replicated, then so is its parent.
 * A vdev's DTL_MISSING is a modified union of its children's DTL_MISSINGs,
 * comprising only those txgs which appear in 'maxfaults' or more children;
 * those are the txgs we don't have enough replication to read.  For example,
 * double-parity RAID-Z can tolerate up to two missing devices (maxfaults == 2);
 * thus, its DTL_MISSING consists of the set of txgs that appear in more than
 * two child DTL_MISSING maps.
 *
 * It should be clear from the above that to compute the DTLs and outage maps
 * for all vdevs, it suffices to know just the leaf vdevs' DTL_MISSING maps.
 * Therefore, that is all we keep on disk.  When loading the pool, or after
 * a configuration change, we generate all other DTLs from first principles.
 */
void
vdev_dtl_dirty(vdev_t *vd, vdev_dtl_type_t t, uint64_t txg, uint64_t size)
{
        range_tree_t *rt = vd->vdev_dtl[t];

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

        mutex_enter(rt->rt_lock);
        if (!range_tree_contains(rt, txg, size))
                range_tree_add(rt, txg, size);
        mutex_exit(rt->rt_lock);
}

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

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

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

        return (dirty);
}

boolean_t
vdev_dtl_empty(vdev_t *vd, vdev_dtl_type_t t)
{
        range_tree_t *rt = vd->vdev_dtl[t];
        boolean_t empty;

        mutex_enter(rt->rt_lock);
```



```
        empty = (range_tree_space(rt) == 0);
        mutex_exit(rt->rt_lock);

        return (empty);
}

/*
 * Returns the lowest txg in the DTL range.
 */
static uint64_t
vdev_dtl_min(vdev_t *vd)
{
        range_seg_t *rs;

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

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

/*
 * Returns the highest txg in the DTL.
 */
static uint64_t
vdev_dtl_max(vdev_t *vd)
{
        range_seg_t *rs;

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

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

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

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

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

        /*
         * When a resilver is initiated the scan will assign the scn_max_txg
         * value to the highest txg value that exists in all DTLs. If this
         * device's max DTL is not part of this scan (i.e. it is not in
         * the range (scn_min_txg, scn_max_txg] then it is not eligible
         * for excision.
         */
```

```
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);
```

```
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];
2143
2144                         if (msp == NULL || msp->ms_sm == NULL)
2145                                 continue;
2146
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.
```

```
 */
int
vdev_degrade(spa_t *spa, uint64_t guid, vdev_aux_t aux)
{
        vdev_t *vd;

        spa_vdev_state_enter(spa, SCL_NONE);

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

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

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

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

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

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

        spa_vdev_state_enter(spa, SCL_NONE);

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

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

        tvd = vd->vdev_top;
        vd->vdev_offline = B_FALSE;
        vd->vdev_tmpoffline = B_FALSE;
        vd->vdev_checkremove = !!(flags & ZFS_ONLINE_CHECKREMOVE);
        vd->vdev_forcefault = !!(flags & ZFS_ONLINE_FORCEFAULT);

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

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

        if (!vd->vdev_aux) {
                for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
```



```
                        pvd->vdev_expanding = B_FALSE;
        }

        if (newstate)
                *newstate = vd->vdev_state;
        if ((flags & ZFS_ONLINE_UNSPARE) &&
            !vdev_is_dead(vd) && vd->vdev_parent &&
            vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
            vd->vdev_parent->vdev_child[0] == vd)
                vd->vdev_unspare = B_TRUE;

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

                /* XXX - L2ARC 1.0 does not support expansion */
                if (vd->vdev_aux)
                        return (spa_vdev_state_exit(spa, vd, ENOTSUP));
                spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
        }
        return (spa_vdev_state_exit(spa, vd, 0));
}

static int
vdev_offline_locked(spa_t *spa, uint64_t guid, uint64_t flags)
{
        vdev_t *vd, *tvd;
        int error = 0;
        uint64_t generation;
        metaslab_group_t *mg;

top:
        spa_vdev_state_enter(spa, SCL_ALLOC);

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

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

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

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

                /*
                 * If the top-level is a slog and it has had allocations
                 * then proceed.  We check that the vdev's metaslab group
                 * is not NULL since it's possible that we may have just
                 * added this vdev but not yet initialized its metaslabs.
                 */
                if (tvd->vdev_islog && mg != NULL) {
                        /*
                         * Prevent any future allocations.
                         */
                        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);
3078
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);
3091
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;
3119
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;
3139
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                         }
3162
3163                         zfs_ereport_post(class, spa, vd, NULL, save_state, 0);
3164                 }
3165
3166                 /* Erase any notion of persistent removed state */
3167                 vd->vdev_removed = B_FALSE;
3168         } else {
3169                 vd->vdev_removed = B_FALSE;
3170         }
3171
3172         if (!isopen && vd->vdev_parent)
3173                 vdev_propagate_state(vd->vdev_parent);
3174 }
3175
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;
3187
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         }
3198
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 }
```

```
**********************************************************
   57769 Thu Oct 16 19:15:52 2014
new/usr/src/uts/common/fs/zfs/zil.c
zpool import speedup
**********************************************************
______unchanged_portion_omitted_

 626 int
 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_nolock(osname, DMU_OST_ANY, B_FALSE, FTAG, &os);
 636         error = dmu_objset_own(osname, DMU_OST_ANY, B_FALSE, FTAG, &os);
 637         if (error != 0) {
 638                 cmn_err(CE_WARN, "can't open objset for %s", osname);
 639                 return (0);
 640         }

 642         zilog = dmu_objset_zil(os);
 643         zh = zil_header_in_syncing_context(zilog);

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

 654         /*
 655          * Claim all log blocks if we haven't already done so, and remember
 656          * the highest claimed sequence number.  This ensures that if we can
 657          * read only part of the log now (e.g. due to a missing device),
 658          * but we can read the entire log later, we will not try to replay
 659          * or destroy beyond the last block we successfully claimed.
 660          */
 661         ASSERT3U(zh->zh_claim_txg, <=, first_txg);
 662         if (zh->zh_claim_txg == 0 && !BP_IS_HOLE(&zh->zh_log)) {
 663                 (void) zil_parse(zilog, zil_claim_log_block,
 664                     zil_claim_log_record, tx, first_txg);
 665                 zh->zh_claim_txg = first_txg;
 666                 zh->zh_claim_blk_seq = zilog->zl_parse_blk_seq;
 667                 zh->zh_claim_lr_seq = zilog->zl_parse_lr_seq;
 668                 if (zilog->zl_parse_lr_count || zilog->zl_parse_blk_count > 1)
 669                         zh->zh_flags |= ZIL_REPLAY_NEEDED;
 670                 zh->zh_flags |= ZIL_CLAIM_LR_SEQ_VALID;
 671                 dsl_dataset_dirty(dmu_objset_ds(os), tx);
 672         }

 674         ASSERT3U(first_txg, ==, (spa_last_synced_txg(zilog->zl_spa) + 1));
 675         dmu_objset_disown(os, FTAG);
 676         return (0);
 677 }

 679 /*
 680  * Check the log by walking the log chain.
 681  * Checksum errors are ok as they indicate the end of the chain.
 682  * Any other error (no device or read failure) returns an error.
 683  */
```



```
 684 int
 685 zil_check_log_chain(const char *osname, void *tx)
 686 {
 687         zilog_t *zilog;
 688         objset_t *os;
 689         blkptr_t *bp;
 690         int error;

 692         ASSERT(tx == NULL);

 694         error = dmu_objset_hold_nolock(osname, FTAG, &os);
 694         error = dmu_objset_hold(osname, FTAG, &os);
 695         if (error != 0) {
 696                 cmn_err(CE_WARN, "can't open objset for %s", osname);
 697                 return (0);
 698         }

 700         zilog = dmu_objset_zil(os);
 701         bp = (blkptr_t *)&zilog->zl_header->zh_log;

 703         /*
 704          * Check the first block and determine if it's on a log device
 705          * which may have been removed or faulted prior to loading this
 706          * pool.  If so, there's no point in checking the rest of the log
 707          * as its content should have already been synced to the pool.
 708          */
 709         if (!BP_IS_HOLE(bp)) {
 710                 vdev_t *vd;
 711                 boolean_t valid = B_TRUE;

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

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

 725         /*
 726          * Because tx == NULL, zil_claim_log_block() will not actually claim
 727          * any blocks, but just determine whether it is possible to do so.
 728          * In addition to checking the log chain, zil_claim_log_block()
 729          * will invoke zio_claim() with a done func of spa_claim_notify(),
 730          * which will update spa_max_claim_txg.  See spa_load() for details.
 731          */
 732         error = zil_parse(zilog, zil_claim_log_block, zil_claim_log_record, tx,
 733             zilog->zl_header->zh_claim_txg ? -1ULL : spa_first_txg(os->os_spa));

 735         dmu_objset_rele(os, FTAG);

 737         return ((error == ECKSUM || error == ENOENT) ? 0 : error);
 738 }
______unchanged_portion_omitted_
```