```
**********************************************************
   13077 Tue Oct  1 13:06:33 2013
new/usr/src/cmd/zhack/zhack.c
4168 ztest assertion failure in dbuf_undirty
4169 verbatim import causes zdb to segfault
4170 zhack leaves pool in ACTIVE state
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Eric Schrock <eric.schrock@delphix.com>
Reviewed by: Matthew Ahrens <mahrens@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) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 */

/*
 * zhack is a debugging tool that can write changes to ZFS pool using libzpool
 * for testing purposes. Altering pools with zhack is unsupported and may
 * result in corrupted pools.
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/zfs_context.h>
#include <sys/spa.h>
#include <sys/spa_impl.h>
#include <sys/dmu.h>
#include <sys/zap.h>
#include <sys/zfs_znode.h>
#include <sys/dsl_synctask.h>
#include <sys/vdev.h>
#include <sys/fs/zfs.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_pool.h>
#include <sys/zio_checksum.h>
#include <sys/zio_compress.h>
#include <sys/zfeature.h>
#include <sys/dmu_tx.h>
#undef ZFS_MAXNAMELEN
#undef verify
#include <libzfs.h>

extern boolean_t zfeature_checks_disable;
```



```
const char cmdname[] = "zhack";
libzfs_handle_t *g_zfs;
static importargs_t g_importargs;
static char *g_pool;
static boolean_t g_readonly;

static void
usage(void)
{
        (void) fprintf(stderr,
            "Usage: %s [-c cachefile] [-d dir] <subcommand> <args> ...\n"
            "where <subcommand> <args> is one of the following:\n"
            "\n", cmdname);

        (void) fprintf(stderr,
            "    feature stat <pool>\n"
            "        print information about enabled features\n"
            "    feature enable [-d desc] <pool> <feature>\n"
            "        add a new enabled feature to the pool\n"
            "        -d <desc> sets the feature's description\n"
            "    feature ref [-md] <pool> <feature>\n"
            "        change the refcount on the given feature\n"
            "        -d decrease instead of increase the refcount\n"
            "        -m add the feature to the label if increasing refcount\n"
            "\n"
            "    <feature> : should be a feature guid\n");
        exit(1);
}


static void
fatal(spa_t *spa, void *tag, const char *fmt, ...)
fatal(const char *fmt, ...)
{
        va_list ap;

        if (spa != NULL) {
                spa_close(spa, tag);
                (void) spa_export(g_pool, NULL, B_TRUE, B_FALSE);
        }

        va_start(ap, fmt);
        (void) fprintf(stderr, "%s: ", cmdname);
        (void) vfprintf(stderr, fmt, ap);
        va_end(ap);
        (void) fprintf(stderr, "\n");

        exit(1);
}
```

_____*unchanged_portion_omitted_*

```
/*
 * Target is the dataset whose pool we want to open.
 */
static void
import_pool(const char *target, boolean_t readonly)
{
        nvlist_t *config;
        nvlist_t *pools;
        int error;
        char *sepp;
        spa_t *spa;
        nvpair_t *elem;
        nvlist_t *props;
        const char *name;
```

```
136         kernel_init(readonly ? FREAD : (FREAD | FWRITE));
137         g_zfs = libzfs_init();
138         ASSERT(g_zfs != NULL);

140         dmu_objset_register_type(DMU_OST_ZFS, space_delta_cb);

142         g_readonly = readonly;

144         /*
145          * If we only want readonly access, it's OK if we find
146          * a potentially-active (ie, imported into the kernel) pool from the
147          * default cachefile.
148          */
149         if (readonly && spa_open(target, &spa, FTAG) == 0) {
150                 spa_close(spa, FTAG);
151                 return;
152         }

154         g_importargs.unique = B_TRUE;
155         g_importargs.can_be_active = readonly;
156         g_pool = strdup(target);
157         if ((sepp = strpbrk(g_pool, "/@")) != NULL)
158                 *sepp = '\0';
159         g_importargs.poolname = g_pool;
160         pools = zpool_search_import(g_zfs, &g_importargs);

162         if (nvlist_empty(pools)) {
163                 if (!g_importargs.can_be_active) {
164                         g_importargs.can_be_active = B_TRUE;
165                         if (zpool_search_import(g_zfs, &g_importargs) != NULL ||
166                             spa_open(target, &spa, FTAG) == 0) {
167                                 fatal(spa, FTAG, "cannot import '%s': pool is "
168                                     "active; run " "\"zpool export %s\" "
169                                     "first\n", g_pool, g_pool);
162                                 fatal("cannot import '%s': pool is active; run "
163                                     "\"zpool export %s\" first\n",
164                                     g_pool, g_pool);
170                         }
171                 }

173                 fatal(NULL, FTAG, "cannot import '%s': no such pool "
174                     "available\n", g_pool);
168                 fatal("cannot import '%s': no such pool available\n", g_pool);
175         }

177         elem = nvlist_next_nvpair(pools, NULL);
178         name = nvpair_name(elem);
179         verify(nvpair_value_nvlist(elem, &config) == 0);

181         props = NULL;
182         if (readonly) {
183                 verify(nvlist_alloc(&props, NV_UNIQUE_NAME, 0) == 0);
184                 verify(nvlist_add_uint64(props,
185                     zpool_prop_to_name(ZPOOL_PROP_READONLY), 1) == 0);
186         }

188         zfeature_checks_disable = B_TRUE;
189         error = spa_import(name, config, props, ZFS_IMPORT_NORMAL);
190         zfeature_checks_disable = B_FALSE;
191         if (error == EEXIST)
192                 error = 0;

194         if (error)
195                 fatal(NULL, FTAG, "can't import '%s': %s", name,
196                     strerror(error));
```



```
189                 fatal("can't import '%s': %s", name, strerror(error));
197 }

199 static void
200 zhack_spa_open(const char *target, boolean_t readonly, void *tag, spa_t **spa)
201 {
202         int err;

204         import_pool(target, readonly);

206         zfeature_checks_disable = B_TRUE;
207         err = spa_open(target, spa, tag);
208         zfeature_checks_disable = B_FALSE;

210         if (err != 0)
211                 fatal(*spa, FTAG, "cannot open '%s': %s", target,
212                     strerror(err));
204                 fatal("cannot open '%s': %s", target, strerror(err));
213         if (spa_version(*spa) < SPA_VERSION_FEATURES) {
214                 fatal(*spa, FTAG, "'%s' has version %d, features not enabled",
215                     target, (int)spa_version(*spa));
206                 fatal("'%s' has version %d, features not enabled", target,
207                     (int)spa_version(*spa));
216         }
217 }
_____unchanged_portion_omitted_

297 static void
298 zhack_do_feature_enable(int argc, char **argv)
299 {
300         char c;
301         char *desc, *target;
302         spa_t *spa;
303         objset_t *mos;
304         zfeature_info_t feature;
305         zfeature_info_t *nodeps[] = { NULL };

307         /*
308          * Features are not added to the pool's label until their refcounts
309          * are incremented, so fi_mos can just be left as false for now.
310          */
311         desc = NULL;
312         feature.fi_uname = "zhack";
313         feature.fi_mos = B_FALSE;
314         feature.fi_can_readonly = B_FALSE;
315         feature.fi_depends = nodeps;

317         optind = 1;
318         while ((c = getopt(argc, argv, "rmd:")) != -1) {
319                 switch (c) {
320                 case 'r':
321                         feature.fi_can_readonly = B_TRUE;
322                         break;
323                 case 'd':
324                         desc = strdup(optarg);
325                         break;
326                 default:
327                         usage();
328                         break;
329                 }
330         }

332         if (desc == NULL)
333                 desc = strdup("zhack injected");
334         feature.fi_desc = desc;
```

```
336         argc -= optind;
337         argv += optind;
338
339         if (argc < 2) {
340                 (void) fprintf(stderr, "error: missing feature or pool name\n");
341                 usage();
342         }
343         target = argv[0];
344         feature.fi_guid = argv[1];
345
346         if (!zfeature_is_valid_guid(feature.fi_guid))
347                 fatal(NULL, FTAG, "invalid feature guid: %s", feature.fi_guid);
339                 fatal("invalid feature guid: %s", feature.fi_guid);
348
349         zhack_spa_open(target, B_FALSE, FTAG, &spa);
350         mos = spa->spa_meta_objset;
351
352         if (0 == zfeature_lookup_guid(feature.fi_guid, NULL))
353                 fatal(spa, FTAG, "'%s' is a real feature, will not enable");
345                 fatal("'%s' is a real feature, will not enable");
354         if (0 == zap_contains(mos, spa->spa_feat_desc_obj, feature.fi_guid))
355                 fatal(spa, FTAG, "feature already enabled: %s",
356                     feature.fi_guid);
347                 fatal("feature already enabled: %s", feature.fi_guid);
357
358         VERIFY0(dsl_sync_task(spa_name(spa), NULL,
359             feature_enable_sync, &feature, 5));
360
361         spa_close(spa, FTAG);
362
363         free(desc);
364 }
```

*unchanged_portion_omitted_*

```
388 static void
389 zhack_do_feature_ref(int argc, char **argv)
390 {
391         char c;
392         char *target;
393         boolean_t decr = B_FALSE;
394         spa_t *spa;
395         objset_t *mos;
396         zfeature_info_t feature;
397         zfeature_info_t *nodeps[] = { NULL };
398
399         /*
400          * fi_desc does not matter here because it was written to disk
401          * when the feature was enabled, but we need to properly set the
402          * feature for read or write based on the information we read off
403          * disk later.
404          */
405         feature.fi_uname = "zhack";
406         feature.fi_mos = B_FALSE;
407         feature.fi_desc = NULL;
408         feature.fi_depends = nodeps;
409
410         optind = 1;
411         while ((c = getopt(argc, argv, "md")) != -1) {
412                 switch (c) {
413                 case 'm':
414                         feature.fi_mos = B_TRUE;
415                         break;
416                 case 'd':
417                         decr = B_TRUE;
418                         break;
419                 default:
```



```
420                         usage();
421                         break;
422                 }
423         }
424         argc -= optind;
425         argv += optind;
426
427         if (argc < 2) {
428                 (void) fprintf(stderr, "error: missing feature or pool name\n");
429                 usage();
430         }
431         target = argv[0];
432         feature.fi_guid = argv[1];
433
434         if (!zfeature_is_valid_guid(feature.fi_guid))
435                 fatal(NULL, FTAG, "invalid feature guid: %s", feature.fi_guid);
426                 fatal("invalid feature guid: %s", feature.fi_guid);
436
437         zhack_spa_open(target, B_FALSE, FTAG, &spa);
438         mos = spa->spa_meta_objset;
439
440         if (0 == zfeature_lookup_guid(feature.fi_guid, NULL))
441                 fatal(spa, FTAG, "'%s' is a real feature, will not change "
442                     "refcount");
432                 fatal("'%s' is a real feature, will not change refcount");
443
444         if (0 == zap_contains(mos, spa->spa_feat_for_read_obj,
445             feature.fi_guid)) {
446                 feature.fi_can_readonly = B_FALSE;
447         } else if (0 == zap_contains(mos, spa->spa_feat_for_write_obj,
448             feature.fi_guid)) {
449                 feature.fi_can_readonly = B_TRUE;
450         } else {
451                 fatal(spa, FTAG, "feature is not enabled: %s", feature.fi_guid);
441                 fatal("feature is not enabled: %s", feature.fi_guid);
452         }
453
454         if (decr && !spa_feature_is_active(spa, &feature))
455                 fatal(spa, FTAG, "feature refcount already 0: %s",
456                     feature.fi_guid);
445                 fatal("feature refcount already 0: %s", feature.fi_guid);
457
458         VERIFY0(dsl_sync_task(spa_name(spa), NULL,
459             decr ? feature_decr_sync : feature_incr_sync, &feature, 5));
460
461         spa_close(spa, FTAG);
462 }
```

*unchanged_portion_omitted_*

```
493 #define MAX_NUM_PATHS 1024
494
495 int
496 main(int argc, char **argv)
497 {
498         extern void zfs_prop_init(void);
499
500         char *path[MAX_NUM_PATHS];
501         const char *subcommand;
502         int rv = 0;
503         char c;
504
505         g_importargs.path = path;
506
507         dprintf_setup(&argc, argv);
508         zfs_prop_init();
```

```
510         while ((c = getopt(argc, argv, "c:d:")) != -1) {
511                 switch (c) {
512                 case 'c':
513                         g_importargs.cachefile = optarg;
514                         break;
515                 case 'd':
516                         assert(g_importargs.paths < MAX_NUM_PATHS);
517                         g_importargs.path[g_importargs.paths++] = optarg;
518                         break;
519                 default:
520                         usage();
521                         break;
522                 }
523         }

525         argc -= optind;
526         argv += optind;
527         optind = 1;

529         if (argc == 0) {
530                 (void) fprintf(stderr, "error: no command specified\n");
531                 usage();
532         }

534         subcommand = argv[0];

536         if (strcmp(subcommand, "feature") == 0) {
537                 rv = zhack_do_feature(argc, argv);
538         } else {
539                 (void) fprintf(stderr, "error: unknown subcommand: %s\n",
540                     subcommand);
541                 usage();
542         }

544         if (!g_readonly && spa_export(g_pool, NULL, B_TRUE, B_FALSE) != 0) {
545                 fatal(NULL, FTAG, "pool export failed; "
533         if (!g_readonly && spa_export(g_pool, NULL, B_TRUE, B_TRUE) != 0) {
534                 fatal("pool export failed; "
546                     "changes may not be committed to disk\n");
547         }

549         libzfs_fini(g_zfs);
550         kernel_fini();

552         return (rv);
553 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   76779 Tue Oct  1 13:06:34 2013
new/usr/src/uts/common/fs/zfs/dbuf.c
4168 ztest assertion failure in dbuf_undirty
4169 verbatim import causes zdb to segfault
4170 zhack leaves pool in ACTIVE state
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Eric Schrock <eric.schrock@delphix.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1307 /*
1308  * Undirty a buffer in the transaction group referenced by the given
1309  * transaction.  Return whether this evicted the dbuf.
1310  */
1311 static boolean_t
1312 dbuf_undirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1313 {
1314         dnode_t *dn;
1315         uint64_t txg = tx->tx_txg;
1316         dbuf_dirty_record_t *dr, **drp;

1318         ASSERT(txg != 0);
1319         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1320         ASSERT0(db->db_level);
1321         ASSERT(MUTEX_HELD(&db->db_mtx));

1323         /*
1324          * If this buffer is not dirty, we're done.
1325          */
1326         for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1327                 if (dr->dr_txg <= txg)
1328                         break;
1329         if (dr == NULL || dr->dr_txg < txg)
1330                 return (B_FALSE);
1331         ASSERT(dr->dr_txg == txg);
1332         ASSERT(dr->dr_dbuf == db);

1334         DB_DNODE_ENTER(db);
1335         dn = DB_DNODE(db);

1337         /*
1338          * Note:  This code will probably work even if there are concurrent
1339          * holders, but it is untested in that scenerio, as the ZPL and
1340          * ztest have additional locking (the range locks) that prevents
1341          * that type of concurrent access.
1342          */
1343         ASSERT3U(refcount_count(&db->db_holds), ==, db->db_dirtycnt);

1337         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1339         ASSERT(db->db.db_size != 0);

1341         /*
1342          * Any space we accounted for in dp_dirty_* will be cleaned up by
1343          * dsl_pool_sync().  This is relatively rare so the discrepancy
1344          * is not a big deal.
1345          */

1347         *drp = dr->dr_next;

1349         /*
1350          * Note that there are three places in dbuf_dirty()
1351          * where this dirty record may be put on a list.
1352          * Make sure to do a list_remove corresponding to
```



```
1353          * every one of those list_insert calls.
1354          */
1355         if (dr->dr_parent) {
1356                 mutex_enter(&dr->dr_parent->dt.di.dr_mtx);
1357                 list_remove(&dr->dr_parent->dt.di.dr_children, dr);
1358                 mutex_exit(&dr->dr_parent->dt.di.dr_mtx);
1359         } else if (db->db_blkid == DMU_SPILL_BLKID ||
1360             db->db_level+1 == dn->dn_nlevels) {
1361                 ASSERT(db->db_blkptr == NULL || db->db_parent == dn->dn_dbuf);
1362                 mutex_enter(&dn->dn_mtx);
1363                 list_remove(&dn->dn_dirty_records[txg & TXG_MASK], dr);
1364                 mutex_exit(&dn->dn_mtx);
1365         }
1366         DB_DNODE_EXIT(db);

1368         if (db->db_state != DB_NOFILL) {
1369                 dbuf_unoverride(dr);

1371                 ASSERT(db->db_buf != NULL);
1372                 ASSERT(dr->dt.dl.dr_data != NULL);
1373                 if (dr->dt.dl.dr_data != db->db_buf)
1374                         VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data, db));
1375         }
1376         kmem_free(dr, sizeof (dbuf_dirty_record_t));

1378         ASSERT(db->db_dirtycnt > 0);
1379         db->db_dirtycnt -= 1;

1381         if (refcount_remove(&db->db_holds, (void *)(uintptr_t)txg) == 0) {
1382                 arc_buf_t *buf = db->db_buf;

1384                 ASSERT(db->db_state == DB_NOFILL || arc_released(buf));
1385                 dbuf_set_data(db, NULL);
1386                 VERIFY(arc_buf_remove_ref(buf, db));
1387                 dbuf_evict(db);
1388                 return (B_TRUE);
1389         }

1391         return (B_FALSE);
1392 }
______unchanged_portion_omitted_
```

```
**********************************************************
  175960 Tue Oct  1 13:06:35 2013
new/usr/src/uts/common/fs/zfs/spa.c
4168 ztest assertion failure in dbuf_undirty
4169 verbatim import causes zdb to segfault
4170 zhack leaves pool in ACTIVE state
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Eric Schrock <eric.schrock@delphix.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_

3794 #endif

3796 /*
3797  * Import a non-root pool into the system.
3798  */
3799 int
3800 spa_import(const char *pool, nvlist_t *config, nvlist_t *props, uint64_t flags)
3801 {
3802         spa_t *spa;
3803         char *altroot = NULL;
3804         spa_load_state_t state = SPA_LOAD_IMPORT;
3805         zpool_rewind_policy_t policy;
3806         uint64_t mode = spa_mode_global;
3807         uint64_t readonly = B_FALSE;
3808         int error;
3809         nvlist_t *nvroot;
3810         nvlist_t **spares, **l2cache;
3811         uint_t nspares, nl2cache;

3813         /*
3814          * If a pool with this name exists, return failure.
3815          */
3816         mutex_enter(&spa_namespace_lock);
3817         if (spa_lookup(pool) != NULL) {
3818                 mutex_exit(&spa_namespace_lock);
3819                 return (SET_ERROR(EEXIST));
3820         }

3822         /*
3823          * Create and initialize the spa structure.
3824          */
3825         (void) nvlist_lookup_string(props,
3826             zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3827         (void) nvlist_lookup_uint64(props,
3828             zpool_prop_to_name(ZPOOL_PROP_READONLY), &readonly);
3829         if (readonly)
3830                 mode = FREAD;
3831         spa = spa_add(pool, config, altroot);
3832         spa->spa_import_flags = flags;

3834         /*
3835          * Verbatim import - Take a pool and insert it into the namespace
3836          * as if it had been loaded at boot.
3837          */
3838         if (spa->spa_import_flags & ZFS_IMPORT_VERBATIM) {
3839                 if (props != NULL)
3840                         spa_configfile_set(spa, props, B_FALSE);

3842                 spa_config_sync(spa, B_FALSE, B_TRUE);

3844                 mutex_exit(&spa_namespace_lock);
3845                 spa_history_log_version(spa, "import");

3845                 return (0);
```



```
3846         }

3848         spa_activate(spa, mode);

3850         /*
3851          * Don't start async tasks until we know everything is healthy.
3852          */
3853         spa_async_suspend(spa);

3855         zpool_get_rewind_policy(config, &policy);
3856         if (policy.zrp_request & ZPOOL_DO_REWIND)
3857                 state = SPA_LOAD_RECOVER;

3859         /*
3860          * Pass off the heavy lifting to spa_load().  Pass TRUE for mosconfig
3861          * because the user-supplied config is actually the one to trust when
3862          * doing an import.
3863          */
3864         if (state != SPA_LOAD_RECOVER)
3865                 spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;

3867         error = spa_load_best(spa, state, B_TRUE, policy.zrp_txg,
3868             policy.zrp_request);

3870         /*
3871          * Propagate anything learned while loading the pool and pass it
3872          * back to caller (i.e. rewind info, missing devices, etc).
3873          */
3874         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_LOAD_INFO,
3875             spa->spa_load_info) == 0);

3877         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3878         /*
3879          * Toss any existing sparelist, as it doesn't have any validity
3880          * anymore, and conflicts with spa_has_spare().
3881          */
3882         if (spa->spa_spares.sav_config) {
3883                 nvlist_free(spa->spa_spares.sav_config);
3884                 spa->spa_spares.sav_config = NULL;
3885                 spa_load_spares(spa);
3886         }
3887         if (spa->spa_l2cache.sav_config) {
3888                 nvlist_free(spa->spa_l2cache.sav_config);
3889                 spa->spa_l2cache.sav_config = NULL;
3890                 spa_load_l2cache(spa);
3891         }

3893         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3894             &nvroot) == 0);
3895         if (error == 0)
3896                 error = spa_validate_aux(spa, nvroot, -1ULL,
3897                     VDEV_ALLOC_SPARE);
3898         if (error == 0)
3899                 error = spa_validate_aux(spa, nvroot, -1ULL,
3900                     VDEV_ALLOC_L2CACHE);
3901         spa_config_exit(spa, SCL_ALL, FTAG);

3903         if (props != NULL)
3904                 spa_configfile_set(spa, props, B_FALSE);

3906         if (error != 0 || (props && spa_writeable(spa) &&
3907             (error = spa_prop_set(spa, props)))) {
3908                 spa_unload(spa);
3909                 spa_deactivate(spa);
3910                 spa_remove(spa);
3911                 mutex_exit(&spa_namespace_lock);
```

```
3912                 return (error);
3913         }

3915         spa_async_resume(spa);

3917         /*
3918          * Override any spares and level 2 cache devices as specified by
3919          * the user, as these may have correct device names/devids, etc.
3920          */
3921         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3922             &spares, &nspares) == 0) {
3923                 if (spa->spa_spares.sav_config)
3924                         VERIFY(nvlist_remove(spa->spa_spares.sav_config,
3925                             ZPOOL_CONFIG_SPARES, DATA_TYPE_NVLIST_ARRAY) == 0);
3926                 else
3927                         VERIFY(nvlist_alloc(&spa->spa_spares.sav_config,
3928                             NV_UNIQUE_NAME, KM_SLEEP) == 0);
3929                 VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
3930                     ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3931                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3932                 spa_load_spares(spa);
3933                 spa_config_exit(spa, SCL_ALL, FTAG);
3934                 spa->spa_spares.sav_sync = B_TRUE;
3935         }
3936         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3937             &l2cache, &nl2cache) == 0) {
3938                 if (spa->spa_l2cache.sav_config)
3939                         VERIFY(nvlist_remove(spa->spa_l2cache.sav_config,
3940                             ZPOOL_CONFIG_L2CACHE, DATA_TYPE_NVLIST_ARRAY) == 0);
3941                 else
3942                         VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3943                             NV_UNIQUE_NAME, KM_SLEEP) == 0);
3944                 VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3945                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3946                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3947                 spa_load_l2cache(spa);
3948                 spa_config_exit(spa, SCL_ALL, FTAG);
3949                 spa->spa_l2cache.sav_sync = B_TRUE;
3950         }

3952         /*
3953          * Check for any removed devices.
3954          */
3955         if (spa->spa_autoreplace) {
3956                 spa_aux_check_removed(&spa->spa_spares);
3957                 spa_aux_check_removed(&spa->spa_l2cache);
3958         }

3960         if (spa_writeable(spa)) {
3961                 /*
3962                  * Update the config cache to include the newly-imported pool.
3963                  */
3964                 spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
3965         }

3967         /*
3968          * It's possible that the pool was expanded while it was exported.
3969          * We kick off an async task to handle this for us.
3970          */
3971         spa_async_request(spa, SPA_ASYNC_AUTOEXPAND);

3973         mutex_exit(&spa_namespace_lock);
3974         spa_history_log_version(spa, "import");

3976         return (0);
3977 }
```

_____***unchanged_portion_omitted_***