```
**********************************************************
  175376 Mon Apr 29 12:58:31 2013
new/usr/src/uts/common/fs/zfs/spa.c
3749 zfs event processing should work on R/O root filesystems
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 */

/*
 * This file contains all the routines used when modifying on-disk SPA state.
 * This includes opening, importing, destroying, exporting a pool, and syncing a
 * pool.
 */

#include <sys/zfs_context.h>
#include <sys/fm/fs/zfs.h>
#include <sys/spa_impl.h>
#include <sys/zio.h>
#include <sys/zio_checksum.h>
#include <sys/dmu.h>
#include <sys/dmu_tx.h>
#include <sys/zap.h>
#include <sys/zil.h>
#include <sys/ddt.h>
#include <sys/vdev_impl.h>
#include <sys/metaslab.h>
#include <sys/metaslab_impl.h>
#include <sys/uberblock_impl.h>
#include <sys/txg.h>
#include <sys/avl.h>
#include <sys/dmu_traverse.h>
#include <sys/dmu_objset.h>
#include <sys/unique.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/fs/zfs.h>
```



```
#include <sys/arc.h>
#include <sys/callb.h>
#include <sys/systeminfo.h>
#include <sys/spa_boot.h>
#include <sys/zfs_ioctl.h>
#include <sys/dsl_scan.h>
#include <sys/zfeature.h>
#include <sys/dsl_destroy.h>

#ifdef  _KERNEL
#include <sys/bootprops.h>
#include <sys/callb.h>
#include <sys/cpupart.h>
#include <sys/pool.h>
#include <sys/sysdc.h>
#include <sys/zone.h>
#endif  /* _KERNEL */

#include "zfs_prop.h"
#include "zfs_comutil.h"

/*
 * The interval, in seconds, at which failed configuration cache file writes
 * should be retried.
 */
static int zfs_ccw_retry_interval = 300;

#endif /* ! codereview */
typedef enum zti_modes {
        ZTI_MODE_FIXED,                 /* value is # of threads (min 1) */
        ZTI_MODE_ONLINE_PERCENT,        /* value is % of online CPUs */
        ZTI_MODE_BATCH,                 /* cpu-intensive; value is ignored */
        ZTI_MODE_NULL,                  /* don't create a taskq */
        ZTI_NMODES
} zti_modes_t;

#define ZTI_P(n, q)     { ZTI_MODE_FIXED, (n), (q) }
#define ZTI_PCT(n)      { ZTI_MODE_ONLINE_PERCENT, (n), 1 }
#define ZTI_BATCH       { ZTI_MODE_BATCH, 0, 1 }
#define ZTI_NULL        { ZTI_MODE_NULL, 0, 0 }

#define ZTI_N(n)        ZTI_P(n, 1)
#define ZTI_ONE         ZTI_N(1)

typedef struct zio_taskq_info {
        zti_modes_t zti_mode;
        uint_t zti_value;
        uint_t zti_count;
} zio_taskq_info_t;

static const char *const zio_taskq_types[ZIO_TASKQ_TYPES] = {
        "issue", "issue_high", "intr", "intr_high"
};

/*
 * This table defines the taskq settings for each ZFS I/O type. When
 * initializing a pool, we use this table to create an appropriately sized
 * taskq. Some operations are low volume and therefore have a small, static
 * number of threads assigned to their taskqs using the ZTI_N(#) or ZTI_ONE
 * macros. Other operations process a large amount of data; the ZTI_BATCH
 * macro causes us to create a taskq oriented for throughput. Some operations
 * are so high frequency and short-lived that the taskq itself can become a a
 * point of lock contention. The ZTI_P(#, #) macro indicates that we need an
 * additional degree of parallelism specified by the number of threads per-
 * taskq and the number of taskqs; when dispatching an event in this case, the
 * particular taskq is chosen at random.
```

```
 *
 * The different taskq priorities are to handle the different contexts (issue
 * and interrupt) and then to reserve threads for ZIO_PRIORITY_NOW I/Os that
 * need to be handled with minimum delay.
 */
const zio_taskq_info_t zio_taskqs[ZIO_TYPES][ZIO_TASKQ_TYPES] = {
        /* ISSUE        ISSUE_HIGH      INTR            INTR_HIGH */
        { ZTI_ONE,      ZTI_NULL,       ZTI_ONE,        ZTI_NULL }, /* NULL */
        { ZTI_N(8),     ZTI_NULL,       ZTI_BATCH,      ZTI_NULL }, /* READ */
        { ZTI_BATCH,    ZTI_N(5),       ZTI_N(8),       ZTI_N(5) }, /* WRITE */
        { ZTI_P(12, 8), ZTI_NULL,       ZTI_ONE,        ZTI_NULL }, /* FREE */
        { ZTI_ONE,      ZTI_NULL,       ZTI_ONE,        ZTI_NULL }, /* CLAIM */
        { ZTI_ONE,      ZTI_NULL,       ZTI_ONE,        ZTI_NULL }, /* IOCTL */
};

static void spa_sync_version(void *arg, dmu_tx_t *tx);
static void spa_sync_props(void *arg, dmu_tx_t *tx);
static boolean_t spa_has_active_shared_spare(spa_t *spa);
static int spa_load_impl(spa_t *spa, uint64_t, nvlist_t *config,
    spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
    char **ereport);
static void spa_vdev_resilver_done(spa_t *spa);

uint_t          zio_taskq_batch_pct = 100;      /* 1 thread per cpu in pset */
id_t            zio_taskq_psrset_bind = PS_NONE;
boolean_t       zio_taskq_sysdc = B_TRUE;       /* use SDC scheduling class */
uint_t          zio_taskq_basedc = 80;          /* base duty cycle */

boolean_t       spa_create_process = B_TRUE;    /* no process ==> no sysdc */
extern int      zfs_sync_pass_deferred_free;

/*
 * This (illegal) pool name is used when temporarily importing a spa_t in order
 * to get the vdev stats associated with the imported devices.
 */
#define TRYIMPORT_NAME  "$import"

/*
 * ==========================================================================
 * SPA properties routines
 * ==========================================================================
 */

/*
 * Add a (source=src, propname=propval) list to an nvlist.
 */
static void
spa_prop_add_list(nvlist_t *nvl, zpool_prop_t prop, char *strval,
    uint64_t intval, zprop_source_t src)
{
        const char *propname = zpool_prop_to_name(prop);
        nvlist_t *propval;

        VERIFY(nvlist_alloc(&propval, NV_UNIQUE_NAME, KM_SLEEP) == 0);
        VERIFY(nvlist_add_uint64(propval, ZPROP_SOURCE, src) == 0);

        if (strval != NULL)
                VERIFY(nvlist_add_string(propval, ZPROP_VALUE, strval) == 0);
        else
                VERIFY(nvlist_add_uint64(propval, ZPROP_VALUE, intval) == 0);

        VERIFY(nvlist_add_nvlist(nvl, propname, propval) == 0);
        nvlist_free(propval);
}

/*
```



```
 * Get property values from the spa configuration.
 */
static void
spa_prop_get_config(spa_t *spa, nvlist_t **nvp)
{
        vdev_t *rvd = spa->spa_root_vdev;
        dsl_pool_t *pool = spa->spa_dsl_pool;
        uint64_t size;
        uint64_t alloc;
        uint64_t space;
        uint64_t cap, version;
        zprop_source_t src = ZPROP_SRC_NONE;
        spa_config_dirent_t *dp;

        ASSERT(MUTEX_HELD(&spa->spa_props_lock));

        if (rvd != NULL) {
                alloc = metaslab_class_get_alloc(spa_normal_class(spa));
                size = metaslab_class_get_space(spa_normal_class(spa));
                spa_prop_add_list(*nvp, ZPOOL_PROP_NAME, spa_name(spa), 0, src);
                spa_prop_add_list(*nvp, ZPOOL_PROP_SIZE, NULL, size, src);
                spa_prop_add_list(*nvp, ZPOOL_PROP_ALLOCATED, NULL, alloc, src);
                spa_prop_add_list(*nvp, ZPOOL_PROP_FREE, NULL,
                    size - alloc, src);

                space = 0;
                for (int c = 0; c < rvd->vdev_children; c++) {
                        vdev_t *tvd = rvd->vdev_child[c];
                        space += tvd->vdev_max_asize - tvd->vdev_asize;
                }
                spa_prop_add_list(*nvp, ZPOOL_PROP_EXPANDSZ, NULL, space,
                    src);

                spa_prop_add_list(*nvp, ZPOOL_PROP_READONLY, NULL,
                    (spa_mode(spa) == FREAD), src);

                cap = (size == 0) ? 0 : (alloc * 100 / size);
                spa_prop_add_list(*nvp, ZPOOL_PROP_CAPACITY, NULL, cap, src);

                spa_prop_add_list(*nvp, ZPOOL_PROP_DEDUPRATIO, NULL,
                    ddt_get_pool_dedup_ratio(spa), src);

                spa_prop_add_list(*nvp, ZPOOL_PROP_HEALTH, NULL,
                    rvd->vdev_state, src);

                version = spa_version(spa);
                if (version == zpool_prop_default_numeric(ZPOOL_PROP_VERSION))
                        src = ZPROP_SRC_DEFAULT;
                else
                        src = ZPROP_SRC_LOCAL;
                spa_prop_add_list(*nvp, ZPOOL_PROP_VERSION, NULL, version, src);
        }

        if (pool != NULL) {
                dsl_dir_t *freedir = pool->dp_free_dir;

                /*
                 * The $FREE directory was introduced in SPA_VERSION_DEADLISTS,
                 * when opening pools before this version freedir will be NULL.
                 */
                if (freedir != NULL) {
                        spa_prop_add_list(*nvp, ZPOOL_PROP_FREEING, NULL,
                            freedir->dd_phys->dd_used_bytes, src);
                } else {
                        spa_prop_add_list(*nvp, ZPOOL_PROP_FREEING,
                            NULL, 0, src);
```

```
257                 }
258         }

260         spa_prop_add_list(*nvp, ZPOOL_PROP_GUID, NULL, spa_guid(spa), src);

262         if (spa->spa_comment != NULL) {
263                 spa_prop_add_list(*nvp, ZPOOL_PROP_COMMENT, spa->spa_comment,
264                     0, ZPROP_SRC_LOCAL);
265         }

267         if (spa->spa_root != NULL)
268                 spa_prop_add_list(*nvp, ZPOOL_PROP_ALTROOT, spa->spa_root,
269                     0, ZPROP_SRC_LOCAL);

271         if ((dp = list_head(&spa->spa_config_list)) != NULL) {
272                 if (dp->scd_path == NULL) {
273                         spa_prop_add_list(*nvp, ZPOOL_PROP_CACHEFILE,
274                             "none", 0, ZPROP_SRC_LOCAL);
275                 } else if (strcmp(dp->scd_path, spa_config_path) != 0) {
276                         spa_prop_add_list(*nvp, ZPOOL_PROP_CACHEFILE,
277                             dp->scd_path, 0, ZPROP_SRC_LOCAL);
278                 }
279         }
280 }

282 /*
283  * Get zpool property values.
284  */
285 int
286 spa_prop_get(spa_t *spa, nvlist_t **nvp)
287 {
288         objset_t *mos = spa->spa_meta_objset;
289         zap_cursor_t zc;
290         zap_attribute_t za;
291         int err;

293         VERIFY(nvlist_alloc(nvp, NV_UNIQUE_NAME, KM_SLEEP) == 0);

295         mutex_enter(&spa->spa_props_lock);

297         /*
298          * Get properties from the spa config.
299          */
300         spa_prop_get_config(spa, nvp);

302         /* If no pool property object, no more prop to get. */
303         if (mos == NULL || spa->spa_pool_props_object == 0) {
304                 mutex_exit(&spa->spa_props_lock);
305                 return (0);
306         }

308         /*
309          * Get properties from the MOS pool property object.
310          */
311         for (zap_cursor_init(&zc, mos, spa->spa_pool_props_object);
312             (err = zap_cursor_retrieve(&zc, &za)) == 0;
313             zap_cursor_advance(&zc)) {
314                 uint64_t intval = 0;
315                 char *strval = NULL;
316                 zprop_source_t src = ZPROP_SRC_DEFAULT;
317                 zpool_prop_t prop;

319                 if ((prop = zpool_name_to_prop(za.za_name)) == ZPROP_INVAL)
320                         continue;

322                 switch (za.za_integer_length) {
```



```
323                 case 8:
324                         /* integer property */
325                         if (za.za_first_integer !=
326                             zpool_prop_default_numeric(prop))
327                                 src = ZPROP_SRC_LOCAL;

329                         if (prop == ZPOOL_PROP_BOOTFS) {
330                                 dsl_pool_t *dp;
331                                 dsl_dataset_t *ds = NULL;

333                                 dp = spa_get_dsl(spa);
334                                 dsl_pool_config_enter(dp, FTAG);
335                                 if (err = dsl_dataset_hold_obj(dp,
336                                     za.za_first_integer, FTAG, &ds)) {
337                                         dsl_pool_config_exit(dp, FTAG);
338                                         break;
339                                 }

341                                 strval = kmem_alloc(
342                                     MAXNAMELEN + strlen(MOS_DIR_NAME) + 1,
343                                     KM_SLEEP);
344                                 dsl_dataset_name(ds, strval);
345                                 dsl_dataset_rele(ds, FTAG);
346                                 dsl_pool_config_exit(dp, FTAG);
347                         } else {
348                                 strval = NULL;
349                                 intval = za.za_first_integer;
350                         }

352                         spa_prop_add_list(*nvp, prop, strval, intval, src);

354                         if (strval != NULL)
355                                 kmem_free(strval,
356                                     MAXNAMELEN + strlen(MOS_DIR_NAME) + 1);

358                         break;

360                 case 1:
361                         /* string property */
362                         strval = kmem_alloc(za.za_num_integers, KM_SLEEP);
363                         err = zap_lookup(mos, spa->spa_pool_props_object,
364                             za.za_name, 1, za.za_num_integers, strval);
365                         if (err) {
366                                 kmem_free(strval, za.za_num_integers);
367                                 break;
368                         }
369                         spa_prop_add_list(*nvp, prop, strval, 0, src);
370                         kmem_free(strval, za.za_num_integers);
371                         break;

373                 default:
374                         break;
375                 }
376         }
377         zap_cursor_fini(&zc);
378         mutex_exit(&spa->spa_props_lock);
379 out:
380         if (err && err != ENOENT) {
381                 nvlist_free(*nvp);
382                 *nvp = NULL;
383                 return (err);
384         }

386         return (0);
387 }
```

```
389 /*
390  * Validate the given pool properties nvlist and modify the list
391  * for the property values to be set.
392  */
393 static int
394 spa_prop_validate(spa_t *spa, nvlist_t *props)
395 {
396         nvpair_t *elem;
397         int error = 0, reset_bootfs = 0;
398         uint64_t objnum = 0;
399         boolean_t has_feature = B_FALSE;
400
401         elem = NULL;
402         while ((elem = nvlist_next_nvpair(props, elem)) != NULL) {
403                 uint64_t intval;
404                 char *strval, *slash, *check, *fname;
405                 const char *propname = nvpair_name(elem);
406                 zpool_prop_t prop = zpool_name_to_prop(propname);
407
408                 switch (prop) {
409                 case ZPROP_INVAL:
410                         if (!zpool_prop_feature(propname)) {
411                                 error = SET_ERROR(EINVAL);
412                                 break;
413                         }
414
415                         /*
416                          * Sanitize the input.
417                          */
418                         if (nvpair_type(elem) != DATA_TYPE_UINT64) {
419                                 error = SET_ERROR(EINVAL);
420                                 break;
421                         }
422
423                         if (nvpair_value_uint64(elem, &intval) != 0) {
424                                 error = SET_ERROR(EINVAL);
425                                 break;
426                         }
427
428                         if (intval != 0) {
429                                 error = SET_ERROR(EINVAL);
430                                 break;
431                         }
432
433                         fname = strchr(propname, '@') + 1;
434                         if (zfeature_lookup_name(fname, NULL) != 0) {
435                                 error = SET_ERROR(EINVAL);
436                                 break;
437                         }
438
439                         has_feature = B_TRUE;
440                         break;
441
442                 case ZPOOL_PROP_VERSION:
443                         error = nvpair_value_uint64(elem, &intval);
444                         if (!error &&
445                             (intval < spa_version(spa) ||
446                             intval > SPA_VERSION_BEFORE_FEATURES ||
447                             has_feature))
448                                 error = SET_ERROR(EINVAL);
449                         break;
450
451                 case ZPOOL_PROP_DELEGATION:
452                 case ZPOOL_PROP_AUTOREPLACE:
453                 case ZPOOL_PROP_LISTSNAPS:
454                 case ZPOOL_PROP_AUTOEXPAND:
```



```
455                         error = nvpair_value_uint64(elem, &intval);
456                         if (!error && intval > 1)
457                                 error = SET_ERROR(EINVAL);
458                         break;
459
460                 case ZPOOL_PROP_BOOTFS:
461                         /*
462                          * If the pool version is less than SPA_VERSION_BOOTFS,
463                          * or the pool is still being created (version == 0),
464                          * the bootfs property cannot be set.
465                          */
466                         if (spa_version(spa) < SPA_VERSION_BOOTFS) {
467                                 error = SET_ERROR(ENOTSUP);
468                                 break;
469                         }
470
471                         /*
472                          * Make sure the vdev config is bootable
473                          */
474                         if (!vdev_is_bootable(spa->spa_root_vdev)) {
475                                 error = SET_ERROR(ENOTSUP);
476                                 break;
477                         }
478
479                         reset_bootfs = 1;
480
481                         error = nvpair_value_string(elem, &strval);
482
483                         if (!error) {
484                                 objset_t *os;
485                                 uint64_t compress;
486
487                                 if (strval == NULL || strval[0] == '\0') {
488                                         objnum = zpool_prop_default_numeric(
489                                             ZPOOL_PROP_BOOTFS);
490                                         break;
491                                 }
492
493                                 if (error = dmu_objset_hold(strval, FTAG, &os))
494                                         break;
495
496                                 /* Must be ZPL and not gzip compressed. */
497
498                                 if (dmu_objset_type(os) != DMU_OST_ZFS) {
499                                         error = SET_ERROR(ENOTSUP);
500                                 } else if ((error =
501                                     dsl_prop_get_int_ds(dmu_objset_ds(os),
502                                     zfs_prop_to_name(ZFS_PROP_COMPRESSION),
503                                     &compress)) == 0 &&
504                                     !BOOTFS_COMPRESS_VALID(compress)) {
505                                         error = SET_ERROR(ENOTSUP);
506                                 } else {
507                                         objnum = dmu_objset_id(os);
508                                 }
509                                 dmu_objset_rele(os, FTAG);
510                         }
511                         break;
512
513                 case ZPOOL_PROP_FAILUREMODE:
514                         error = nvpair_value_uint64(elem, &intval);
515                         if (!error && (intval < ZIO_FAILURE_MODE_WAIT ||
516                             intval > ZIO_FAILURE_MODE_PANIC))
517                                 error = SET_ERROR(EINVAL);
518
519                         /*
520                          * This is a special case which only occurs when
```

```
521                          * the pool has completely failed. This allows
522                          * the user to change the in-core failmode property
523                          * without syncing it out to disk (I/Os might
524                          * currently be blocked). We do this by returning
525                          * EIO to the caller (spa_prop_set) to trick it
526                          * into thinking we encountered a property validation
527                          * error.
528                          */
529                         if (!error && spa_suspended(spa)) {
530                                 spa->spa_failmode = intval;
531                                 error = SET_ERROR(EIO);
532                         }
533                         break;
534
535                 case ZPOOL_PROP_CACHEFILE:
536                         if ((error = nvpair_value_string(elem, &strval)) != 0)
537                                 break;
538
539                         if (strval[0] == '\0')
540                                 break;
541
542                         if (strcmp(strval, "none") == 0)
543                                 break;
544
545                         if (strval[0] != '/') {
546                                 error = SET_ERROR(EINVAL);
547                                 break;
548                         }
549
550                         slash = strrchr(strval, '/');
551                         ASSERT(slash != NULL);
552
553                         if (slash[1] == '\0' || strcmp(slash, "/.") == 0 ||
554                             strcmp(slash, "/..") == 0)
555                                 error = SET_ERROR(EINVAL);
556                         break;
557
558                 case ZPOOL_PROP_COMMENT:
559                         if ((error = nvpair_value_string(elem, &strval)) != 0)
560                                 break;
561                         for (check = strval; *check != '\0'; check++) {
562                                 /*
563                                  * The kernel doesn't have an easy isprint()
564                                  * check.  For this kernel check, we merely
565                                  * check ASCII apart from DEL.  Fix this if
566                                  * there is an easy-to-use kernel isprint().
567                                  */
568                                 if (*check >= 0x7f) {
569                                         error = SET_ERROR(EINVAL);
570                                         break;
571                                 }
572                                 check++;
573                         }
574                         if (strlen(strval) > ZPROP_MAX_COMMENT)
575                                 error = E2BIG;
576                         break;
577
578                 case ZPOOL_PROP_DEDUPDITTO:
579                         if (spa_version(spa) < SPA_VERSION_DEDUP)
580                                 error = SET_ERROR(ENOTSUP);
581                         else
582                                 error = nvpair_value_uint64(elem, &intval);
583                         if (error == 0 &&
584                             intval != 0 && intval < ZIO_DEDUPDITTO_MIN)
585                                 error = SET_ERROR(EINVAL);
586                         break;
```



```
587                 }
588
589                 if (error)
590                         break;
591         }
592
593         if (!error && reset_bootfs) {
594                 error = nvlist_remove(props,
595                     zpool_prop_to_name(ZPOOL_PROP_BOOTFS), DATA_TYPE_STRING);
596
597                 if (!error) {
598                         error = nvlist_add_uint64(props,
599                             zpool_prop_to_name(ZPOOL_PROP_BOOTFS), objnum);
600                 }
601         }
602
603         return (error);
604 }
605
606 void
607 spa_configfile_set(spa_t *spa, nvlist_t *nvp, boolean_t need_sync)
608 {
609         char *cachefile;
610         spa_config_dirent_t *dp;
611
612         if (nvlist_lookup_string(nvp, zpool_prop_to_name(ZPOOL_PROP_CACHEFILE),
613             &cachefile) != 0)
614                 return;
615
616         dp = kmem_alloc(sizeof (spa_config_dirent_t),
617             KM_SLEEP);
618
619         if (cachefile[0] == '\0')
620                 dp->scd_path = spa_strdup(spa_config_path);
621         else if (strcmp(cachefile, "none") == 0)
622                 dp->scd_path = NULL;
623         else
624                 dp->scd_path = spa_strdup(cachefile);
625
626         list_insert_head(&spa->spa_config_list, dp);
627         if (need_sync)
628                 spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
629 }
630
631 int
632 spa_prop_set(spa_t *spa, nvlist_t *nvp)
633 {
634         int error;
635         nvpair_t *elem = NULL;
636         boolean_t need_sync = B_FALSE;
637
638         if ((error = spa_prop_validate(spa, nvp)) != 0)
639                 return (error);
640
641         while ((elem = nvlist_next_nvpair(nvp, elem)) != NULL) {
642                 zpool_prop_t prop = zpool_name_to_prop(nvpair_name(elem));
643
644                 if (prop == ZPOOL_PROP_CACHEFILE ||
645                     prop == ZPOOL_PROP_ALTROOT ||
646                     prop == ZPOOL_PROP_READONLY)
647                         continue;
648
649                 if (prop == ZPOOL_PROP_VERSION || prop == ZPROP_INVAL) {
650                         uint64_t ver;
651
652                         if (prop == ZPOOL_PROP_VERSION) {
```

```
653                             VERIFY(nvpair_value_uint64(elem, &ver) == 0);
654                     } else {
655                             ASSERT(zpool_prop_feature(nvpair_name(elem)));
656                             ver = SPA_VERSION_FEATURES;
657                             need_sync = B_TRUE;
658                     }
659
660                     /* Save time if the version is already set. */
661                     if (ver == spa_version(spa))
662                             continue;
663
664                     /*
665                      * In addition to the pool directory object, we might
666                      * create the pool properties object, the features for
667                      * read object, the features for write object, or the
668                      * feature descriptions object.
669                      */
670                     error = dsl_sync_task(spa->spa_name, NULL,
671                         spa_sync_version, &ver, 6);
672                     if (error)
673                             return (error);
674                     continue;
675             }
676
677             need_sync = B_TRUE;
678             break;
679     }
680
681     if (need_sync) {
682             return (dsl_sync_task(spa->spa_name, NULL, spa_sync_props,
683                 nvp, 6));
684     }
685
686     return (0);
687 }
688
689 /*
690  * If the bootfs property value is dsobj, clear it.
691  */
692 void
693 spa_prop_clear_bootfs(spa_t *spa, uint64_t dsobj, dmu_tx_t *tx)
694 {
695     if (spa->spa_bootfs == dsobj && spa->spa_pool_props_object != 0) {
696             VERIFY(zap_remove(spa->spa_meta_objset,
697                 spa->spa_pool_props_object,
698                 zpool_prop_to_name(ZPOOL_PROP_BOOTFS), tx) == 0);
699             spa->spa_bootfs = 0;
700     }
701 }
702
703 /*ARGSUSED*/
704 static int
705 spa_change_guid_check(void *arg, dmu_tx_t *tx)
706 {
707     uint64_t *newguid = arg;
708     spa_t *spa = dmu_tx_pool(tx)->dp_spa;
709     vdev_t *rvd = spa->spa_root_vdev;
710     uint64_t vdev_state;
711
712     spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
713     vdev_state = rvd->vdev_state;
714     spa_config_exit(spa, SCL_STATE, FTAG);
715
716     if (vdev_state != VDEV_STATE_HEALTHY)
717             return (SET_ERROR(ENXIO));
```



```
719     ASSERT3U(spa_guid(spa), !=, *newguid);
720
721     return (0);
722 }
723
724 static void
725 spa_change_guid_sync(void *arg, dmu_tx_t *tx)
726 {
727     uint64_t *newguid = arg;
728     spa_t *spa = dmu_tx_pool(tx)->dp_spa;
729     uint64_t oldguid;
730     vdev_t *rvd = spa->spa_root_vdev;
731
732     oldguid = spa_guid(spa);
733
734     spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
735     rvd->vdev_guid = *newguid;
736     rvd->vdev_guid_sum += (*newguid - oldguid);
737     vdev_config_dirty(rvd);
738     spa_config_exit(spa, SCL_STATE, FTAG);
739
740     spa_history_log_internal(spa, "guid change", tx, "old=%llu new=%llu",
741         oldguid, *newguid);
742 }
743
744 /*
745  * Change the GUID for the pool.  This is done so that we can later
746  * re-import a pool built from a clone of our own vdevs.  We will modify
747  * the root vdev's guid, our own pool guid, and then mark all of our
748  * vdevs dirty.  Note that we must make sure that all our vdevs are
749  * online when we do this, or else any vdevs that weren't present
750  * would be orphaned from our pool.  We are also going to issue a
751  * sysevent to update any watchers.
752  */
753 int
754 spa_change_guid(spa_t *spa)
755 {
756     int error;
757     uint64_t guid;
758
759     mutex_enter(&spa_namespace_lock);
760     guid = spa_generate_guid(NULL);
761
762     error = dsl_sync_task(spa->spa_name, spa_change_guid_check,
763         spa_change_guid_sync, &guid, 5);
764
765     if (error == 0) {
766             spa_config_sync(spa, B_FALSE, B_TRUE);
767             spa_event_notify(spa, NULL, ESC_ZFS_POOL_REGUID);
768     }
769
770     mutex_exit(&spa_namespace_lock);
771
772     return (error);
773 }
774
775 /*
776  * ============================================================================
777  * SPA state manipulation (open/create/destroy/import/export)
778  * ============================================================================
779  */
780
781 static int
782 spa_error_entry_compare(const void *a, const void *b)
783 {
784     spa_error_entry_t *sa = (spa_error_entry_t *)a;
```

```
785         spa_error_entry_t *sb = (spa_error_entry_t *)b;
786         int ret;
788         ret = bcmp(&sa->se_bookmark, &sb->se_bookmark,
789             sizeof (zbookmark_t));
791         if (ret < 0)
792                 return (-1);
793         else if (ret > 0)
794                 return (1);
795         else
796                 return (0);
797 }
799 /*
800  * Utility function which retrieves copies of the current logs and
801  * re-initializes them in the process.
802  */
803 void
804 spa_get_errlists(spa_t *spa, avl_tree_t *last, avl_tree_t *scrub)
805 {
806         ASSERT(MUTEX_HELD(&spa->spa_errlist_lock));
808         bcopy(&spa->spa_errlist_last, last, sizeof (avl_tree_t));
809         bcopy(&spa->spa_errlist_scrub, scrub, sizeof (avl_tree_t));
811         avl_create(&spa->spa_errlist_scrub,
812             spa_error_entry_compare, sizeof (spa_error_entry_t),
813             offsetof(spa_error_entry_t, se_avl));
814         avl_create(&spa->spa_errlist_last,
815             spa_error_entry_compare, sizeof (spa_error_entry_t),
816             offsetof(spa_error_entry_t, se_avl));
817 }
819 static void
820 spa_taskqs_init(spa_t *spa, zio_type_t t, zio_taskq_type_t q)
821 {
822         const zio_taskq_info_t *ztip = &zio_taskqs[t][q];
823         enum zti_modes mode = ztip->zti_mode;
824         uint_t value = ztip->zti_value;
825         uint_t count = ztip->zti_count;
826         spa_taskqs_t *tqs = &spa->spa_zio_taskq[t][q];
827         char name[32];
828         uint_t flags = 0;
829         boolean_t batch = B_FALSE;
831         if (mode == ZTI_MODE_NULL) {
832                 tqs->stqs_count = 0;
833                 tqs->stqs_taskq = NULL;
834                 return;
835         }
837         ASSERT3U(count, >, 0);
839         tqs->stqs_count = count;
840         tqs->stqs_taskq = kmem_alloc(count * sizeof (taskq_t *), KM_SLEEP);
842         for (uint_t i = 0; i < count; i++) {
843                 taskq_t *tq;
845                 switch (mode) {
846                 case ZTI_MODE_FIXED:
847                         ASSERT3U(value, >=, 1);
848                         value = MAX(value, 1);
849                         break;
```



```
851                 case ZTI_MODE_BATCH:
852                         batch = B_TRUE;
853                         flags |= TASKQ_THREADS_CPU_PCT;
854                         value = zio_taskq_batch_pct;
855                         break;
857                 case ZTI_MODE_ONLINE_PERCENT:
858                         flags |= TASKQ_THREADS_CPU_PCT;
859                         break;
861                 default:
862                         panic("unrecognized mode for %s_%s taskq (%u:%u) in "
863                             "spa_activate()",
864                             zio_type_name[t], zio_taskq_types[q], mode, value);
865                         break;
866                 }
868                 if (count > 1) {
869                         (void) snprintf(name, sizeof (name), "%s_%s_%u",
870                             zio_type_name[t], zio_taskq_types[q], i);
871                 } else {
872                         (void) snprintf(name, sizeof (name), "%s_%s",
873                             zio_type_name[t], zio_taskq_types[q]);
874                 }
876                 if (zio_taskq_sysdc && spa->spa_proc != &p0) {
877                         if (batch)
878                                 flags |= TASKQ_DC_BATCH;
880                         tq = taskq_create_sysdc(name, value, 50, INT_MAX,
881                             spa->spa_proc, zio_taskq_basedc, flags);
882                 } else {
883                         tq = taskq_create_proc(name, value, maxclsyspri, 50,
884                             INT_MAX, spa->spa_proc, flags);
885                 }
887                 tqs->stqs_taskq[i] = tq;
888         }
889 }
891 static void
892 spa_taskqs_fini(spa_t *spa, zio_type_t t, zio_taskq_type_t q)
893 {
894         spa_taskqs_t *tqs = &spa->spa_zio_taskq[t][q];
896         if (tqs->stqs_taskq == NULL) {
897                 ASSERT0(tqs->stqs_count);
898                 return;
899         }
901         for (uint_t i = 0; i < tqs->stqs_count; i++) {
902                 ASSERT3P(tqs->stqs_taskq[i], !=, NULL);
903                 taskq_destroy(tqs->stqs_taskq[i]);
904         }
906         kmem_free(tqs->stqs_taskq, tqs->stqs_count * sizeof (taskq_t *));
907         tqs->stqs_taskq = NULL;
908 }
910 /*
911  * Dispatch a task to the appropriate taskq for the ZFS I/O type and priority.
912  * Note that a type may have multiple discrete taskqs to avoid lock contention
913  * on the taskq itself. In that case we choose which taskq at random by using
914  * the low bits of gethrtime().
915  */
916 void
```

```
spa_taskq_dispatch_ent(spa_t *spa, zio_type_t t, zio_taskq_type_t q,
    task_func_t *func, void *arg, uint_t flags, taskq_ent_t *ent)
{
        spa_taskqs_t *tqs = &spa->spa_zio_taskq[t][q];
        taskq_t *tq;

        ASSERT3P(tqs->stqs_taskq, !=, NULL);
        ASSERT3U(tqs->stqs_count, !=, 0);

        if (tqs->stqs_count == 1) {
                tq = tqs->stqs_taskq[0];
        } else {
                tq = tqs->stqs_taskq[gethrtime() % tqs->stqs_count];
        }

        taskq_dispatch_ent(tq, func, arg, flags, ent);
}

static void
spa_create_zio_taskqs(spa_t *spa)
{
        for (int t = 0; t < ZIO_TYPES; t++) {
                for (int q = 0; q < ZIO_TASKQ_TYPES; q++) {
                        spa_taskqs_init(spa, t, q);
                }
        }
}

#ifdef _KERNEL
static void
spa_thread(void *arg)
{
        callb_cpr_t cprinfo;

        spa_t *spa = arg;
        user_t *pu = PTOU(curproc);

        CALLB_CPR_INIT(&cprinfo, &spa->spa_proc_lock, callb_generic_cpr,
            spa->spa_name);

        ASSERT(curproc != &p0);
        (void) snprintf(pu->u_psargs, sizeof (pu->u_psargs),
            "zpool-%s", spa->spa_name);
        (void) strlcpy(pu->u_comm, pu->u_psargs, sizeof (pu->u_comm));

        /* bind this thread to the requested psrset */
        if (zio_taskq_psrset_bind != PS_NONE) {
                pool_lock();
                mutex_enter(&cpu_lock);
                mutex_enter(&pidlock);
                mutex_enter(&curproc->p_lock);

                if (cpupart_bind_thread(curthread, zio_taskq_psrset_bind,
                    0, NULL, NULL) == 0)  {
                        curthread->t_bind_pset = zio_taskq_psrset_bind;
                } else {
                        cmn_err(CE_WARN,
                            "Couldn't bind process for zfs pool \"%s\" to "
                            "pset %d\n", spa->spa_name, zio_taskq_psrset_bind);
                }

                mutex_exit(&curproc->p_lock);
                mutex_exit(&pidlock);
                mutex_exit(&cpu_lock);
                pool_unlock();
        }
```



```
        if (zio_taskq_sysdc) {
                sysdc_thread_enter(curthread, 100, 0);
        }

        spa->spa_proc = curproc;
        spa->spa_did = curthread->t_did;

        spa_create_zio_taskqs(spa);

        mutex_enter(&spa->spa_proc_lock);
        ASSERT(spa->spa_proc_state == SPA_PROC_CREATED);

        spa->spa_proc_state = SPA_PROC_ACTIVE;
        cv_broadcast(&spa->spa_proc_cv);

        CALLB_CPR_SAFE_BEGIN(&cprinfo);
        while (spa->spa_proc_state == SPA_PROC_ACTIVE)
                cv_wait(&spa->spa_proc_cv, &spa->spa_proc_lock);
        CALLB_CPR_SAFE_END(&cprinfo, &spa->spa_proc_lock);

        ASSERT(spa->spa_proc_state == SPA_PROC_DEACTIVATE);
        spa->spa_proc_state = SPA_PROC_GONE;
        spa->spa_proc = &p0;
        cv_broadcast(&spa->spa_proc_cv);
        CALLB_CPR_EXIT(&cprinfo);       /* drops spa_proc_lock */

        mutex_enter(&curproc->p_lock);
        lwp_exit();
}
#endif

/*
 * Activate an uninitialized pool.
 */
static void
spa_activate(spa_t *spa, int mode)
{
        ASSERT(spa->spa_state == POOL_STATE_UNINITIALIZED);

        spa->spa_state = POOL_STATE_ACTIVE;
        spa->spa_mode = mode;

        spa->spa_normal_class = metaslab_class_create(spa, zfs_metaslab_ops);
        spa->spa_log_class = metaslab_class_create(spa, zfs_metaslab_ops);

        /* Try to create a covering process */
        mutex_enter(&spa->spa_proc_lock);
        ASSERT(spa->spa_proc_state == SPA_PROC_NONE);
        ASSERT(spa->spa_proc == &p0);
        spa->spa_did = 0;

        /* Only create a process if we're going to be around a while. */
        if (spa_create_process && strcmp(spa->spa_name, TRYIMPORT_NAME) != 0) {
                if (newproc(spa_thread, (caddr_t)spa, syscid, maxclsyspri,
                    NULL, 0) == 0) {
                        spa->spa_proc_state = SPA_PROC_CREATED;
                        while (spa->spa_proc_state == SPA_PROC_CREATED) {
                                cv_wait(&spa->spa_proc_cv,
                                    &spa->spa_proc_lock);
                        }
                        ASSERT(spa->spa_proc_state == SPA_PROC_ACTIVE);
                        ASSERT(spa->spa_proc != &p0);
                        ASSERT(spa->spa_did != 0);
                } else {
#ifdef _KERNEL
```

```
1049                        cmn_err(CE_WARN,
1050                            "Couldn't create process for zfs pool \"%s\"\n",
1051                            spa->spa_name);
1052 #endif
1053                 }
1054         }
1055         mutex_exit(&spa->spa_proc_lock);

1057         /* If we didn't create a process, we need to create our taskqs. */
1058         if (spa->spa_proc == &p0) {
1059                 spa_create_zio_taskqs(spa);
1060         }

1062         list_create(&spa->spa_config_dirty_list, sizeof (vdev_t),
1063             offsetof(vdev_t, vdev_config_dirty_node));
1064         list_create(&spa->spa_state_dirty_list, sizeof (vdev_t),
1065             offsetof(vdev_t, vdev_state_dirty_node));

1067         txg_list_create(&spa->spa_vdev_txg_list,
1068             offsetof(struct vdev, vdev_txg_node));

1070         avl_create(&spa->spa_errlist_scrub,
1071             spa_error_entry_compare, sizeof (spa_error_entry_t),
1072             offsetof(spa_error_entry_t, se_avl));
1073         avl_create(&spa->spa_errlist_last,
1074             spa_error_entry_compare, sizeof (spa_error_entry_t),
1075             offsetof(spa_error_entry_t, se_avl));
1076 }

1078 /*
1079  * Opposite of spa_activate().
1080  */
1081 static void
1082 spa_deactivate(spa_t *spa)
1083 {
1084         ASSERT(spa->spa_sync_on == B_FALSE);
1085         ASSERT(spa->spa_dsl_pool == NULL);
1086         ASSERT(spa->spa_root_vdev == NULL);
1087         ASSERT(spa->spa_async_zio_root == NULL);
1088         ASSERT(spa->spa_state != POOL_STATE_UNINITIALIZED);

1090         txg_list_destroy(&spa->spa_vdev_txg_list);

1092         list_destroy(&spa->spa_config_dirty_list);
1093         list_destroy(&spa->spa_state_dirty_list);

1095         for (int t = 0; t < ZIO_TYPES; t++) {
1096                 for (int q = 0; q < ZIO_TASKQ_TYPES; q++) {
1097                         spa_taskqs_fini(spa, t, q);
1098                 }
1099         }

1101         metaslab_class_destroy(spa->spa_normal_class);
1102         spa->spa_normal_class = NULL;

1104         metaslab_class_destroy(spa->spa_log_class);
1105         spa->spa_log_class = NULL;

1107         /*
1108          * If this was part of an import or the open otherwise failed, we may
1109          * still have errors left in the queues.  Empty them just in case.
1110          */
1111         spa_errlog_drain(spa);

1113         avl_destroy(&spa->spa_errlist_scrub);
1114         avl_destroy(&spa->spa_errlist_last);
```



```
1116         spa->spa_state = POOL_STATE_UNINITIALIZED;

1118         mutex_enter(&spa->spa_proc_lock);
1119         if (spa->spa_proc_state != SPA_PROC_NONE) {
1120                 ASSERT(spa->spa_proc_state == SPA_PROC_ACTIVE);
1121                 spa->spa_proc_state = SPA_PROC_DEACTIVATE;
1122                 cv_broadcast(&spa->spa_proc_cv);
1123                 while (spa->spa_proc_state == SPA_PROC_DEACTIVATE) {
1124                         ASSERT(spa->spa_proc != &p0);
1125                         cv_wait(&spa->spa_proc_cv, &spa->spa_proc_lock);
1126                 }
1127                 ASSERT(spa->spa_proc_state == SPA_PROC_GONE);
1128                 spa->spa_proc_state = SPA_PROC_NONE;
1129         }
1130         ASSERT(spa->spa_proc == &p0);
1131         mutex_exit(&spa->spa_proc_lock);

1133         /*
1134          * We want to make sure spa_thread() has actually exited the ZFS
1135          * module, so that the module can't be unloaded out from underneath
1136          * it.
1137          */
1138         if (spa->spa_did != 0) {
1139                 thread_join(spa->spa_did);
1140                 spa->spa_did = 0;
1141         }
1142 }

1144 /*
1145  * Verify a pool configuration, and construct the vdev tree appropriately.  This
1146  * will create all the necessary vdevs in the appropriate layout, with each vdev
1147  * in the CLOSED state.  This will prep the pool before open/creation/import.
1148  * All vdev validation is done by the vdev_alloc() routine.
1149  */
1150 static int
1151 spa_config_parse(spa_t *spa, vdev_t **vdp, nvlist_t *nv, vdev_t *parent,
1152     uint_t id, int atype)
1153 {
1154         nvlist_t **child;
1155         uint_t children;
1156         int error;

1158         if ((error = vdev_alloc(spa, vdp, nv, parent, id, atype)) != 0)
1159                 return (error);

1161         if ((*vdp)->vdev_ops->vdev_op_leaf)
1162                 return (0);

1164         error = nvlist_lookup_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN,
1165             &child, &children);

1167         if (error == ENOENT)
1168                 return (0);

1170         if (error) {
1171                 vdev_free(*vdp);
1172                 *vdp = NULL;
1173                 return (SET_ERROR(EINVAL));
1174         }

1176         for (int c = 0; c < children; c++) {
1177                 vdev_t *vd;
1178                 if ((error = spa_config_parse(spa, &vd, child[c], *vdp, c,
1179                     atype)) != 0) {
1180                         vdev_free(*vdp);
```

```
1181                        *vdp = NULL;
1182                        return (error);
1183                }
1184        }

1186        ASSERT(*vdp != NULL);

1188        return (0);
1189 }

1191 /*
1192  * Opposite of spa_load().
1193  */
1194 static void
1195 spa_unload(spa_t *spa)
1196 {
1197        int i;

1199        ASSERT(MUTEX_HELD(&spa_namespace_lock));

1201        /*
1202         * Stop async tasks.
1203         */
1204        spa_async_suspend(spa);

1206        /*
1207         * Stop syncing.
1208         */
1209        if (spa->spa_sync_on) {
1210                txg_sync_stop(spa->spa_dsl_pool);
1211                spa->spa_sync_on = B_FALSE;
1212        }

1214        /*
1215         * Wait for any outstanding async I/O to complete.
1216         */
1217        if (spa->spa_async_zio_root != NULL) {
1218                (void) zio_wait(spa->spa_async_zio_root);
1219                spa->spa_async_zio_root = NULL;
1220        }

1222        bpobj_close(&spa->spa_deferred_bpobj);

1224        /*
1225         * Close the dsl pool.
1226         */
1227        if (spa->spa_dsl_pool) {
1228                dsl_pool_close(spa->spa_dsl_pool);
1229                spa->spa_dsl_pool = NULL;
1230                spa->spa_meta_objset = NULL;
1231        }

1233        ddt_unload(spa);

1235        spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

1237        /*
1238         * Drop and purge level 2 cache
1239         */
1240        spa_l2cache_drop(spa);

1242        /*
1243         * Close all vdevs.
1244         */
1245        if (spa->spa_root_vdev)
1246                vdev_free(spa->spa_root_vdev);
```



```
1247        ASSERT(spa->spa_root_vdev == NULL);

1249        for (i = 0; i < spa->spa_spares.sav_count; i++)
1250                vdev_free(spa->spa_spares.sav_vdevs[i]);
1251        if (spa->spa_spares.sav_vdevs) {
1252                kmem_free(spa->spa_spares.sav_vdevs,
1253                    spa->spa_spares.sav_count * sizeof (void *));
1254                spa->spa_spares.sav_vdevs = NULL;
1255        }
1256        if (spa->spa_spares.sav_config) {
1257                nvlist_free(spa->spa_spares.sav_config);
1258                spa->spa_spares.sav_config = NULL;
1259        }
1260        spa->spa_spares.sav_count = 0;

1262        for (i = 0; i < spa->spa_l2cache.sav_count; i++) {
1263                vdev_clear_stats(spa->spa_l2cache.sav_vdevs[i]);
1264                vdev_free(spa->spa_l2cache.sav_vdevs[i]);
1265        }
1266        if (spa->spa_l2cache.sav_vdevs) {
1267                kmem_free(spa->spa_l2cache.sav_vdevs,
1268                    spa->spa_l2cache.sav_count * sizeof (void *));
1269                spa->spa_l2cache.sav_vdevs = NULL;
1270        }
1271        if (spa->spa_l2cache.sav_config) {
1272                nvlist_free(spa->spa_l2cache.sav_config);
1273                spa->spa_l2cache.sav_config = NULL;
1274        }
1275        spa->spa_l2cache.sav_count = 0;

1277        spa->spa_async_suspended = 0;

1279        if (spa->spa_comment != NULL) {
1280                spa_strfree(spa->spa_comment);
1281                spa->spa_comment = NULL;
1282        }

1284        spa_config_exit(spa, SCL_ALL, FTAG);
1285 }

1287 /*
1288  * Load (or re-load) the current list of vdevs describing the active spares for
1289  * this pool.  When this is called, we have some form of basic information in
1290  * 'spa_spares.sav_config'.  We parse this into vdevs, try to open them, and
1291  * then re-generate a more complete list including status information.
1292  */
1293 static void
1294 spa_load_spares(spa_t *spa)
1295 {
1296        nvlist_t **spares;
1297        uint_t nspares;
1298        int i;
1299        vdev_t *vd, *tvd;

1301        ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

1303        /*
1304         * First, close and free any existing spare vdevs.
1305         */
1306        for (i = 0; i < spa->spa_spares.sav_count; i++) {
1307                vd = spa->spa_spares.sav_vdevs[i];

1309                /* Undo the call to spa_activate() below */
1310                if ((tvd = spa_lookup_by_guid(spa, vd->vdev_guid,
1311                    B_FALSE)) != NULL && tvd->vdev_isspare)
1312                        spa_spare_remove(tvd);
```

```
1313                vdev_close(vd);
1314                vdev_free(vd);
1315        }
1316
1317        if (spa->spa_spares.sav_vdevs)
1318                kmem_free(spa->spa_spares.sav_vdevs,
1319                    spa->spa_spares.sav_count * sizeof (void *));
1320
1321        if (spa->spa_spares.sav_config == NULL)
1322                nspares = 0;
1323        else
1324                VERIFY(nvlist_lookup_nvlist_array(spa->spa_spares.sav_config,
1325                    ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);
1326
1327        spa->spa_spares.sav_count = (int)nspares;
1328        spa->spa_spares.sav_vdevs = NULL;
1329
1330        if (nspares == 0)
1331                return;
1332
1333        /*
1334         * Construct the array of vdevs, opening them to get status in the
1335         * process.   For each spare, there is potentially two different vdev_t
1336         * structures associated with it: one in the list of spares (used only
1337         * for basic validation purposes) and one in the active vdev
1338         * configuration (if it's spared in).  During this phase we open and
1339         * validate each vdev on the spare list.  If the vdev also exists in the
1340         * active configuration, then we also mark this vdev as an active spare.
1341         */
1342        spa->spa_spares.sav_vdevs = kmem_alloc(nspares * sizeof (void *),
1343            KM_SLEEP);
1344        for (i = 0; i < spa->spa_spares.sav_count; i++) {
1345                VERIFY(spa_config_parse(spa, &vd, spares[i], NULL, 0,
1346                    VDEV_ALLOC_SPARE) == 0);
1347                ASSERT(vd != NULL);
1348
1349                spa->spa_spares.sav_vdevs[i] = vd;
1350
1351                if ((tvd = spa_lookup_by_guid(spa, vd->vdev_guid,
1352                    B_FALSE)) != NULL) {
1353                        if (!tvd->vdev_isspare)
1354                                spa_spare_add(tvd);
1355
1356                        /*
1357                         * We only mark the spare active if we were successfully
1358                         * able to load the vdev.  Otherwise, importing a pool
1359                         * with a bad active spare would result in strange
1360                         * behavior, because multiple pool would think the spare
1361                         * is actively in use.
1362                         *
1363                         * There is a vulnerability here to an equally bizarre
1364                         * circumstance, where a dead active spare is later
1365                         * brought back to life (onlined or otherwise).  Given
1366                         * the rarity of this scenario, and the extra complexity
1367                         * it adds, we ignore the possibility.
1368                         */
1369                        if (!vdev_is_dead(tvd))
1370                                spa_spare_activate(tvd);
1371                }
1372
1373                vd->vdev_top = vd;
1374                vd->vdev_aux = &spa->spa_spares;
1375
1376                if (vdev_open(vd) != 0)
1377                        continue;
```



```
1379                if (vdev_validate_aux(vd) == 0)
1380                        spa_spare_add(vd);
1381        }
1382
1383        /*
1384         * Recompute the stashed list of spares, with status information
1385         * this time.
1386         */
1387        VERIFY(nvlist_remove(spa->spa_spares.sav_config, ZPOOL_CONFIG_SPARES,
1388            DATA_TYPE_NVLIST_ARRAY) == 0);
1389
1390        spares = kmem_alloc(spa->spa_spares.sav_count * sizeof (void *),
1391            KM_SLEEP);
1392        for (i = 0; i < spa->spa_spares.sav_count; i++)
1393                spares[i] = vdev_config_generate(spa,
1394                    spa->spa_spares.sav_vdevs[i], B_TRUE, VDEV_CONFIG_SPARE);
1395        VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
1396            ZPOOL_CONFIG_SPARES, spares, spa->spa_spares.sav_count) == 0);
1397        for (i = 0; i < spa->spa_spares.sav_count; i++)
1398                nvlist_free(spares[i]);
1399        kmem_free(spares, spa->spa_spares.sav_count * sizeof (void *));
1400 }
1401
1402 /*
1403  * Load (or re-load) the current list of vdevs describing the active l2cache for
1404  * this pool.  When this is called, we have some form of basic information in
1405  * 'spa_l2cache.sav_config'.  We parse this into vdevs, try to open them, and
1406  * then re-generate a more complete list including status information.
1407  * Devices which are already active have their details maintained, and are
1408  * not re-opened.
1409  */
1410 static void
1411 spa_load_l2cache(spa_t *spa)
1412 {
1413        nvlist_t **l2cache;
1414        uint_t nl2cache;
1415        int i, j, oldnvdevs;
1416        uint64_t guid;
1417        vdev_t *vd, **oldvdevs, **newvdevs;
1418        spa_aux_vdev_t *sav = &spa->spa_l2cache;
1419
1420        ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
1421
1422        if (sav->sav_config != NULL) {
1423                VERIFY(nvlist_lookup_nvlist_array(sav->sav_config,
1424                    ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
1425                newvdevs = kmem_alloc(nl2cache * sizeof (void *), KM_SLEEP);
1426        } else {
1427                nl2cache = 0;
1428                newvdevs = NULL;
1429        }
1430
1431        oldvdevs = sav->sav_vdevs;
1432        oldnvdevs = sav->sav_count;
1433        sav->sav_vdevs = NULL;
1434        sav->sav_count = 0;
1435
1436        /*
1437         * Process new nvlist of vdevs.
1438         */
1439        for (i = 0; i < nl2cache; i++) {
1440                VERIFY(nvlist_lookup_uint64(l2cache[i], ZPOOL_CONFIG_GUID,
1441                    &guid) == 0);
1442
1443                newvdevs[i] = NULL;
1444                for (j = 0; j < oldnvdevs; j++) {
```

```
1445                        vd = oldvdevs[j];
1446                        if (vd != NULL && guid == vd->vdev_guid) {
1447                                /*
1448                                 * Retain previous vdev for add/remove ops.
1449                                 */
1450                                newvdevs[i] = vd;
1451                                oldvdevs[j] = NULL;
1452                                break;
1453                        }
1454                }

1456                if (newvdevs[i] == NULL) {
1457                        /*
1458                         * Create new vdev
1459                         */
1460                        VERIFY(spa_config_parse(spa, &vd, l2cache[i], NULL, 0,
1461                            VDEV_ALLOC_L2CACHE) == 0);
1462                        ASSERT(vd != NULL);
1463                        newvdevs[i] = vd;

1465                        /*
1466                         * Commit this vdev as an l2cache device,
1467                         * even if it fails to open.
1468                         */
1469                        spa_l2cache_add(vd);

1471                        vd->vdev_top = vd;
1472                        vd->vdev_aux = sav;

1474                        spa_l2cache_activate(vd);

1476                        if (vdev_open(vd) != 0)
1477                                continue;

1479                        (void) vdev_validate_aux(vd);

1481                        if (!vdev_is_dead(vd))
1482                                l2arc_add_vdev(spa, vd);
1483                }
1484        }

1486        /*
1487         * Purge vdevs that were dropped
1488         */
1489        for (i = 0; i < oldnvdevs; i++) {
1490                uint64_t pool;

1492                vd = oldvdevs[i];
1493                if (vd != NULL) {
1494                        ASSERT(vd->vdev_isl2cache);

1496                        if (spa_l2cache_exists(vd->vdev_guid, &pool) &&
1497                            pool != 0ULL && l2arc_vdev_present(vd))
1498                                l2arc_remove_vdev(vd);
1499                        vdev_clear_stats(vd);
1500                        vdev_free(vd);
1501                }
1502        }

1504        if (oldvdevs)
1505                kmem_free(oldvdevs, oldnvdevs * sizeof (void *));

1507        if (sav->sav_config == NULL)
1508                goto out;

1510        sav->sav_vdevs = newvdevs;
```



```
1511        sav->sav_count = (int)nl2cache;

1513        /*
1514         * Recompute the stashed list of l2cache devices, with status
1515         * information this time.
1516         */
1517        VERIFY(nvlist_remove(sav->sav_config, ZPOOL_CONFIG_L2CACHE,
1518            DATA_TYPE_NVLIST_ARRAY) == 0);

1520        l2cache = kmem_alloc(sav->sav_count * sizeof (void *), KM_SLEEP);
1521        for (i = 0; i < sav->sav_count; i++)
1522                l2cache[i] = vdev_config_generate(spa,
1523                    sav->sav_vdevs[i], B_TRUE, VDEV_CONFIG_L2CACHE);
1524        VERIFY(nvlist_add_nvlist_array(sav->sav_config,
1525            ZPOOL_CONFIG_L2CACHE, l2cache, sav->sav_count) == 0);
1526 out:
1527        for (i = 0; i < sav->sav_count; i++)
1528                nvlist_free(l2cache[i]);
1529        if (sav->sav_count)
1530                kmem_free(l2cache, sav->sav_count * sizeof (void *));
1531 }

1533 static int
1534 load_nvlist(spa_t *spa, uint64_t obj, nvlist_t **value)
1535 {
1536        dmu_buf_t *db;
1537        char *packed = NULL;
1538        size_t nvsize = 0;
1539        int error;
1540        *value = NULL;

1542        VERIFY(0 == dmu_bonus_hold(spa->spa_meta_objset, obj, FTAG, &db));
1543        nvsize = *(uint64_t *)db->db_data;
1544        dmu_buf_rele(db, FTAG);

1546        packed = kmem_alloc(nvsize, KM_SLEEP);
1547        error = dmu_read(spa->spa_meta_objset, obj, 0, nvsize, packed,
1548            DMU_READ_PREFETCH);
1549        if (error == 0)
1550                error = nvlist_unpack(packed, nvsize, value, 0);
1551        kmem_free(packed, nvsize);

1553        return (error);
1554 }

1556 /*
1557  * Checks to see if the given vdev could not be opened, in which case we post a
1558  * sysevent to notify the autoreplace code that the device has been removed.
1559  */
1560 static void
1561 spa_check_removed(vdev_t *vd)
1562 {
1563        for (int c = 0; c < vd->vdev_children; c++)
1564                spa_check_removed(vd->vdev_child[c]);

1566        if (vd->vdev_ops->vdev_op_leaf && vdev_is_dead(vd) &&
1567            !vd->vdev_ishole) {
1568                zfs_post_autoreplace(vd->vdev_spa, vd);
1569                spa_event_notify(vd->vdev_spa, vd, ESC_ZFS_VDEV_CHECK);
1570        }
1571 }

1573 /*
1574  * Validate the current config against the MOS config
1575  */
1576 static boolean_t
```

```
1577 spa_config_valid(spa_t *spa, nvlist_t *config)
1578 {
1579         vdev_t *mrvd, *rvd = spa->spa_root_vdev;
1580         nvlist_t *nv;

1582         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nv) == 0);

1584         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
1585         VERIFY(spa_config_parse(spa, &mrvd, nv, NULL, 0, VDEV_ALLOC_LOAD) == 0);

1587         ASSERT3U(rvd->vdev_children, ==, mrvd->vdev_children);

1589         /*
1590          * If we're doing a normal import, then build up any additional
1591          * diagnostic information about missing devices in this config.
1592          * We'll pass this up to the user for further processing.
1593          */
1594         if (!(spa->spa_import_flags & ZFS_IMPORT_MISSING_LOG)) {
1595                 nvlist_t **child, *nv;
1596                 uint64_t idx = 0;

1598                 child = kmem_alloc(rvd->vdev_children * sizeof (nvlist_t **),
1599                     KM_SLEEP);
1600                 VERIFY(nvlist_alloc(&nv, NV_UNIQUE_NAME, KM_SLEEP) == 0);

1602                 for (int c = 0; c < rvd->vdev_children; c++) {
1603                         vdev_t *tvd = rvd->vdev_child[c];
1604                         vdev_t *mtvd  = mrvd->vdev_child[c];

1606                         if (tvd->vdev_ops == &vdev_missing_ops &&
1607                             mtvd->vdev_ops != &vdev_missing_ops &&
1608                             mtvd->vdev_islog)
1609                                 child[idx++] = vdev_config_generate(spa, mtvd,
1610                                     B_FALSE, 0);
1611                 }

1613                 if (idx) {
1614                         VERIFY(nvlist_add_nvlist_array(nv,
1615                             ZPOOL_CONFIG_CHILDREN, child, idx) == 0);
1616                         VERIFY(nvlist_add_nvlist(spa->spa_load_info,
1617                             ZPOOL_CONFIG_MISSING_DEVICES, nv) == 0);

1619                         for (int i = 0; i < idx; i++)
1620                                 nvlist_free(child[i]);
1621                 }
1622                 nvlist_free(nv);
1623                 kmem_free(child, rvd->vdev_children * sizeof (char **));
1624         }

1626         /*
1627          * Compare the root vdev tree with the information we have
1628          * from the MOS config (mrvd). Check each top-level vdev
1629          * with the corresponding MOS config top-level (mtvd).
1630          */
1631         for (int c = 0; c < rvd->vdev_children; c++) {
1632                 vdev_t *tvd = rvd->vdev_child[c];
1633                 vdev_t *mtvd  = mrvd->vdev_child[c];

1635                 /*
1636                  * Resolve any "missing" vdevs in the current configuration.
1637                  * If we find that the MOS config has more accurate information
1638                  * about the top-level vdev then use that vdev instead.
1639                  */
1640                 if (tvd->vdev_ops == &vdev_missing_ops &&
1641                     mtvd->vdev_ops != &vdev_missing_ops) {
```



```
1643                         if (!(spa->spa_import_flags & ZFS_IMPORT_MISSING_LOG))
1644                                 continue;

1646                         /*
1647                          * Device specific actions.
1648                          */
1649                         if (mtvd->vdev_islog) {
1650                                 spa_set_log_state(spa, SPA_LOG_CLEAR);
1651                         } else {
1652                                 /*
1653                                  * XXX - once we have 'readonly' pool
1654                                  * support we should be able to handle
1655                                  * missing data devices by transitioning
1656                                  * the pool to readonly.
1657                                  */
1658                                 continue;
1659                         }

1661                         /*
1662                          * Swap the missing vdev with the data we were
1663                          * able to obtain from the MOS config.
1664                          */
1665                         vdev_remove_child(rvd, tvd);
1666                         vdev_remove_child(mrvd, mtvd);

1668                         vdev_add_child(rvd, mtvd);
1669                         vdev_add_child(mrvd, tvd);

1671                         spa_config_exit(spa, SCL_ALL, FTAG);
1672                         vdev_load(mtvd);
1673                         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

1675                         vdev_reopen(rvd);
1676                 } else if (mtvd->vdev_islog) {
1677                         /*
1678                          * Load the slog device's state from the MOS config
1679                          * since it's possible that the label does not
1680                          * contain the most up-to-date information.
1681                          */
1682                         vdev_load_log_state(tvd, mtvd);
1683                         vdev_reopen(tvd);
1684                 }
1685         }
1686         vdev_free(mrvd);
1687         spa_config_exit(spa, SCL_ALL, FTAG);

1689         /*
1690          * Ensure we were able to validate the config.
1691          */
1692         return (rvd->vdev_guid_sum == spa->spa_uberblock.ub_guid_sum);
1693 }

1695 /*
1696  * Check for missing log devices
1697  */
1698 static boolean_t
1699 spa_check_logs(spa_t *spa)
1700 {
1701         boolean_t rv = B_FALSE;

1703         switch (spa->spa_log_state) {
1704         case SPA_LOG_MISSING:
1705                 /* need to recheck in case slog has been restored */
1706         case SPA_LOG_UNKNOWN:
1707                 rv = (dmu_objset_find(spa->spa_name, zil_check_log_chain,
1708                     NULL, DS_FIND_CHILDREN) != 0);
```

```
1709                 if (rv)
1710                         spa_set_log_state(spa, SPA_LOG_MISSING);
1711                 break;
1712         }
1713         return (rv);
1714 }

1716 static boolean_t
1717 spa_passivate_log(spa_t *spa)
1718 {
1719         vdev_t *rvd = spa->spa_root_vdev;
1720         boolean_t slog_found = B_FALSE;

1722         ASSERT(spa_config_held(spa, SCL_ALLOC, RW_WRITER));

1724         if (!spa_has_slogs(spa))
1725                 return (B_FALSE);

1727         for (int c = 0; c < rvd->vdev_children; c++) {
1728                 vdev_t *tvd = rvd->vdev_child[c];
1729                 metaslab_group_t *mg = tvd->vdev_mg;

1731                 if (tvd->vdev_islog) {
1732                         metaslab_group_passivate(mg);
1733                         slog_found = B_TRUE;
1734                 }
1735         }

1737         return (slog_found);
1738 }

1740 static void
1741 spa_activate_log(spa_t *spa)
1742 {
1743         vdev_t *rvd = spa->spa_root_vdev;

1745         ASSERT(spa_config_held(spa, SCL_ALLOC, RW_WRITER));

1747         for (int c = 0; c < rvd->vdev_children; c++) {
1748                 vdev_t *tvd = rvd->vdev_child[c];
1749                 metaslab_group_t *mg = tvd->vdev_mg;

1751                 if (tvd->vdev_islog)
1752                         metaslab_group_activate(mg);
1753         }
1754 }

1756 int
1757 spa_offline_log(spa_t *spa)
1758 {
1759         int error;

1761         error = dmu_objset_find(spa_name(spa), zil_vdev_offline,
1762             NULL, DS_FIND_CHILDREN);
1763         if (error == 0) {
1764                 /*
1765                  * We successfully offlined the log device, sync out the
1766                  * current txg so that the "stubby" block can be removed
1767                  * by zil_sync().
1768                  */
1769                 txg_wait_synced(spa->spa_dsl_pool, 0);
1770         }
1771         return (error);
1772 }

1774 static void
```



```
1775 spa_aux_check_removed(spa_aux_vdev_t *sav)
1776 {
1777         for (int i = 0; i < sav->sav_count; i++)
1778                 spa_check_removed(sav->sav_vdevs[i]);
1779 }

1781 void
1782 spa_claim_notify(zio_t *zio)
1783 {
1784         spa_t *spa = zio->io_spa;

1786         if (zio->io_error)
1787                 return;

1789         mutex_enter(&spa->spa_props_lock);      /* any mutex will do */
1790         if (spa->spa_claim_max_txg < zio->io_bp->blk_birth)
1791                 spa->spa_claim_max_txg = zio->io_bp->blk_birth;
1792         mutex_exit(&spa->spa_props_lock);
1793 }

1795 typedef struct spa_load_error {
1796         uint64_t        sle_meta_count;
1797         uint64_t        sle_data_count;
1798 } spa_load_error_t;

1800 static void
1801 spa_load_verify_done(zio_t *zio)
1802 {
1803         blkptr_t *bp = zio->io_bp;
1804         spa_load_error_t *sle = zio->io_private;
1805         dmu_object_type_t type = BP_GET_TYPE(bp);
1806         int error = zio->io_error;

1808         if (error) {
1809                 if ((BP_GET_LEVEL(bp) != 0 || DMU_OT_IS_METADATA(type)) &&
1810                     type != DMU_OT_INTENT_LOG)
1811                         atomic_add_64(&sle->sle_meta_count, 1);
1812                 else
1813                         atomic_add_64(&sle->sle_data_count, 1);
1814         }
1815         zio_data_buf_free(zio->io_data, zio->io_size);
1816 }

1818 /*ARGSUSED*/
1819 static int
1820 spa_load_verify_cb(spa_t *spa, zilog_t *zilog, const blkptr_t *bp,
1821     const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
1822 {
1823         if (bp != NULL) {
1824                 zio_t *rio = arg;
1825                 size_t size = BP_GET_PSIZE(bp);
1826                 void *data = zio_data_buf_alloc(size);

1828                 zio_nowait(zio_read(rio, spa, bp, data, size,
1829                     spa_load_verify_done, rio->io_private, ZIO_PRIORITY_SCRUB,
1830                     ZIO_FLAG_SPECULATIVE | ZIO_FLAG_CANFAIL |
1831                     ZIO_FLAG_SCRUB | ZIO_FLAG_RAW, zb));
1832         }
1833         return (0);
1834 }

1836 static int
1837 spa_load_verify(spa_t *spa)
1838 {
1839         zio_t *rio;
1840         spa_load_error_t sle = { 0 };
```

```
1841         zpool_rewind_policy_t policy;
1842         boolean_t verify_ok = B_FALSE;
1843         int error;

1845         zpool_get_rewind_policy(spa->spa_config, &policy);

1847         if (policy.zrp_request & ZPOOL_NEVER_REWIND)
1848                 return (0);

1850         rio = zio_root(spa, NULL, &sle,
1851             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE);

1853         error = traverse_pool(spa, spa->spa_verify_min_txg,
1854             TRAVERSE_PRE | TRAVERSE_PREFETCH, spa_load_verify_cb, rio);

1856         (void) zio_wait(rio);

1858         spa->spa_load_meta_errors = sle.sle_meta_count;
1859         spa->spa_load_data_errors = sle.sle_data_count;

1861         if (!error && sle.sle_meta_count <= policy.zrp_maxmeta &&
1862             sle.sle_data_count <= policy.zrp_maxdata) {
1863                 int64_t loss = 0;

1865                 verify_ok = B_TRUE;
1866                 spa->spa_load_txg = spa->spa_uberblock.ub_txg;
1867                 spa->spa_load_txg_ts = spa->spa_uberblock.ub_timestamp;

1869                 loss = spa->spa_last_ubsync_txg_ts - spa->spa_load_txg_ts;
1870                 VERIFY(nvlist_add_uint64(spa->spa_load_info,
1871                     ZPOOL_CONFIG_LOAD_TIME, spa->spa_load_txg_ts) == 0);
1872                 VERIFY(nvlist_add_int64(spa->spa_load_info,
1873                     ZPOOL_CONFIG_REWIND_TIME, loss) == 0);
1874                 VERIFY(nvlist_add_uint64(spa->spa_load_info,
1875                     ZPOOL_CONFIG_LOAD_DATA_ERRORS, sle.sle_data_count) == 0);
1876         } else {
1877                 spa->spa_load_max_txg = spa->spa_uberblock.ub_txg;
1878         }

1880         if (error) {
1881                 if (error != ENXIO && error != EIO)
1882                         error = SET_ERROR(EIO);
1883                 return (error);
1884         }

1886         return (verify_ok ? 0 : EIO);
1887 }

1889 /*
1890  * Find a value in the pool props object.
1891  */
1892 static void
1893 spa_prop_find(spa_t *spa, zpool_prop_t prop, uint64_t *val)
1894 {
1895         (void) zap_lookup(spa->spa_meta_objset, spa->spa_pool_props_object,
1896             zpool_prop_to_name(prop), sizeof (uint64_t), 1, val);
1897 }

1899 /*
1900  * Find a value in the pool directory object.
1901  */
1902 static int
1903 spa_dir_prop(spa_t *spa, const char *name, uint64_t *val)
1904 {
1905         return (zap_lookup(spa->spa_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
1906             name, sizeof (uint64_t), 1, val));
```



```
1907 }

1909 static int
1910 spa_vdev_err(vdev_t *vdev, vdev_aux_t aux, int err)
1911 {
1912         vdev_set_state(vdev, B_TRUE, VDEV_STATE_CANT_OPEN, aux);
1913         return (err);
1914 }

1916 /*
1917  * Fix up config after a partly-completed split.  This is done with the
1918  * ZPOOL_CONFIG_SPLIT nvlist.  Both the splitting pool and the split-off
1919  * pool have that entry in their config, but only the splitting one contains
1920  * a list of all the guids of the vdevs that are being split off.
1921  *
1922  * This function determines what to do with that list: either rejoin
1923  * all the disks to the pool, or complete the splitting process.  To attempt
1924  * the rejoin, each disk that is offlined is marked online again, and
1925  * we do a reopen() call.  If the vdev label for every disk that was
1926  * marked online indicates it was successfully split off (VDEV_AUX_SPLIT_POOL)
1927  * then we call vdev_split() on each disk, and complete the split.
1928  *
1929  * Otherwise we leave the config alone, with all the vdevs in place in
1930  * the original pool.
1931  */
1932 static void
1933 spa_try_repair(spa_t *spa, nvlist_t *config)
1934 {
1935         uint_t extracted;
1936         uint64_t *glist;
1937         uint_t i, gcount;
1938         nvlist_t *nvl;
1939         vdev_t **vd;
1940         boolean_t attempt_reopen;

1942         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_SPLIT, &nvl) != 0)
1943                 return;

1945         /* check that the config is complete */
1946         if (nvlist_lookup_uint64_array(nvl, ZPOOL_CONFIG_SPLIT_LIST,
1947             &glist, &gcount) != 0)
1948                 return;

1950         vd = kmem_zalloc(gcount * sizeof (vdev_t *), KM_SLEEP);

1952         /* attempt to online all the vdevs & validate */
1953         attempt_reopen = B_TRUE;
1954         for (i = 0; i < gcount; i++) {
1955                 if (glist[i] == 0)      /* vdev is hole */
1956                         continue;

1958                 vd[i] = spa_lookup_by_guid(spa, glist[i], B_FALSE);
1959                 if (vd[i] == NULL) {
1960                         /*
1961                          * Don't bother attempting to reopen the disks;
1962                          * just do the split.
1963                          */
1964                         attempt_reopen = B_FALSE;
1965                 } else {
1966                         /* attempt to re-online it */
1967                         vd[i]->vdev_offline = B_FALSE;
1968                 }
1969         }

1971         if (attempt_reopen) {
1972                 vdev_reopen(spa->spa_root_vdev);
```

```
1974                 /* check each device to see what state it's in */
1975                 for (extracted = 0, i = 0; i < gcount; i++) {
1976                         if (vd[i] != NULL &&
1977                             vd[i]->vdev_stat.vs_aux != VDEV_AUX_SPLIT_POOL)
1978                                 break;
1979                         ++extracted;
1980                 }
1981         }
1983         /*
1984          * If every disk has been moved to the new pool, or if we never
1985          * even attempted to look at them, then we split them off for
1986          * good.
1987          */
1988         if (!attempt_reopen || gcount == extracted) {
1989                 for (i = 0; i < gcount; i++)
1990                         if (vd[i] != NULL)
1991                                 vdev_split(vd[i]);
1992                 vdev_reopen(spa->spa_root_vdev);
1993         }
1995         kmem_free(vd, gcount * sizeof (vdev_t *));
1996 }
1998 static int
1999 spa_load(spa_t *spa, spa_load_state_t state, spa_import_type_t type,
2000     boolean_t mosconfig)
2001 {
2002         nvlist_t *config = spa->spa_config;
2003         char *ereport = FM_EREPORT_ZFS_POOL;
2004         char *comment;
2005         int error;
2006         uint64_t pool_guid;
2007         nvlist_t *nvl;
2009         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID, &pool_guid))
2010                 return (SET_ERROR(EINVAL));
2012         ASSERT(spa->spa_comment == NULL);
2013         if (nvlist_lookup_string(config, ZPOOL_CONFIG_COMMENT, &comment) == 0)
2014                 spa->spa_comment = spa_strdup(comment);
2016         /*
2017          * Versioning wasn't explicitly added to the label until later, so if
2018          * it's not present treat it as the initial version.
2019          */
2020         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VERSION,
2021             &spa->spa_ubsync.ub_version) != 0)
2022                 spa->spa_ubsync.ub_version = SPA_VERSION_INITIAL;
2024         (void) nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG,
2025             &spa->spa_config_txg);
2027         if ((state == SPA_LOAD_IMPORT || state == SPA_LOAD_TRYIMPORT) &&
2028             spa_guid_exists(pool_guid, 0)) {
2029                 error = SET_ERROR(EEXIST);
2030         } else {
2031                 spa->spa_config_guid = pool_guid;
2033                 if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_SPLIT,
2034                     &nvl) == 0) {
2035                         VERIFY(nvlist_dup(nvl, &spa->spa_config_splitting,
2036                             KM_SLEEP) == 0);
2037                 }
```



```
2039                 nvlist_free(spa->spa_load_info);
2040                 spa->spa_load_info = fnvlist_alloc();
2042                 gethrestime(&spa->spa_loaded_ts);
2043                 error = spa_load_impl(spa, pool_guid, config, state, type,
2044                     mosconfig, &ereport);
2045         }
2047         spa->spa_minref = refcount_count(&spa->spa_refcount);
2048         if (error) {
2049                 if (error != EEXIST) {
2050                         spa->spa_loaded_ts.tv_sec = 0;
2051                         spa->spa_loaded_ts.tv_nsec = 0;
2052                 }
2053                 if (error != EBADF) {
2054                         zfs_ereport_post(ereport, spa, NULL, NULL, 0, 0);
2055                 }
2056         }
2057         spa->spa_load_state = error ? SPA_LOAD_ERROR : SPA_LOAD_NONE;
2058         spa->spa_ena = 0;
2060         return (error);
2061 }
2063 /*
2064  * Load an existing storage pool, using the pool's builtin spa_config as a
2065  * source of configuration information.
2066  */
2067 static int
2068 spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config,
2069     spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
2070     char **ereport)
2071 {
2072         int error = 0;
2073         nvlist_t *nvroot = NULL;
2074         nvlist_t *label;
2075         vdev_t *rvd;
2076         uberblock_t *ub = &spa->spa_uberblock;
2077         uint64_t children, config_cache_txg = spa->spa_config_txg;
2078         int orig_mode = spa->spa_mode;
2079         int parse;
2080         uint64_t obj;
2081         boolean_t missing_feat_write = B_FALSE;
2083         /*
2084          * If this is an untrusted config, access the pool in read-only mode.
2085          * This prevents things like resilvering recently removed devices.
2086          */
2087         if (!mosconfig)
2088                 spa->spa_mode = FREAD;
2090         ASSERT(MUTEX_HELD(&spa_namespace_lock));
2092         spa->spa_load_state = state;
2094         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot))
2095                 return (SET_ERROR(EINVAL));
2097         parse = (type == SPA_IMPORT_EXISTING ?
2098             VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);
2100         /*
2101          * Create "The Godfather" zio to hold all async IOs
2102          */
2103         spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
2104             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);
```

```
2106         /*
2107          * Parse the configuration into a vdev tree.  We explicitly set the
2108          * value that will be returned by spa_version() since parsing the
2109          * configuration requires knowing the version number.
2110          */
2111         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2112         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse);
2113         spa_config_exit(spa, SCL_ALL, FTAG);
2114
2115         if (error != 0)
2116                 return (error);
2117
2118         ASSERT(spa->spa_root_vdev == rvd);
2119
2120         if (type != SPA_IMPORT_ASSEMBLE) {
2121                 ASSERT(spa_guid(spa) == pool_guid);
2122         }
2123
2124         /*
2125          * Try to open all vdevs, loading each label in the process.
2126          */
2127         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2128         error = vdev_open(rvd);
2129         spa_config_exit(spa, SCL_ALL, FTAG);
2130         if (error != 0)
2131                 return (error);
2132
2133         /*
2134          * We need to validate the vdev labels against the configuration that
2135          * we have in hand, which is dependent on the setting of mosconfig. If
2136          * mosconfig is true then we're validating the vdev labels based on
2137          * that config.  Otherwise, we're validating against the cached config
2138          * (zpool.cache) that was read when we loaded the zfs module, and then
2139          * later we will recursively call spa_load() and validate against
2140          * the vdev config.
2141          *
2142          * If we're assembling a new pool that's been split off from an
2143          * existing pool, the labels haven't yet been updated so we skip
2144          * validation for now.
2145          */
2146         if (type != SPA_IMPORT_ASSEMBLE) {
2147                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2148                 error = vdev_validate(rvd, mosconfig);
2149                 spa_config_exit(spa, SCL_ALL, FTAG);
2150
2151                 if (error != 0)
2152                         return (error);
2153
2154                 if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2155                         return (SET_ERROR(ENXIO));
2156         }
2157
2158         /*
2159          * Find the best uberblock.
2160          */
2161         vdev_uberblock_load(rvd, ub, &label);
2162
2163         /*
2164          * If we weren't able to find a single valid uberblock, return failure.
2165          */
2166         if (ub->ub_txg == 0) {
2167                 nvlist_free(label);
2168                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, ENXIO));
2169         }
```



```
2171         /*
2172          * If the pool has an unsupported version we can't open it.
2173          */
2174         if (!SPA_VERSION_IS_SUPPORTED(ub->ub_version)) {
2175                 nvlist_free(label);
2176                 return (spa_vdev_err(rvd, VDEV_AUX_VERSION_NEWER, ENOTSUP));
2177         }
2178
2179         if (ub->ub_version >= SPA_VERSION_FEATURES) {
2180                 nvlist_t *features;
2181
2182                 /*
2183                  * If we weren't able to find what's necessary for reading the
2184                  * MOS in the label, return failure.
2185                  */
2186                 if (label == NULL || nvlist_lookup_nvlist(label,
2187                     ZPOOL_CONFIG_FEATURES_FOR_READ, &features) != 0) {
2188                         nvlist_free(label);
2189                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2190                             ENXIO));
2191                 }
2192
2193                 /*
2194                  * Update our in-core representation with the definitive values
2195                  * from the label.
2196                  */
2197                 nvlist_free(spa->spa_label_features);
2198                 VERIFY(nvlist_dup(features, &spa->spa_label_features, 0) == 0);
2199         }
2200
2201         nvlist_free(label);
2202
2203         /*
2204          * Look through entries in the label nvlist's features_for_read. If
2205          * there is a feature listed there which we don't understand then we
2206          * cannot open a pool.
2207          */
2208         if (ub->ub_version >= SPA_VERSION_FEATURES) {
2209                 nvlist_t *unsup_feat;
2210
2211                 VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
2212                     0);
2213
2214                 for (nvpair_t *nvp = nvlist_next_nvpair(spa->spa_label_features,
2215                     NULL); nvp != NULL;
2216                     nvp = nvlist_next_nvpair(spa->spa_label_features, nvp)) {
2217                         if (!zfeature_is_supported(nvpair_name(nvp))) {
2218                                 VERIFY(nvlist_add_string(unsup_feat,
2219                                     nvpair_name(nvp), "") == 0);
2220                         }
2221                 }
2222
2223                 if (!nvlist_empty(unsup_feat)) {
2224                         VERIFY(nvlist_add_nvlist(spa->spa_load_info,
2225                             ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
2226                         nvlist_free(unsup_feat);
2227                         return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2228                             ENOTSUP));
2229                 }
2230
2231                 nvlist_free(unsup_feat);
2232         }
2233
2234         /*
2235          * If the vdev guid sum doesn't match the uberblock, we have an
2236          * incomplete configuration.  We first check to see if the pool
```

```
2237          * is aware of the complete config (i.e ZPOOL_CONFIG_VDEV_CHILDREN).
2238          * If it is, defer the vdev_guid_sum check till later so we
2239          * can handle missing vdevs.
2240          */
2241         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VDEV_CHILDREN,
2242             &children) != 0 && mosconfig && type != SPA_IMPORT_ASSEMBLE &&
2243             rvd->vdev_guid_sum != ub->ub_guid_sum)
2244                 return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));

2246         if (type != SPA_IMPORT_ASSEMBLE && spa->spa_config_splitting) {
2247                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2248                 spa_try_repair(spa, config);
2249                 spa_config_exit(spa, SCL_ALL, FTAG);
2250                 nvlist_free(spa->spa_config_splitting);
2251                 spa->spa_config_splitting = NULL;
2252         }

2254         /*
2255          * Initialize internal SPA structures.
2256          */
2257         spa->spa_state = POOL_STATE_ACTIVE;
2258         spa->spa_ubsync = spa->spa_uberblock;
2259         spa->spa_verify_min_txg = spa->spa_extreme_rewind ?
2260             TXG_INITIAL - 1 : spa_last_synced_txg(spa) - TXG_DEFER_SIZE - 1;
2261         spa->spa_first_txg = spa->spa_last_ubsync_txg ?
2262             spa->spa_last_ubsync_txg : spa_last_synced_txg(spa) + 1;
2263         spa->spa_claim_max_txg = spa->spa_first_txg;
2264         spa->spa_prev_software_version = ub->ub_software_version;

2266         error = dsl_pool_init(spa, spa->spa_first_txg, &spa->spa_dsl_pool);
2267         if (error)
2268                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2269         spa->spa_meta_objset = spa->spa_dsl_pool->dp_meta_objset;

2271         if (spa_dir_prop(spa, DMU_POOL_CONFIG, &spa->spa_config_object) != 0)
2272                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2274         if (spa_version(spa) >= SPA_VERSION_FEATURES) {
2275                 boolean_t missing_feat_read = B_FALSE;
2276                 nvlist_t *unsup_feat, *enabled_feat;

2278                 if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_READ,
2279                     &spa->spa_feat_for_read_obj) != 0) {
2280                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2281                 }

2283                 if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_WRITE,
2284                     &spa->spa_feat_for_write_obj) != 0) {
2285                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2286                 }

2288                 if (spa_dir_prop(spa, DMU_POOL_FEATURE_DESCRIPTIONS,
2289                     &spa->spa_feat_desc_obj) != 0) {
2290                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2291                 }

2293                 enabled_feat = fnvlist_alloc();
2294                 unsup_feat = fnvlist_alloc();

2296                 if (!feature_is_supported(spa->spa_meta_objset,
2297                     spa->spa_feat_for_read_obj, spa->spa_feat_desc_obj,
2298                     unsup_feat, enabled_feat))
2299                         missing_feat_read = B_TRUE;

2301                 if (spa_writeable(spa) || state == SPA_LOAD_TRYIMPORT) {
2302                         if (!feature_is_supported(spa->spa_meta_objset,
```



```
2303                             spa->spa_feat_for_write_obj, spa->spa_feat_desc_obj,
2304                             unsup_feat, enabled_feat)) {
2305                                 missing_feat_write = B_TRUE;
2306                         }
2307                 }

2309                 fnvlist_add_nvlist(spa->spa_load_info,
2310                     ZPOOL_CONFIG_ENABLED_FEAT, enabled_feat);

2312                 if (!nvlist_empty(unsup_feat)) {
2313                         fnvlist_add_nvlist(spa->spa_load_info,
2314                             ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat);
2315                 }

2317                 fnvlist_free(enabled_feat);
2318                 fnvlist_free(unsup_feat);

2320                 if (!missing_feat_read) {
2321                         fnvlist_add_boolean(spa->spa_load_info,
2322                             ZPOOL_CONFIG_CAN_RDONLY);
2323                 }

2325                 /*
2326                  * If the state is SPA_LOAD_TRYIMPORT, our objective is
2327                  * twofold: to determine whether the pool is available for
2328                  * import in read-write mode and (if it is not) whether the
2329                  * pool is available for import in read-only mode. If the pool
2330                  * is available for import in read-write mode, it is displayed
2331                  * as available in userland; if it is not available for import
2332                  * in read-only mode, it is displayed as unavailable in
2333                  * userland. If the pool is available for import in read-only
2334                  * mode but not read-write mode, it is displayed as unavailable
2335                  * in userland with a special note that the pool is actually
2336                  * available for open in read-only mode.
2337                  *
2338                  * As a result, if the state is SPA_LOAD_TRYIMPORT and we are
2339                  * missing a feature for write, we must first determine whether
2340                  * the pool can be opened read-only before returning to
2341                  * userland in order to know whether to display the
2342                  * abovementioned note.
2343                  */
2344                 if (missing_feat_read || (missing_feat_write &&
2345                     spa_writeable(spa))) {
2346                         return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2347                             ENOTSUP));
2348                 }
2349         }

2351         spa->spa_is_initializing = B_TRUE;
2352         error = dsl_pool_open(spa->spa_dsl_pool);
2353         spa->spa_is_initializing = B_FALSE;
2354         if (error != 0)
2355                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2357         if (!mosconfig) {
2358                 uint64_t hostid;
2359                 nvlist_t *policy = NULL, *nvconfig;

2361                 if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2362                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2364                 if (!spa_is_root(spa) && nvlist_lookup_uint64(nvconfig,
2365                     ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
2366                         char *hostname;
2367                         unsigned long myhostid = 0;
```

```
2369                        VERIFY(nvlist_lookup_string(nvconfig,
2370                            ZPOOL_CONFIG_HOSTNAME, &hostname) == 0);

2372 #ifdef  _KERNEL
2373                        myhostid = zone_get_hostid(NULL);
2374 #else   /* _KERNEL */
2375                        /*
2376                         * We're emulating the system's hostid in userland, so
2377                         * we can't use zone_get_hostid().
2378                         */
2379                        (void) ddi_strtoul(hw_serial, NULL, 10, &myhostid);
2380 #endif  /* _KERNEL */
2381                        if (hostid != 0 && myhostid != 0 &&
2382                            hostid != myhostid) {
2383                                nvlist_free(nvconfig);
2384                                cmn_err(CE_WARN, "pool '%s' could not be "
2385                                    "loaded as it was last accessed by "
2386                                    "another system (host: %s hostid: 0x%lx). "
2387                                    "See: http://illumos.org/msg/ZFS-8000-EY",
2388                                    spa_name(spa), hostname,
2389                                    (unsigned long)hostid);
2390                                return (SET_ERROR(EBADF));
2391                        }
2392                }
2393                if (nvlist_lookup_nvlist(spa->spa_config,
2394                    ZPOOL_REWIND_POLICY, &policy) == 0)
2395                        VERIFY(nvlist_add_nvlist(nvconfig,
2396                            ZPOOL_REWIND_POLICY, policy) == 0);

2398                spa_config_set(spa, nvconfig);
2399                spa_unload(spa);
2400                spa_deactivate(spa);
2401                spa_activate(spa, orig_mode);

2403                return (spa_load(spa, state, SPA_IMPORT_EXISTING, B_TRUE));
2404        }

2406        if (spa_dir_prop(spa, DMU_POOL_SYNC_BPOBJ, &obj) != 0)
2407                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2408        error = bpobj_open(&spa->spa_deferred_bpobj, spa->spa_meta_objset, obj);
2409        if (error != 0)
2410                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2412        /*
2413         * Load the bit that tells us to use the new accounting function
2414         * (raid-z deflation).  If we have an older pool, this will not
2415         * be present.
2416         */
2417        error = spa_dir_prop(spa, DMU_POOL_DEFLATE, &spa->spa_deflate);
2418        if (error != 0 && error != ENOENT)
2419                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2421        error = spa_dir_prop(spa, DMU_POOL_CREATION_VERSION,
2422            &spa->spa_creation_version);
2423        if (error != 0 && error != ENOENT)
2424                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2426        /*
2427         * Load the persistent error log.  If we have an older pool, this will
2428         * not be present.
2429         */
2430        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_LAST, &spa->spa_errlog_last);
2431        if (error != 0 && error != ENOENT)
2432                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2434        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_SCRUB,
```



```
2435            &spa->spa_errlog_scrub);
2436        if (error != 0 && error != ENOENT)
2437                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2439        /*
2440         * Load the history object.  If we have an older pool, this
2441         * will not be present.
2442         */
2443        error = spa_dir_prop(spa, DMU_POOL_HISTORY, &spa->spa_history);
2444        if (error != 0 && error != ENOENT)
2445                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2447        /*
2448         * If we're assembling the pool from the split-off vdevs of
2449         * an existing pool, we don't want to attach the spares & cache
2450         * devices.
2451         */

2453        /*
2454         * Load any hot spares for this pool.
2455         */
2456        error = spa_dir_prop(spa, DMU_POOL_SPARES, &spa->spa_spares.sav_object);
2457        if (error != 0 && error != ENOENT)
2458                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2459        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2460                ASSERT(spa_version(spa) >= SPA_VERSION_SPARES);
2461                if (load_nvlist(spa, spa->spa_spares.sav_object,
2462                    &spa->spa_spares.sav_config) != 0)
2463                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2465                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2466                spa_load_spares(spa);
2467                spa_config_exit(spa, SCL_ALL, FTAG);
2468        } else if (error == 0) {
2469                spa->spa_spares.sav_sync = B_TRUE;
2470        }

2472        /*
2473         * Load any level 2 ARC devices for this pool.
2474         */
2475        error = spa_dir_prop(spa, DMU_POOL_L2CACHE,
2476            &spa->spa_l2cache.sav_object);
2477        if (error != 0 && error != ENOENT)
2478                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2479        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2480                ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
2481                if (load_nvlist(spa, spa->spa_l2cache.sav_object,
2482                    &spa->spa_l2cache.sav_config) != 0)
2483                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2485                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2486                spa_load_l2cache(spa);
2487                spa_config_exit(spa, SCL_ALL, FTAG);
2488        } else if (error == 0) {
2489                spa->spa_l2cache.sav_sync = B_TRUE;
2490        }

2492        spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);

2494        error = spa_dir_prop(spa, DMU_POOL_PROPS, &spa->spa_pool_props_object);
2495        if (error && error != ENOENT)
2496                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2498        if (error == 0) {
2499                uint64_t autoreplace;
```

```
2501                spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
2502                spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
2503                spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
2504                spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
2505                spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
2506                spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO,
2507                    &spa->spa_dedup_ditto);

2509                spa->spa_autoreplace = (autoreplace != 0);
2510        }

2512        /*
2513         * If the 'autoreplace' property is set, then post a resource notifying
2514         * the ZFS DE that it should not issue any faults for unopenable
2515         * devices.  We also iterate over the vdevs, and post a sysevent for any
2516         * unopenable vdevs so that the normal autoreplace handler can take
2517         * over.
2518         */
2519        if (spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT) {
2520                spa_check_removed(spa->spa_root_vdev);
2521                /*
2522                 * For the import case, this is done in spa_import(), because
2523                 * at this point we're using the spare definitions from
2524                 * the MOS config, not necessarily from the userland config.
2525                 */
2526                if (state != SPA_LOAD_IMPORT) {
2527                        spa_aux_check_removed(&spa->spa_spares);
2528                        spa_aux_check_removed(&spa->spa_l2cache);
2529                }
2530        }

2532        /*
2533         * Load the vdev state for all toplevel vdevs.
2534         */
2535        vdev_load(rvd);

2537        /*
2538         * Propagate the leaf DTLs we just loaded all the way up the tree.
2539         */
2540        spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2541        vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
2542        spa_config_exit(spa, SCL_ALL, FTAG);

2544        /*
2545         * Load the DDTs (dedup tables).
2546         */
2547        error = ddt_load(spa);
2548        if (error != 0)
2549                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2551        spa_update_dspace(spa);

2553        /*
2554         * Validate the config, using the MOS config to fill in any
2555         * information which might be missing.  If we fail to validate
2556         * the config then declare the pool unfit for use. If we're
2557         * assembling a pool from a split, the log is not transferred
2558         * over.
2559         */
2560        if (type != SPA_IMPORT_ASSEMBLE) {
2561                nvlist_t *nvconfig;

2563                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2564                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2566                if (!spa_config_valid(spa, nvconfig)) {
```



```
2567                        nvlist_free(nvconfig);
2568                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM,
2569                            ENXIO));
2570                }
2571                nvlist_free(nvconfig);

2573                /*
2574                 * Now that we've validated the config, check the state of the
2575                 * root vdev.  If it can't be opened, it indicates one or
2576                 * more toplevel vdevs are faulted.
2577                 */
2578                if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2579                        return (SET_ERROR(ENXIO));

2581                if (spa_check_logs(spa)) {
2582                        *ereport = FM_EREPORT_ZFS_LOG_REPLAY;
2583                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
2584                }
2585        }

2587        if (missing_feat_write) {
2588                ASSERT(state == SPA_LOAD_TRYIMPORT);

2590                /*
2591                 * At this point, we know that we can open the pool in
2592                 * read-only mode but not read-write mode. We now have enough
2593                 * information and can return to userland.
2594                 */
2595                return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
2596        }

2598        /*
2599         * We've successfully opened the pool, verify that we're ready
2600         * to start pushing transactions.
2601         */
2602        if (state != SPA_LOAD_TRYIMPORT) {
2603                if (error = spa_load_verify(spa))
2604                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2605                            error));
2606        }

2608        if (spa_writeable(spa) && (state == SPA_LOAD_RECOVER ||
2609            spa->spa_load_max_txg == UINT64_MAX)) {
2610                dmu_tx_t *tx;
2611                int need_update = B_FALSE;

2613                ASSERT(state != SPA_LOAD_TRYIMPORT);

2615                /*
2616                 * Claim log blocks that haven't been committed yet.
2617                 * This must all happen in a single txg.
2618                 * Note: spa_claim_max_txg is updated by spa_claim_notify(),
2619                 * invoked from zil_claim_log_block()'s i/o done callback.
2620                 * Price of rollback is that we abandon the log.
2621                 */
2622                spa->spa_claiming = B_TRUE;

2624                tx = dmu_tx_create_assigned(spa_get_dsl(spa),
2625                    spa_first_txg(spa));
2626                (void) dmu_objset_find(spa_name(spa),
2627                    zil_claim, tx, DS_FIND_CHILDREN);
2628                dmu_tx_commit(tx);

2630                spa->spa_claiming = B_FALSE;

2632                spa_set_log_state(spa, SPA_LOG_GOOD);
```

```
2633                 spa->spa_sync_on = B_TRUE;
2634                 txg_sync_start(spa->spa_dsl_pool);

2636                 /*
2637                  * Wait for all claims to sync.  We sync up to the highest
2638                  * claimed log block birth time so that claimed log blocks
2639                  * don't appear to be from the future.  spa_claim_max_txg
2640                  * will have been set for us by either zil_check_log_chain()
2641                  * (invoked from spa_check_logs()) or zil_claim() above.
2642                  */
2643                 txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_txg);

2645                 /*
2646                  * If the config cache is stale, or we have uninitialized
2647                  * metaslabs (see spa_vdev_add()), then update the config.
2648                  *
2649                  * If this is a verbatim import, trust the current
2650                  * in-core spa_config and update the disk labels.
2651                  */
2652                 if (config_cache_txg != spa->spa_config_txg ||
2653                     state == SPA_LOAD_IMPORT ||
2654                     state == SPA_LOAD_RECOVER ||
2655                     (spa->spa_import_flags & ZFS_IMPORT_VERBATIM))
2656                         need_update = B_TRUE;

2658                 for (int c = 0; c < rvd->vdev_children; c++)
2659                         if (rvd->vdev_child[c]->vdev_ms_array == 0)
2660                                 need_update = B_TRUE;

2662                 /*
2663                  * Update the config cache asychronously in case we're the
2664                  * root pool, in which case the config cache isn't writable yet.
2665                  */
2666                 if (need_update)
2667                         spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);

2669                 /*
2670                  * Check all DTLs to see if anything needs resilvering.
2671                  */
2672                 if (!dsl_scan_resilvering(spa->spa_dsl_pool) &&
2673                     vdev_resilver_needed(rvd, NULL, NULL))
2674                         spa_async_request(spa, SPA_ASYNC_RESILVER);

2676                 /*
2677                  * Log the fact that we booted up (so that we can detect if
2678                  * we rebooted in the middle of an operation).
2679                  */
2680                 spa_history_log_version(spa, "open");

2682                 /*
2683                  * Delete any inconsistent datasets.
2684                  */
2685                 (void) dmu_objset_find(spa_name(spa),
2686                     dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);

2688                 /*
2689                  * Clean up any stale temporary dataset userrefs.
2690                  */
2691                 dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
2692         }

2694         return (0);
2695 }

2697 static int
2698 spa_load_retry(spa_t *spa, spa_load_state_t state, int mosconfig)
```



```
2699 {
2700         int mode = spa->spa_mode;

2702         spa_unload(spa);
2703         spa_deactivate(spa);

2705         spa->spa_load_max_txg--;

2707         spa_activate(spa, mode);
2708         spa_async_suspend(spa);

2710         return (spa_load(spa, state, SPA_IMPORT_EXISTING, mosconfig));
2711 }

2713 /*
2714  * If spa_load() fails this function will try loading prior txg's. If
2715  * 'state' is SPA_LOAD_RECOVER and one of these loads succeeds the pool
2716  * will be rewound to that txg. If 'state' is not SPA_LOAD_RECOVER this
2717  * function will not rewind the pool and will return the same error as
2718  * spa_load().
2719  */
2720 static int
2721 spa_load_best(spa_t *spa, spa_load_state_t state, int mosconfig,
2722     uint64_t max_request, int rewind_flags)
2723 {
2724         nvlist_t *loadinfo = NULL;
2725         nvlist_t *config = NULL;
2726         int load_error, rewind_error;
2727         uint64_t safe_rewind_txg;
2728         uint64_t min_txg;

2730         if (spa->spa_load_txg && state == SPA_LOAD_RECOVER) {
2731                 spa->spa_load_max_txg = spa->spa_load_txg;
2732                 spa_set_log_state(spa, SPA_LOG_CLEAR);
2733         } else {
2734                 spa->spa_load_max_txg = max_request;
2735         }

2737         load_error = rewind_error = spa_load(spa, state, SPA_IMPORT_EXISTING,
2738             mosconfig);
2739         if (load_error == 0)
2740                 return (0);

2742         if (spa->spa_root_vdev != NULL)
2743                 config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);

2745         spa->spa_last_ubsync_txg = spa->spa_uberblock.ub_txg;
2746         spa->spa_last_ubsync_txg_ts = spa->spa_uberblock.ub_timestamp;

2748         if (rewind_flags & ZPOOL_NEVER_REWIND) {
2749                 nvlist_free(config);
2750                 return (load_error);
2751         }

2753         if (state == SPA_LOAD_RECOVER) {
2754                 /* Price of rolling back is discarding txgs, including log */
2755                 spa_set_log_state(spa, SPA_LOG_CLEAR);
2756         } else {
2757                 /*
2758                  * If we aren't rolling back save the load info from our first
2759                  * import attempt so that we can restore it after attempting
2760                  * to rewind.
2761                  */
2762                 loadinfo = spa->spa_load_info;
2763                 spa->spa_load_info = fnvlist_alloc();
2764         }
```

```
2766         spa->spa_load_max_txg = spa->spa_last_ubsync_txg;
2767         safe_rewind_txg = spa->spa_last_ubsync_txg - TXG_DEFER_SIZE;
2768         min_txg = (rewind_flags & ZPOOL_EXTREME_REWIND) ?
2769             TXG_INITIAL : safe_rewind_txg;

2771         /*
2772          * Continue as long as we're finding errors, we're still within
2773          * the acceptable rewind range, and we're still finding uberblocks
2774          */
2775         while (rewind_error && spa->spa_uberblock.ub_txg >= min_txg &&
2776             spa->spa_uberblock.ub_txg <= spa->spa_load_max_txg) {
2777                 if (spa->spa_load_max_txg < safe_rewind_txg)
2778                         spa->spa_extreme_rewind = B_TRUE;
2779                 rewind_error = spa_load_retry(spa, state, mosconfig);
2780         }

2782         spa->spa_extreme_rewind = B_FALSE;
2783         spa->spa_load_max_txg = UINT64_MAX;

2785         if (config && (rewind_error || state != SPA_LOAD_RECOVER))
2786                 spa_config_set(spa, config);

2788         if (state == SPA_LOAD_RECOVER) {
2789                 ASSERT3P(loadinfo, ==, NULL);
2790                 return (rewind_error);
2791         } else {
2792                 /* Store the rewind info as part of the initial load info */
2793                 fnvlist_add_nvlist(loadinfo, ZPOOL_CONFIG_REWIND_INFO,
2794                     spa->spa_load_info);

2796                 /* Restore the initial load info */
2797                 fnvlist_free(spa->spa_load_info);
2798                 spa->spa_load_info = loadinfo;

2800                 return (load_error);
2801         }
2802 }

2804 /*
2805  * Pool Open/Import
2806  *
2807  * The import case is identical to an open except that the configuration is sent
2808  * down from userland, instead of grabbed from the configuration cache.  For the
2809  * case of an open, the pool configuration will exist in the
2810  * POOL_STATE_UNINITIALIZED state.
2811  *
2812  * The stats information (gen/count/ustats) is used to gather vdev statistics at
2813  * the same time open the pool, without having to keep around the spa_t in some
2814  * ambiguous state.
2815  */
2816 static int
2817 spa_open_common(const char *pool, spa_t **spapp, void *tag, nvlist_t *nvpolicy,
2818     nvlist_t **config)
2819 {
2820         spa_t *spa;
2821         spa_load_state_t state = SPA_LOAD_OPEN;
2822         int error;
2823         int locked = B_FALSE;

2825         *spapp = NULL;

2827         /*
2828          * As disgusting as this is, we need to support recursive calls to this
2829          * function because dsl_dir_open() is called during spa_load(), and ends
2830          * up calling spa_open() again.  The real fix is to figure out how to
```



```
2831          * avoid dsl_dir_open() calling this in the first place.
2832          */
2833         if (mutex_owner(&spa_namespace_lock) != curthread) {
2834                 mutex_enter(&spa_namespace_lock);
2835                 locked = B_TRUE;
2836         }

2838         if ((spa = spa_lookup(pool)) == NULL) {
2839                 if (locked)
2840                         mutex_exit(&spa_namespace_lock);
2841                 return (SET_ERROR(ENOENT));
2842         }

2844         if (spa->spa_state == POOL_STATE_UNINITIALIZED) {
2845                 zpool_rewind_policy_t policy;

2847                 zpool_get_rewind_policy(nvpolicy ? nvpolicy : spa->spa_config,
2848                     &policy);
2849                 if (policy.zrp_request & ZPOOL_DO_REWIND)
2850                         state = SPA_LOAD_RECOVER;

2852                 spa_activate(spa, spa_mode_global);

2854                 if (state != SPA_LOAD_RECOVER)
2855                         spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;

2857                 error = spa_load_best(spa, state, B_FALSE, policy.zrp_txg,
2858                     policy.zrp_request);

2860                 if (error == EBADF) {
2861                         /*
2862                          * If vdev_validate() returns failure (indicated by
2863                          * EBADF), it indicates that one of the vdevs indicates
2864                          * that the pool has been exported or destroyed.  If
2865                          * this is the case, the config cache is out of sync and
2866                          * we should remove the pool from the namespace.
2867                          */
2868                         spa_unload(spa);
2869                         spa_deactivate(spa);
2870                         spa_config_sync(spa, B_TRUE, B_TRUE);
2871                         spa_remove(spa);
2872                         if (locked)
2873                                 mutex_exit(&spa_namespace_lock);
2874                         return (SET_ERROR(ENOENT));
2875                 }

2877                 if (error) {
2878                         /*
2879                          * We can't open the pool, but we still have useful
2880                          * information: the state of each vdev after the
2881                          * attempted vdev_open().  Return this to the user.
2882                          */
2883                         if (config != NULL && spa->spa_config) {
2884                                 VERIFY(nvlist_dup(spa->spa_config, config,
2885                                     KM_SLEEP) == 0);
2886                                 VERIFY(nvlist_add_nvlist(*config,
2887                                     ZPOOL_CONFIG_LOAD_INFO,
2888                                     spa->spa_load_info) == 0);
2889                         }
2890                         spa_unload(spa);
2891                         spa_deactivate(spa);
2892                         spa->spa_last_open_failed = error;
2893                         if (locked)
2894                                 mutex_exit(&spa_namespace_lock);
2895                         *spapp = NULL;
2896                         return (error);
```

```
2897                }
2898        }

2900        spa_open_ref(spa, tag);

2902        if (config != NULL)
2903                *config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);

2905        /*
2906         * If we've recovered the pool, pass back any information we
2907         * gathered while doing the load.
2908         */
2909        if (state == SPA_LOAD_RECOVER) {
2910                VERIFY(nvlist_add_nvlist(*config, ZPOOL_CONFIG_LOAD_INFO,
2911                    spa->spa_load_info) == 0);
2912        }

2914        if (locked) {
2915                spa->spa_last_open_failed = 0;
2916                spa->spa_last_ubsync_txg = 0;
2917                spa->spa_load_txg = 0;
2918                mutex_exit(&spa_namespace_lock);
2919        }

2921        *spapp = spa;

2923        return (0);
2924 }

2926 int
2927 spa_open_rewind(const char *name, spa_t **spapp, void *tag, nvlist_t *policy,
2928     nvlist_t **config)
2929 {
2930        return (spa_open_common(name, spapp, tag, policy, config));
2931 }

2933 int
2934 spa_open(const char *name, spa_t **spapp, void *tag)
2935 {
2936        return (spa_open_common(name, spapp, tag, NULL, NULL));
2937 }

2939 /*
2940  * Lookup the given spa_t, incrementing the inject count in the process,
2941  * preventing it from being exported or destroyed.
2942  */
2943 spa_t *
2944 spa_inject_addref(char *name)
2945 {
2946        spa_t *spa;

2948        mutex_enter(&spa_namespace_lock);
2949        if ((spa = spa_lookup(name)) == NULL) {
2950                mutex_exit(&spa_namespace_lock);
2951                return (NULL);
2952        }
2953        spa->spa_inject_ref++;
2954        mutex_exit(&spa_namespace_lock);

2956        return (spa);
2957 }

2959 void
2960 spa_inject_delref(spa_t *spa)
2961 {
2962        mutex_enter(&spa_namespace_lock);
```



```
2963        spa->spa_inject_ref--;
2964        mutex_exit(&spa_namespace_lock);
2965 }

2967 /*
2968  * Add spares device information to the nvlist.
2969  */
2970 static void
2971 spa_add_spares(spa_t *spa, nvlist_t *config)
2972 {
2973        nvlist_t **spares;
2974        uint_t i, nspares;
2975        nvlist_t *nvroot;
2976        uint64_t guid;
2977        vdev_stat_t *vs;
2978        uint_t vsc;
2979        uint64_t pool;

2981        ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));

2983        if (spa->spa_spares.sav_count == 0)
2984                return;

2986        VERIFY(nvlist_lookup_nvlist(config,
2987            ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
2988        VERIFY(nvlist_lookup_nvlist_array(spa->spa_spares.sav_config,
2989            ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);
2990        if (nspares != 0) {
2991                VERIFY(nvlist_add_nvlist_array(nvroot,
2992                    ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
2993                VERIFY(nvlist_lookup_nvlist_array(nvroot,
2994                    ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);

2996                /*
2997                 * Go through and find any spares which have since been
2998                 * repurposed as an active spare.  If this is the case, update
2999                 * their status appropriately.
3000                 */
3001                for (i = 0; i < nspares; i++) {
3002                        VERIFY(nvlist_lookup_uint64(spares[i],
3003                            ZPOOL_CONFIG_GUID, &guid) == 0);
3004                        if (spa_spare_exists(guid, &pool, NULL) &&
3005                            pool != 0ULL) {
3006                                VERIFY(nvlist_lookup_uint64_array(
3007                                    spares[i], ZPOOL_CONFIG_VDEV_STATS,
3008                                    (uint64_t **)&vs, &vsc) == 0);
3009                                vs->vs_state = VDEV_STATE_CANT_OPEN;
3010                                vs->vs_aux = VDEV_AUX_SPARED;
3011                        }
3012                }
3013        }
3014 }

3016 /*
3017  * Add l2cache device information to the nvlist, including vdev stats.
3018  */
3019 static void
3020 spa_add_l2cache(spa_t *spa, nvlist_t *config)
3021 {
3022        nvlist_t **l2cache;
3023        uint_t i, j, nl2cache;
3024        nvlist_t *nvroot;
3025        uint64_t guid;
3026        vdev_t *vd;
3027        vdev_stat_t *vs;
3028        uint_t vsc;
```

```
3030         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));

3032         if (spa->spa_l2cache.sav_count == 0)
3033                 return;

3035         VERIFY(nvlist_lookup_nvlist(config,
3036             ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
3037         VERIFY(nvlist_lookup_nvlist_array(spa->spa_l2cache.sav_config,
3038             ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
3039         if (nl2cache != 0) {
3040                 VERIFY(nvlist_add_nvlist_array(nvroot,
3041                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3042                 VERIFY(nvlist_lookup_nvlist_array(nvroot,
3043                     ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);

3045                 /*
3046                  * Update level 2 cache device stats.
3047                  */

3049                 for (i = 0; i < nl2cache; i++) {
3050                         VERIFY(nvlist_lookup_uint64(l2cache[i],
3051                             ZPOOL_CONFIG_GUID, &guid) == 0);

3053                         vd = NULL;
3054                         for (j = 0; j < spa->spa_l2cache.sav_count; j++) {
3055                                 if (guid ==
3056                                     spa->spa_l2cache.sav_vdevs[j]->vdev_guid) {
3057                                         vd = spa->spa_l2cache.sav_vdevs[j];
3058                                         break;
3059                                 }
3060                         }
3061                         ASSERT(vd != NULL);

3063                         VERIFY(nvlist_lookup_uint64_array(l2cache[i],
3064                             ZPOOL_CONFIG_VDEV_STATS, (uint64_t **)&vs, &vsc)
3065                             == 0);
3066                         vdev_get_stats(vd, vs);
3067                 }
3068         }
3069 }

3071 static void
3072 spa_add_feature_stats(spa_t *spa, nvlist_t *config)
3073 {
3074         nvlist_t *features;
3075         zap_cursor_t zc;
3076         zap_attribute_t za;

3078         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));
3079         VERIFY(nvlist_alloc(&features, NV_UNIQUE_NAME, KM_SLEEP) == 0);

3081         if (spa->spa_feat_for_read_obj != 0) {
3082                 for (zap_cursor_init(&zc, spa->spa_meta_objset,
3083                     spa->spa_feat_for_read_obj);
3084                     zap_cursor_retrieve(&zc, &za) == 0;
3085                     zap_cursor_advance(&zc)) {
3086                         ASSERT(za.za_integer_length == sizeof (uint64_t) &&
3087                             za.za_num_integers == 1);
3088                         VERIFY3U(0, ==, nvlist_add_uint64(features, za.za_name,
3089                             za.za_first_integer));
3090                 }
3091                 zap_cursor_fini(&zc);
3092         }

3094         if (spa->spa_feat_for_write_obj != 0) {
```



```
3095                 for (zap_cursor_init(&zc, spa->spa_meta_objset,
3096                     spa->spa_feat_for_write_obj);
3097                     zap_cursor_retrieve(&zc, &za) == 0;
3098                     zap_cursor_advance(&zc)) {
3099                         ASSERT(za.za_integer_length == sizeof (uint64_t) &&
3100                             za.za_num_integers == 1);
3101                         VERIFY3U(0, ==, nvlist_add_uint64(features, za.za_name,
3102                             za.za_first_integer));
3103                 }
3104                 zap_cursor_fini(&zc);
3105         }

3107         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_FEATURE_STATS,
3108             features) == 0);
3109         nvlist_free(features);
3110 }

3112 int
3113 spa_get_stats(const char *name, nvlist_t **config,
3114     char *altroot, size_t buflen)
3115 {
3116         int error;
3117         spa_t *spa;

3119         *config = NULL;
3120         error = spa_open_common(name, &spa, FTAG, NULL, config);

3122         if (spa != NULL) {
3123                 /*
3124                  * This still leaves a window of inconsistency where the spares
3125                  * or l2cache devices could change and the config would be
3126                  * self-inconsistent.
3127                  */
3128                 spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);

3130                 if (*config != NULL) {
3131                         uint64_t loadtimes[2];

3133                         loadtimes[0] = spa->spa_loaded_ts.tv_sec;
3134                         loadtimes[1] = spa->spa_loaded_ts.tv_nsec;
3135                         VERIFY(nvlist_add_uint64_array(*config,
3136                             ZPOOL_CONFIG_LOADED_TIME, loadtimes, 2) == 0);

3138                         VERIFY(nvlist_add_uint64(*config,
3139                             ZPOOL_CONFIG_ERRCOUNT,
3140                             spa_get_errlog_size(spa)) == 0);

3142                         if (spa_suspended(spa))
3143                                 VERIFY(nvlist_add_uint64(*config,
3144                                     ZPOOL_CONFIG_SUSPENDED,
3145                                     spa->spa_failmode) == 0);

3147                         spa_add_spares(spa, *config);
3148                         spa_add_l2cache(spa, *config);
3149                         spa_add_feature_stats(spa, *config);
3150                 }
3151         }

3153         /*
3154          * We want to get the alternate root even for faulted pools, so we cheat
3155          * and call spa_lookup() directly.
3156          */
3157         if (altroot) {
3158                 if (spa == NULL) {
3159                         mutex_enter(&spa_namespace_lock);
3160                         spa = spa_lookup(name);
```

```
3161                        if (spa)
3162                                spa_altroot(spa, altroot, buflen);
3163                        else
3164                                altroot[0] = '\0';
3165                        spa = NULL;
3166                        mutex_exit(&spa_namespace_lock);
3167                } else {
3168                        spa_altroot(spa, altroot, buflen);
3169                }
3170        }

3172        if (spa != NULL) {
3173                spa_config_exit(spa, SCL_CONFIG, FTAG);
3174                spa_close(spa, FTAG);
3175        }

3177        return (error);
3178 }

3180 /*
3181  * Validate that the auxiliary device array is well formed.  We must have an
3182  * array of nvlists, each which describes a valid leaf vdev.  If this is an
3183  * import (mode is VDEV_ALLOC_SPARE), then we allow corrupted spares to be
3184  * specified, as long as they are well-formed.
3185  */
3186 static int
3187 spa_validate_aux_devs(spa_t *spa, nvlist_t *nvroot, uint64_t crtxg, int mode,
3188     spa_aux_vdev_t *sav, const char *config, uint64_t version,
3189     vdev_labeltype_t label)
3190 {
3191        nvlist_t **dev;
3192        uint_t i, ndev;
3193        vdev_t *vd;
3194        int error;

3196        ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

3198        /*
3199         * It's acceptable to have no devs specified.
3200         */
3201        if (nvlist_lookup_nvlist_array(nvroot, config, &dev, &ndev) != 0)
3202                return (0);

3204        if (ndev == 0)
3205                return (SET_ERROR(EINVAL));

3207        /*
3208         * Make sure the pool is formatted with a version that supports this
3209         * device type.
3210         */
3211        if (spa_version(spa) < version)
3212                return (SET_ERROR(ENOTSUP));

3214        /*
3215         * Set the pending device list so we correctly handle device in-use
3216         * checking.
3217         */
3218        sav->sav_pending = dev;
3219        sav->sav_npending = ndev;

3221        for (i = 0; i < ndev; i++) {
3222                if ((error = spa_config_parse(spa, &vd, dev[i], NULL, 0,
3223                    mode)) != 0)
3224                        goto out;

3226                if (!vd->vdev_ops->vdev_op_leaf) {
```



```
3227                        vdev_free(vd);
3228                        error = SET_ERROR(EINVAL);
3229                        goto out;
3230                }

3232                /*
3233                 * The L2ARC currently only supports disk devices in
3234                 * kernel context.  For user-level testing, we allow it.
3235                 */
3236 #ifdef _KERNEL
3237                if ((strcmp(config, ZPOOL_CONFIG_L2CACHE) == 0) &&
3238                    strcmp(vd->vdev_ops->vdev_op_type, VDEV_TYPE_DISK) != 0) {
3239                        error = SET_ERROR(ENOTBLK);
3240                        vdev_free(vd);
3241                        goto out;
3242                }
3243 #endif
3244                vd->vdev_top = vd;

3246                if ((error = vdev_open(vd)) == 0 &&
3247                    (error = vdev_label_init(vd, crtxg, label)) == 0) {
3248                        VERIFY(nvlist_add_uint64(dev[i], ZPOOL_CONFIG_GUID,
3249                            vd->vdev_guid) == 0);
3250                }

3252                vdev_free(vd);

3254                if (error &&
3255                    (mode != VDEV_ALLOC_SPARE && mode != VDEV_ALLOC_L2CACHE))
3256                        goto out;
3257                else
3258                        error = 0;
3259        }

3261 out:
3262        sav->sav_pending = NULL;
3263        sav->sav_npending = 0;
3264        return (error);
3265 }

3267 static int
3268 spa_validate_aux(spa_t *spa, nvlist_t *nvroot, uint64_t crtxg, int mode)
3269 {
3270        int error;

3272        ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

3274        if ((error = spa_validate_aux_devs(spa, nvroot, crtxg, mode,
3275            &spa->spa_spares, ZPOOL_CONFIG_SPARES, SPA_VERSION_SPARES,
3276            VDEV_LABEL_SPARE)) != 0) {
3277                return (error);
3278        }

3280        return (spa_validate_aux_devs(spa, nvroot, crtxg, mode,
3281            &spa->spa_l2cache, ZPOOL_CONFIG_L2CACHE, SPA_VERSION_L2CACHE,
3282            VDEV_LABEL_L2CACHE));
3283 }

3285 static void
3286 spa_set_aux_vdevs(spa_aux_vdev_t *sav, nvlist_t **devs, int ndevs,
3287     const char *config)
3288 {
3289        int i;

3291        if (sav->sav_config != NULL) {
3292                nvlist_t **olddevs;
```

```
3293                 uint_t oldndevs;
3294                 nvlist_t **newdevs;
3296                 /*
3297                  * Generate new dev list by concatentating with the
3298                  * current dev list.
3299                  */
3300                 VERIFY(nvlist_lookup_nvlist_array(sav->sav_config, config,
3301                     &olddevs, &oldndevs) == 0);
3303                 newdevs = kmem_alloc(sizeof (void *) *
3304                     (ndevs + oldndevs), KM_SLEEP);
3305                 for (i = 0; i < oldndevs; i++)
3306                         VERIFY(nvlist_dup(olddevs[i], &newdevs[i],
3307                             KM_SLEEP) == 0);
3308                 for (i = 0; i < ndevs; i++)
3309                         VERIFY(nvlist_dup(devs[i], &newdevs[i + oldndevs],
3310                             KM_SLEEP) == 0);
3312                 VERIFY(nvlist_remove(sav->sav_config, config,
3313                     DATA_TYPE_NVLIST_ARRAY) == 0);
3315                 VERIFY(nvlist_add_nvlist_array(sav->sav_config,
3316                     config, newdevs, ndevs + oldndevs) == 0);
3317                 for (i = 0; i < oldndevs + ndevs; i++)
3318                         nvlist_free(newdevs[i]);
3319                 kmem_free(newdevs, (oldndevs + ndevs) * sizeof (void *));
3320         } else {
3321                 /*
3322                  * Generate a new dev list.
3323                  */
3324                 VERIFY(nvlist_alloc(&sav->sav_config, NV_UNIQUE_NAME,
3325                     KM_SLEEP) == 0);
3326                 VERIFY(nvlist_add_nvlist_array(sav->sav_config, config,
3327                     devs, ndevs) == 0);
3328         }
3329 }
3331 /*
3332  * Stop and drop level 2 ARC devices
3333  */
3334 void
3335 spa_l2cache_drop(spa_t *spa)
3336 {
3337         vdev_t *vd;
3338         int i;
3339         spa_aux_vdev_t *sav = &spa->spa_l2cache;
3341         for (i = 0; i < sav->sav_count; i++) {
3342                 uint64_t pool;
3344                 vd = sav->sav_vdevs[i];
3345                 ASSERT(vd != NULL);
3347                 if (spa_l2cache_exists(vd->vdev_guid, &pool) &&
3348                     pool != 0ULL && l2arc_vdev_present(vd))
3349                         l2arc_remove_vdev(vd);
3350         }
3351 }
3353 /*
3354  * Pool Creation
3355  */
3356 int
3357 spa_create(const char *pool, nvlist_t *nvroot, nvlist_t *props,
3358     nvlist_t *zplprops)
```



```
3359 {
3360         spa_t *spa;
3361         char *altroot = NULL;
3362         vdev_t *rvd;
3363         dsl_pool_t *dp;
3364         dmu_tx_t *tx;
3365         int error = 0;
3366         uint64_t txg = TXG_INITIAL;
3367         nvlist_t **spares, **l2cache;
3368         uint_t nspares, nl2cache;
3369         uint64_t version, obj;
3370         boolean_t has_features;
3372         /*
3373          * If this pool already exists, return failure.
3374          */
3375         mutex_enter(&spa_namespace_lock);
3376         if (spa_lookup(pool) != NULL) {
3377                 mutex_exit(&spa_namespace_lock);
3378                 return (SET_ERROR(EEXIST));
3379         }
3381         /*
3382          * Allocate a new spa_t structure.
3383          */
3384         (void) nvlist_lookup_string(props,
3385             zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3386         spa = spa_add(pool, NULL, altroot);
3387         spa_activate(spa, spa_mode_global);
3389         if (props && (error = spa_prop_validate(spa, props))) {
3390                 spa_deactivate(spa);
3391                 spa_remove(spa);
3392                 mutex_exit(&spa_namespace_lock);
3393                 return (error);
3394         }
3396         has_features = B_FALSE;
3397         for (nvpair_t *elem = nvlist_next_nvpair(props, NULL);
3398             elem != NULL; elem = nvlist_next_nvpair(props, elem)) {
3399                 if (zpool_prop_feature(nvpair_name(elem)))
3400                         has_features = B_TRUE;
3401         }
3403         if (has_features || nvlist_lookup_uint64(props,
3404             zpool_prop_to_name(ZPOOL_PROP_VERSION), &version) != 0) {
3405                 version = SPA_VERSION;
3406         }
3407         ASSERT(SPA_VERSION_IS_SUPPORTED(version));
3409         spa->spa_first_txg = txg;
3410         spa->spa_uberblock.ub_txg = txg - 1;
3411         spa->spa_uberblock.ub_version = version;
3412         spa->spa_ubsync = spa->spa_uberblock;
3414         /*
3415          * Create "The Godfather" zio to hold all async IOs
3416          */
3417         spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
3418             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);
3420         /*
3421          * Create the root vdev.
3422          */
3423         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
```

```
3425         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, VDEV_ALLOC_ADD);

3427         ASSERT(error != 0 || rvd != NULL);
3428         ASSERT(error != 0 || spa->spa_root_vdev == rvd);

3430         if (error == 0 && !zfs_allocatable_devs(nvroot))
3431                 error = SET_ERROR(EINVAL);

3433         if (error == 0 &&
3434             (error = vdev_create(rvd, txg, B_FALSE)) == 0 &&
3435             (error = spa_validate_aux(spa, nvroot, txg,
3436             VDEV_ALLOC_ADD)) == 0) {
3437                 for (int c = 0; c < rvd->vdev_children; c++) {
3438                         vdev_metaslab_set_size(rvd->vdev_child[c]);
3439                         vdev_expand(rvd->vdev_child[c], txg);
3440                 }
3441         }

3443         spa_config_exit(spa, SCL_ALL, FTAG);

3445         if (error != 0) {
3446                 spa_unload(spa);
3447                 spa_deactivate(spa);
3448                 spa_remove(spa);
3449                 mutex_exit(&spa_namespace_lock);
3450                 return (error);
3451         }

3453         /*
3454          * Get the list of spares, if specified.
3455          */
3456         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3457             &spares, &nspares) == 0) {
3458                 VERIFY(nvlist_alloc(&spa->spa_spares.sav_config, NV_UNIQUE_NAME,
3459                     KM_SLEEP) == 0);
3460                 VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
3461                     ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3462                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3463                 spa_load_spares(spa);
3464                 spa_config_exit(spa, SCL_ALL, FTAG);
3465                 spa->spa_spares.sav_sync = B_TRUE;
3466         }

3468         /*
3469          * Get the list of level 2 cache devices, if specified.
3470          */
3471         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3472             &l2cache, &nl2cache) == 0) {
3473                 VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3474                     NV_UNIQUE_NAME, KM_SLEEP) == 0);
3475                 VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3476                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3477                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3478                 spa_load_l2cache(spa);
3479                 spa_config_exit(spa, SCL_ALL, FTAG);
3480                 spa->spa_l2cache.sav_sync = B_TRUE;
3481         }

3483         spa->spa_is_initializing = B_TRUE;
3484         spa->spa_dsl_pool = dp = dsl_pool_create(spa, zplprops, txg);
3485         spa->spa_meta_objset = dp->dp_meta_objset;
3486         spa->spa_is_initializing = B_FALSE;

3488         /*
3489          * Create DDTs (dedup tables).
3490          */
```



```
3491         ddt_create(spa);

3493         spa_update_dspace(spa);

3495         tx = dmu_tx_create_assigned(dp, txg);

3497         /*
3498          * Create the pool config object.
3499          */
3500         spa->spa_config_object = dmu_object_alloc(spa->spa_meta_objset,
3501             DMU_OT_PACKED_NVLIST, SPA_CONFIG_BLOCKSIZE,
3502             DMU_OT_PACKED_NVLIST_SIZE, sizeof (uint64_t), tx);

3504         if (zap_add(spa->spa_meta_objset,
3505             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CONFIG,
3506             sizeof (uint64_t), 1, &spa->spa_config_object, tx) != 0) {
3507                 cmn_err(CE_PANIC, "failed to add pool config");
3508         }

3510         if (spa_version(spa) >= SPA_VERSION_FEATURES)
3511                 spa_feature_create_zap_objects(spa, tx);

3513         if (zap_add(spa->spa_meta_objset,
3514             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CREATION_VERSION,
3515             sizeof (uint64_t), 1, &version, tx) != 0) {
3516                 cmn_err(CE_PANIC, "failed to add pool version");
3517         }

3519         /* Newly created pools with the right version are always deflated. */
3520         if (version >= SPA_VERSION_RAIDZ_DEFLATE) {
3521                 spa->spa_deflate = TRUE;
3522                 if (zap_add(spa->spa_meta_objset,
3523                     DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_DEFLATE,
3524                     sizeof (uint64_t), 1, &spa->spa_deflate, tx) != 0) {
3525                         cmn_err(CE_PANIC, "failed to add deflate");
3526                 }
3527         }

3529         /*
3530          * Create the deferred-free bpobj.  Turn off compression
3531          * because sync-to-convergence takes longer if the blocksize
3532          * keeps changing.
3533          */
3534         obj = bpobj_alloc(spa->spa_meta_objset, 1 << 14, tx);
3535         dmu_object_set_compress(spa->spa_meta_objset, obj,
3536             ZIO_COMPRESS_OFF, tx);
3537         if (zap_add(spa->spa_meta_objset,
3538             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_SYNC_BPOBJ,
3539             sizeof (uint64_t), 1, &obj, tx) != 0) {
3540                 cmn_err(CE_PANIC, "failed to add bpobj");
3541         }
3542         VERIFY3U(0, ==, bpobj_open(&spa->spa_deferred_bpobj,
3543             spa->spa_meta_objset, obj));

3545         /*
3546          * Create the pool's history object.
3547          */
3548         if (version >= SPA_VERSION_ZPOOL_HISTORY)
3549                 spa_history_create_obj(spa, tx);

3551         /*
3552          * Set pool properties.
3553          */
3554         spa->spa_bootfs = zpool_prop_default_numeric(ZPOOL_PROP_BOOTFS);
3555         spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
3556         spa->spa_failmode = zpool_prop_default_numeric(ZPOOL_PROP_FAILUREMODE);
```

```
3557         spa->spa_autoexpand = zpool_prop_default_numeric(ZPOOL_PROP_AUTOEXPAND);

3559         if (props != NULL) {
3560                 spa_configfile_set(spa, props, B_FALSE);
3561                 spa_sync_props(props, tx);
3562         }

3564         dmu_tx_commit(tx);

3566         spa->spa_sync_on = B_TRUE;
3567         txg_sync_start(spa->spa_dsl_pool);

3569         /*
3570          * We explicitly wait for the first transaction to complete so that our
3571          * bean counters are appropriately updated.
3572          */
3573         txg_wait_synced(spa->spa_dsl_pool, txg);

3575         spa_config_sync(spa, B_FALSE, B_TRUE);

3577         spa_history_log_version(spa, "create");

3579         spa->spa_minref = refcount_count(&spa->spa_refcount);

3581         mutex_exit(&spa_namespace_lock);

3583         return (0);
3584 }

3586 #ifdef _KERNEL
3587 /*
3588  * Get the root pool information from the root disk, then import the root pool
3589  * during the system boot up time.
3590  */
3591 extern int vdev_disk_read_rootlabel(char *, char *, nvlist_t **);

3593 static nvlist_t *
3594 spa_generate_rootconf(char *devpath, char *devid, uint64_t *guid)
3595 {
3596         nvlist_t *config;
3597         nvlist_t *nvtop, *nvroot;
3598         uint64_t pgid;

3600         if (vdev_disk_read_rootlabel(devpath, devid, &config) != 0)
3601                 return (NULL);

3603         /*
3604          * Add this top-level vdev to the child array.
3605          */
3606         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3607             &nvtop) == 0);
3608         VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID,
3609             &pgid) == 0);
3610         VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID, guid) == 0);

3612         /*
3613          * Put this pool's top-level vdevs into a root vdev.
3614          */
3615         VERIFY(nvlist_alloc(&nvroot, NV_UNIQUE_NAME, KM_SLEEP) == 0);
3616         VERIFY(nvlist_add_string(nvroot, ZPOOL_CONFIG_TYPE,
3617             VDEV_TYPE_ROOT) == 0);
3618         VERIFY(nvlist_add_uint64(nvroot, ZPOOL_CONFIG_ID, 0ULL) == 0);
3619         VERIFY(nvlist_add_uint64(nvroot, ZPOOL_CONFIG_GUID, pgid) == 0);
3620         VERIFY(nvlist_add_nvlist_array(nvroot, ZPOOL_CONFIG_CHILDREN,
3621             &nvtop, 1) == 0);
```



```
3623         /*
3624          * Replace the existing vdev_tree with the new root vdev in
3625          * this pool's configuration (remove the old, add the new).
3626          */
3627         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, nvroot) == 0);
3628         nvlist_free(nvroot);
3629         return (config);
3630 }

3632 /*
3633  * Walk the vdev tree and see if we can find a device with "better"
3634  * configuration. A configuration is "better" if the label on that
3635  * device has a more recent txg.
3636  */
3637 static void
3638 spa_alt_rootvdev(vdev_t *vd, vdev_t **avd, uint64_t *txg)
3639 {
3640         for (int c = 0; c < vd->vdev_children; c++)
3641                 spa_alt_rootvdev(vd->vdev_child[c], avd, txg);

3643         if (vd->vdev_ops->vdev_op_leaf) {
3644                 nvlist_t *label;
3645                 uint64_t label_txg;

3647                 if (vdev_disk_read_rootlabel(vd->vdev_physpath, vd->vdev_devid,
3648                     &label) != 0)
3649                         return;

3651                 VERIFY(nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_TXG,
3652                     &label_txg) == 0);

3654                 /*
3655                  * Do we have a better boot device?
3656                  */
3657                 if (label_txg > *txg) {
3658                         *txg = label_txg;
3659                         *avd = vd;
3660                 }
3661                 nvlist_free(label);
3662         }
3663 }

3665 /*
3666  * Import a root pool.
3667  *
3668  * For x86. devpath_list will consist of devid and/or physpath name of
3669  * the vdev (e.g. "id1,sd@SSEAGATE..." or "/pci@1f,0/ide@d/disk@0,0:a").
3670  * The GRUB "findroot" command will return the vdev we should boot.
3671  *
3672  * For Sparc, devpath_list consists the physpath name of the booting device
3673  * no matter the rootpool is a single device pool or a mirrored pool.
3674  * e.g.
3675  *      "/pci@1f,0/ide@d/disk@0,0:a"
3676  */
3677 int
3678 spa_import_rootpool(char *devpath, char *devid)
3679 {
3680         spa_t *spa;
3681         vdev_t *rvd, *bvd, *avd = NULL;
3682         nvlist_t *config, *nvtop;
3683         uint64_t guid, txg;
3684         char *pname;
3685         int error;

3687         /*
3688          * Read the label from the boot device and generate a configuration.
```

```
3689          */
3690         config = spa_generate_rootconf(devpath, devid, &guid);
3691 #if defined(_OBP) && defined(_KERNEL)
3692         if (config == NULL) {
3693                 if (strstr(devpath, "/iscsi/ssd") != NULL) {
3694                         /* iscsi boot */
3695                         get_iscsi_bootpath_phy(devpath);
3696                         config = spa_generate_rootconf(devpath, devid, &guid);
3697                 }
3698         }
3699 #endif
3700         if (config == NULL) {
3701                 cmn_err(CE_NOTE, "Cannot read the pool label from '%s'",
3702                     devpath);
3703                 return (SET_ERROR(EIO));
3704         }

3706         VERIFY(nvlist_lookup_string(config, ZPOOL_CONFIG_POOL_NAME,
3707             &pname) == 0);
3708         VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG, &txg) == 0);

3710         mutex_enter(&spa_namespace_lock);
3711         if ((spa = spa_lookup(pname)) != NULL) {
3712                 /*
3713                  * Remove the existing root pool from the namespace so that we
3714                  * can replace it with the correct config we just read in.
3715                  */
3716                 spa_remove(spa);
3717         }

3719         spa = spa_add(pname, config, NULL);
3720         spa->spa_is_root = B_TRUE;
3721         spa->spa_import_flags = ZFS_IMPORT_VERBATIM;

3723         /*
3724          * Build up a vdev tree based on the boot device's label config.
3725          */
3726         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3727             &nvtop) == 0);
3728         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3729         error = spa_config_parse(spa, &rvd, nvtop, NULL, 0,
3730             VDEV_ALLOC_ROOTPOOL);
3731         spa_config_exit(spa, SCL_ALL, FTAG);
3732         if (error) {
3733                 mutex_exit(&spa_namespace_lock);
3734                 nvlist_free(config);
3735                 cmn_err(CE_NOTE, "Can not parse the config for pool '%s'",
3736                     pname);
3737                 return (error);
3738         }

3740         /*
3741          * Get the boot vdev.
3742          */
3743         if ((bvd = vdev_lookup_by_guid(rvd, guid)) == NULL) {
3744                 cmn_err(CE_NOTE, "Can not find the boot vdev for guid %llu",
3745                     (u_longlong_t)guid);
3746                 error = SET_ERROR(ENOENT);
3747                 goto out;
3748         }

3750         /*
3751          * Determine if there is a better boot device.
3752          */
3753         avd = bvd;
3754         spa_alt_rootvdev(rvd, &avd, &txg);
```



```
3755         if (avd != bvd) {
3756                 cmn_err(CE_NOTE, "The boot device is 'degraded'. Please "
3757                     "try booting from '%s'", avd->vdev_path);
3758                 error = SET_ERROR(EINVAL);
3759                 goto out;
3760         }

3762         /*
3763          * If the boot device is part of a spare vdev then ensure that
3764          * we're booting off the active spare.
3765          */
3766         if (bvd->vdev_parent->vdev_ops == &vdev_spare_ops &&
3767             !bvd->vdev_isspare) {
3768                 cmn_err(CE_NOTE, "The boot device is currently spared. Please "
3769                     "try booting from '%s'",
3770                     bvd->vdev_parent->
3771                     vdev_child[bvd->vdev_parent->vdev_children - 1]->vdev_path);
3772                 error = SET_ERROR(EINVAL);
3773                 goto out;
3774         }

3776         error = 0;
3777 out:
3778         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3779         vdev_free(rvd);
3780         spa_config_exit(spa, SCL_ALL, FTAG);
3781         mutex_exit(&spa_namespace_lock);

3783         nvlist_free(config);
3784         return (error);
3785 }

3787 #endif

3789 /*
3790  * Import a non-root pool into the system.
3791  */
3792 int
3793 spa_import(const char *pool, nvlist_t *config, nvlist_t *props, uint64_t flags)
3794 {
3795         spa_t *spa;
3796         char *altroot = NULL;
3797         spa_load_state_t state = SPA_LOAD_IMPORT;
3798         zpool_rewind_policy_t policy;
3799         uint64_t mode = spa_mode_global;
3800         uint64_t readonly = B_FALSE;
3801         int error;
3802         nvlist_t *nvroot;
3803         nvlist_t **spares, **l2cache;
3804         uint_t nspares, nl2cache;

3806         /*
3807          * If a pool with this name exists, return failure.
3808          */
3809         mutex_enter(&spa_namespace_lock);
3810         if (spa_lookup(pool) != NULL) {
3811                 mutex_exit(&spa_namespace_lock);
3812                 return (SET_ERROR(EEXIST));
3813         }

3815         /*
3816          * Create and initialize the spa structure.
3817          */
3818         (void) nvlist_lookup_string(props,
3819             zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3820         (void) nvlist_lookup_uint64(props,
```

```
3821             zpool_prop_to_name(ZPOOL_PROP_READONLY), &readonly);
3822         if (readonly)
3823                 mode = FREAD;
3824         spa = spa_add(pool, config, altroot);
3825         spa->spa_import_flags = flags;

3827         /*
3828          * Verbatim import - Take a pool and insert it into the namespace
3829          * as if it had been loaded at boot.
3830          */
3831         if (spa->spa_import_flags & ZFS_IMPORT_VERBATIM) {
3832                 if (props != NULL)
3833                         spa_configfile_set(spa, props, B_FALSE);

3835                 spa_config_sync(spa, B_FALSE, B_TRUE);

3837                 mutex_exit(&spa_namespace_lock);
3838                 spa_history_log_version(spa, "import");

3840                 return (0);
3841         }

3843         spa_activate(spa, mode);

3845         /*
3846          * Don't start async tasks until we know everything is healthy.
3847          */
3848         spa_async_suspend(spa);

3850         zpool_get_rewind_policy(config, &policy);
3851         if (policy.zrp_request & ZPOOL_DO_REWIND)
3852                 state = SPA_LOAD_RECOVER;

3854         /*
3855          * Pass off the heavy lifting to spa_load().  Pass TRUE for mosconfig
3856          * because the user-supplied config is actually the one to trust when
3857          * doing an import.
3858          */
3859         if (state != SPA_LOAD_RECOVER)
3860                 spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;

3862         error = spa_load_best(spa, state, B_TRUE, policy.zrp_txg,
3863             policy.zrp_request);

3865         /*
3866          * Propagate anything learned while loading the pool and pass it
3867          * back to caller (i.e. rewind info, missing devices, etc).
3868          */
3869         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_LOAD_INFO,
3870             spa->spa_load_info) == 0);

3872         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3873         /*
3874          * Toss any existing sparelist, as it doesn't have any validity
3875          * anymore, and conflicts with spa_has_spare().
3876          */
3877         if (spa->spa_spares.sav_config) {
3878                 nvlist_free(spa->spa_spares.sav_config);
3879                 spa->spa_spares.sav_config = NULL;
3880                 spa_load_spares(spa);
3881         }
3882         if (spa->spa_l2cache.sav_config) {
3883                 nvlist_free(spa->spa_l2cache.sav_config);
3884                 spa->spa_l2cache.sav_config = NULL;
3885                 spa_load_l2cache(spa);
3886         }
```



```
3888         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3889             &nvroot) == 0);
3890         if (error == 0)
3891                 error = spa_validate_aux(spa, nvroot, -1ULL,
3892                     VDEV_ALLOC_SPARE);
3893         if (error == 0)
3894                 error = spa_validate_aux(spa, nvroot, -1ULL,
3895                     VDEV_ALLOC_L2CACHE);
3896         spa_config_exit(spa, SCL_ALL, FTAG);

3898         if (props != NULL)
3899                 spa_configfile_set(spa, props, B_FALSE);

3901         if (error != 0 || (props && spa_writeable(spa) &&
3902             (error = spa_prop_set(spa, props)))) {
3903                 spa_unload(spa);
3904                 spa_deactivate(spa);
3905                 spa_remove(spa);
3906                 mutex_exit(&spa_namespace_lock);
3907                 return (error);
3908         }

3910         spa_async_resume(spa);

3912         /*
3913          * Override any spares and level 2 cache devices as specified by
3914          * the user, as these may have correct device names/devids, etc.
3915          */
3916         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3917             &spares, &nspares) == 0) {
3918                 if (spa->spa_spares.sav_config)
3919                         VERIFY(nvlist_remove(spa->spa_spares.sav_config,
3920                             ZPOOL_CONFIG_SPARES, DATA_TYPE_NVLIST_ARRAY) == 0);
3921                 else
3922                         VERIFY(nvlist_alloc(&spa->spa_spares.sav_config,
3923                             NV_UNIQUE_NAME, KM_SLEEP) == 0);
3924                 VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
3925                     ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3926                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3927                 spa_load_spares(spa);
3928                 spa_config_exit(spa, SCL_ALL, FTAG);
3929                 spa->spa_spares.sav_sync = B_TRUE;
3930         }
3931         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3932             &l2cache, &nl2cache) == 0) {
3933                 if (spa->spa_l2cache.sav_config)
3934                         VERIFY(nvlist_remove(spa->spa_l2cache.sav_config,
3935                             ZPOOL_CONFIG_L2CACHE, DATA_TYPE_NVLIST_ARRAY) == 0);
3936                 else
3937                         VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3938                             NV_UNIQUE_NAME, KM_SLEEP) == 0);
3939                 VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3940                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3941                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3942                 spa_load_l2cache(spa);
3943                 spa_config_exit(spa, SCL_ALL, FTAG);
3944                 spa->spa_l2cache.sav_sync = B_TRUE;
3945         }

3947         /*
3948          * Check for any removed devices.
3949          */
3950         if (spa->spa_autoreplace) {
3951                 spa_aux_check_removed(&spa->spa_spares);
3952                 spa_aux_check_removed(&spa->spa_l2cache);
```

```
3953         }
3954
3955         if (spa_writeable(spa)) {
3956                 /*
3957                  * Update the config cache to include the newly-imported pool.
3958                  */
3959                 spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
3960         }
3961
3962         /*
3963          * It's possible that the pool was expanded while it was exported.
3964          * We kick off an async task to handle this for us.
3965          */
3966         spa_async_request(spa, SPA_ASYNC_AUTOEXPAND);
3967
3968         mutex_exit(&spa_namespace_lock);
3969         spa_history_log_version(spa, "import");
3970
3971         return (0);
3972 }
3973
3974 nvlist_t *
3975 spa_tryimport(nvlist_t *tryconfig)
3976 {
3977         nvlist_t *config = NULL;
3978         char *poolname;
3979         spa_t *spa;
3980         uint64_t state;
3981         int error;
3982
3983         if (nvlist_lookup_string(tryconfig, ZPOOL_CONFIG_POOL_NAME, &poolname))
3984                 return (NULL);
3985
3986         if (nvlist_lookup_uint64(tryconfig, ZPOOL_CONFIG_POOL_STATE, &state))
3987                 return (NULL);
3988
3989         /*
3990          * Create and initialize the spa structure.
3991          */
3992         mutex_enter(&spa_namespace_lock);
3993         spa = spa_add(TRYIMPORT_NAME, tryconfig, NULL);
3994         spa_activate(spa, FREAD);
3995
3996         /*
3997          * Pass off the heavy lifting to spa_load().
3998          * Pass TRUE for mosconfig because the user-supplied config
3999          * is actually the one to trust when doing an import.
4000          */
4001         error = spa_load(spa, SPA_LOAD_TRYIMPORT, SPA_IMPORT_EXISTING, B_TRUE);
4002
4003         /*
4004          * If 'tryconfig' was at least parsable, return the current config.
4005          */
4006         if (spa->spa_root_vdev != NULL) {
4007                 config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);
4008                 VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME,
4009                     poolname) == 0);
4010                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE,
4011                     state) == 0);
4012                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_TIMESTAMP,
4013                     spa->spa_uberblock.ub_timestamp) == 0);
4014                 VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_LOAD_INFO,
4015                     spa->spa_load_info) == 0);
4016
4017                 /*
4018                  * If the bootfs property exists on this pool then we
```



```
4019                  * copy it out so that external consumers can tell which
4020                  * pools are bootable.
4021                  */
4022                 if ((!error || error == EEXIST) && spa->spa_bootfs) {
4023                         char *tmpname = kmem_alloc(MAXPATHLEN, KM_SLEEP);
4024
4025                         /*
4026                          * We have to play games with the name since the
4027                          * pool was opened as TRYIMPORT_NAME.
4028                          */
4029                         if (dsl_dsobj_to_dsname(spa_name(spa),
4030                             spa->spa_bootfs, tmpname) == 0) {
4031                                 char *cp;
4032                                 char *dsname = kmem_alloc(MAXPATHLEN, KM_SLEEP);
4033
4034                                 cp = strchr(tmpname, '/');
4035                                 if (cp == NULL) {
4036                                         (void) strlcpy(dsname, tmpname,
4037                                             MAXPATHLEN);
4038                                 } else {
4039                                         (void) snprintf(dsname, MAXPATHLEN,
4040                                             "%s/%s", poolname, ++cp);
4041                                 }
4042                                 VERIFY(nvlist_add_string(config,
4043                                     ZPOOL_CONFIG_BOOTFS, dsname) == 0);
4044                                 kmem_free(dsname, MAXPATHLEN);
4045                         }
4046                         kmem_free(tmpname, MAXPATHLEN);
4047                 }
4048
4049                 /*
4050                  * Add the list of hot spares and level 2 cache devices.
4051                  */
4052                 spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
4053                 spa_add_spares(spa, config);
4054                 spa_add_l2cache(spa, config);
4055                 spa_config_exit(spa, SCL_CONFIG, FTAG);
4056         }
4057
4058         spa_unload(spa);
4059         spa_deactivate(spa);
4060         spa_remove(spa);
4061         mutex_exit(&spa_namespace_lock);
4062
4063         return (config);
4064 }
4065
4066 /*
4067  * Pool export/destroy
4068  *
4069  * The act of destroying or exporting a pool is very simple.  We make sure there
4070  * is no more pending I/O and any references to the pool are gone.  Then, we
4071  * update the pool state and sync all the labels to disk, removing the
4072  * configuration from the cache afterwards. If the 'hardforce' flag is set, then
4073  * we don't sync the labels or remove the configuration cache.
4074  */
4075 static int
4076 spa_export_common(char *pool, int new_state, nvlist_t **oldconfig,
4077     boolean_t force, boolean_t hardforce)
4078 {
4079         spa_t *spa;
4080
4081         if (oldconfig)
4082                 *oldconfig = NULL;
4083
4084         if (!(spa_mode_global & FWRITE))
```

```
4085                return (SET_ERROR(EROFS));
4087        mutex_enter(&spa_namespace_lock);
4088        if ((spa = spa_lookup(pool)) == NULL) {
4089                mutex_exit(&spa_namespace_lock);
4090                return (SET_ERROR(ENOENT));
4091        }
4093        /*
4094         * Put a hold on the pool, drop the namespace lock, stop async tasks,
4095         * reacquire the namespace lock, and see if we can export.
4096         */
4097        spa_open_ref(spa, FTAG);
4098        mutex_exit(&spa_namespace_lock);
4099        spa_async_suspend(spa);
4100        mutex_enter(&spa_namespace_lock);
4101        spa_close(spa, FTAG);
4103        /*
4104         * The pool will be in core if it's openable,
4105         * in which case we can modify its state.
4106         */
4107        if (spa->spa_state != POOL_STATE_UNINITIALIZED && spa->spa_sync_on) {
4108                /*
4109                 * Objsets may be open only because they're dirty, so we
4110                 * have to force it to sync before checking spa_refcnt.
4111                 */
4112                txg_wait_synced(spa->spa_dsl_pool, 0);
4114                /*
4115                 * A pool cannot be exported or destroyed if there are active
4116                 * references.  If we are resetting a pool, allow references by
4117                 * fault injection handlers.
4118                 */
4119                if (!spa_refcount_zero(spa) ||
4120                    (spa->spa_inject_ref != 0 &&
4121                    new_state != POOL_STATE_UNINITIALIZED)) {
4122                        spa_async_resume(spa);
4123                        mutex_exit(&spa_namespace_lock);
4124                        return (SET_ERROR(EBUSY));
4125                }
4127                /*
4128                 * A pool cannot be exported if it has an active shared spare.
4129                 * This is to prevent other pools stealing the active spare
4130                 * from an exported pool. At user's own will, such pool can
4131                 * be forcedly exported.
4132                 */
4133                if (!force && new_state == POOL_STATE_EXPORTED &&
4134                    spa_has_active_shared_spare(spa)) {
4135                        spa_async_resume(spa);
4136                        mutex_exit(&spa_namespace_lock);
4137                        return (SET_ERROR(EXDEV));
4138                }
4140                /*
4141                 * We want this to be reflected on every label,
4142                 * so mark them all dirty.  spa_unload() will do the
4143                 * final sync that pushes these changes out.
4144                 */
4145                if (new_state != POOL_STATE_UNINITIALIZED && !hardforce) {
4146                        spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
4147                        spa->spa_state = new_state;
4148                        spa->spa_final_txg = spa_last_synced_txg(spa) +
4149                            TXG_DEFER_SIZE + 1;
4150                        vdev_config_dirty(spa->spa_root_vdev);
```



```
4151                        spa_config_exit(spa, SCL_ALL, FTAG);
4152                }
4153        }
4155        spa_event_notify(spa, NULL, ESC_ZFS_POOL_DESTROY);
4157        if (spa->spa_state != POOL_STATE_UNINITIALIZED) {
4158                spa_unload(spa);
4159                spa_deactivate(spa);
4160        }
4162        if (oldconfig && spa->spa_config)
4163                VERIFY(nvlist_dup(spa->spa_config, oldconfig, 0) == 0);
4165        if (new_state != POOL_STATE_UNINITIALIZED) {
4166                if (!hardforce)
4167                        spa_config_sync(spa, B_TRUE, B_TRUE);
4168                spa_remove(spa);
4169        }
4170        mutex_exit(&spa_namespace_lock);
4172        return (0);
4173 }
4175 /*
4176  * Destroy a storage pool.
4177  */
4178 int
4179 spa_destroy(char *pool)
4180 {
4181        return (spa_export_common(pool, POOL_STATE_DESTROYED, NULL,
4182            B_FALSE, B_FALSE));
4183 }
4185 /*
4186  * Export a storage pool.
4187  */
4188 int
4189 spa_export(char *pool, nvlist_t **oldconfig, boolean_t force,
4190     boolean_t hardforce)
4191 {
4192        return (spa_export_common(pool, POOL_STATE_EXPORTED, oldconfig,
4193            force, hardforce));
4194 }
4196 /*
4197  * Similar to spa_export(), this unloads the spa_t without actually removing it
4198  * from the namespace in any way.
4199  */
4200 int
4201 spa_reset(char *pool)
4202 {
4203        return (spa_export_common(pool, POOL_STATE_UNINITIALIZED, NULL,
4204            B_FALSE, B_FALSE));
4205 }
4207 /*
4208  * ========================================================================
4209  * Device manipulation
4210  * ========================================================================
4211  */
4213 /*
4214  * Add a device to a storage pool.
4215  */
4216 int
```

```
4217 spa_vdev_add(spa_t *spa, nvlist_t *nvroot)
4218 {
4219         uint64_t txg, id;
4220         int error;
4221         vdev_t *rvd = spa->spa_root_vdev;
4222         vdev_t *vd, *tvd;
4223         nvlist_t **spares, **l2cache;
4224         uint_t nspares, nl2cache;

4226         ASSERT(spa_writeable(spa));

4228         txg = spa_vdev_enter(spa);

4230         if ((error = spa_config_parse(spa, &vd, nvroot, NULL, 0,
4231             VDEV_ALLOC_ADD)) != 0)
4232                 return (spa_vdev_exit(spa, NULL, txg, error));

4234         spa->spa_pending_vdev = vd;     /* spa_vdev_exit() will clear this */

4236         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES, &spares,
4237             &nspares) != 0)
4238                 nspares = 0;

4240         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE, &l2cache,
4241             &nl2cache) != 0)
4242                 nl2cache = 0;

4244         if (vd->vdev_children == 0 && nspares == 0 && nl2cache == 0)
4245                 return (spa_vdev_exit(spa, vd, txg, EINVAL));

4247         if (vd->vdev_children != 0 &&
4248             (error = vdev_create(vd, txg, B_FALSE)) != 0)
4249                 return (spa_vdev_exit(spa, vd, txg, error));

4251         /*
4252          * We must validate the spares and l2cache devices after checking the
4253          * children.  Otherwise, vdev_inuse() will blindly overwrite the spare.
4254          */
4255         if ((error = spa_validate_aux(spa, nvroot, txg, VDEV_ALLOC_ADD)) != 0)
4256                 return (spa_vdev_exit(spa, vd, txg, error));

4258         /*
4259          * Transfer each new top-level vdev from vd to rvd.
4260          */
4261         for (int c = 0; c < vd->vdev_children; c++) {

4263                 /*
4264                  * Set the vdev id to the first hole, if one exists.
4265                  */
4266                 for (id = 0; id < rvd->vdev_children; id++) {
4267                         if (rvd->vdev_child[id]->vdev_ishole) {
4268                                 vdev_free(rvd->vdev_child[id]);
4269                                 break;
4270                         }
4271                 }
4272                 tvd = vd->vdev_child[c];
4273                 vdev_remove_child(vd, tvd);
4274                 tvd->vdev_id = id;
4275                 vdev_add_child(rvd, tvd);
4276                 vdev_config_dirty(tvd);
4277         }

4279         if (nspares != 0) {
4280                 spa_set_aux_vdevs(&spa->spa_spares, spares, nspares,
4281                     ZPOOL_CONFIG_SPARES);
4282                 spa_load_spares(spa);
```



```
4283                 spa->spa_spares.sav_sync = B_TRUE;
4284         }

4286         if (nl2cache != 0) {
4287                 spa_set_aux_vdevs(&spa->spa_l2cache, l2cache, nl2cache,
4288                     ZPOOL_CONFIG_L2CACHE);
4289                 spa_load_l2cache(spa);
4290                 spa->spa_l2cache.sav_sync = B_TRUE;
4291         }

4293         /*
4294          * We have to be careful when adding new vdevs to an existing pool.
4295          * If other threads start allocating from these vdevs before we
4296          * sync the config cache, and we lose power, then upon reboot we may
4297          * fail to open the pool because there are DVAs that the config cache
4298          * can't translate.  Therefore, we first add the vdevs without
4299          * initializing metaslabs; sync the config cache (via spa_vdev_exit());
4300          * and then let spa_config_update() initialize the new metaslabs.
4301          *
4302          * spa_load() checks for added-but-not-initialized vdevs, so that
4303          * if we lose power at any point in this sequence, the remaining
4304          * steps will be completed the next time we load the pool.
4305          */
4306         (void) spa_vdev_exit(spa, vd, txg, 0);

4308         mutex_enter(&spa_namespace_lock);
4309         spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
4310         mutex_exit(&spa_namespace_lock);

4312         return (0);
4313 }

4315 /*
4316  * Attach a device to a mirror.  The arguments are the path to any device
4317  * in the mirror, and the nvroot for the new device.  If the path specifies
4318  * a device that is not mirrored, we automatically insert the mirror vdev.
4319  *
4320  * If 'replacing' is specified, the new device is intended to replace the
4321  * existing device; in this case the two devices are made into their own
4322  * mirror using the 'replacing' vdev, which is functionally identical to
4323  * the mirror vdev (it actually reuses all the same ops) but has a few
4324  * extra rules: you can't attach to it after it's been created, and upon
4325  * completion of resilvering, the first disk (the one being replaced)
4326  * is automatically detached.
4327  */
4328 int
4329 spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot, int replacing)
4330 {
4331         uint64_t txg, dtl_max_txg;
4332         vdev_t *rvd = spa->spa_root_vdev;
4333         vdev_t *oldvd, *newvd, *newrootvd, *pvd, *tvd;
4334         vdev_ops_t *pvops;
4335         char *oldvdpath, *newvdpath;
4336         int newvd_isspare;
4337         int error;

4339         ASSERT(spa_writeable(spa));

4341         txg = spa_vdev_enter(spa);

4343         oldvd = spa_lookup_by_guid(spa, guid, B_FALSE);

4345         if (oldvd == NULL)
4346                 return (spa_vdev_exit(spa, NULL, txg, ENODEV));

4348         if (!oldvd->vdev_ops->vdev_op_leaf)
```

```
4349                return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4351        pvd = oldvd->vdev_parent;

4353        if ((error = spa_config_parse(spa, &newrootvd, nvroot, NULL, 0,
4354            VDEV_ALLOC_ATTACH)) != 0)
4355                return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4357        if (newrootvd->vdev_children != 1)
4358                return (spa_vdev_exit(spa, newrootvd, txg, EINVAL));

4360        newvd = newrootvd->vdev_child[0];

4362        if (!newvd->vdev_ops->vdev_op_leaf)
4363                return (spa_vdev_exit(spa, newrootvd, txg, EINVAL));

4365        if ((error = vdev_create(newrootvd, txg, replacing)) != 0)
4366                return (spa_vdev_exit(spa, newrootvd, txg, error));

4368        /*
4369         * Spares can't replace logs
4370         */
4371        if (oldvd->vdev_top->vdev_islog && newvd->vdev_isspare)
4372                return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));

4374        if (!replacing) {
4375                /*
4376                 * For attach, the only allowable parent is a mirror or the root
4377                 * vdev.
4378                 */
4379                if (pvd->vdev_ops != &vdev_mirror_ops &&
4380                    pvd->vdev_ops != &vdev_root_ops)
4381                        return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));

4383                pvops = &vdev_mirror_ops;
4384        } else {
4385                /*
4386                 * Active hot spares can only be replaced by inactive hot
4387                 * spares.
4388                 */
4389                if (pvd->vdev_ops == &vdev_spare_ops &&
4390                    oldvd->vdev_isspare &&
4391                    !spa_has_spare(spa, newvd->vdev_guid))
4392                        return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));

4394                /*
4395                 * If the source is a hot spare, and the parent isn't already a
4396                 * spare, then we want to create a new hot spare.  Otherwise, we
4397                 * want to create a replacing vdev.  The user is not allowed to
4398                 * attach to a spared vdev child unless the 'isspare' state is
4399                 * the same (spare replaces spare, non-spare replaces
4400                 * non-spare).
4401                 */
4402                if (pvd->vdev_ops == &vdev_replacing_ops &&
4403                    spa_version(spa) < SPA_VERSION_MULTI_REPLACE) {
4404                        return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4405                } else if (pvd->vdev_ops == &vdev_spare_ops &&
4406                    newvd->vdev_isspare != oldvd->vdev_isspare) {
4407                        return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4408                }

4410                if (newvd->vdev_isspare)
4411                        pvops = &vdev_spare_ops;
4412                else
4413                        pvops = &vdev_replacing_ops;
4414        }
```



```
4416        /*
4417         * Make sure the new device is big enough.
4418         */
4419        if (newvd->vdev_asize < vdev_get_min_asize(oldvd))
4420                return (spa_vdev_exit(spa, newrootvd, txg, EOVERFLOW));

4422        /*
4423         * The new device cannot have a higher alignment requirement
4424         * than the top-level vdev.
4425         */
4426        if (newvd->vdev_ashift > oldvd->vdev_top->vdev_ashift)
4427                return (spa_vdev_exit(spa, newrootvd, txg, EDOM));

4429        /*
4430         * If this is an in-place replacement, update oldvd's path and devid
4431         * to make it distinguishable from newvd, and unopenable from now on.
4432         */
4433        if (strcmp(oldvd->vdev_path, newvd->vdev_path) == 0) {
4434                spa_strfree(oldvd->vdev_path);
4435                oldvd->vdev_path = kmem_alloc(strlen(newvd->vdev_path) + 5,
4436                    KM_SLEEP);
4437                (void) sprintf(oldvd->vdev_path, "%s/%s",
4438                    newvd->vdev_path, "old");
4439                if (oldvd->vdev_devid != NULL) {
4440                        spa_strfree(oldvd->vdev_devid);
4441                        oldvd->vdev_devid = NULL;
4442                }
4443        }

4445        /* mark the device being resilvered */
4446        newvd->vdev_resilvering = B_TRUE;

4448        /*
4449         * If the parent is not a mirror, or if we're replacing, insert the new
4450         * mirror/replacing/spare vdev above oldvd.
4451         */
4452        if (pvd->vdev_ops != pvops)
4453                pvd = vdev_add_parent(oldvd, pvops);

4455        ASSERT(pvd->vdev_top->vdev_parent == rvd);
4456        ASSERT(pvd->vdev_ops == pvops);
4457        ASSERT(oldvd->vdev_parent == pvd);

4459        /*
4460         * Extract the new device from its root and add it to pvd.
4461         */
4462        vdev_remove_child(newrootvd, newvd);
4463        newvd->vdev_id = pvd->vdev_children;
4464        newvd->vdev_crtxg = oldvd->vdev_crtxg;
4465        vdev_add_child(pvd, newvd);

4467        tvd = newvd->vdev_top;
4468        ASSERT(pvd->vdev_top == tvd);
4469        ASSERT(tvd->vdev_parent == rvd);

4471        vdev_config_dirty(tvd);

4473        /*
4474         * Set newvd's DTL to [TXG_INITIAL, dtl_max_txg) so that we account
4475         * for any dmu_sync-ed blocks.  It will propagate upward when
4476         * spa_vdev_exit() calls vdev_dtl_reassess().
4477         */
4478        dtl_max_txg = txg + TXG_CONCURRENT_STATES;

4480        vdev_dtl_dirty(newvd, DTL_MISSING, TXG_INITIAL,
```

```
4481             dtl_max_txg - TXG_INITIAL);

4483         if (newvd->vdev_isspare) {
4484                 spa_spare_activate(newvd);
4485                 spa_event_notify(spa, newvd, ESC_ZFS_VDEV_SPARE);
4486         }

4488         oldvdpath = spa_strdup(oldvd->vdev_path);
4489         newvdpath = spa_strdup(newvd->vdev_path);
4490         newvd_isspare = newvd->vdev_isspare;

4492         /*
4493          * Mark newvd's DTL dirty in this txg.
4494          */
4495         vdev_dirty(tvd, VDD_DTL, newvd, txg);

4497         /*
4498          * Restart the resilver
4499          */
4500         dsl_resilver_restart(spa->spa_dsl_pool, dtl_max_txg);

4502         /*
4503          * Commit the config
4504          */
4505         (void) spa_vdev_exit(spa, newrootvd, dtl_max_txg, 0);

4507         spa_history_log_internal(spa, "vdev attach", NULL,
4508             "%s vdev=%s %s vdev=%s",
4509             replacing && newvd_isspare ? "spare in" :
4510             replacing ? "replace" : "attach", newvdpath,
4511             replacing ? "for" : "to", oldvdpath);

4513         spa_strfree(oldvdpath);
4514         spa_strfree(newvdpath);

4516         if (spa->spa_bootfs)
4517                 spa_event_notify(spa, newvd, ESC_ZFS_BOOTFS_VDEV_ATTACH);

4519         return (0);
4520 }

4522 /*
4523  * Detach a device from a mirror or replacing vdev.
4524  * If 'replace_done' is specified, only detach if the parent
4525  * is a replacing vdev.
4526  */
4527 int
4528 spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid, int replace_done)
4529 {
4530         uint64_t txg;
4531         int error;
4532         vdev_t *rvd = spa->spa_root_vdev;
4533         vdev_t *vd, *pvd, *cvd, *tvd;
4534         boolean_t unspare = B_FALSE;
4535         uint64_t unspare_guid = 0;
4536         char *vdpath;

4538         ASSERT(spa_writeable(spa));

4540         txg = spa_vdev_enter(spa);

4542         vd = spa_lookup_by_guid(spa, guid, B_FALSE);

4544         if (vd == NULL)
4545                 return (spa_vdev_exit(spa, NULL, txg, ENODEV));
```



```
4547         if (!vd->vdev_ops->vdev_op_leaf)
4548                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4550         pvd = vd->vdev_parent;

4552         /*
4553          * If the parent/child relationship is not as expected, don't do it.
4554          * Consider M(A,R(B,C)) -- that is, a mirror of A with a replacing
4555          * vdev that's replacing B with C.  The user's intent in replacing
4556          * is to go from M(A,B) to M(A,C).  If the user decides to cancel
4557          * the replace by detaching C, the expected behavior is to end up
4558          * M(A,B).  But suppose that right after deciding to detach C,
4559          * the replacement of B completes.  We would have M(A,C), and then
4560          * ask to detach C, which would leave us with just A -- not what
4561          * the user wanted.  To prevent this, we make sure that the
4562          * parent/child relationship hasn't changed -- in this example,
4563          * that C's parent is still the replacing vdev R.
4564          */
4565         if (pvd->vdev_guid != pguid && pguid != 0)
4566                 return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4568         /*
4569          * Only 'replacing' or 'spare' vdevs can be replaced.
4570          */
4571         if (replace_done && pvd->vdev_ops != &vdev_replacing_ops &&
4572             pvd->vdev_ops != &vdev_spare_ops)
4573                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4575         ASSERT(pvd->vdev_ops != &vdev_spare_ops ||
4576             spa_version(spa) >= SPA_VERSION_SPARES);

4578         /*
4579          * Only mirror, replacing, and spare vdevs support detach.
4580          */
4581         if (pvd->vdev_ops != &vdev_replacing_ops &&
4582             pvd->vdev_ops != &vdev_mirror_ops &&
4583             pvd->vdev_ops != &vdev_spare_ops)
4584                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4586         /*
4587          * If this device has the only valid copy of some data,
4588          * we cannot safely detach it.
4589          */
4590         if (vdev_dtl_required(vd))
4591                 return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4593         ASSERT(pvd->vdev_children >= 2);

4595         /*
4596          * If we are detaching the second disk from a replacing vdev, then
4597          * check to see if we changed the original vdev's path to have "/old"
4598          * at the end in spa_vdev_attach().  If so, undo that change now.
4599          */
4600         if (pvd->vdev_ops == &vdev_replacing_ops && vd->vdev_id > 0 &&
4601             vd->vdev_path != NULL) {
4602                 size_t len = strlen(vd->vdev_path);

4604                 for (int c = 0; c < pvd->vdev_children; c++) {
4605                         cvd = pvd->vdev_child[c];

4607                         if (cvd == vd || cvd->vdev_path == NULL)
4608                                 continue;

4610                         if (strncmp(cvd->vdev_path, vd->vdev_path, len) == 0 &&
4611                             strcmp(cvd->vdev_path + len, "/old") == 0) {
4612                                 spa_strfree(cvd->vdev_path);
```

```
4613                                cvd->vdev_path = spa_strdup(vd->vdev_path);
4614                                break;
4615                        }
4616                }
4617        }

4619        /*
4620         * If we are detaching the original disk from a spare, then it implies
4621         * that the spare should become a real disk, and be removed from the
4622         * active spare list for the pool.
4623         */
4624        if (pvd->vdev_ops == &vdev_spare_ops &&
4625            vd->vdev_id == 0 &&
4626            pvd->vdev_child[pvd->vdev_children - 1]->vdev_isspare)
4627                unspare = B_TRUE;

4629        /*
4630         * Erase the disk labels so the disk can be used for other things.
4631         * This must be done after all other error cases are handled,
4632         * but before we disembowel vd (so we can still do I/O to it).
4633         * But if we can't do it, don't treat the error as fatal --
4634         * it may be that the unwritability of the disk is the reason
4635         * it's being detached!
4636         */
4637        error = vdev_label_init(vd, 0, VDEV_LABEL_REMOVE);

4639        /*
4640         * Remove vd from its parent and compact the parent's children.
4641         */
4642        vdev_remove_child(pvd, vd);
4643        vdev_compact_children(pvd);

4645        /*
4646         * Remember one of the remaining children so we can get tvd below.
4647         */
4648        cvd = pvd->vdev_child[pvd->vdev_children - 1];

4650        /*
4651         * If we need to remove the remaining child from the list of hot spares,
4652         * do it now, marking the vdev as no longer a spare in the process.
4653         * We must do this before vdev_remove_parent(), because that can
4654         * change the GUID if it creates a new toplevel GUID.  For a similar
4655         * reason, we must remove the spare now, in the same txg as the detach;
4656         * otherwise someone could attach a new sibling, change the GUID, and
4657         * the subsequent attempt to spa_vdev_remove(unspare_guid) would fail.
4658         */
4659        if (unspare) {
4660                ASSERT(cvd->vdev_isspare);
4661                spa_spare_remove(cvd);
4662                unspare_guid = cvd->vdev_guid;
4663                (void) spa_vdev_remove(spa, unspare_guid, B_TRUE);
4664                cvd->vdev_unspare = B_TRUE;
4665        }

4667        /*
4668         * If the parent mirror/replacing vdev only has one child,
4669         * the parent is no longer needed.  Remove it from the tree.
4670         */
4671        if (pvd->vdev_children == 1) {
4672                if (pvd->vdev_ops == &vdev_spare_ops)
4673                        cvd->vdev_unspare = B_FALSE;
4674                vdev_remove_parent(cvd);
4675                cvd->vdev_resilvering = B_FALSE;
4676        }
```



```
4679        /*
4680         * We don't set tvd until now because the parent we just removed
4681         * may have been the previous top-level vdev.
4682         */
4683        tvd = cvd->vdev_top;
4684        ASSERT(tvd->vdev_parent == rvd);

4686        /*
4687         * Reevaluate the parent vdev state.
4688         */
4689        vdev_propagate_state(cvd);

4691        /*
4692         * If the 'autoexpand' property is set on the pool then automatically
4693         * try to expand the size of the pool. For example if the device we
4694         * just detached was smaller than the others, it may be possible to
4695         * add metaslabs (i.e. grow the pool). We need to reopen the vdev
4696         * first so that we can obtain the updated sizes of the leaf vdevs.
4697         */
4698        if (spa->spa_autoexpand) {
4699                vdev_reopen(tvd);
4700                vdev_expand(tvd, txg);
4701        }

4703        vdev_config_dirty(tvd);

4705        /*
4706         * Mark vd's DTL as dirty in this txg.  vdev_dtl_sync() will see that
4707         * vd->vdev_detached is set and free vd's DTL object in syncing context.
4708         * But first make sure we're not on any *other* txg's DTL list, to
4709         * prevent vd from being accessed after it's freed.
4710         */
4711        vdpath = spa_strdup(vd->vdev_path);
4712        for (int t = 0; t < TXG_SIZE; t++)
4713                (void) txg_list_remove_this(&tvd->vdev_dtl_list, vd, t);
4714        vd->vdev_detached = B_TRUE;
4715        vdev_dirty(tvd, VDD_DTL, vd, txg);

4717        spa_event_notify(spa, vd, ESC_ZFS_VDEV_REMOVE);

4719        /* hang on to the spa before we release the lock */
4720        spa_open_ref(spa, FTAG);

4722        error = spa_vdev_exit(spa, vd, txg, 0);

4724        spa_history_log_internal(spa, "detach", NULL,
4725            "vdev=%s", vdpath);
4726        spa_strfree(vdpath);

4728        /*
4729         * If this was the removal of the original device in a hot spare vdev,
4730         * then we want to go through and remove the device from the hot spare
4731         * list of every other pool.
4732         */
4733        if (unspare) {
4734                spa_t *altspa = NULL;

4736                mutex_enter(&spa_namespace_lock);
4737                while ((altspa = spa_next(altspa)) != NULL) {
4738                        if (altspa->spa_state != POOL_STATE_ACTIVE ||
4739                            altspa == spa)
4740                                continue;

4742                        spa_open_ref(altspa, FTAG);
4743                        mutex_exit(&spa_namespace_lock);
4744                        (void) spa_vdev_remove(altspa, unspare_guid, B_TRUE);
```

```
4745                        mutex_enter(&spa_namespace_lock);
4746                        spa_close(altspa, FTAG);
4747                }
4748                mutex_exit(&spa_namespace_lock);

4750                /* search the rest of the vdevs for spares to remove */
4751                spa_vdev_resilver_done(spa);
4752        }

4754        /* all done with the spa; OK to release */
4755        mutex_enter(&spa_namespace_lock);
4756        spa_close(spa, FTAG);
4757        mutex_exit(&spa_namespace_lock);

4759        return (error);
4760 }

4762 /*
4763  * Split a set of devices from their mirrors, and create a new pool from them.
4764  */
4765 int
4766 spa_vdev_split_mirror(spa_t *spa, char *newname, nvlist_t *config,
4767     nvlist_t *props, boolean_t exp)
4768 {
4769        int error = 0;
4770        uint64_t txg, *glist;
4771        spa_t *newspa;
4772        uint_t c, children, lastlog;
4773        nvlist_t **child, *nvl, *tmp;
4774        dmu_tx_t *tx;
4775        char *altroot = NULL;
4776        vdev_t *rvd, **vml = NULL;                      /* vdev modify list */
4777        boolean_t activate_slog;

4779        ASSERT(spa_writeable(spa));

4781        txg = spa_vdev_enter(spa);

4783        /* clear the log and flush everything up to now */
4784        activate_slog = spa_passivate_log(spa);
4785        (void) spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
4786        error = spa_offline_log(spa);
4787        txg = spa_vdev_config_enter(spa);

4789        if (activate_slog)
4790                spa_activate_log(spa);

4792        if (error != 0)
4793                return (spa_vdev_exit(spa, NULL, txg, error));

4795        /* check new spa name before going any further */
4796        if (spa_lookup(newname) != NULL)
4797                return (spa_vdev_exit(spa, NULL, txg, EEXIST));

4799        /*
4800         * scan through all the children to ensure they're all mirrors
4801         */
4802        if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvl) != 0 ||
4803            nvlist_lookup_nvlist_array(nvl, ZPOOL_CONFIG_CHILDREN, &child,
4804            &children) != 0)
4805                return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4807        /* first, check to ensure we've got the right child count */
4808        rvd = spa->spa_root_vdev;
4809        lastlog = 0;
4810        for (c = 0; c < rvd->vdev_children; c++) {
```



```
4811                vdev_t *vd = rvd->vdev_child[c];

4813                /* don't count the holes & logs as children */
4814                if (vd->vdev_islog || vd->vdev_ishole) {
4815                        if (lastlog == 0)
4816                                lastlog = c;
4817                        continue;
4818                }

4820                lastlog = 0;
4821        }
4822        if (children != (lastlog != 0 ? lastlog : rvd->vdev_children))
4823                return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4825        /* next, ensure no spare or cache devices are part of the split */
4826        if (nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_SPARES, &tmp) == 0 ||
4827            nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_L2CACHE, &tmp) == 0)
4828                return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4830        vml = kmem_zalloc(children * sizeof (vdev_t *), KM_SLEEP);
4831        glist = kmem_zalloc(children * sizeof (uint64_t), KM_SLEEP);

4833        /* then, loop over each vdev and validate it */
4834        for (c = 0; c < children; c++) {
4835                uint64_t is_hole = 0;

4837                (void) nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_IS_HOLE,
4838                    &is_hole);

4840                if (is_hole != 0) {
4841                        if (spa->spa_root_vdev->vdev_child[c]->vdev_ishole ||
4842                            spa->spa_root_vdev->vdev_child[c]->vdev_islog) {
4843                                continue;
4844                        } else {
4845                                error = SET_ERROR(EINVAL);
4846                                break;
4847                        }
4848                }

4850                /* which disk is going to be split? */
4851                if (nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_GUID,
4852                    &glist[c]) != 0) {
4853                        error = SET_ERROR(EINVAL);
4854                        break;
4855                }

4857                /* look it up in the spa */
4858                vml[c] = spa_lookup_by_guid(spa, glist[c], B_FALSE);
4859                if (vml[c] == NULL) {
4860                        error = SET_ERROR(ENODEV);
4861                        break;
4862                }

4864                /* make sure there's nothing stopping the split */
4865                if (vml[c]->vdev_parent->vdev_ops != &vdev_mirror_ops ||
4866                    vml[c]->vdev_islog ||
4867                    vml[c]->vdev_ishole ||
4868                    vml[c]->vdev_isspare ||
4869                    vml[c]->vdev_isl2cache ||
4870                    !vdev_writeable(vml[c]) ||
4871                    vml[c]->vdev_children != 0 ||
4872                    vml[c]->vdev_state != VDEV_STATE_HEALTHY ||
4873                    c != spa->spa_root_vdev->vdev_child[c]->vdev_id) {
4874                        error = SET_ERROR(EINVAL);
4875                        break;
4876                }
```

```
4878                if (vdev_dtl_required(vml[c])) {
4879                        error = SET_ERROR(EBUSY);
4880                        break;
4881                }

4883                /* we need certain info from the top level */
4884                VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_ARRAY,
4885                    vml[c]->vdev_top->vdev_ms_array) == 0);
4886                VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_SHIFT,
4887                    vml[c]->vdev_top->vdev_ms_shift) == 0);
4888                VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASIZE,
4889                    vml[c]->vdev_top->vdev_asize) == 0);
4890                VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASHIFT,
4891                    vml[c]->vdev_top->vdev_ashift) == 0);
4892        }

4894        if (error != 0) {
4895                kmem_free(vml, children * sizeof (vdev_t *));
4896                kmem_free(glist, children * sizeof (uint64_t));
4897                return (spa_vdev_exit(spa, NULL, txg, error));
4898        }

4900        /* stop writers from using the disks */
4901        for (c = 0; c < children; c++) {
4902                if (vml[c] != NULL)
4903                        vml[c]->vdev_offline = B_TRUE;
4904        }
4905        vdev_reopen(spa->spa_root_vdev);

4907        /*
4908         * Temporarily record the splitting vdevs in the spa config.  This
4909         * will disappear once the config is regenerated.
4910         */
4911        VERIFY(nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP) == 0);
4912        VERIFY(nvlist_add_uint64_array(nvl, ZPOOL_CONFIG_SPLIT_LIST,
4913            glist, children) == 0);
4914        kmem_free(glist, children * sizeof (uint64_t));

4916        mutex_enter(&spa->spa_props_lock);
4917        VERIFY(nvlist_add_nvlist(spa->spa_config, ZPOOL_CONFIG_SPLIT,
4918            nvl) == 0);
4919        mutex_exit(&spa->spa_props_lock);
4920        spa->spa_config_splitting = nvl;
4921        vdev_config_dirty(spa->spa_root_vdev);

4923        /* configure and create the new pool */
4924        VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME, newname) == 0);
4925        VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE,
4926            exp ? POOL_STATE_EXPORTED : POOL_STATE_ACTIVE) == 0);
4927        VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_VERSION,
4928            spa_version(spa)) == 0);
4929        VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_TXG,
4930            spa->spa_config_txg) == 0);
4931        VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_GUID,
4932            spa_generate_guid(NULL)) == 0);
4933        (void) nvlist_lookup_string(props,
4934            zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);

4936        /* add the new pool to the namespace */
4937        newspa = spa_add(newname, config, altroot);
4938        newspa->spa_config_txg = spa->spa_config_txg;
4939        spa_set_log_state(newspa, SPA_LOG_CLEAR);

4941        /* release the spa config lock, retaining the namespace lock */
4942        spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
```



```
4944        if (zio_injection_enabled)
4945                zio_handle_panic_injection(spa, FTAG, 1);

4947        spa_activate(newspa, spa_mode_global);
4948        spa_async_suspend(newspa);

4950        /* create the new pool from the disks of the original pool */
4951        error = spa_load(newspa, SPA_LOAD_IMPORT, SPA_IMPORT_ASSEMBLE, B_TRUE);
4952        if (error)
4953                goto out;

4955        /* if that worked, generate a real config for the new pool */
4956        if (newspa->spa_root_vdev != NULL) {
4957                VERIFY(nvlist_alloc(&newspa->spa_config_splitting,
4958                    NV_UNIQUE_NAME, KM_SLEEP) == 0);
4959                VERIFY(nvlist_add_uint64(newspa->spa_config_splitting,
4960                    ZPOOL_CONFIG_SPLIT_GUID, spa_guid(spa)) == 0);
4961                spa_config_set(newspa, spa_config_generate(newspa, NULL, -1ULL,
4962                    B_TRUE));
4963        }

4965        /* set the props */
4966        if (props != NULL) {
4967                spa_configfile_set(newspa, props, B_FALSE);
4968                error = spa_prop_set(newspa, props);
4969                if (error)
4970                        goto out;
4971        }

4973        /* flush everything */
4974        txg = spa_vdev_config_enter(newspa);
4975        vdev_config_dirty(newspa->spa_root_vdev);
4976        (void) spa_vdev_config_exit(newspa, NULL, txg, 0, FTAG);

4978        if (zio_injection_enabled)
4979                zio_handle_panic_injection(spa, FTAG, 2);

4981        spa_async_resume(newspa);

4983        /* finally, update the original pool's config */
4984        txg = spa_vdev_config_enter(spa);
4985        tx = dmu_tx_create_dd(spa_get_dsl(spa)->dp_mos_dir);
4986        error = dmu_tx_assign(tx, TXG_WAIT);
4987        if (error != 0)
4988                dmu_tx_abort(tx);
4989        for (c = 0; c < children; c++) {
4990                if (vml[c] != NULL) {
4991                        vdev_split(vml[c]);
4992                        if (error == 0)
4993                                spa_history_log_internal(spa, "detach", tx,
4994                                    "vdev=%s", vml[c]->vdev_path);
4995                        vdev_free(vml[c]);
4996                }
4997        }
4998        vdev_config_dirty(spa->spa_root_vdev);
4999        spa->spa_config_splitting = NULL;
5000        nvlist_free(nvl);
5001        if (error == 0)
5002                dmu_tx_commit(tx);
5003        (void) spa_vdev_exit(spa, NULL, txg, 0);

5005        if (zio_injection_enabled)
5006                zio_handle_panic_injection(spa, FTAG, 3);

5008        /* split is complete; log a history record */
```

```
5009         spa_history_log_internal(newspa, "split", NULL,
5010             "from pool %s", spa_name(spa));

5012         kmem_free(vml, children * sizeof (vdev_t *));

5014         /* if we're not going to mount the filesystems in userland, export */
5015         if (exp)
5016                 error = spa_export_common(newname, POOL_STATE_EXPORTED, NULL,
5017                     B_FALSE, B_FALSE);

5019         return (error);

5021 out:
5022         spa_unload(newspa);
5023         spa_deactivate(newspa);
5024         spa_remove(newspa);

5026         txg = spa_vdev_config_enter(spa);

5028         /* re-online all offlined disks */
5029         for (c = 0; c < children; c++) {
5030                 if (vml[c] != NULL)
5031                         vml[c]->vdev_offline = B_FALSE;
5032         }
5033         vdev_reopen(spa->spa_root_vdev);

5035         nvlist_free(spa->spa_config_splitting);
5036         spa->spa_config_splitting = NULL;
5037         (void) spa_vdev_exit(spa, NULL, txg, error);

5039         kmem_free(vml, children * sizeof (vdev_t *));
5040         return (error);
5041 }

5043 static nvlist_t *
5044 spa_nvlist_lookup_by_guid(nvlist_t **nvpp, int count, uint64_t target_guid)
5045 {
5046         for (int i = 0; i < count; i++) {
5047                 uint64_t guid;

5049                 VERIFY(nvlist_lookup_uint64(nvpp[i], ZPOOL_CONFIG_GUID,
5050                     &guid) == 0);

5052                 if (guid == target_guid)
5053                         return (nvpp[i]);
5054         }

5056         return (NULL);
5057 }

5059 static void
5060 spa_vdev_remove_aux(nvlist_t *config, char *name, nvlist_t **dev, int count,
5061         nvlist_t *dev_to_remove)
5062 {
5063         nvlist_t **newdev = NULL;

5065         if (count > 1)
5066                 newdev = kmem_alloc((count - 1) * sizeof (void *), KM_SLEEP);

5068         for (int i = 0, j = 0; i < count; i++) {
5069                 if (dev[i] == dev_to_remove)
5070                         continue;
5071                 VERIFY(nvlist_dup(dev[i], &newdev[j++], KM_SLEEP) == 0);
5072         }

5074         VERIFY(nvlist_remove(config, name, DATA_TYPE_NVLIST_ARRAY) == 0);
```



```
5075         VERIFY(nvlist_add_nvlist_array(config, name, newdev, count - 1) == 0);

5077         for (int i = 0; i < count - 1; i++)
5078                 nvlist_free(newdev[i]);

5080         if (count > 1)
5081                 kmem_free(newdev, (count - 1) * sizeof (void *));
5082 }

5084 /*
5085  * Evacuate the device.
5086  */
5087 static int
5088 spa_vdev_remove_evacuate(spa_t *spa, vdev_t *vd)
5089 {
5090         uint64_t txg;
5091         int error = 0;

5093         ASSERT(MUTEX_HELD(&spa_namespace_lock));
5094         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == 0);
5095         ASSERT(vd == vd->vdev_top);

5097         /*
5098          * Evacuate the device.  We don't hold the config lock as writer
5099          * since we need to do I/O but we do keep the
5100          * spa_namespace_lock held.  Once this completes the device
5101          * should no longer have any blocks allocated on it.
5102          */
5103         if (vd->vdev_islog) {
5104                 if (vd->vdev_stat.vs_alloc != 0)
5105                         error = spa_offline_log(spa);
5106         } else {
5107                 error = SET_ERROR(ENOTSUP);
5108         }

5110         if (error)
5111                 return (error);

5113         /*
5114          * The evacuation succeeded.  Remove any remaining MOS metadata
5115          * associated with this vdev, and wait for these changes to sync.
5116          */
5117         ASSERT0(vd->vdev_stat.vs_alloc);
5118         txg = spa_vdev_config_enter(spa);
5119         vd->vdev_removing = B_TRUE;
5120         vdev_dirty(vd, 0, NULL, txg);
5121         vdev_config_dirty(vd);
5122         spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);

5124         return (0);
5125 }

5127 /*
5128  * Complete the removal by cleaning up the namespace.
5129  */
5130 static void
5131 spa_vdev_remove_from_namespace(spa_t *spa, vdev_t *vd)
5132 {
5133         vdev_t *rvd = spa->spa_root_vdev;
5134         uint64_t id = vd->vdev_id;
5135         boolean_t last_vdev = (id == (rvd->vdev_children - 1));

5137         ASSERT(MUTEX_HELD(&spa_namespace_lock));
5138         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
5139         ASSERT(vd == vd->vdev_top);
```

```
5141         /*
5142          * Only remove any devices which are empty.
5143          */
5144         if (vd->vdev_stat.vs_alloc != 0)
5145                 return;

5147         (void) vdev_label_init(vd, 0, VDEV_LABEL_REMOVE);

5149         if (list_link_active(&vd->vdev_state_dirty_node))
5150                 vdev_state_clean(vd);
5151         if (list_link_active(&vd->vdev_config_dirty_node))
5152                 vdev_config_clean(vd);

5154         vdev_free(vd);

5156         if (last_vdev) {
5157                 vdev_compact_children(rvd);
5158         } else {
5159                 vd = vdev_alloc_common(spa, id, 0, &vdev_hole_ops);
5160                 vdev_add_child(rvd, vd);
5161         }
5162         vdev_config_dirty(rvd);

5164         /*
5165          * Reassess the health of our root vdev.
5166          */
5167         vdev_reopen(rvd);
5168 }

5170 /*
5171  * Remove a device from the pool -
5172  *
5173  * Removing a device from the vdev namespace requires several steps
5174  * and can take a significant amount of time.  As a result we use
5175  * the spa_vdev_config_[enter/exit] functions which allow us to
5176  * grab and release the spa_config_lock while still holding the namespace
5177  * lock.  During each step the configuration is synced out.
5178  */

5180 /*
5181  * Remove a device from the pool.  Currently, this supports removing only hot
5182  * spares, slogs, and level 2 ARC devices.
5183  */
5184 int
5185 spa_vdev_remove(spa_t *spa, uint64_t guid, boolean_t unspare)
5186 {
5187         vdev_t *vd;
5188         metaslab_group_t *mg;
5189         nvlist_t **spares, **l2cache, *nv;
5190         uint64_t txg = 0;
5191         uint_t nspares, nl2cache;
5192         int error = 0;
5193         boolean_t locked = MUTEX_HELD(&spa_namespace_lock);

5195         ASSERT(spa_writeable(spa));

5197         if (!locked)
5198                 txg = spa_vdev_enter(spa);

5200         vd = spa_lookup_by_guid(spa, guid, B_FALSE);

5202         if (spa->spa_spares.sav_vdevs != NULL &&
5203             nvlist_lookup_nvlist_array(spa->spa_spares.sav_config,
5204             ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0 &&
5205             (nv = spa_nvlist_lookup_by_guid(spares, nspares, guid)) != NULL) {
5206                 /*
```



```
5207                  * Only remove the hot spare if it's not currently in use
5208                  * in this pool.
5209                  */
5210                 if (vd == NULL || unspare) {
5211                         spa_vdev_remove_aux(spa->spa_spares.sav_config,
5212                             ZPOOL_CONFIG_SPARES, spares, nspares, nv);
5213                         spa_load_spares(spa);
5214                         spa->spa_spares.sav_sync = B_TRUE;
5215                 } else {
5216                         error = SET_ERROR(EBUSY);
5217                 }
5218         } else if (spa->spa_l2cache.sav_vdevs != NULL &&
5219             nvlist_lookup_nvlist_array(spa->spa_l2cache.sav_config,
5220             ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0 &&
5221             (nv = spa_nvlist_lookup_by_guid(l2cache, nl2cache, guid)) != NULL) {
5222                 /*
5223                  * Cache devices can always be removed.
5224                  */
5225                 spa_vdev_remove_aux(spa->spa_l2cache.sav_config,
5226                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache, nv);
5227                 spa_load_l2cache(spa);
5228                 spa->spa_l2cache.sav_sync = B_TRUE;
5229         } else if (vd != NULL && vd->vdev_islog) {
5230                 ASSERT(!locked);
5231                 ASSERT(vd == vd->vdev_top);

5233                 /*
5234                  * XXX - Once we have bp-rewrite this should
5235                  * become the common case.
5236                  */

5238                 mg = vd->vdev_mg;

5240                 /*
5241                  * Stop allocating from this vdev.
5242                  */
5243                 metaslab_group_passivate(mg);

5245                 /*
5246                  * Wait for the youngest allocations and frees to sync,
5247                  * and then wait for the deferral of those frees to finish.
5248                  */
5249                 spa_vdev_config_exit(spa, NULL,
5250                     txg + TXG_CONCURRENT_STATES + TXG_DEFER_SIZE, 0, FTAG);

5252                 /*
5253                  * Attempt to evacuate the vdev.
5254                  */
5255                 error = spa_vdev_remove_evacuate(spa, vd);

5257                 txg = spa_vdev_config_enter(spa);

5259                 /*
5260                  * If we couldn't evacuate the vdev, unwind.
5261                  */
5262                 if (error) {
5263                         metaslab_group_activate(mg);
5264                         return (spa_vdev_exit(spa, NULL, txg, error));
5265                 }

5267                 /*
5268                  * Clean up the vdev namespace.
5269                  */
5270                 spa_vdev_remove_from_namespace(spa, vd);

5272         } else if (vd != NULL) {
```

```
5273                 /*
5274                  * Normal vdevs cannot be removed (yet).
5275                  */
5276                 error = SET_ERROR(ENOTSUP);
5277         } else {
5278                 /*
5279                  * There is no vdev of any kind with the specified guid.
5280                  */
5281                 error = SET_ERROR(ENOENT);
5282         }

5284         if (!locked)
5285                 return (spa_vdev_exit(spa, NULL, txg, error));

5287         return (error);
5288 }

5290 /*
5291  * Find any device that's done replacing, or a vdev marked 'unspare' that's
5292  * current spared, so we can detach it.
5293  */
5294 static vdev_t *
5295 spa_vdev_resilver_done_hunt(vdev_t *vd)
5296 {
5297         vdev_t *newvd, *oldvd;

5299         for (int c = 0; c < vd->vdev_children; c++) {
5300                 oldvd = spa_vdev_resilver_done_hunt(vd->vdev_child[c]);
5301                 if (oldvd != NULL)
5302                         return (oldvd);
5303         }

5305         /*
5306          * Check for a completed replacement.  We always consider the first
5307          * vdev in the list to be the oldest vdev, and the last one to be
5308          * the newest (see spa_vdev_attach() for how that works).  In
5309          * the case where the newest vdev is faulted, we will not automatically
5310          * remove it after a resilver completes.  This is OK as it will require
5311          * user intervention to determine which disk the admin wishes to keep.
5312          */
5313         if (vd->vdev_ops == &vdev_replacing_ops) {
5314                 ASSERT(vd->vdev_children > 1);

5316                 newvd = vd->vdev_child[vd->vdev_children - 1];
5317                 oldvd = vd->vdev_child[0];

5319                 if (vdev_dtl_empty(newvd, DTL_MISSING) &&
5320                     vdev_dtl_empty(newvd, DTL_OUTAGE) &&
5321                     !vdev_dtl_required(oldvd))
5322                         return (oldvd);
5323         }

5325         /*
5326          * Check for a completed resilver with the 'unspare' flag set.
5327          */
5328         if (vd->vdev_ops == &vdev_spare_ops) {
5329                 vdev_t *first = vd->vdev_child[0];
5330                 vdev_t *last = vd->vdev_child[vd->vdev_children - 1];

5332                 if (last->vdev_unspare) {
5333                         oldvd = first;
5334                         newvd = last;
5335                 } else if (first->vdev_unspare) {
5336                         oldvd = last;
5337                         newvd = first;
5338                 } else {
```



```
5339                         oldvd = NULL;
5340                 }

5342                 if (oldvd != NULL &&
5343                     vdev_dtl_empty(newvd, DTL_MISSING) &&
5344                     vdev_dtl_empty(newvd, DTL_OUTAGE) &&
5345                     !vdev_dtl_required(oldvd))
5346                         return (oldvd);

5348                 /*
5349                  * If there are more than two spares attached to a disk,
5350                  * and those spares are not required, then we want to
5351                  * attempt to free them up now so that they can be used
5352                  * by other pools.  Once we're back down to a single
5353                  * disk+spare, we stop removing them.
5354                  */
5355                 if (vd->vdev_children > 2) {
5356                         newvd = vd->vdev_child[1];

5358                         if (newvd->vdev_isspare && last->vdev_isspare &&
5359                             vdev_dtl_empty(last, DTL_MISSING) &&
5360                             vdev_dtl_empty(last, DTL_OUTAGE) &&
5361                             !vdev_dtl_required(newvd))
5362                                 return (newvd);
5363                 }
5364         }

5366         return (NULL);
5367 }

5369 static void
5370 spa_vdev_resilver_done(spa_t *spa)
5371 {
5372         vdev_t *vd, *pvd, *ppvd;
5373         uint64_t guid, sguid, pguid, ppguid;

5375         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

5377         while ((vd = spa_vdev_resilver_done_hunt(spa->spa_root_vdev)) != NULL) {
5378                 pvd = vd->vdev_parent;
5379                 ppvd = pvd->vdev_parent;
5380                 guid = vd->vdev_guid;
5381                 pguid = pvd->vdev_guid;
5382                 ppguid = ppvd->vdev_guid;
5383                 sguid = 0;
5384                 /*
5385                  * If we have just finished replacing a hot spared device, then
5386                  * we need to detach the parent's first child (the original hot
5387                  * spare) as well.
5388                  */
5389                 if (ppvd->vdev_ops == &vdev_spare_ops && pvd->vdev_id == 0 &&
5390                     ppvd->vdev_children == 2) {
5391                         ASSERT(pvd->vdev_ops == &vdev_replacing_ops);
5392                         sguid = ppvd->vdev_child[1]->vdev_guid;
5393                 }
5394                 spa_config_exit(spa, SCL_ALL, FTAG);
5395                 if (spa_vdev_detach(spa, guid, pguid, B_TRUE) != 0)
5396                         return;
5397                 if (sguid && spa_vdev_detach(spa, sguid, ppguid, B_TRUE) != 0)
5398                         return;
5399                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
5400         }

5402         spa_config_exit(spa, SCL_ALL, FTAG);
5403 }
```

```
5405 /*
5406  * Update the stored path or FRU for this vdev.
5407  */
5408 int
5409 spa_vdev_set_common(spa_t *spa, uint64_t guid, const char *value,
5410     boolean_t ispath)
5411 {
5412         vdev_t *vd;
5413         boolean_t sync = B_FALSE;

5415         ASSERT(spa_writeable(spa));

5417         spa_vdev_state_enter(spa, SCL_ALL);

5419         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
5420                 return (spa_vdev_state_exit(spa, NULL, ENOENT));

5422         if (!vd->vdev_ops->vdev_op_leaf)
5423                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

5425         if (ispath) {
5426                 if (strcmp(value, vd->vdev_path) != 0) {
5427                         spa_strfree(vd->vdev_path);
5428                         vd->vdev_path = spa_strdup(value);
5429                         sync = B_TRUE;
5430                 }
5431         } else {
5432                 if (vd->vdev_fru == NULL) {
5433                         vd->vdev_fru = spa_strdup(value);
5434                         sync = B_TRUE;
5435                 } else if (strcmp(value, vd->vdev_fru) != 0) {
5436                         spa_strfree(vd->vdev_fru);
5437                         vd->vdev_fru = spa_strdup(value);
5438                         sync = B_TRUE;
5439                 }
5440         }

5442         return (spa_vdev_state_exit(spa, sync ? vd : NULL, 0));
5443 }

5445 int
5446 spa_vdev_setpath(spa_t *spa, uint64_t guid, const char *newpath)
5447 {
5448         return (spa_vdev_set_common(spa, guid, newpath, B_TRUE));
5449 }

5451 int
5452 spa_vdev_setfru(spa_t *spa, uint64_t guid, const char *newfru)
5453 {
5454         return (spa_vdev_set_common(spa, guid, newfru, B_FALSE));
5455 }

5457 /*
5458  * ========================================================================
5459  * SPA Scanning
5460  * ========================================================================
5461  */

5463 int
5464 spa_scan_stop(spa_t *spa)
5465 {
5466         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == 0);
5467         if (dsl_scan_resilvering(spa->spa_dsl_pool))
5468                 return (SET_ERROR(EBUSY));
5469         return (dsl_scan_cancel(spa->spa_dsl_pool));
5470 }
```



```
5472 int
5473 spa_scan(spa_t *spa, pool_scan_func_t func)
5474 {
5475         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == 0);

5477         if (func >= POOL_SCAN_FUNCS || func == POOL_SCAN_NONE)
5478                 return (SET_ERROR(ENOTSUP));

5480         /*
5481          * If a resilver was requested, but there is no DTL on a
5482          * writeable leaf device, we have nothing to do.
5483          */
5484         if (func == POOL_SCAN_RESILVER &&
5485             !vdev_resilver_needed(spa->spa_root_vdev, NULL, NULL)) {
5486                 spa_async_request(spa, SPA_ASYNC_RESILVER_DONE);
5487                 return (0);
5488         }

5490         return (dsl_scan(spa->spa_dsl_pool, func));
5491 }

5493 /*
5494  * ========================================================================
5495  * SPA async task processing
5496  * ========================================================================
5497  */

5499 static void
5500 spa_async_remove(spa_t *spa, vdev_t *vd)
5501 {
5502         if (vd->vdev_remove_wanted) {
5503                 vd->vdev_remove_wanted = B_FALSE;
5504                 vd->vdev_delayed_close = B_FALSE;
5505                 vdev_set_state(vd, B_FALSE, VDEV_STATE_REMOVED, VDEV_AUX_NONE);

5507                 /*
5508                  * We want to clear the stats, but we don't want to do a full
5509                  * vdev_clear() as that will cause us to throw away
5510                  * degraded/faulted state as well as attempt to reopen the
5511                  * device, all of which is a waste.
5512                  */
5513                 vd->vdev_stat.vs_read_errors = 0;
5514                 vd->vdev_stat.vs_write_errors = 0;
5515                 vd->vdev_stat.vs_checksum_errors = 0;

5517                 vdev_state_dirty(vd->vdev_top);
5518         }

5520         for (int c = 0; c < vd->vdev_children; c++)
5521                 spa_async_remove(spa, vd->vdev_child[c]);
5522 }

5524 static void
5525 spa_async_probe(spa_t *spa, vdev_t *vd)
5526 {
5527         if (vd->vdev_probe_wanted) {
5528                 vd->vdev_probe_wanted = B_FALSE;
5529                 vdev_reopen(vd);        /* vdev_open() does the actual probe */
5530         }

5532         for (int c = 0; c < vd->vdev_children; c++)
5533                 spa_async_probe(spa, vd->vdev_child[c]);
5534 }

5536 static void
```

```
5537 spa_async_autoexpand(spa_t *spa, vdev_t *vd)
5538 {
5539         sysevent_id_t eid;
5540         nvlist_t *attr;
5541         char *physpath;
5543         if (!spa->spa_autoexpand)
5544                 return;
5546         for (int c = 0; c < vd->vdev_children; c++) {
5547                 vdev_t *cvd = vd->vdev_child[c];
5548                 spa_async_autoexpand(spa, cvd);
5549         }
5551         if (!vd->vdev_ops->vdev_op_leaf || vd->vdev_physpath == NULL)
5552                 return;
5554         physpath = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
5555         (void) snprintf(physpath, MAXPATHLEN, "/devices%s", vd->vdev_physpath);
5557         VERIFY(nvlist_alloc(&attr, NV_UNIQUE_NAME, KM_SLEEP) == 0);
5558         VERIFY(nvlist_add_string(attr, DEV_PHYS_PATH, physpath) == 0);
5560         (void) ddi_log_sysevent(zfs_dip, SUNW_VENDOR, EC_DEV_STATUS,
5561             ESC_DEV_DLE, attr, &eid, DDI_SLEEP);
5563         nvlist_free(attr);
5564         kmem_free(physpath, MAXPATHLEN);
5565 }
5567 static void
5568 spa_async_thread(spa_t *spa)
5569 {
5570         int tasks;
5572         ASSERT(spa->spa_sync_on);
5574         mutex_enter(&spa->spa_async_lock);
5575         tasks = spa->spa_async_tasks;
5576         spa->spa_async_tasks = 0;
5577         mutex_exit(&spa->spa_async_lock);
5579         /*
5580          * See if the config needs to be updated.
5581          */
5582         if (tasks & SPA_ASYNC_CONFIG_UPDATE) {
5583                 uint64_t old_space, new_space;
5585                 mutex_enter(&spa_namespace_lock);
5586                 old_space = metaslab_class_get_space(spa_normal_class(spa));
5587                 spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
5588                 new_space = metaslab_class_get_space(spa_normal_class(spa));
5589                 mutex_exit(&spa_namespace_lock);
5591                 /*
5592                  * If the pool grew as a result of the config update,
5593                  * then log an internal history event.
5594                  */
5595                 if (new_space != old_space) {
5596                         spa_history_log_internal(spa, "vdev online", NULL,
5597                             "pool '%s' size: %llu(+%llu)",
5598                             spa_name(spa), new_space, new_space - old_space);
5599                 }
5600         }
5602         /*
```



```
5603          * See if any devices need to be marked REMOVED.
5604          */
5605         if (tasks & SPA_ASYNC_REMOVE) {
5606                 spa_vdev_state_enter(spa, SCL_NONE);
5607                 spa_async_remove(spa, spa->spa_root_vdev);
5608                 for (int i = 0; i < spa->spa_l2cache.sav_count; i++)
5609                         spa_async_remove(spa, spa->spa_l2cache.sav_vdevs[i]);
5610                 for (int i = 0; i < spa->spa_spares.sav_count; i++)
5611                         spa_async_remove(spa, spa->spa_spares.sav_vdevs[i]);
5612                 (void) spa_vdev_state_exit(spa, NULL, 0);
5613         }
5615         if ((tasks & SPA_ASYNC_AUTOEXPAND) && !spa_suspended(spa)) {
5616                 spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
5617                 spa_async_autoexpand(spa, spa->spa_root_vdev);
5618                 spa_config_exit(spa, SCL_CONFIG, FTAG);
5619         }
5621         /*
5622          * See if any devices need to be probed.
5623          */
5624         if (tasks & SPA_ASYNC_PROBE) {
5625                 spa_vdev_state_enter(spa, SCL_NONE);
5626                 spa_async_probe(spa, spa->spa_root_vdev);
5627                 (void) spa_vdev_state_exit(spa, NULL, 0);
5628         }
5630         /*
5631          * If any devices are done replacing, detach them.
5632          */
5633         if (tasks & SPA_ASYNC_RESILVER_DONE)
5634                 spa_vdev_resilver_done(spa);
5636         /*
5637          * Kick off a resilver.
5638          */
5639         if (tasks & SPA_ASYNC_RESILVER)
5640                 dsl_resilver_restart(spa->spa_dsl_pool, 0);
5642         /*
5643          * Let the world know that we're done.
5644          */
5645         mutex_enter(&spa->spa_async_lock);
5646         spa->spa_async_thread = NULL;
5647         cv_broadcast(&spa->spa_async_cv);
5648         mutex_exit(&spa->spa_async_lock);
5649         thread_exit();
5650 }
5652 void
5653 spa_async_suspend(spa_t *spa)
5654 {
5655         mutex_enter(&spa->spa_async_lock);
5656         spa->spa_async_suspended++;
5657         while (spa->spa_async_thread != NULL)
5658                 cv_wait(&spa->spa_async_cv, &spa->spa_async_lock);
5659         mutex_exit(&spa->spa_async_lock);
5660 }
5662 void
5663 spa_async_resume(spa_t *spa)
5664 {
5665         mutex_enter(&spa->spa_async_lock);
5666         ASSERT(spa->spa_async_suspended != 0);
5667         spa->spa_async_suspended--;
5668         mutex_exit(&spa->spa_async_lock);
```

```
5669 }

5671 static boolean_t
5672 spa_async_tasks_pending(spa_t *spa)
5673 {
5674         u_int non_config_tasks;
5675         u_int config_task;
5676         boolean_t config_task_suspended;

5678         non_config_tasks = spa->spa_async_tasks & ~SPA_ASYNC_CONFIG_UPDATE;
5679         config_task = spa->spa_async_tasks & SPA_ASYNC_CONFIG_UPDATE;
5680         if (spa->spa_ccw_fail_time == 0) {
5681                 config_task_suspended = B_FALSE;
5682         } else {
5683                 config_task_suspended =
5684                     (gethrtime() - spa->spa_ccw_fail_time) <
5685                     (zfs_ccw_retry_interval * NANOSEC);
5686         }

5688         return (non_config_tasks || (config_task && !config_task_suspended));
5689 }

5691 #endif /* ! codereview */
5692 static void
5693 spa_async_dispatch(spa_t *spa)
5694 {
5695         mutex_enter(&spa->spa_async_lock);
5696         if (spa_async_tasks_pending(spa) &&
5697             !spa->spa_async_suspended &&
  80         if (spa->spa_async_tasks && !spa->spa_async_suspended &&
5698             spa->spa_async_thread == NULL &&
5699             rootdir != NULL)
  82             rootdir != NULL && !vn_is_readonly(rootdir))
5700                 spa->spa_async_thread = thread_create(NULL, 0,
5701                     spa_async_thread, spa, 0, &p0, TS_RUN, maxclsyspri);
5702         mutex_exit(&spa->spa_async_lock);
5703 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   15064 Mon Apr 29 12:58:32 2013
new/usr/src/uts/common/fs/zfs/spa_config.c
3749 zfs event processing should work on R/O root filesystems
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#include <sys/spa.h>
#include <sys/fm/fs/zfs.h>
#endif /* ! codereview */
#include <sys/spa_impl.h>
#include <sys/nvpair.h>
#include <sys/uio.h>
#include <sys/fs/zfs.h>
#include <sys/vdev_impl.h>
#include <sys/zfs_ioctl.h>
#include <sys/utsname.h>
#include <sys/systeminfo.h>
#include <sys/sunddi.h>
#include <sys/zfeature.h>
#ifdef _KERNEL
#include <sys/kobj.h>
#include <sys/zone.h>
#endif

/*
 * Pool configuration repository.
 *
 * Pool configuration is stored as a packed nvlist on the filesystem.  By
 * default, all pools are stored in /etc/zfs/zpool.cache and loaded on boot
 * (when the ZFS module is loaded).  Pools can also have the 'cachefile'
 * property set that allows them to be stored in an alternate location until
 * the control of external software.
 *
 * For each cache file, we have a single nvlist which holds all the
 * configuration information.  When the module loads, we read this information
 * from /etc/zfs/zpool.cache and populate the SPA namespace.  This namespace is
 * maintained independently in spa.c.  Whenever the namespace is modified, or
```



```
 * the configuration of a pool is changed, we call spa_config_sync(), which
 * walks through all the active pools and writes the configuration to disk.
 */

static uint64_t spa_config_generation = 1;

/*
 * This can be overridden in userland to preserve an alternate namespace for
 * userland pools when doing testing.
 */
const char *spa_config_path = ZPOOL_CACHE;

/*
 * Called when the module is first loaded, this routine loads the configuration
 * file into the SPA namespace.  It does not actually open or load the pools; it
 * only populates the namespace.
 */
void
spa_config_load(void)
{
        void *buf = NULL;
        nvlist_t *nvlist, *child;
        nvpair_t *nvpair;
        char *pathname;
        struct _buf *file;
        uint64_t fsize;

        /*
         * Open the configuration file.
         */
        pathname = kmem_alloc(MAXPATHLEN, KM_SLEEP);

        (void) snprintf(pathname, MAXPATHLEN, "%s%s",
            (rootdir != NULL) ? "./" : "", spa_config_path);

        file = kobj_open_file(pathname);

        kmem_free(pathname, MAXPATHLEN);

        if (file == (struct _buf *)-1)
                return;

        if (kobj_get_filesize(file, &fsize) != 0)
                goto out;

        buf = kmem_alloc(fsize, KM_SLEEP);

        /*
         * Read the nvlist from the file.
         */
        if (kobj_read_file(file, buf, fsize, 0) < 0)
                goto out;

        /*
         * Unpack the nvlist.
         */
        if (nvlist_unpack(buf, fsize, &nvlist, KM_SLEEP) != 0)
                goto out;

        /*
         * Iterate over all elements in the nvlist, creating a new spa_t for
         * each one with the specified configuration.
         */
        mutex_enter(&spa_namespace_lock);
        nvpair = NULL;
        while ((nvpair = nvlist_next_nvpair(nvlist, nvpair)) != NULL) {
```

```
125                 if (nvpair_type(nvpair) != DATA_TYPE_NVLIST)
126                         continue;
127
128                 VERIFY(nvpair_value_nvlist(nvpair, &child) == 0);
129
130                 if (spa_lookup(nvpair_name(nvpair)) != NULL)
131                         continue;
132                 (void) spa_add(nvpair_name(nvpair), child, NULL);
133         }
134         mutex_exit(&spa_namespace_lock);
135
136         nvlist_free(nvlist);
137
138 out:
139         if (buf != NULL)
140                 kmem_free(buf, fsize);
141
142         kobj_close_file(file);
143 }
144
145 static int
 29 static void
146 spa_config_write(spa_config_dirent_t *dp, nvlist_t *nvl)
147 {
148         size_t buflen;
149         char *buf;
150         vnode_t *vp;
151         int oflags = FWRITE | FTRUNC | FCREAT | FOFFMAX;
152         char *temp;
153         int err;
154 #endif /* ! codereview */
155
156         /*
157          * If the nvlist is empty (NULL), then remove the old cachefile.
158          */
159         if (nvl == NULL) {
160                 err = vn_remove(dp->scd_path, UIO_SYSSPACE, RMFILE);
161                 return (err);
 37                 (void) vn_remove(dp->scd_path, UIO_SYSSPACE, RMFILE);
 38                 return;
162         }
163
164         /*
165          * Pack the configuration into a buffer.
166          */
167         VERIFY(nvlist_size(nvl, &buflen, NV_ENCODE_XDR) == 0);
168
169         buf = kmem_alloc(buflen, KM_SLEEP);
170         temp = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
171
172         VERIFY(nvlist_pack(nvl, &buf, &buflen, NV_ENCODE_XDR,
173             KM_SLEEP) == 0);
174
175         /*
176          * Write the configuration to disk.  We need to do the traditional
177          * 'write to temporary file, sync, move over original' to make sure we
178          * always have a consistent view of the data.
179          */
180         (void) snprintf(temp, MAXPATHLEN, "%s.tmp", dp->scd_path);
181
182         err = vn_open(temp, UIO_SYSSPACE, oflags, 0644, &vp, CRCREAT, 0);
183         if (err == 0) {
184                 err = vn_rdwr(UIO_WRITE, vp, buf, buflen, 0, UIO_SYSSPACE,
185                     0, RLIM64_INFINITY, kcred, NULL);
186                 if (err == 0)
187                         err = VOP_FSYNC(vp, FSYNC, kcred, NULL);
```



```
188                 if (err == 0)
189                         err = vn_rename(temp, dp->scd_path, UIO_SYSSPACE);
 59         if (vn_open(temp, UIO_SYSSPACE, oflags, 0644, &vp, CRCREAT, 0) == 0) {
 60                 if (vn_rdwr(UIO_WRITE, vp, buf, buflen, 0, UIO_SYSSPACE,
 61                     0, RLIM64_INFINITY, kcred, NULL) == 0 &&
 62                     VOP_FSYNC(vp, FSYNC, kcred, NULL) == 0) {
 63                         (void) vn_rename(temp, dp->scd_path, UIO_SYSSPACE);
 64                 }
190                 (void) VOP_CLOSE(vp, oflags, 1, 0, kcred, NULL);
191                 VN_RELE(vp);
192         }
193
194         (void) vn_remove(temp, UIO_SYSSPACE, RMFILE);
195
196         kmem_free(buf, buflen);
197         kmem_free(temp, MAXPATHLEN);
198         return (err);
199 #endif /* ! codereview */
200 }
201
202 /*
203  * Synchronize pool configuration to disk.  This must be called with the
204  * namespace lock held.
205  */
206 void
207 spa_config_sync(spa_t *target, boolean_t removing, boolean_t postsysevent)
208 {
209         spa_config_dirent_t *dp, *tdp;
210         nvlist_t *nvl;
211         boolean_t ccw_failure;
212         int error;
213 #endif /* ! codereview */
214
215         ASSERT(MUTEX_HELD(&spa_namespace_lock));
216
217         if (rootdir == NULL || !(spa_mode_global & FWRITE))
218                 return;
219
220         /*
221          * Iterate over all cachefiles for the pool, past or present.  When the
222          * cachefile is changed, the new one is pushed onto this list, allowing
223          * us to update previous cachefiles that no longer contain this pool.
224          */
225         ccw_failure = B_FALSE;
226 #endif /* ! codereview */
227         for (dp = list_head(&target->spa_config_list); dp != NULL;
228             dp = list_next(&target->spa_config_list, dp)) {
229                 spa_t *spa = NULL;
230                 if (dp->scd_path == NULL)
231                         continue;
232
233                 /*
234                  * Iterate over all pools, adding any matching pools to 'nvl'.
235                  */
236                 nvl = NULL;
237                 while ((spa = spa_next(spa)) != NULL) {
238                         /*
239                          * Skip over our own pool if we're about to remove
240                          * ourselves from the spa namespace or any pool that
241                          * is readonly. Since we cannot guarantee that a
242                          * readonly pool would successfully import upon reboot,
243                          * we don't allow them to be written to the cache file.
244                          */
245                         if ((spa == target && removing) ||
246                             !spa_writeable(spa))
247                                 continue;
```

```
249                         mutex_enter(&spa->spa_props_lock);
250                         tdp = list_head(&spa->spa_config_list);
251                         if (spa->spa_config == NULL ||
252                             tdp->scd_path == NULL ||
253                             strcmp(tdp->scd_path, dp->scd_path) != 0) {
254                                 mutex_exit(&spa->spa_props_lock);
255                                 continue;
256                         }

258                         if (nvl == NULL)
259                                 VERIFY(nvlist_alloc(&nvl, NV_UNIQUE_NAME,
260                                     KM_SLEEP) == 0);

262                         VERIFY(nvlist_add_nvlist(nvl, spa->spa_name,
263                             spa->spa_config) == 0);
264                         mutex_exit(&spa->spa_props_lock);
265                 }

267                 error = spa_config_write(dp, nvl);
268                 if (error != 0)
269                         ccw_failure = B_TRUE;
 73                 spa_config_write(dp, nvl);
270                 nvlist_free(nvl);
271         }

273         if (ccw_failure) {
274                 /*
275                  * Keep trying so that configuration data is
276                  * written if/when any temporary filesystem
277                  * resource issues are resolved.
278                  */
279                 if (target->spa_ccw_fail_time == 0) {
280                         zfs_ereport_post(FM_EREPORT_ZFS_CONFIG_CACHE_WRITE,
281                             target, NULL, NULL, 0, 0);
282                 }
283                 target->spa_ccw_fail_time = gethrtime();
284                 spa_async_request(target, SPA_ASYNC_CONFIG_UPDATE);
285         } else {
286                 /*
287                  * Do not rate limit future attempts to update
288                  * the config cache.
289                  */
290                 target->spa_ccw_fail_time = 0;
291         }

293 #endif /* ! codereview */
294         /*
295          * Remove any config entries older than the current one.
296          */
297         dp = list_head(&target->spa_config_list);
298         while ((tdp = list_next(&target->spa_config_list, dp)) != NULL) {
299                 list_remove(&target->spa_config_list, tdp);
300                 if (tdp->scd_path != NULL)
301                         spa_strfree(tdp->scd_path);
302                 kmem_free(tdp, sizeof (spa_config_dirent_t));
303         }

305         spa_config_generation++;

307         if (postsysevent)
308                 spa_event_notify(target, NULL, ESC_ZFS_CONFIG_SYNC);
309 }

311 /*
312  * Sigh.  Inside a local zone, we don't have access to /etc/zfs/zpool.cache,
```



```
313  * and we don't want to allow the local zone to see all the pools anyway.
314  * So we have to invent the ZFS_IOC_CONFIG ioctl to grab the configuration
315  * information for all pool visible within the zone.
316  */
317 nvlist_t *
318 spa_all_configs(uint64_t *generation)
319 {
320         nvlist_t *pools;
321         spa_t *spa = NULL;

323         if (*generation == spa_config_generation)
324                 return (NULL);

326         VERIFY(nvlist_alloc(&pools, NV_UNIQUE_NAME, KM_SLEEP) == 0);

328         mutex_enter(&spa_namespace_lock);
329         while ((spa = spa_next(spa)) != NULL) {
330                 if (INGLOBALZONE(curproc) ||
331                     zone_dataset_visible(spa_name(spa), NULL)) {
332                         mutex_enter(&spa->spa_props_lock);
333                         VERIFY(nvlist_add_nvlist(pools, spa_name(spa),
334                             spa->spa_config) == 0);
335                         mutex_exit(&spa->spa_props_lock);
336                 }
337         }
338         *generation = spa_config_generation;
339         mutex_exit(&spa_namespace_lock);

341         return (pools);
342 }

344 void
345 spa_config_set(spa_t *spa, nvlist_t *config)
346 {
347         mutex_enter(&spa->spa_props_lock);
348         if (spa->spa_config != NULL)
349                 nvlist_free(spa->spa_config);
350         spa->spa_config = config;
351         mutex_exit(&spa->spa_props_lock);
352 }

354 /*
355  * Generate the pool's configuration based on the current in-core state.
356  * We infer whether to generate a complete config or just one top-level config
357  * based on whether vd is the root vdev.
358  */
359 nvlist_t *
360 spa_config_generate(spa_t *spa, vdev_t *vd, uint64_t txg, int getstats)
361 {
362         nvlist_t *config, *nvroot;
363         vdev_t *rvd = spa->spa_root_vdev;
364         unsigned long hostid = 0;
365         boolean_t locked = B_FALSE;
366         uint64_t split_guid;

368         if (vd == NULL) {
369                 vd = rvd;
370                 locked = B_TRUE;
371                 spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_READER);
372         }

374         ASSERT(spa_config_held(spa, SCL_CONFIG | SCL_STATE, RW_READER) ==
375             (SCL_CONFIG | SCL_STATE));

377         /*
378          * If txg is -1, report the current value of spa->spa_config_txg.
```

```
379          */
380         if (txg == -1ULL)
381                 txg = spa->spa_config_txg;
383         VERIFY(nvlist_alloc(&config, NV_UNIQUE_NAME, KM_SLEEP) == 0);
385         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_VERSION,
386             spa_version(spa)) == 0);
387         VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME,
388             spa_name(spa)) == 0);
389         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE,
390             spa_state(spa)) == 0);
391         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_TXG,
392             txg) == 0);
393         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_GUID,
394             spa_guid(spa)) == 0);
395         VERIFY(spa->spa_comment == NULL || nvlist_add_string(config,
396             ZPOOL_CONFIG_COMMENT, spa->spa_comment) == 0);
399 #ifdef  _KERNEL
400         hostid = zone_get_hostid(NULL);
401 #else   /* _KERNEL */
402         /*
403          * We're emulating the system's hostid in userland, so we can't use
404          * zone_get_hostid().
405          */
406         (void) ddi_strtoul(hw_serial, NULL, 10, &hostid);
407 #endif  /* _KERNEL */
408         if (hostid != 0) {
409                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_HOSTID,
410                     hostid) == 0);
411         }
412         VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_HOSTNAME,
413             utsname.nodename) == 0);
415         if (vd != rvd) {
416                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_TOP_GUID,
417                     vd->vdev_top->vdev_guid) == 0);
418                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_GUID,
419                     vd->vdev_guid) == 0);
420                 if (vd->vdev_isspare)
421                         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_IS_SPARE,
422                             1ULL) == 0);
423                 if (vd->vdev_islog)
424                         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_IS_LOG,
425                             1ULL) == 0);
426                 vd = vd->vdev_top;              /* label contains top config */
427         } else {
428                 /*
429                  * Only add the (potentially large) split information
430                  * in the mos config, and not in the vdev labels
431                  */
432                 if (spa->spa_config_splitting != NULL)
433                         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_SPLIT,
434                             spa->spa_config_splitting) == 0);
435         }
437         /*
438          * Add the top-level config.  We even add this on pools which
439          * don't support holes in the namespace.
440          */
441         vdev_top_config_generate(spa, config);
443         /*
444          * If we're splitting, record the original pool's guid.
```



```
445          */
446         if (spa->spa_config_splitting != NULL &&
447             nvlist_lookup_uint64(spa->spa_config_splitting,
448             ZPOOL_CONFIG_SPLIT_GUID, &split_guid) == 0) {
449                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_SPLIT_GUID,
450                     split_guid) == 0);
451         }
453         nvroot = vdev_config_generate(spa, vd, getstats, 0);
454         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, nvroot) == 0);
455         nvlist_free(nvroot);
457         /*
458          * Store what's necessary for reading the MOS in the label.
459          */
460         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_FEATURES_FOR_READ,
461             spa->spa_label_features) == 0);
463         if (getstats && spa_load_state(spa) == SPA_LOAD_NONE) {
464                 ddt_histogram_t *ddh;
465                 ddt_stat_t *dds;
466                 ddt_object_t *ddo;
468                 ddh = kmem_zalloc(sizeof (ddt_histogram_t), KM_SLEEP);
469                 ddt_get_dedup_histogram(spa, ddh);
470                 VERIFY(nvlist_add_uint64_array(config,
471                     ZPOOL_CONFIG_DDT_HISTOGRAM,
472                     (uint64_t *)ddh, sizeof (*ddh) / sizeof (uint64_t)) == 0);
473                 kmem_free(ddh, sizeof (ddt_histogram_t));
475                 ddo = kmem_zalloc(sizeof (ddt_object_t), KM_SLEEP);
476                 ddt_get_dedup_object_stats(spa, ddo);
477                 VERIFY(nvlist_add_uint64_array(config,
478                     ZPOOL_CONFIG_DDT_OBJ_STATS,
479                     (uint64_t *)ddo, sizeof (*ddo) / sizeof (uint64_t)) == 0);
480                 kmem_free(ddo, sizeof (ddt_object_t));
482                 dds = kmem_zalloc(sizeof (ddt_stat_t), KM_SLEEP);
483                 ddt_get_dedup_stats(spa, dds);
484                 VERIFY(nvlist_add_uint64_array(config,
485                     ZPOOL_CONFIG_DDT_STATS,
486                     (uint64_t *)dds, sizeof (*dds) / sizeof (uint64_t)) == 0);
487                 kmem_free(dds, sizeof (ddt_stat_t));
488         }
490         if (locked)
491                 spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);
493         return (config);
494 }
496 /*
497  * Update all disk labels, generate a fresh config based on the current
498  * in-core state, and sync the global config cache (do not sync the config
499  * cache if this is a booting rootpool).
500  */
501 void
502 spa_config_update(spa_t *spa, int what)
503 {
504         vdev_t *rvd = spa->spa_root_vdev;
505         uint64_t txg;
506         int c;
508         ASSERT(MUTEX_HELD(&spa_namespace_lock));
510         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
```

```
511         txg = spa_last_synced_txg(spa) + 1;
512         if (what == SPA_CONFIG_UPDATE_POOL) {
513                 vdev_config_dirty(rvd);
514         } else {
515                 /*
516                  * If we have top-level vdevs that were added but have
517                  * not yet been prepared for allocation, do that now.
518                  * (It's safe now because the config cache is up to date,
519                  * so it will be able to translate the new DVAs.)
520                  * See comments in spa_vdev_add() for full details.
521                  */
522                 for (c = 0; c < rvd->vdev_children; c++) {
523                         vdev_t *tvd = rvd->vdev_child[c];
524                         if (tvd->vdev_ms_array == 0)
525                                 vdev_metaslab_set_size(tvd);
526                         vdev_expand(tvd, txg);
527                 }
528         }
529         spa_config_exit(spa, SCL_ALL, FTAG);

531         /*
532          * Wait for the mosconfig to be regenerated and synced.
533          */
534         txg_wait_synced(spa->spa_dsl_pool, txg);

536         /*
537          * Update the global config cache to reflect the new mosconfig.
538          */
539         if (!spa->spa_is_root)
540                 spa_config_sync(spa, B_FALSE, what != SPA_CONFIG_UPDATE_POOL);

542         if (what == SPA_CONFIG_UPDATE_POOL)
543                 spa_config_update(spa, SPA_CONFIG_UPDATE_VDEVS);
544 }
```

```
**********************************************************
   10929 Mon Apr 29 12:58:32 2013
new/usr/src/uts/common/fs/zfs/sys/spa_impl.h
3749 zfs event processing should work on R/O root filesystems
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 115 struct spa {
 116         /*
 117          * Fields protected by spa_namespace_lock.
 118          */
 119         char            spa_name[MAXNAMELEN];   /* pool name */
 120         char            *spa_comment;           /* comment */
 121         avl_node_t      spa_avl;                /* node in spa_namespace_avl */
 122         nvlist_t        *spa_config;            /* last synced config */
 123         nvlist_t        *spa_config_syncing;    /* currently syncing config */
 124         nvlist_t        *spa_config_splitting;  /* config for splitting */
 125         nvlist_t        *spa_load_info;         /* info and errors from load */
 126         uint64_t        spa_config_txg;         /* txg of last config change */
 127         int             spa_sync_pass;          /* iterate-to-convergence */
 128         pool_state_t    spa_state;              /* pool state */
 129         int             spa_inject_ref;         /* injection references */
 130         uint8_t         spa_sync_on;            /* sync threads are running */
 131         spa_load_state_t spa_load_state;        /* current load operation */
 132         uint64_t        spa_import_flags;       /* import specific flags */
 133         spa_taskqs_t    spa_zio_taskq[ZIO_TYPES][ZIO_TASKQ_TYPES];
 134         dsl_pool_t      *spa_dsl_pool;
 135         boolean_t       spa_is_initializing;    /* true while opening pool */
 136         metaslab_class_t *spa_normal_class;     /* normal data class */
 137         metaslab_class_t *spa_log_class;        /* intent log data class */
 138         uint64_t        spa_first_txg;          /* first txg after spa_open() */
 139         uint64_t        spa_final_txg;          /* txg of export/destroy */
 140         uint64_t        spa_freeze_txg;         /* freeze pool at this txg */
 141         uint64_t        spa_load_max_txg;       /* best initial ub_txg */
 142         uint64_t        spa_claim_max_txg;      /* highest claimed birth txg */
 143         timespec_t      spa_loaded_ts;          /* 1st successful open time */
 144         objset_t        *spa_meta_objset;       /* copy of dp->dp_meta_objset */
 145         txg_list_t      spa_vdev_txg_list;      /* per-txg dirty vdev list */
 146         vdev_t          *spa_root_vdev;         /* top-level vdev container */
 147         uint64_t        spa_config_guid;        /* config pool guid */
 148         uint64_t        spa_load_guid;          /* spa_load initialized guid */
 149         uint64_t        spa_last_synced_guid;   /* last synced guid */
 150         list_t          spa_config_dirty_list;  /* vdevs with dirty config */
 151         list_t          spa_state_dirty_list;   /* vdevs with dirty state */
 152         spa_aux_vdev_t  spa_spares;             /* hot spares */
 153         spa_aux_vdev_t  spa_l2cache;            /* L2ARC cache devices */
 154         nvlist_t        *spa_label_features;    /* Features for reading MOS */
 155         uint64_t        spa_config_object;      /* MOS object for pool config */
 156         uint64_t        spa_config_generation;  /* config generation number */
 157         uint64_t        spa_syncing_txg;        /* txg currently syncing */
 158         bpobj_t         spa_deferred_bpobj;     /* deferred-free bplist */
 159         bplist_t        spa_free_bplist[TXG_SIZE]; /* bplist of stuff to free */
 160         uberblock_t     spa_ubsync;             /* last synced uberblock */
 161         uberblock_t     spa_uberblock;          /* current uberblock */
 162         boolean_t       spa_extreme_rewind;     /* rewind past deferred frees */
 163         uint64_t        spa_last_io;            /* lbolt of last non-scan I/O */
 164         kmutex_t        spa_scrub_lock;         /* resilver/scrub lock */
 165         uint64_t        spa_scrub_inflight;     /* in-flight scrub I/Os */
 166         kcondvar_t      spa_scrub_io_cv;        /* scrub I/O completion */
 167         uint8_t         spa_scrub_active;       /* active or suspended? */
 168         uint8_t         spa_scrub_type;         /* type of scrub we're doing */
 169         uint8_t         spa_scrub_finished;     /* indicator to rotate logs */
 170         uint8_t         spa_scrub_started;      /* started since last boot */
```



```
 171         uint8_t         spa_scrub_reopen;       /* scrub doing vdev_reopen */
 172         uint64_t        spa_scan_pass_start;    /* start time per pass/reboot */
 173         uint64_t        spa_scan_pass_exam;     /* examined bytes per pass */
 174         kmutex_t        spa_async_lock;         /* protect async state */
 175         kthread_t       *spa_async_thread;      /* thread doing async task */
 176         int             spa_async_suspended;    /* async tasks suspended */
 177         kcondvar_t      spa_async_cv;           /* wait for thread_exit() */
 178         uint16_t        spa_async_tasks;        /* async task mask */
 179         char            *spa_root;              /* alternate root directory */
 180         uint64_t        spa_ena;                /* spa-wide ereport ENA */
 181         int             spa_last_open_failed;   /* error if last open failed */
 182         uint64_t        spa_last_ubsync_txg;    /* "best" uberblock txg */
 183         uint64_t        spa_last_ubsync_txg_ts; /* timestamp from that ub */
 184         uint64_t        spa_load_txg;           /* ub txg that loaded */
 185         uint64_t        spa_load_txg_ts;        /* timestamp from that ub */
 186         uint64_t        spa_load_meta_errors;   /* verify metadata err count */
 187         uint64_t        spa_load_data_errors;   /* verify data err count */
 188         uint64_t        spa_verify_min_txg;     /* start txg of verify scrub */
 189         kmutex_t        spa_errlog_lock;        /* error log lock */
 190         uint64_t        spa_errlog_last;        /* last error log object */
 191         uint64_t        spa_errlog_scrub;       /* scrub error log object */
 192         kmutex_t        spa_errlist_lock;       /* error list/ereport lock */
 193         avl_tree_t      spa_errlist_last;       /* last error list */
 194         avl_tree_t      spa_errlist_scrub;      /* scrub error list */
 195         uint64_t        spa_deflate;            /* should we deflate? */
 196         uint64_t        spa_history;            /* history object */
 197         kmutex_t        spa_history_lock;       /* history lock */
 198         vdev_t          *spa_pending_vdev;      /* pending vdev additions */
 199         kmutex_t        spa_props_lock;         /* property lock */
 200         uint64_t        spa_pool_props_object;  /* object for properties */
 201         uint64_t        spa_bootfs;             /* default boot filesystem */
 202         uint64_t        spa_failmode;           /* failure mode for the pool */
 203         uint64_t        spa_delegation;         /* delegation on/off */
 204         list_t          spa_config_list;        /* previous cache file(s) */
 205         zio_t           *spa_async_zio_root;    /* root of all async I/O */
 206         zio_t           *spa_suspend_zio_root;  /* root of all suspended I/O */
 207         kmutex_t        spa_suspend_lock;       /* protects suspend_zio_root */
 208         kcondvar_t      spa_suspend_cv;         /* notification of resume */
 209         uint8_t         spa_suspended;          /* pool is suspended */
 210         uint8_t         spa_claiming;           /* pool is doing zil_claim() */
 211         boolean_t       spa_debug;              /* debug enabled? */
 212         boolean_t       spa_is_root;            /* pool is root */
 213         int             spa_minref;             /* num refs when first opened */
 214         int             spa_mode;               /* FREAD | FWRITE */
 215         spa_log_state_t spa_log_state;          /* log state */
 216         uint64_t        spa_autoexpand;         /* lun expansion on/off */
 217         ddt_t           *spa_ddt[ZIO_CHECKSUM_FUNCTIONS]; /* in-core DDTs */
 218         uint64_t        spa_ddt_stat_object;    /* DDT statistics */
 219         uint64_t        spa_dedup_ditto;        /* dedup ditto threshold */
 220         uint64_t        spa_dedup_checksum;     /* default dedup checksum */
 221         uint64_t        spa_dspace;             /* dspace in normal class */
 222         kmutex_t        spa_vdev_top_lock;      /* dueling offline/remove */
 223         kmutex_t        spa_proc_lock;          /* protects spa_proc* */
 224         kcondvar_t      spa_proc_cv;            /* spa_proc_state transitions */
 225         spa_proc_state_t spa_proc_state;        /* see definition */
 226         struct proc     *spa_proc;              /* "zpool-poolname" process */
 227         uint64_t        spa_did;                /* if procp != p0, did of t1 */
 228         boolean_t       spa_autoreplace;        /* autoreplace set in open */
 229         int             spa_vdev_locks;         /* locks grabbed */
 230         uint64_t        spa_creation_version;   /* version at pool creation */
 231         uint64_t        spa_prev_software_version; /* See ub_software_version */
 232         uint64_t        spa_feat_for_write_obj; /* required to write to pool */
 233         uint64_t        spa_feat_for_read_obj;  /* required to read from pool */
 234         uint64_t        spa_feat_desc_obj;      /* Feature descriptions */
 235         cyclic_id_t     spa_deadman_cycid;      /* cyclic id */
 236         uint64_t        spa_deadman_calls;      /* number of deadman calls */
```

```
237         uint64_t        spa_sync_starttime;     /* starting time fo spa_sync */
238         uint64_t        spa_deadman_synctime;   /* deadman expiration timer */
239         kmutex_t        spa_iokstat_lock;       /* protects spa_iokstat_* */
240         struct kstat    *spa_iokstat;           /* kstat of io to this pool */
241         hrtime_t        spa_ccw_fail_time;      /* Conf cache write fail time */
242 #endif /* ! codereview */
243         /*
244          * spa_refcnt & spa_config_lock must be the last elements
245          * because refcount_t changes size based on compilation options.
246          * In order for the MDB module to function correctly, the other
247          * fields must remain in the same location.
248          */
249         spa_config_lock_t spa_config_lock[SCL_LOCKS]; /* config changes */
250         refcount_t      spa_refcount;           /* number of opens */
251 };

253 extern const char *spa_config_path;

255 extern void spa_taskq_dispatch_ent(spa_t *spa, zio_type_t t, zio_taskq_type_t q,
256     task_func_t *func, void *arg, uint_t flags, taskq_ent_t *ent);

258 #ifdef  __cplusplus
259 }
260 #endif

262 #endif  /* _SYS_SPA_IMPL_H */
```

```
**********************************************************
    4091 Mon Apr 29 12:58:32 2013
new/usr/src/uts/common/sys/fm/fs/zfs.h
3749 zfs event processing should work on R/O root filesystems
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_FM_FS_ZFS_H
#define _SYS_FM_FS_ZFS_H

#ifdef  __cplusplus
extern "C" {
#endif

#define ZFS_ERROR_CLASS                         "fs.zfs"

#define FM_EREPORT_ZFS_CHECKSUM                 "checksum"
#define FM_EREPORT_ZFS_IO                       "io"
#define FM_EREPORT_ZFS_DATA                     "data"
#define FM_EREPORT_ZFS_POOL                     "zpool"
#define FM_EREPORT_ZFS_DEVICE_UNKNOWN           "vdev.unknown"
#define FM_EREPORT_ZFS_DEVICE_OPEN_FAILED       "vdev.open_failed"
#define FM_EREPORT_ZFS_DEVICE_CORRUPT_DATA      "vdev.corrupt_data"
#define FM_EREPORT_ZFS_DEVICE_NO_REPLICAS       "vdev.no_replicas"
#define FM_EREPORT_ZFS_DEVICE_BAD_GUID_SUM      "vdev.bad_guid_sum"
#define FM_EREPORT_ZFS_DEVICE_TOO_SMALL         "vdev.too_small"
#define FM_EREPORT_ZFS_DEVICE_BAD_LABEL         "vdev.bad_label"
#define FM_EREPORT_ZFS_IO_FAILURE               "io_failure"
#define FM_EREPORT_ZFS_PROBE_FAILURE            "probe_failure"
#define FM_EREPORT_ZFS_LOG_REPLAY               "log_replay"
#define FM_EREPORT_ZFS_CONFIG_CACHE_WRITE       "config_cache_write"
#endif /* ! codereview */

#define FM_EREPORT_PAYLOAD_ZFS_POOL             "pool"
#define FM_EREPORT_PAYLOAD_ZFS_POOL_FAILMODE    "pool_failmode"
#define FM_EREPORT_PAYLOAD_ZFS_POOL_GUID        "pool_guid"
#define FM_EREPORT_PAYLOAD_ZFS_POOL_CONTEXT     "pool_context"
#define FM_EREPORT_PAYLOAD_ZFS_VDEV_GUID        "vdev_guid"
#define FM_EREPORT_PAYLOAD_ZFS_VDEV_TYPE        "vdev_type"
#define FM_EREPORT_PAYLOAD_ZFS_VDEV_PATH        "vdev_path"
```



```
#define FM_EREPORT_PAYLOAD_ZFS_VDEV_DEVID       "vdev_devid"
#define FM_EREPORT_PAYLOAD_ZFS_VDEV_FRU         "vdev_fru"
#define FM_EREPORT_PAYLOAD_ZFS_PARENT_GUID      "parent_guid"
#define FM_EREPORT_PAYLOAD_ZFS_PARENT_TYPE      "parent_type"
#define FM_EREPORT_PAYLOAD_ZFS_PARENT_PATH      "parent_path"
#define FM_EREPORT_PAYLOAD_ZFS_PARENT_DEVID     "parent_devid"
#define FM_EREPORT_PAYLOAD_ZFS_ZIO_OBJSET       "zio_objset"
#define FM_EREPORT_PAYLOAD_ZFS_ZIO_OBJECT       "zio_object"
#define FM_EREPORT_PAYLOAD_ZFS_ZIO_LEVEL        "zio_level"
#define FM_EREPORT_PAYLOAD_ZFS_ZIO_BLKID        "zio_blkid"
#define FM_EREPORT_PAYLOAD_ZFS_ZIO_ERR          "zio_err"
#define FM_EREPORT_PAYLOAD_ZFS_ZIO_OFFSET       "zio_offset"
#define FM_EREPORT_PAYLOAD_ZFS_ZIO_SIZE         "zio_size"
#define FM_EREPORT_PAYLOAD_ZFS_PREV_STATE       "prev_state"
#define FM_EREPORT_PAYLOAD_ZFS_CKSUM_EXPECTED   "cksum_expected"
#define FM_EREPORT_PAYLOAD_ZFS_CKSUM_ACTUAL     "cksum_actual"
#define FM_EREPORT_PAYLOAD_ZFS_CKSUM_ALGO       "cksum_algorithm"
#define FM_EREPORT_PAYLOAD_ZFS_CKSUM_BYTESWAP   "cksum_byteswap"
#define FM_EREPORT_PAYLOAD_ZFS_BAD_OFFSET_RANGES "bad_ranges"
#define FM_EREPORT_PAYLOAD_ZFS_BAD_RANGE_MIN_GAP "bad_ranges_min_gap"
#define FM_EREPORT_PAYLOAD_ZFS_BAD_RANGE_SETS   "bad_range_sets"
#define FM_EREPORT_PAYLOAD_ZFS_BAD_RANGE_CLEARS "bad_range_clears"
#define FM_EREPORT_PAYLOAD_ZFS_BAD_SET_BITS     "bad_set_bits"
#define FM_EREPORT_PAYLOAD_ZFS_BAD_CLEARED_BITS "bad_cleared_bits"
#define FM_EREPORT_PAYLOAD_ZFS_BAD_SET_HISTOGRAM "bad_set_histogram"
#define FM_EREPORT_PAYLOAD_ZFS_BAD_CLEARED_HISTOGRAM "bad_cleared_histogram"

#define FM_EREPORT_FAILMODE_WAIT                "wait"
#define FM_EREPORT_FAILMODE_CONTINUE            "continue"
#define FM_EREPORT_FAILMODE_PANIC               "panic"

#define FM_RESOURCE_REMOVED                     "removed"
#define FM_RESOURCE_AUTOREPLACE                 "autoreplace"
#define FM_RESOURCE_STATECHANGE                 "statechange"

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_FM_FS_ZFS_H */
```