```
**********************************************************
   76620 Thu Aug  1 22:44:30 2013
new/usr/src/cmd/mdb/common/modules/zfs/zfs.c
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1115 /*
1116  * ::vdev
1117  *
1118  * Print out a summarized vdev_t, in the following form:
1119  *
1120  * ADDR             STATE       AUX            DESC
1121  * ffffffffbcde23df0 HEALTHY     -              /dev/dsk/c0t0d0
1122  *
1123  * If '-r' is specified, recursively visit all children.
1124  *
1125  * With '-e', the statistics associated with the vdev are printed as well.
1126  */
1127 static int
1128 do_print_vdev(uintptr_t addr, int flags, int depth, int stats,
1129     int recursive)
1130 {
1131         vdev_t vdev;
1132         char desc[MAXNAMELEN];
1133         int c, children;
1134         uintptr_t *child;
1135         const char *state, *aux;

1137         if (mdb_vread(&vdev, sizeof (vdev), (uintptr_t)addr) == -1) {
1138                 mdb_warn("failed to read vdev_t at %p\n", (uintptr_t)addr);
1139                 return (DCMD_ERR);
1140         }

1142         if (flags & DCMD_PIPE_OUT) {
1143                 mdb_printf("%#lr\n", addr);
1143                 mdb_printf("%#lr", addr);
1144         } else {
1145                 if (vdev.vdev_path != NULL) {
1146                         if (mdb_readstr(desc, sizeof (desc),
1147                             (uintptr_t)vdev.vdev_path) == -1) {
1148                                 mdb_warn("failed to read vdev_path at %p\n",
1149                                     vdev.vdev_path);
1150                                 return (DCMD_ERR);
1151                         }
1152                 } else if (vdev.vdev_ops != NULL) {
1153                         vdev_ops_t ops;
1154                         if (mdb_vread(&ops, sizeof (ops),
1155                             (uintptr_t)vdev.vdev_ops) == -1) {
1156                                 mdb_warn("failed to read vdev_ops at %p\n",
1157                                     vdev.vdev_ops);
1158                                 return (DCMD_ERR);
1159                         }
1160                         (void) strcpy(desc, ops.vdev_op_type);
1161                 } else {
1162                         (void) strcpy(desc, "<unknown>");
1163                 }
```



```
1165                 if (depth == 0 && DCMD_HDRSPEC(flags))
1166                         mdb_printf("%<u>%-?s %-9s %-12s %-*s%</u>\n",
1167                             "ADDR", "STATE", "AUX",
1168                             sizeof (uintptr_t) == 4 ? 43 : 35,
1169                             "DESCRIPTION");

1171                 mdb_printf("%0?p ", addr);

1173                 switch (vdev.vdev_state) {
1174                 case VDEV_STATE_CLOSED:
1175                         state = "CLOSED";
1176                         break;
1177                 case VDEV_STATE_OFFLINE:
1178                         state = "OFFLINE";
1179                         break;
1180                 case VDEV_STATE_CANT_OPEN:
1181                         state = "CANT_OPEN";
1182                         break;
1183                 case VDEV_STATE_DEGRADED:
1184                         state = "DEGRADED";
1185                         break;
1186                 case VDEV_STATE_HEALTHY:
1187                         state = "HEALTHY";
1188                         break;
1189                 case VDEV_STATE_REMOVED:
1190                         state = "REMOVED";
1191                         break;
1192                 case VDEV_STATE_FAULTED:
1193                         state = "FAULTED";
1194                         break;
1195                 default:
1196                         state = "UNKNOWN";
1197                         break;
1198                 }

1200                 switch (vdev.vdev_stat.vs_aux) {
1201                 case VDEV_AUX_NONE:
1202                         aux = "-";
1203                         break;
1204                 case VDEV_AUX_OPEN_FAILED:
1205                         aux = "OPEN_FAILED";
1206                         break;
1207                 case VDEV_AUX_CORRUPT_DATA:
1208                         aux = "CORRUPT_DATA";
1209                         break;
1210                 case VDEV_AUX_NO_REPLICAS:
1211                         aux = "NO_REPLICAS";
1212                         break;
1213                 case VDEV_AUX_BAD_GUID_SUM:
1214                         aux = "BAD_GUID_SUM";
1215                         break;
1216                 case VDEV_AUX_TOO_SMALL:
1217                         aux = "TOO_SMALL";
1218                         break;
1219                 case VDEV_AUX_BAD_LABEL:
1220                         aux = "BAD_LABEL";
1221                         break;
1222                 case VDEV_AUX_VERSION_NEWER:
1223                         aux = "VERS_NEWER";
1224                         break;
1225                 case VDEV_AUX_VERSION_OLDER:
1226                         aux = "VERS_OLDER";
1227                         break;
1228                 case VDEV_AUX_UNSUP_FEAT:
1229                         aux = "UNSUP_FEAT";
1230                         break;
```

```
1231                 case VDEV_AUX_SPARED:
1232                         aux = "SPARED";
1233                         break;
1234                 case VDEV_AUX_ERR_EXCEEDED:
1235                         aux = "ERR_EXCEEDED";
1236                         break;
1237                 case VDEV_AUX_IO_FAILURE:
1238                         aux = "IO_FAILURE";
1239                         break;
1240                 case VDEV_AUX_BAD_LOG:
1241                         aux = "BAD_LOG";
1242                         break;
1243                 case VDEV_AUX_EXTERNAL:
1244                         aux = "EXTERNAL";
1245                         break;
1246                 case VDEV_AUX_SPLIT_POOL:
1247                         aux = "SPLIT_POOL";
1248                         break;
1249                 default:
1250                         aux = "UNKNOWN";
1251                         break;
1252                 }

1254                 mdb_printf("%-9s %-12s %*s%s\n", state, aux, depth, "", desc);

1256                 if (stats) {
1257                         vdev_stat_t *vs = &vdev.vdev_stat;
1258                         int i;

1260                         mdb_inc_indent(4);
1261                         mdb_printf("\n");
1262                         mdb_printf("%<u>       %12s %12s %12s %12s "
1263                             "%12s%</u>\n", "READ", "WRITE", "FREE", "CLAIM",
1264                             "IOCTL");
1265                         mdb_printf("OPS     ");
1266                         for (i = 1; i < ZIO_TYPES; i++)
1267                                 mdb_printf("%11#llx%s", vs->vs_ops[i],
1268                                     i == ZIO_TYPES - 1 ? "" : "  ");
1269                         mdb_printf("\n");
1270                         mdb_printf("BYTES   ");
1271                         for (i = 1; i < ZIO_TYPES; i++)
1272                                 mdb_printf("%11#llx%s", vs->vs_bytes[i],
1273                                     i == ZIO_TYPES - 1 ? "" : "  ");


1276                         mdb_printf("\n");
1277                         mdb_printf("EREAD    %10#llx\n", vs->vs_read_errors);
1278                         mdb_printf("EWRITE   %10#llx\n", vs->vs_write_errors);
1279                         mdb_printf("ECKSUM   %10#llx\n",
1280                             vs->vs_checksum_errors);
1281                         mdb_dec_indent(4);
1282                 }

1284                 if (stats)
1285                         mdb_printf("\n");
1286         }

1288         children = vdev.vdev_children;

1290         if (children == 0 || !recursive)
1291                 return (DCMD_OK);

1293         child = mdb_alloc(children * sizeof (void *), UM_SLEEP | UM_GC);
1294         if (mdb_vread(child, children * sizeof (void *),
1295             (uintptr_t)vdev.vdev_child) == -1) {
1296                 mdb_warn("failed to read vdev children at %p", vdev.vdev_child);
```



```
1297                         return (DCMD_ERR);
1298                 }

1300                 for (c = 0; c < children; c++) {
1301                         if (do_print_vdev(child[c], flags, depth + 2, stats,
1302                             recursive))
1303                                 return (DCMD_ERR);
1304                 }

1306                 return (DCMD_OK);
1307 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  160647 Thu Aug  1 22:44:32 2013
new/usr/src/cmd/ztest/ztest.c
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@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 (c) 2012 by Delphix. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 */

/*
 * The objective of this program is to provide a DMU/ZAP/SPA stress test
 * that runs entirely in userland, is easy to use, and easy to extend.
 *
 * The overall design of the ztest program is as follows:
 *
 * (1) For each major functional area (e.g. adding vdevs to a pool,
 *     creating and destroying datasets, reading and writing objects, etc)
 *     we have a simple routine to test that functionality.  These
 *     individual routines do not have to do anything "stressful".
 *
 * (2) We turn these simple functionality tests into a stress test by
 *     running them all in parallel, with as many threads as desired,
 *     and spread across as many datasets, objects, and vdevs as desired.
 *
 * (3) While all this is happening, we inject faults into the pool to
 *     verify that self-healing data really works.
 *
 * (4) Every time we open a dataset, we change its checksum and compression
 *     functions.  Thus even individual objects vary from block to block
 *     in which checksum they use and whether they're compressed.
 *
 * (5) To verify that we never lose on-disk consistency after a crash,
 *     we run the entire test in a child of the main process.
 *     At random times, the child self-immolates with a SIGKILL.
```



```
 *     This is the software equivalent of pulling the power cord.
 *     The parent then runs the test again, using the existing
 *     storage pool, as many times as desired. If backwards compatability
 *     testing is enabled ztest will sometimes run the "older" version
 *     of ztest after a SIGKILL.
 *
 * (6) To verify that we don't have future leaks or temporal incursions,
 *     many of the functional tests record the transaction group number
 *     as part of their data.  When reading old data, they verify that
 *     the transaction group number is less than the current, open txg.
 *     If you add a new test, please do this if applicable.
 *
 * When run with no arguments, ztest runs for about five minutes and
 * produces no output if successful.  To get a little bit of information,
 * specify -V.  To get more information, specify -VV, and so on.
 *
 * To turn this into an overnight stress test, use -T to specify run time.
 *
 * You can ask more more vdevs [-v], datasets [-d], or threads [-t]
 * to increase the pool capacity, fanout, and overall stress level.
 *
 * Use the -k option to set the desired frequency of kills.
 *
 * When ztest invokes itself it passes all relevant information through a
 * temporary file which is mmap-ed in the child process. This allows shared
 * memory to survive the exec syscall. The ztest_shared_hdr_t struct is always
 * stored at offset 0 of this file and contains information on the size and
 * number of shared structures in the file. The information stored in this file
 * must remain backwards compatible with older versions of ztest so that
 * ztest can invoke them during backwards compatibility testing (-B).
 */

#include <sys/zfs_context.h>
#include <sys/spa.h>
#include <sys/dmu.h>
#include <sys/txg.h>
#include <sys/dbuf.h>
#include <sys/zap.h>
#include <sys/dmu_objset.h>
#include <sys/poll.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <sys/resource.h>
#include <sys/zio.h>
#include <sys/zil.h>
#include <sys/zil_impl.h>
#include <sys/vdev_impl.h>
#include <sys/vdev_file.h>
#include <sys/spa_impl.h>
#include <sys/metaslab_impl.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_destroy.h>
#include <sys/dsl_scan.h>
#include <sys/zio_checksum.h>
#include <sys/refcount.h>
#include <sys/zfeature.h>
#include <sys/dsl_userhold.h>
#include <stdio.h>
#include <stdio_ext.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <umem.h>
```

```
 119 #include <dlfcn.h>
 120 #include <ctype.h>
 121 #include <math.h>
 122 #include <sys/fs/zfs.h>
 123 #include <libnvpair.h>

 125 static int ztest_fd_data = -1;
 126 static int ztest_fd_rand = -1;

 128 typedef struct ztest_shared_hdr {
 129         uint64_t        zh_hdr_size;
 130         uint64_t        zh_opts_size;
 131         uint64_t        zh_size;
 132         uint64_t        zh_stats_size;
 133         uint64_t        zh_stats_count;
 134         uint64_t        zh_ds_size;
 135         uint64_t        zh_ds_count;
 136 } ztest_shared_hdr_t;
_____unchanged_portion_omitted_

 765 static void
 766 ztest_kill(ztest_shared_t *zs)
 767 {
 768         zs->zs_alloc = metaslab_class_get_alloc(spa_normal_class(ztest_spa));
 769         zs->zs_space = metaslab_class_get_space(spa_normal_class(ztest_spa));

 771         /*
 772          * Before we kill off ztest, make sure that the config is updated.
 773          * See comment above spa_config_sync().
 774          */
 775         mutex_enter(&spa_namespace_lock);
 776         spa_config_sync(ztest_spa, B_FALSE, B_FALSE);
 777         mutex_exit(&spa_namespace_lock);

 779         zfs_dbgmsg_print(FTAG);
 780         (void) kill(getpid(), SIGKILL);
 781 }
_____unchanged_portion_omitted_

2727 /*
2728  * Verify that we can attach and detach devices.
2729  */
2730 /* ARGSUSED */
2731 void
2732 ztest_vdev_attach_detach(ztest_ds_t *zd, uint64_t id)
2733 {
2734         ztest_shared_t *zs = ztest_shared;
2735         spa_t *spa = ztest_spa;
2736         spa_aux_vdev_t *sav = &spa->spa_spares;
2737         vdev_t *rvd = spa->spa_root_vdev;
2738         vdev_t *oldvd, *newvd, *pvd;
2739         nvlist_t *root;
2740         uint64_t leaves;
2741         uint64_t leaf, top;
2742         uint64_t ashift = ztest_get_ashift();
2743         uint64_t oldguid, pguid;
2744         uint64_t oldsize, newsize;
2734         size_t oldsize, newsize;
2745         char oldpath[MAXPATHLEN], newpath[MAXPATHLEN];
2746         int replacing;
2747         int oldvd_has_siblings = B_FALSE;
2748         int newvd_is_spare = B_FALSE;
2749         int oldvd_is_log;
2750         int error, expected_error;

2752         VERIFY(mutex_lock(&ztest_vdev_lock) == 0);
```



```
2753         leaves = MAX(zs->zs_mirrors, 1) * ztest_opts.zo_raidz;

2755         spa_config_enter(spa, SCL_VDEV, FTAG, RW_READER);

2757         /*
2758          * Decide whether to do an attach or a replace.
2759          */
2760         replacing = ztest_random(2);

2762         /*
2763          * Pick a random top-level vdev.
2764          */
2765         top = ztest_random_vdev_top(spa, B_TRUE);

2767         /*
2768          * Pick a random leaf within it.
2769          */
2770         leaf = ztest_random(leaves);

2772         /*
2773          * Locate this vdev.
2774          */
2775         oldvd = rvd->vdev_child[top];
2776         if (zs->zs_mirrors >= 1) {
2777                 ASSERT(oldvd->vdev_ops == &vdev_mirror_ops);
2778                 ASSERT(oldvd->vdev_children >= zs->zs_mirrors);
2779                 oldvd = oldvd->vdev_child[leaf / ztest_opts.zo_raidz];
2780         }
2781         if (ztest_opts.zo_raidz > 1) {
2782                 ASSERT(oldvd->vdev_ops == &vdev_raidz_ops);
2783                 ASSERT(oldvd->vdev_children == ztest_opts.zo_raidz);
2784                 oldvd = oldvd->vdev_child[leaf % ztest_opts.zo_raidz];
2785         }

2787         /*
2788          * If we're already doing an attach or replace, oldvd may be a
2789          * mirror vdev -- in which case, pick a random child.
2790          */
2791         while (oldvd->vdev_children != 0) {
2792                 oldvd_has_siblings = B_TRUE;
2793                 ASSERT(oldvd->vdev_children >= 2);
2794                 oldvd = oldvd->vdev_child[ztest_random(oldvd->vdev_children)];
2795         }

2797         oldguid = oldvd->vdev_guid;
2798         oldsize = vdev_get_min_asize(oldvd);
2799         oldvd_is_log = oldvd->vdev_top->vdev_islog;
2800         (void) strcpy(oldpath, oldvd->vdev_path);
2801         pvd = oldvd->vdev_parent;
2802         pguid = pvd->vdev_guid;

2804         /*
2805          * If oldvd has siblings, then half of the time, detach it.
2806          */
2807         if (oldvd_has_siblings && ztest_random(2) == 0) {
2808                 spa_config_exit(spa, SCL_VDEV, FTAG);
2809                 error = spa_vdev_detach(spa, oldguid, pguid, B_FALSE);
2810                 if (error != 0 && error != ENODEV && error != EBUSY &&
2811                     error != ENOTSUP)
2812                         fatal(0, "detach (%s) returned %d", oldpath, error);
2813                 VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);
2814                 return;
2815         }

2817         /*
2818          * For the new vdev, choose with equal probability between the two
```

```
2819          * standard paths (ending in either 'a' or 'b') or a random hot spare.
2820          */
2821         if (sav->sav_count != 0 && ztest_random(3) == 0) {
2822                 newvd = sav->sav_vdevs[ztest_random(sav->sav_count)];
2823                 newvd_is_spare = B_TRUE;
2824                 (void) strcpy(newpath, newvd->vdev_path);
2825         } else {
2826                 (void) snprintf(newpath, sizeof (newpath), ztest_dev_template,
2827                     ztest_opts.zo_dir, ztest_opts.zo_pool,
2828                     top * leaves + leaf);
2829                 if (ztest_random(2) == 0)
2830                         newpath[strlen(newpath) - 1] = 'b';
2831                 newvd = vdev_lookup_by_path(rvd, newpath);
2832         }

2834         if (newvd) {
2835                 newsize = vdev_get_min_asize(newvd);
2836         } else {
2837                 /*
2838                  * Make newsize a little bigger or smaller than oldsize.
2839                  * If it's smaller, the attach should fail.
2840                  * If it's larger, and we're doing a replace,
2841                  * we should get dynamic LUN growth when we're done.
2842                  */
2843                 newsize = 10 * oldsize / (9 + ztest_random(3));
2844         }

2846         /*
2847          * If pvd is not a mirror or root, the attach should fail with ENOTSUP,
2848          * unless it's a replace; in that case any non-replacing parent is OK.
2849          *
2850          * If newvd is already part of the pool, it should fail with EBUSY.
2851          *
2852          * If newvd is too small, it should fail with EOVERFLOW.
2853          */
2854         if (pvd->vdev_ops != &vdev_mirror_ops &&
2855             pvd->vdev_ops != &vdev_root_ops && (!replacing ||
2856             pvd->vdev_ops == &vdev_replacing_ops ||
2857             pvd->vdev_ops == &vdev_spare_ops))
2858                 expected_error = ENOTSUP;
2859         else if (newvd_is_spare && (!replacing || oldvd_is_log))
2860                 expected_error = ENOTSUP;
2861         else if (newvd == oldvd)
2862                 expected_error = replacing ? 0 : EBUSY;
2863         else if (vdev_lookup_by_path(rvd, newpath) != NULL)
2864                 expected_error = EBUSY;
2865         else if (newsize < oldsize)
2866                 expected_error = EOVERFLOW;
2867         else if (ashift > oldvd->vdev_top->vdev_ashift)
2868                 expected_error = EDOM;
2869         else
2870                 expected_error = 0;

2872         spa_config_exit(spa, SCL_VDEV, FTAG);

2874         /*
2875          * Build the nvlist describing newpath.
2876          */
2877         root = make_vdev_root(newpath, NULL, NULL, newvd == NULL ? newsize : 0,
2878             ashift, 0, 0, 0, 1);

2880         error = spa_vdev_attach(spa, oldguid, root, replacing);

2882         nvlist_free(root);

2884         /*
```



```
2885          * If our parent was the replacing vdev, but the replace completed,
2886          * then instead of failing with ENOTSUP we may either succeed,
2887          * fail with ENODEV, or fail with EOVERFLOW.
2888          */
2889         if (expected_error == ENOTSUP &&
2890             (error == 0 || error == ENODEV || error == EOVERFLOW))
2891                 expected_error = error;

2893         /*
2894          * If someone grew the LUN, the replacement may be too small.
2895          */
2896         if (error == EOVERFLOW || error == EBUSY)
2897                 expected_error = error;

2899         /* XXX workaround 6690467 */
2900         if (error != expected_error && expected_error != EBUSY) {
2901                 fatal(0, "attach (%s %llu, %s %llu, %d) "
2902                     "returned %d, expected %d",
2903                     oldpath, oldsize, newpath,
2904                     newsize, replacing, error, expected_error);
2893                     oldpath, (longlong_t)oldsize, newpath,
2894                     (longlong_t)newsize, replacing, error, expected_error);
2905         }

2907         VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);
2908 }
```

*unchanged_portion_omitted_*

```
4737 /*
4738  * Inject random faults into the on-disk data.
4739  */
4740 /* ARGSUSED */
4741 void
4742 ztest_fault_inject(ztest_ds_t *zd, uint64_t id)
4743 {
4744         ztest_shared_t *zs = ztest_shared;
4745         spa_t *spa = ztest_spa;
4746         int fd;
4747         uint64_t offset;
4748         uint64_t leaves;
4749         uint64_t bad = 0x1990c0ffeedecade;
4750         uint64_t top, leaf;
4751         char path0[MAXPATHLEN];
4752         char pathrand[MAXPATHLEN];
4753         size_t fsize;
4754         int bshift = SPA_MAXBLOCKSHIFT + 2;     /* don't scrog all labels */
4755         int iters = 1000;
4756         int maxfaults;
4757         int mirror_save;
4758         vdev_t *vd0 = NULL;
4759         uint64_t guid0 = 0;
4760         boolean_t islog = B_FALSE;

4762         VERIFY(mutex_lock(&ztest_vdev_lock) == 0);
4763         maxfaults = MAXFAULTS();
4764         leaves = MAX(zs->zs_mirrors, 1) * ztest_opts.zo_raidz;
4765         mirror_save = zs->zs_mirrors;
4766         VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);

4768         ASSERT(leaves >= 1);

4770         /*
4771          * Grab the name lock as reader. There are some operations
4772          * which don't like to have their vdevs changed while
4773          * they are in progress (i.e. spa_change_guid). Those
4774          * operations will have grabbed the name lock as writer.
```

```
4775          */
4776         (void) rw_rdlock(&ztest_name_lock);
4777
4778         /*
4779          * We need SCL_STATE here because we're going to look at vd0->vdev_tsd.
4780          */
4781         spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
4782
4783         if (ztest_random(2) == 0) {
4784                 /*
4785                  * Inject errors on a normal data device or slog device.
4786                  */
4787                 top = ztest_random_vdev_top(spa, B_TRUE);
4788                 leaf = ztest_random(leaves) + zs->zs_splits;
4789
4790                 /*
4791                  * Generate paths to the first leaf in this top-level vdev,
4792                  * and to the random leaf we selected.  We'll induce transient
4793                  * write failures and random online/offline activity on leaf 0,
4794                  * and we'll write random garbage to the randomly chosen leaf.
4795                  */
4796                 (void) snprintf(path0, sizeof (path0), ztest_dev_template,
4797                     ztest_opts.zo_dir, ztest_opts.zo_pool,
4798                     top * leaves + zs->zs_splits);
4799                 (void) snprintf(pathrand, sizeof (pathrand), ztest_dev_template,
4800                     ztest_opts.zo_dir, ztest_opts.zo_pool,
4801                     top * leaves + leaf);
4802
4803                 vd0 = vdev_lookup_by_path(spa->spa_root_vdev, path0);
4804                 if (vd0 != NULL && vd0->vdev_top->vdev_islog)
4805                         islog = B_TRUE;
4806
4807                 /*
4808                  * If the top-level vdev needs to be resilvered
4809                  * then we only allow faults on the device that is
4810                  * resilvering.
4811                  */
4812                 if (vd0 != NULL && maxfaults != 1 &&
4813                     (!vdev_resilver_needed(vd0->vdev_top, NULL, NULL) ||
4814                     vd0->vdev_resilver_txg != 0)) {
4815                     vd0->vdev_resilvering)) {
4816                         /*
4817                          * Make vd0 explicitly claim to be unreadable,
4818                          * or unwriteable, or reach behind its back
4819                          * and close the underlying fd.  We can do this if
4820                          * maxfaults == 0 because we'll fail and reexecute,
4821                          * and we can do it if maxfaults >= 2 because we'll
4822                          * have enough redundancy.  If maxfaults == 1, the
4823                          * combination of this with injection of random data
4824                          * corruption below exceeds the pool's fault tolerance.
4825                          */
4826                         vdev_file_t *vf = vd0->vdev_tsd;
4827
4828                         if (vf != NULL && ztest_random(3) == 0) {
4829                                 (void) close(vf->vf_vnode->v_fd);
4830                                 vf->vf_vnode->v_fd = -1;
4831                         } else if (ztest_random(2) == 0) {
4832                                 vd0->vdev_cant_read = B_TRUE;
4833                         } else {
4834                                 vd0->vdev_cant_write = B_TRUE;
4835                         }
4836                         guid0 = vd0->vdev_guid;
4837                 }
4838         } else {
4839                 /*
4840                  * Inject errors on an l2cache device.
```



```
4841                  */
4842                 spa_aux_vdev_t *sav = &spa->spa_l2cache;
4843
4844                 if (sav->sav_count == 0) {
4845                         spa_config_exit(spa, SCL_STATE, FTAG);
4846                         (void) rw_unlock(&ztest_name_lock);
4847                         return;
4848                 }
4849                 vd0 = sav->sav_vdevs[ztest_random(sav->sav_count)];
4850                 guid0 = vd0->vdev_guid;
4851                 (void) strcpy(path0, vd0->vdev_path);
4852                 (void) strcpy(pathrand, vd0->vdev_path);
4853
4854                 leaf = 0;
4855                 leaves = 1;
4856                 maxfaults = INT_MAX;    /* no limit on cache devices */
4857         }
4858
4859         spa_config_exit(spa, SCL_STATE, FTAG);
4860         (void) rw_unlock(&ztest_name_lock);
4861
4862         /*
4863          * If we can tolerate two or more faults, or we're dealing
4864          * with a slog, randomly online/offline vd0.
4865          */
4866         if ((maxfaults >= 2 || islog) && guid0 != 0) {
4867                 if (ztest_random(10) < 6) {
4868                         int flags = (ztest_random(2) == 0 ?
4869                             ZFS_OFFLINE_TEMPORARY : 0);
4870
4871                         /*
4872                          * We have to grab the zs_name_lock as writer to
4873                          * prevent a race between offlining a slog and
4874                          * destroying a dataset. Offlining the slog will
4875                          * grab a reference on the dataset which may cause
4876                          * dmu_objset_destroy() to fail with EBUSY thus
4877                          * leaving the dataset in an inconsistent state.
4878                          */
4879                         if (islog)
4880                                 (void) rw_wrlock(&ztest_name_lock);
4881
4882                         VERIFY(vdev_offline(spa, guid0, flags) != EBUSY);
4883
4884                         if (islog)
4885                                 (void) rw_unlock(&ztest_name_lock);
4886                 } else {
4887                         /*
4888                          * Ideally we would like to be able to randomly
4889                          * call vdev_[on|off]line without holding locks
4890                          * to force unpredictable failures but the side
4891                          * effects of vdev_[on|off]line prevent us from
4892                          * doing so. We grab the ztest_vdev_lock here to
4893                          * prevent a race between injection testing and
4894                          * aux_vdev removal.
4895                          */
4896                         VERIFY(mutex_lock(&ztest_vdev_lock) == 0);
4897                         (void) vdev_online(spa, guid0, 0, NULL);
4898                         VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);
4899                 }
4900         }
4901
4902         if (maxfaults == 0)
4903                 return;
4904
4905         /*
4906          * We have at least single-fault tolerance, so inject data corruption.
```

```
4906          */
4907         fd = open(pathrand, O_RDWR);

4909         if (fd == -1)   /* we hit a gap in the device namespace */
4910                 return;

4912         fsize = lseek(fd, 0, SEEK_END);

4914         while (--iters != 0) {
4915                 offset = ztest_random(fsize / (leaves << bshift)) *
4916                     (leaves << bshift) + (leaf << bshift) +
4917                     (ztest_random(1ULL << (bshift - 1)) & -8ULL);

4919                 if (offset >= fsize)
4920                         continue;

4922                 VERIFY(mutex_lock(&ztest_vdev_lock) == 0);
4923                 if (mirror_save != zs->zs_mirrors) {
4924                         VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);
4925                         (void) close(fd);
4926                         return;
4927                 }

4929                 if (pwrite(fd, &bad, sizeof (bad), offset) != sizeof (bad))
4930                         fatal(1, "can't inject bad word at 0x%llx in %s",
4931                             offset, pathrand);

4933                 VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);

4935                 if (ztest_opts.zo_verbose >= 7)
4936                         (void) printf("injected bad word into %s,"
4937                             " offset 0x%llx\n", pathrand, (u_longlong_t)offset);
4938         }

4940         (void) close(fd);
4941 }
```

*unchanged_portion_omitted_*

```
5537 /*
5538  * Kick off threads to run tests on all datasets in parallel.
5539  */
5540 static void
5541 ztest_run(ztest_shared_t *zs)
5542 {
5543         thread_t *tid;
5544         spa_t *spa;
5545         objset_t *os;
5546         thread_t resume_tid;
5547         int error;

5549         ztest_exiting = B_FALSE;

5551         /*
5552          * Initialize parent/child shared state.
5553          */
5554         VERIFY(_mutex_init(&ztest_vdev_lock, USYNC_THREAD, NULL) == 0);
5555         VERIFY(rwlock_init(&ztest_name_lock, USYNC_THREAD, NULL) == 0);

5557         zs->zs_thread_start = gethrtime();
5558         zs->zs_thread_stop =
5559             zs->zs_thread_start + ztest_opts.zo_passtime * NANOSEC;
5560         zs->zs_thread_stop = MIN(zs->zs_thread_stop, zs->zs_proc_stop);
5561         zs->zs_thread_kill = zs->zs_thread_stop;
5562         if (ztest_random(100) < ztest_opts.zo_killrate) {
5563                 zs->zs_thread_kill -=
5564                     ztest_random(ztest_opts.zo_passtime * NANOSEC);
```



```
5565         }

5567         (void) _mutex_init(&zcl.zcl_callbacks_lock, USYNC_THREAD, NULL);

5569         list_create(&zcl.zcl_callbacks, sizeof (ztest_cb_data_t),
5570             offsetof(ztest_cb_data_t, zcd_node));

5572         /*
5573          * Open our pool.
5574          */
5575         kernel_init(FREAD | FWRITE);
5576         VERIFY0(spa_open(ztest_opts.zo_pool, &spa, FTAG));
5577         spa->spa_debug = B_TRUE;
5578         ztest_spa = spa;

5580         VERIFY0(dmu_objset_own(ztest_opts.zo_pool,
5581             DMU_OST_ANY, B_TRUE, FTAG, &os));
5582         zs->zs_guid = dmu_objset_fsid_guid(os);
5583         dmu_objset_disown(os, FTAG);

5585         spa->spa_dedup_ditto = 2 * ZIO_DEDUPDITTO_MIN;

5587         /*
5588          * We don't expect the pool to suspend unless maxfaults == 0,
5589          * in which case ztest_fault_inject() temporarily takes away
5590          * the only valid replica.
5591          */
5592         if (MAXFAULTS() == 0)
5593                 spa->spa_failmode = ZIO_FAILURE_MODE_WAIT;
5594         else
5595                 spa->spa_failmode = ZIO_FAILURE_MODE_PANIC;

5597         /*
5598          * Create a thread to periodically resume suspended I/O.
5599          */
5600         VERIFY(thr_create(0, 0, ztest_resume_thread, spa, THR_BOUND,
5601             &resume_tid) == 0);

5603         /*
5604          * Create a deadman thread to abort() if we hang.
5605          */
5606         VERIFY(thr_create(0, 0, ztest_deadman_thread, zs, THR_BOUND,
5607             NULL) == 0);

5609         /*
5610          * Verify that we can safely inquire about about any object,
5611          * whether it's allocated or not.  To make it interesting,
5612          * we probe a 5-wide window around each power of two.
5613          * This hits all edge cases, including zero and the max.
5614          */
5615         for (int t = 0; t < 64; t++) {
5616                 for (int d = -5; d <= 5; d++) {
5617                         error = dmu_object_info(spa->spa_meta_objset,
5618                             (1ULL << t) + d, NULL);
5619                         ASSERT(error == 0 || error == ENOENT ||
5620                             error == EINVAL);
5621                 }
5622         }

5624         /*
5625          * If we got any ENOSPC errors on the previous run, destroy something.
5626          */
5627         if (zs->zs_enospc_count != 0) {
5628                 int d = ztest_random(ztest_opts.zo_datasets);
5629                 ztest_dataset_destroy(d);
5630         }
```

```
5631         zs->zs_enospc_count = 0;

5633         tid = umem_zalloc(ztest_opts.zo_threads * sizeof (thread_t),
5634             UMEM_NOFAIL);

5636         if (ztest_opts.zo_verbose >= 4)
5637                 (void) printf("starting main threads...\n");

5639         /*
5640          * Kick off all the tests that run in parallel.
5641          */
5642         for (int t = 0; t < ztest_opts.zo_threads; t++) {
5643                 if (t < ztest_opts.zo_datasets &&
5644                     ztest_dataset_open(t) != 0)
5645                         return;
5646                 VERIFY(thr_create(0, 0, ztest_thread, (void *)(uintptr_t)t,
5647                     THR_BOUND, &tid[t]) == 0);
5648         }

5650         /*
5651          * Wait for all of the tests to complete.  We go in reverse order
5652          * so we don't close datasets while threads are still using them.
5653          */
5654         for (int t = ztest_opts.zo_threads - 1; t >= 0; t--) {
5655                 VERIFY(thr_join(tid[t], NULL, NULL) == 0);
5656                 if (t < ztest_opts.zo_datasets)
5657                         ztest_dataset_close(t);
5658         }

5660         txg_wait_synced(spa_get_dsl(spa), 0);

5662         zs->zs_alloc = metaslab_class_get_alloc(spa_normal_class(spa));
5663         zs->zs_space = metaslab_class_get_space(spa_normal_class(spa));
5664         zfs_dbgmsg_print(FTAG);

5666         umem_free(tid, ztest_opts.zo_threads * sizeof (thread_t));

5668         /* Kill the resume thread */
5669         ztest_exiting = B_TRUE;
5670         VERIFY(thr_join(resume_tid, NULL, NULL) == 0);
5671         ztest_resume(spa);

5673         /*
5674          * Right before closing the pool, kick off a bunch of async I/O;
5675          * spa_close() should wait for it to complete.
5676          */
5677         for (uint64_t object = 1; object < 50; object++)
5678                 dmu_prefetch(spa->spa_meta_objset, object, 0, 1ULL << 20);

5680         spa_close(spa, FTAG);

5682         /*
5683          * Verify that we can loop over all pools.
5684          */
5685         mutex_enter(&spa_namespace_lock);
5686         for (spa = spa_next(NULL); spa != NULL; spa = spa_next(spa))
5687                 if (ztest_opts.zo_verbose > 3)
5688                         (void) printf("spa_next: found %s\n", spa_name(spa));
5689         mutex_exit(&spa_namespace_lock);

5691         /*
5692          * Verify that we can export the pool and reimport it under a
5693          * different name.
5694          */
5695         if (ztest_random(2) == 0) {
5696                 char name[MAXNAMELEN];
```



```
5697                 (void) snprintf(name, MAXNAMELEN, "%s_import",
5698                     ztest_opts.zo_pool);
5699                 ztest_spa_import_export(ztest_opts.zo_pool, name);
5700                 ztest_spa_import_export(name, ztest_opts.zo_pool);
5701         }

5703         kernel_fini();

5705         list_destroy(&zcl.zcl_callbacks);

5707         (void) _mutex_destroy(&zcl.zcl_callbacks_lock);

5709         (void) rwlock_destroy(&ztest_name_lock);
5710         (void) _mutex_destroy(&ztest_vdev_lock);
5711 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   50454 Thu Aug  1 22:44:34 2013
new/usr/src/uts/common/fs/zfs/dsl_scan.c
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 176 static void
 177 dsl_scan_setup_sync(void *arg, dmu_tx_t *tx)
 178 {
 179         dsl_scan_t *scn = dmu_tx_pool(tx)->dp_scan;
 180         pool_scan_func_t *funcp = arg;
 181         dmu_object_type_t ot = 0;
 182         dsl_pool_t *dp = scn->scn_dp;
 183         spa_t *spa = dp->dp_spa;

 185         ASSERT(scn->scn_phys.scn_state != DSS_SCANNING);
 186         ASSERT(*funcp > POOL_SCAN_NONE && *funcp < POOL_SCAN_FUNCS);
 187         bzero(&scn->scn_phys, sizeof (scn->scn_phys));
 188         scn->scn_phys.scn_func = *funcp;
 189         scn->scn_phys.scn_state = DSS_SCANNING;
 190         scn->scn_phys.scn_min_txg = 0;
 191         scn->scn_phys.scn_max_txg = tx->tx_txg;
 192         scn->scn_phys.scn_ddt_class_max = DDT_CLASSES - 1; /* the entire DDT */
 193         scn->scn_phys.scn_start_time = gethrestime_sec();
 194         scn->scn_phys.scn_errors = 0;
 195         scn->scn_phys.scn_to_examine = spa->spa_root_vdev->vdev_stat.vs_alloc;
 196         scn->scn_restart_txg = 0;
 197         scn->scn_done_txg = 0;
 198         spa_scan_stat_init(spa);

 200         if (DSL_SCAN_IS_SCRUB_RESILVER(scn)) {
 201                 scn->scn_phys.scn_ddt_class_max = zfs_scrub_ddt_class_max;

 203                 /* rewrite all disk labels */
 204                 vdev_config_dirty(spa->spa_root_vdev);

 206                 if (vdev_resilver_needed(spa->spa_root_vdev,
 207                     &scn->scn_phys.scn_min_txg, &scn->scn_phys.scn_max_txg)) {
 208                         spa_event_notify(spa, NULL, ESC_ZFS_RESILVER_START);
 209                 } else {
 210                         spa_event_notify(spa, NULL, ESC_ZFS_SCRUB_START);
 211                 }

 213                 spa->spa_scrub_started = B_TRUE;
 214                 /*
 215                  * If this is an incremental scrub, limit the DDT scrub phase
 216                  * to just the auto-ditto class (for correctness); the rest
 217                  * of the scrub should go faster using top-down pruning.
 218                  */
 219                 if (scn->scn_phys.scn_min_txg > TXG_INITIAL)
 220                         scn->scn_phys.scn_ddt_class_max = DDT_CLASS_DITTO;

 222         }

 224         /* back to the generic stuff */

 226         if (dp->dp_blkstats == NULL) {
```



```
 227                 dp->dp_blkstats =
 228                     kmem_alloc(sizeof (zfs_all_blkstats_t), KM_SLEEP);
 229         }
 230         bzero(dp->dp_blkstats, sizeof (zfs_all_blkstats_t));

 232         if (spa_version(spa) < SPA_VERSION_DSL_SCRUB)
 233                 ot = DMU_OT_ZAP_OTHER;

 235         scn->scn_phys.scn_queue_obj = zap_create(dp->dp_meta_objset,
 236             ot ? ot : DMU_OT_SCAN_QUEUE, DMU_OT_NONE, 0, tx);

 238         dsl_scan_sync_state(scn, tx);

 240         spa_history_log_internal(spa, "scan setup", tx,
 241             "func=%u mintxg=%llu maxtxg=%llu",
 242             *funcp, scn->scn_phys.scn_min_txg, scn->scn_phys.scn_max_txg);
 243 }
______unchanged_portion_omitted_

 715 /*
 716  * The arguments are in this order because mdb can only print the
 717  * first 5; we want them to be useful.
 718  */
 719 static void
 720 dsl_scan_visitbp(blkptr_t *bp, const zbookmark_t *zb,
 721     dnode_phys_t *dnp, arc_buf_t *pbuf,
 722     dsl_dataset_t *ds, dsl_scan_t *scn, dmu_objset_type_t ostype,
 723     dmu_tx_t *tx)
 724 {
 725         dsl_pool_t *dp = scn->scn_dp;
 726         arc_buf_t *buf = NULL;
 727         blkptr_t bp_toread = *bp;

 729         /* ASSERT(pbuf == NULL || arc_released(pbuf)); */

 731         if (dsl_scan_check_pause(scn, zb))
 732                 return;

 734         if (dsl_scan_check_resume(scn, dnp, zb))
 735                 return;

 737         if (bp->blk_birth == 0)
 738                 return;

 740         scn->scn_visited_this_txg++;

 742         dprintf_bp(bp,
 743             "visiting ds=%p/%llu zb=%llx/%llx/%llx/%llx buf=%p bp=%p",
 744             ds, ds ? ds->ds_object : 0,
 745             zb->zb_objset, zb->zb_object, zb->zb_level, zb->zb_blkid,
 746             pbuf, bp);

 748         if (bp->blk_birth <= scn->scn_phys.scn_cur_min_txg)
 749                 return;

 751         if (dsl_scan_recurse(scn, ds, ostype, dnp, &bp_toread, zb, tx,
 752             &buf) != 0)
 753                 return;

 755         /*
 756          * If dsl_scan_ddt() has aready visited this block, it will have
 757          * already done any translations or scrubbing, so don't call the
 758          * callback again.
 759          */
 760         if (ddt_class_contains(dp->dp_spa,
 761             scn->scn_phys.scn_ddt_class_max, bp)) {
```

```
 762                 ASSERT(buf == NULL);
 763                 return;
 764         }

 766         /*
 767          * If this block is from the future (after cur_max_txg), then we
 768          * are doing this on behalf of a deleted snapshot, and we will
 769          * revisit the future block on the next pass of this dataset.
 770          * Don't scan it now unless we need to because something
 771          * under it was modified.
 772          */
 773         if (BP_PHYSICAL_BIRTH(bp) <= scn->scn_phys.scn_cur_max_txg) {
 772         if (bp->blk_birth <= scn->scn_phys.scn_cur_max_txg) {
 774                 scan_funcs[scn->scn_phys.scn_func](dp, bp, zb);
 775         }
 776         if (buf)
 777                 (void) arc_buf_remove_ref(buf, &buf);
 778 }
_____unchanged_portion_omitted_

1203 /* ARGSUSED */
1204 void
1205 dsl_scan_ddt_entry(dsl_scan_t *scn, enum zio_checksum checksum,
1206     ddt_entry_t *dde, dmu_tx_t *tx)
1207 {
1208         const ddt_key_t *ddk = &dde->dde_key;
1209         ddt_phys_t *ddp = dde->dde_phys;
1210         blkptr_t bp;
1211         zbookmark_t zb = { 0 };

1213         if (scn->scn_phys.scn_state != DSS_SCANNING)
1214                 return;

1216         for (int p = 0; p < DDT_PHYS_TYPES; p++, ddp++) {
1217                 if (ddp->ddp_phys_birth == 0 ||
1218                     ddp->ddp_phys_birth > scn->scn_phys.scn_max_txg)
1217                     ddp->ddp_phys_birth > scn->scn_phys.scn_cur_max_txg)
1219                         continue;
1220                 ddt_bp_create(checksum, ddk, ddp, &bp);

1222                 scn->scn_visited_this_txg++;
1223                 scan_funcs[scn->scn_phys.scn_func](scn->scn_dp, &bp, &zb);
1224         }
1225 }
_____unchanged_portion_omitted_

1368 void
1369 dsl_scan_sync(dsl_pool_t *dp, dmu_tx_t *tx)
1370 {
1371         dsl_scan_t *scn = dp->dp_scan;
1372         spa_t *spa = dp->dp_spa;
1373         int err;

1375         /*
1376          * Check for scn_restart_txg before checking spa_load_state, so
1377          * that we can restart an old-style scan while the pool is being
1378          * imported (see dsl_scan_init).
1379          */
1380         if (scn->scn_restart_txg != 0 &&
1381             scn->scn_restart_txg <= tx->tx_txg) {
1382                 pool_scan_func_t func = POOL_SCAN_SCRUB;
1383                 dsl_scan_done(scn, B_FALSE, tx);
1384                 if (vdev_resilver_needed(spa->spa_root_vdev, NULL, NULL))
1385                         func = POOL_SCAN_RESILVER;
1386                 zfs_dbgmsg("restarting scan func=%u txg=%llu",
1387                     func, tx->tx_txg);
```



```
1388                 dsl_scan_setup_sync(&func, tx);
1389         }

1391         if (!dsl_scan_active(scn) ||
1392             spa_sync_pass(dp->dp_spa) > 1)
1393                 return;

1395         scn->scn_visited_this_txg = 0;
1396         scn->scn_pausing = B_FALSE;
1397         scn->scn_sync_start_time = gethrtime();
1398         spa->spa_scrub_active = B_TRUE;

1400         /*
1401          * First process the free list.  If we pause the free, don't do
1402          * any scanning.  This ensures that there is no free list when
1403          * we are scanning, so the scan code doesn't have to worry about
1404          * traversing it.
1405          */
1406         if (spa_version(dp->dp_spa) >= SPA_VERSION_DEADLISTS) {
1407                 scn->scn_is_bptree = B_FALSE;
1408                 scn->scn_zio_root = zio_root(dp->dp_spa, NULL,
1409                     NULL, ZIO_FLAG_MUSTSUCCEED);
1410                 err = bpobj_iterate(&dp->dp_free_bpobj,
1411                     dsl_scan_free_block_cb, scn, tx);
1412                 VERIFY3U(0, ==, zio_wait(scn->scn_zio_root));

1414                 if (err == 0 && spa_feature_is_active(spa,
1415                     &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
1416                         ASSERT(scn->scn_async_destroying);
1417                         scn->scn_is_bptree = B_TRUE;
1418                         scn->scn_zio_root = zio_root(dp->dp_spa, NULL,
1419                             NULL, ZIO_FLAG_MUSTSUCCEED);
1420                         err = bptree_iterate(dp->dp_meta_objset,
1421                             dp->dp_bptree_obj, B_TRUE, dsl_scan_free_block_cb,
1422                             scn, tx);
1423                         VERIFY0(zio_wait(scn->scn_zio_root));

1425                         if (err == 0) {
1426                                 zfeature_info_t *feat = &spa_feature_table
1427                                     [SPA_FEATURE_ASYNC_DESTROY];
1428                                 /* finished; deactivate async destroy feature */
1429                                 spa_feature_decr(spa, feat, tx);
1430                                 ASSERT(!spa_feature_is_active(spa, feat));
1431                                 VERIFY0(zap_remove(dp->dp_meta_objset,
1432                                     DMU_POOL_DIRECTORY_OBJECT,
1433                                     DMU_POOL_BPTREE_OBJ, tx));
1434                                 VERIFY0(bptree_free(dp->dp_meta_objset,
1435                                     dp->dp_bptree_obj, tx));
1436                                 dp->dp_bptree_obj = 0;
1437                                 scn->scn_async_destroying = B_FALSE;
1438                         }
1439                 }
1440                 if (scn->scn_visited_this_txg) {
1441                         zfs_dbgmsg("freed %llu blocks in %llums from "
1442                             "free_bpobj/bptree txg %llu",
1443                             (longlong_t)scn->scn_visited_this_txg,
1444                             (longlong_t)
1445                             NSEC2MSEC(gethrtime() - scn->scn_sync_start_time),
1446                             (longlong_t)tx->tx_txg);
1447                         scn->scn_visited_this_txg = 0;
1448                         /*
1449                          * Re-sync the ddt so that we can further modify
1450                          * it when doing bprewrite.
1451                          */
1452                         ddt_sync(spa, tx->tx_txg);
1453                 }
```

```
1454                 if (err == ERESTART)
1455                         return;
1456         }

1458         if (scn->scn_phys.scn_state != DSS_SCANNING)
1459                 return;

1461         if (scn->scn_done_txg == tx->tx_txg) {
1462                 ASSERT(!scn->scn_pausing);
1463                 /* finished with scan. */
1464                 zfs_dbgmsg("txg %llu scan complete", tx->tx_txg);
1465                 dsl_scan_done(scn, B_TRUE, tx);
1466                 ASSERT3U(spa->spa_scrub_inflight, ==, 0);
1467                 dsl_scan_sync_state(scn, tx);
1468                 return;
1469         }

1471         if (scn->scn_phys.scn_ddt_bookmark.ddb_class <=
1472             scn->scn_phys.scn_ddt_class_max) {
1473                 zfs_dbgmsg("doing scan sync txg %llu; "
1474                     "ddt bm=%llu/%llu/%llu/%llx",
1475                     (longlong_t)tx->tx_txg,
1476                     (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_class,
1477                     (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_type,
1478                     (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_checksum,
1479                     (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_cursor);
1480                 ASSERT(scn->scn_phys.scn_bookmark.zb_objset == 0);
1481                 ASSERT(scn->scn_phys.scn_bookmark.zb_object == 0);
1482                 ASSERT(scn->scn_phys.scn_bookmark.zb_level == 0);
1483                 ASSERT(scn->scn_phys.scn_bookmark.zb_blkid == 0);
1484         } else {
1485                 zfs_dbgmsg("doing scan sync txg %llu; bm=%llu/%llu/%llu/%llu",
1486                     (longlong_t)tx->tx_txg,
1487                     (longlong_t)scn->scn_phys.scn_bookmark.zb_objset,
1488                     (longlong_t)scn->scn_phys.scn_bookmark.zb_object,
1489                     (longlong_t)scn->scn_phys.scn_bookmark.zb_level,
1490                     (longlong_t)scn->scn_phys.scn_bookmark.zb_blkid);
1491         }

1493         scn->scn_zio_root = zio_root(dp->dp_spa, NULL,
1494             NULL, ZIO_FLAG_CANFAIL);
1495         dsl_pool_config_enter(dp, FTAG);
1496         dsl_scan_visit(scn, tx);
1497         dsl_pool_config_exit(dp, FTAG);
1498         (void) zio_wait(scn->scn_zio_root);
1499         scn->scn_zio_root = NULL;

1501         zfs_dbgmsg("visited %llu blocks in %llums",
1502             (longlong_t)scn->scn_visited_this_txg,
1503             (longlong_t)NSEC2MSEC(gethrtime() - scn->scn_sync_start_time));

1505         if (!scn->scn_pausing) {
1506                 scn->scn_done_txg = tx->tx_txg + 1;
1507                 zfs_dbgmsg("txg %llu traversal complete, waiting till txg %llu",
1508                     tx->tx_txg, scn->scn_done_txg);
1495                 /* finished with scan. */
1496                 zfs_dbgmsg("finished scan txg %llu", (longlong_t)tx->tx_txg);
1497                 dsl_scan_done(scn, B_TRUE, tx);
1509         }

1511         if (DSL_SCAN_IS_SCRUB_RESILVER(scn)) {
1512                 mutex_enter(&spa->spa_scrub_lock);
1513                 while (spa->spa_scrub_inflight > 0) {
1514                         cv_wait(&spa->spa_scrub_io_cv,
1515                             &spa->spa_scrub_lock);
1516                 }
```



```
1517                 mutex_exit(&spa->spa_scrub_lock);
1518         }

1520         dsl_scan_sync_state(scn, tx);
1521 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  175485 Thu Aug  1 22:44:35 2013
new/usr/src/uts/common/fs/zfs/spa.c
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

4318 /*
4319  * Attach a device to a mirror.  The arguments are the path to any device
4320  * in the mirror, and the nvroot for the new device.  If the path specifies
4321  * a device that is not mirrored, we automatically insert the mirror vdev.
4322  *
4323  * If 'replacing' is specified, the new device is intended to replace the
4324  * existing device; in this case the two devices are made into their own
4325  * mirror using the 'replacing' vdev, which is functionally identical to
4326  * the mirror vdev (it actually reuses all the same ops) but has a few
4327  * extra rules: you can't attach to it after it's been created, and upon
4328  * completion of resilvering, the first disk (the one being replaced)
4329  * is automatically detached.
4330  */
4331 int
4332 spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot, int replacing)
4333 {
4334         uint64_t txg, dtl_max_txg;
4335         vdev_t *rvd = spa->spa_root_vdev;
4336         vdev_t *oldvd, *newvd, *newrootvd, *pvd, *tvd;
4337         vdev_ops_t *pvops;
4338         char *oldvdpath, *newvdpath;
4339         int newvd_isspare;
4340         int error;

4342         ASSERT(spa_writeable(spa));

4344         txg = spa_vdev_enter(spa);

4346         oldvd = spa_lookup_by_guid(spa, guid, B_FALSE);

4348         if (oldvd == NULL)
4349                 return (spa_vdev_exit(spa, NULL, txg, ENODEV));

4351         if (!oldvd->vdev_ops->vdev_op_leaf)
4352                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4354         pvd = oldvd->vdev_parent;

4356         if ((error = spa_config_parse(spa, &newrootvd, nvroot, NULL, 0,
4357             VDEV_ALLOC_ATTACH)) != 0)
4358                 return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4360         if (newrootvd->vdev_children != 1)
4361                 return (spa_vdev_exit(spa, newrootvd, txg, EINVAL));

4363         newvd = newrootvd->vdev_child[0];

4365         if (!newvd->vdev_ops->vdev_op_leaf)
4366                 return (spa_vdev_exit(spa, newrootvd, txg, EINVAL));

4368         if ((error = vdev_create(newrootvd, txg, replacing)) != 0)
```



```
4369                 return (spa_vdev_exit(spa, newrootvd, txg, error));

4371         /*
4372          * Spares can't replace logs
4373          */
4374         if (oldvd->vdev_top->vdev_islog && newvd->vdev_isspare)
4375                 return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));

4377         if (!replacing) {
4378                 /*
4379                  * For attach, the only allowable parent is a mirror or the root
4380                  * vdev.
4381                  */
4382                 if (pvd->vdev_ops != &vdev_mirror_ops &&
4383                     pvd->vdev_ops != &vdev_root_ops)
4384                         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));

4386                 pvops = &vdev_mirror_ops;
4387         } else {
4388                 /*
4389                  * Active hot spares can only be replaced by inactive hot
4390                  * spares.
4391                  */
4392                 if (pvd->vdev_ops == &vdev_spare_ops &&
4393                     oldvd->vdev_isspare &&
4394                     !spa_has_spare(spa, newvd->vdev_guid))
4395                         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));

4397                 /*
4398                  * If the source is a hot spare, and the parent isn't already a
4399                  * spare, then we want to create a new hot spare.  Otherwise, we
4400                  * want to create a replacing vdev.  The user is not allowed to
4401                  * attach to a spared vdev child unless the 'isspare' state is
4402                  * the same (spare replaces spare, non-spare replaces
4403                  * non-spare).
4404                  */
4405                 if (pvd->vdev_ops == &vdev_replacing_ops &&
4406                     spa_version(spa) < SPA_VERSION_MULTI_REPLACE) {
4407                         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4408                 } else if (pvd->vdev_ops == &vdev_spare_ops &&
4409                     newvd->vdev_isspare != oldvd->vdev_isspare) {
4410                         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4411                 }

4413                 if (newvd->vdev_isspare)
4414                         pvops = &vdev_spare_ops;
4415                 else
4416                         pvops = &vdev_replacing_ops;
4417         }

4419         /*
4420          * Make sure the new device is big enough.
4421          */
4422         if (newvd->vdev_asize < vdev_get_min_asize(oldvd))
4423                 return (spa_vdev_exit(spa, newrootvd, txg, EOVERFLOW));

4425         /*
4426          * The new device cannot have a higher alignment requirement
4427          * than the top-level vdev.
4428          */
4429         if (newvd->vdev_ashift > oldvd->vdev_top->vdev_ashift)
4430                 return (spa_vdev_exit(spa, newrootvd, txg, EDOM));

4432         /*
4433          * If this is an in-place replacement, update oldvd's path and devid
4434          * to make it distinguishable from newvd, and unopenable from now on.
```

```
4435          */
4436         if (strcmp(oldvd->vdev_path, newvd->vdev_path) == 0) {
4437                 spa_strfree(oldvd->vdev_path);
4438                 oldvd->vdev_path = kmem_alloc(strlen(newvd->vdev_path) + 5,
4439                     KM_SLEEP);
4440                 (void) sprintf(oldvd->vdev_path, "%s/%s",
4441                     newvd->vdev_path, "old");
4442                 if (oldvd->vdev_devid != NULL) {
4443                         spa_strfree(oldvd->vdev_devid);
4444                         oldvd->vdev_devid = NULL;
4445                 }
4446         }

4448         /* mark the device being resilvered */
4449         newvd->vdev_resilver_txg = txg;
4449         newvd->vdev_resilvering = B_TRUE;

4451         /*
4452          * If the parent is not a mirror, or if we're replacing, insert the new
4453          * mirror/replacing/spare vdev above oldvd.
4454          */
4455         if (pvd->vdev_ops != pvops)
4456                 pvd = vdev_add_parent(oldvd, pvops);

4458         ASSERT(pvd->vdev_top->vdev_parent == rvd);
4459         ASSERT(pvd->vdev_ops == pvops);
4460         ASSERT(oldvd->vdev_parent == pvd);

4462         /*
4463          * Extract the new device from its root and add it to pvd.
4464          */
4465         vdev_remove_child(newrootvd, newvd);
4466         newvd->vdev_id = pvd->vdev_children;
4467         newvd->vdev_crtxg = oldvd->vdev_crtxg;
4468         vdev_add_child(pvd, newvd);

4470         tvd = newvd->vdev_top;
4471         ASSERT(pvd->vdev_top == tvd);
4472         ASSERT(tvd->vdev_parent == rvd);

4474         vdev_config_dirty(tvd);

4476         /*
4477          * Set newvd's DTL to [TXG_INITIAL, dtl_max_txg) so that we account
4478          * for any dmu_sync-ed blocks.  It will propagate upward when
4479          * spa_vdev_exit() calls vdev_dtl_reassess().
4480          */
4481         dtl_max_txg = txg + TXG_CONCURRENT_STATES;

4483         vdev_dtl_dirty(newvd, DTL_MISSING, TXG_INITIAL,
4484             dtl_max_txg - TXG_INITIAL);

4486         if (newvd->vdev_isspare) {
4487                 spa_spare_activate(newvd);
4488                 spa_event_notify(spa, newvd, ESC_ZFS_VDEV_SPARE);
4489         }

4491         oldvdpath = spa_strdup(oldvd->vdev_path);
4492         newvdpath = spa_strdup(newvd->vdev_path);
4493         newvd_isspare = newvd->vdev_isspare;

4495         /*
4496          * Mark newvd's DTL dirty in this txg.
4497          */
4498         vdev_dirty(tvd, VDD_DTL, newvd, txg);
```



```
4500         /*
4501          * Restart the resilver
4502          */
4503         dsl_resilver_restart(spa->spa_dsl_pool, dtl_max_txg);

4505         /*
4506          * Commit the config
4507          */
4508         (void) spa_vdev_exit(spa, newrootvd, dtl_max_txg, 0);

4510         spa_history_log_internal(spa, "vdev attach", NULL,
4511             "%s vdev=%s %s vdev=%s",
4512             replacing && newvd_isspare ? "spare in" :
4513             replacing ? "replace" : "attach", newvdpath,
4514             replacing ? "for" : "to", oldvdpath);

4516         spa_strfree(oldvdpath);
4517         spa_strfree(newvdpath);

4519         if (spa->spa_bootfs)
4520                 spa_event_notify(spa, newvd, ESC_ZFS_BOOTFS_VDEV_ATTACH);

4522         return (0);
4523 }
_____unchanged_portion_omitted_

5291 /*
5292  * Find any device that's done replacing, or a vdev marked 'unspare' that's
5293  * currently spared, so we can detach it.
5294  */
5295 static vdev_t *
5296 spa_vdev_resilver_done_hunt(vdev_t *vd)
5297 {
5298         vdev_t *newvd, *oldvd;

5300         for (int c = 0; c < vd->vdev_children; c++) {
5301                 oldvd = spa_vdev_resilver_done_hunt(vd->vdev_child[c]);
5302                 if (oldvd != NULL)
5303                         return (oldvd);
5304         }

5306         if (vd->vdev_resilvering && vdev_dtl_empty(vd, DTL_MISSING) &&
5307             vdev_dtl_empty(vd, DTL_OUTAGE)) {
5308                 ASSERT(vd->vdev_ops->vdev_op_leaf);
5309                 vd->vdev_resilvering = B_FALSE;
5310                 vdev_config_dirty(vd->vdev_top);
5311         }

5306         /*
5307          * Check for a completed replacement.  We always consider the first
5308          * vdev in the list to be the oldest vdev, and the last one to be
5309          * the newest (see spa_vdev_attach() for how that works).  In
5310          * the case where the newest vdev is faulted, we will not automatically
5311          * remove it after a resilver completes.  This is OK as it will require
5312          * user intervention to determine which disk the admin wishes to keep.
5313          */
5314         if (vd->vdev_ops == &vdev_replacing_ops) {
5315                 ASSERT(vd->vdev_children > 1);

5317                 newvd = vd->vdev_child[vd->vdev_children - 1];
5318                 oldvd = vd->vdev_child[0];

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

```
5324         }

5326         /*
5327          * Check for a completed resilver with the 'unspare' flag set.
5328          */
5329         if (vd->vdev_ops == &vdev_spare_ops) {
5330                 vdev_t *first = vd->vdev_child[0];
5331                 vdev_t *last = vd->vdev_child[vd->vdev_children - 1];

5333                 if (last->vdev_unspare) {
5334                         oldvd = first;
5335                         newvd = last;
5336                 } else if (first->vdev_unspare) {
5337                         oldvd = last;
5338                         newvd = first;
5339                 } else {
5340                         oldvd = NULL;
5341                 }

5343                 if (oldvd != NULL &&
5344                     vdev_dtl_empty(newvd, DTL_MISSING) &&
5345                     vdev_dtl_empty(newvd, DTL_OUTAGE) &&
5346                     !vdev_dtl_required(oldvd))
5347                         return (oldvd);

5349                 /*
5350                  * If there are more than two spares attached to a disk,
5351                  * and those spares are not required, then we want to
5352                  * attempt to free them up now so that they can be used
5353                  * by other pools.  Once we're back down to a single
5354                  * disk+spare, we stop removing them.
5355                  */
5356                 if (vd->vdev_children > 2) {
5357                         newvd = vd->vdev_child[1];

5359                         if (newvd->vdev_isspare && last->vdev_isspare &&
5360                             vdev_dtl_empty(last, DTL_MISSING) &&
5361                             vdev_dtl_empty(last, DTL_OUTAGE) &&
5362                             !vdev_dtl_required(newvd))
5363                                 return (newvd);
5364                 }
5365         }

5367         return (NULL);
5368 }

5370 static void
5371 spa_vdev_resilver_done(spa_t *spa)
5372 {
5373         vdev_t *vd, *pvd, *ppvd;
5374         uint64_t guid, sguid, pguid, ppguid;

5376         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

5378         while ((vd = spa_vdev_resilver_done_hunt(spa->spa_root_vdev)) != NULL) {
5379                 pvd = vd->vdev_parent;
5380                 ppvd = pvd->vdev_parent;
5381                 guid = vd->vdev_guid;
5382                 pguid = pvd->vdev_guid;
5383                 ppguid = ppvd->vdev_guid;
5384                 sguid = 0;
5385                 /*
5386                  * If we have just finished replacing a hot spared device, then
5387                  * we need to detach the parent's first child (the original hot
5388                  * spare) as well.
5389                  */
```



```
5390                 if (ppvd->vdev_ops == &vdev_spare_ops && pvd->vdev_id == 0 &&
5391                     ppvd->vdev_children == 2) {
5392                         ASSERT(pvd->vdev_ops == &vdev_replacing_ops);
5393                         sguid = ppvd->vdev_child[1]->vdev_guid;
5394                 }
5395                 ASSERT(vd->vdev_resilver_txg == 0 || !vdev_dtl_required(vd));

5397                 spa_config_exit(spa, SCL_ALL, FTAG);
5398                 if (spa_vdev_detach(spa, guid, pguid, B_TRUE) != 0)
5399                         return;
5400                 if (sguid && spa_vdev_detach(spa, sguid, ppguid, B_TRUE) != 0)
5401                         return;
5402                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
5403         }

5405         spa_config_exit(spa, SCL_ALL, FTAG);
5406 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   15442 Thu Aug  1 22:44:37 2013
new/usr/src/uts/common/fs/zfs/spa_config.c
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  24  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
  25  * Copyright (c) 2013 by Delphix. All rights reserved.
  25  * Copyright (c) 2012 by Delphix. All rights reserved.
  26  */

  28 #include <sys/spa.h>
  29 #include <sys/fm/fs/zfs.h>
  30 #include <sys/spa_impl.h>
  31 #include <sys/nvpair.h>
  32 #include <sys/uio.h>
  33 #include <sys/fs/zfs.h>
  34 #include <sys/vdev_impl.h>
  35 #include <sys/zfs_ioctl.h>
  36 #include <sys/utsname.h>
  37 #include <sys/systeminfo.h>
  38 #include <sys/sunddi.h>
  39 #include <sys/zfeature.h>
  40 #ifdef _KERNEL
  41 #include <sys/kobj.h>
  42 #include <sys/zone.h>
  43 #endif

  45 /*
  46  * Pool configuration repository.
  47  *
  48  * Pool configuration is stored as a packed nvlist on the filesystem.  By
  49  * default, all pools are stored in /etc/zfs/zpool.cache and loaded on boot
  50  * (when the ZFS module is loaded).  Pools can also have the 'cachefile'
  51  * property set that allows them to be stored in an alternate location until
  52  * the control of external software.
```



```
  53  *
  54  * For each cache file, we have a single nvlist which holds all the
  55  * configuration information.  When the module loads, we read this information
  56  * from /etc/zfs/zpool.cache and populate the SPA namespace.  This namespace is
  57  * maintained independently in spa.c.  Whenever the namespace is modified, or
  58  * the configuration of a pool is changed, we call spa_config_sync(), which
  59  * walks through all the active pools and writes the configuration to disk.
  60  */

  62 static uint64_t spa_config_generation = 1;

  64 /*
  65  * This can be overridden in userland to preserve an alternate namespace for
  66  * userland pools when doing testing.
  67  */
  68 const char *spa_config_path = ZPOOL_CACHE;

  70 /*
  71  * Called when the module is first loaded, this routine loads the configuration
  72  * file into the SPA namespace.  It does not actually open or load the pools; it
  73  * only populates the namespace.
  74  */
  75 void
  76 spa_config_load(void)
  77 {
  78         void *buf = NULL;
  79         nvlist_t *nvlist, *child;
  80         nvpair_t *nvpair;
  81         char *pathname;
  82         struct _buf *file;
  83         uint64_t fsize;

  85         /*
  86          * Open the configuration file.
  87          */
  88         pathname = kmem_alloc(MAXPATHLEN, KM_SLEEP);

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

  93         file = kobj_open_file(pathname);

  95         kmem_free(pathname, MAXPATHLEN);

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

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

 103         buf = kmem_alloc(fsize, KM_SLEEP);

 105         /*
 106          * Read the nvlist from the file.
 107          */
 108         if (kobj_read_file(file, buf, fsize, 0) < 0)
 109                 goto out;

 111         /*
 112          * Unpack the nvlist.
 113          */
 114         if (nvlist_unpack(buf, fsize, &nvlist, KM_SLEEP) != 0)
 115                 goto out;

 117         /*
 118          * Iterate over all elements in the nvlist, creating a new spa_t for
```

```
119          * each one with the specified configuration.
120          */
121         mutex_enter(&spa_namespace_lock);
122         nvpair = NULL;
123         while ((nvpair = nvlist_next_nvpair(nvlist, nvpair)) != NULL) {
124                 if (nvpair_type(nvpair) != DATA_TYPE_NVLIST)
125                         continue;

127                 VERIFY(nvpair_value_nvlist(nvpair, &child) == 0);

129                 if (spa_lookup(nvpair_name(nvpair)) != NULL)
130                         continue;
131                 (void) spa_add(nvpair_name(nvpair), child, NULL);
132         }
133         mutex_exit(&spa_namespace_lock);

135         nvlist_free(nvlist);

137 out:
138         if (buf != NULL)
139                 kmem_free(buf, fsize);

141         kobj_close_file(file);
142 }
```

unchanged_portion_omitted_

```
199 /*
200  * Synchronize pool configuration to disk.  This must be called with the
201  * namespace lock held. Synchronizing the pool cache is typically done after
202  * the configuration has been synced to the MOS. This exposes a window where
203  * the MOS config will have been updated but the cache file has not. If
204  * the system were to crash at that instant then the cached config may not
205  * contain the correct information to open the pool and an explicity import
206  * would be required.
201  * namespace lock held.
207  */
208 void
209 spa_config_sync(spa_t *target, boolean_t removing, boolean_t postsysevent)
210 {
211         spa_config_dirent_t *dp, *tdp;
212         nvlist_t *nvl;
213         boolean_t ccw_failure;
214         int error;

216         ASSERT(MUTEX_HELD(&spa_namespace_lock));

218         if (rootdir == NULL || !(spa_mode_global & FWRITE))
219                 return;

221         /*
222          * Iterate over all cachefiles for the pool, past or present.  When the
223          * cachefile is changed, the new one is pushed onto this list, allowing
224          * us to update previous cachefiles that no longer contain this pool.
225          */
226         ccw_failure = B_FALSE;
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;

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

```
 304         spa_config_generation++;

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

_____***unchanged_portion_omitted_***

```
**********************************************************
    4891 Thu Aug  1 22:44:38 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_scan.h
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

  75 /*
  76  * Every pool will have one dsl_scan_t and this structure will contain
  77  * in-memory information about the scan and a pointer to the on-disk
  78  * representation (i.e. dsl_scan_phys_t). Most of the state of the scan
  79  * is contained on-disk to allow the scan to resume in the event of a reboot
  80  * or panic. This structure maintains information about the behavior of a
  81  * running scan, some caching information, and how it should traverse the pool.
  82  *
  83  * The following members of this structure direct the behavior of the scan:
  84  *
  85  * scn_pausing -        a scan that cannot be completed in a single txg or
  86  *                      has exceeded its allotted time will need to pause.
  87  *                      When this flag is set the scanner will stop traversing
  88  *                      the pool and write out the current state to disk.
  89  *
  90  * scn_restart_txg -    directs the scanner to either restart or start a
  91  *                      a scan at the specified txg value.
  92  *
  93  * scn_done_txg -       when a scan completes its traversal it will set
  94  *                      the completion txg to the next txg. This is necessary
  95  *                      to ensure that any blocks that were freed during
  96  *                      the scan but have not yet been processed (i.e deferred
  97  *                      frees) are accounted for.
  98  *
  99  * This structure also maintains information about deferred frees which are
 100  * a special kind of traversal. Deferred free can exist in either a bptree or
 101  * a bpobj structure. The scn_is_bptree flag will indicate the type of
 102  * deferred free that is in progress. If the deferred free is part of an
 103  * asynchronous destroy then the scn_async_destroying flag will be set.
 104  */
 105 typedef struct dsl_scan {
 106         struct dsl_pool *scn_dp;

 108         boolean_t scn_pausing;
 109         uint64_t scn_restart_txg;
 110         uint64_t scn_done_txg;
 111         uint64_t scn_sync_start_time;
 112         zio_t *scn_zio_root;

 114         /* for freeing blocks */
 115         boolean_t scn_is_bptree;
 116         boolean_t scn_async_destroying;

 118         /* for debugging / information */
 119         uint64_t scn_visited_this_txg;

 121         dsl_scan_phys_t scn_phys;
 122 } dsl_scan_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   11455 Thu Aug  1 22:44:39 2013
new/usr/src/uts/common/fs/zfs/sys/vdev_impl.h
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright (c) 2013 by Delphix. All rights reserved.
  23  * Copyright (c) 2012 by Delphix. All rights reserved.
  24  */

  26 #ifndef _SYS_VDEV_IMPL_H
  27 #define _SYS_VDEV_IMPL_H

  29 #include <sys/avl.h>
  30 #include <sys/dmu.h>
  31 #include <sys/metaslab.h>
  32 #include <sys/nvpair.h>
  33 #include <sys/space_map.h>
  34 #include <sys/vdev.h>
  35 #include <sys/dkio.h>
  36 #include <sys/uberblock_impl.h>

  38 #ifdef __cplusplus
  39 extern "C" {
  40 #endif

  42 /*
  43  * Virtual device descriptors.
  44  *
  45  * All storage pool operations go through the virtual device framework,
  46  * which provides data replication and I/O scheduling.
  47  */

  49 /*
  50  * Forward declarations that lots of things need.
  51  */
  52 typedef struct vdev_queue vdev_queue_t;
```



```
  53 typedef struct vdev_cache vdev_cache_t;
  54 typedef struct vdev_cache_entry vdev_cache_entry_t;

  56 /*
  57  * Virtual device operations
  58  */
  59 typedef int     vdev_open_func_t(vdev_t *vd, uint64_t *size, uint64_t *max_size,
  60     uint64_t *ashift);
  61 typedef void    vdev_close_func_t(vdev_t *vd);
  62 typedef uint64_t vdev_asize_func_t(vdev_t *vd, uint64_t psize);
  63 typedef int     vdev_io_start_func_t(zio_t *zio);
  64 typedef void    vdev_io_done_func_t(zio_t *zio);
  65 typedef void    vdev_state_change_func_t(vdev_t *vd, int, int);
  66 typedef void    vdev_hold_func_t(vdev_t *vd);
  67 typedef void    vdev_rele_func_t(vdev_t *vd);

  69 typedef struct vdev_ops {
  70         vdev_open_func_t                *vdev_op_open;
  71         vdev_close_func_t               *vdev_op_close;
  72         vdev_asize_func_t               *vdev_op_asize;
  73         vdev_io_start_func_t            *vdev_op_io_start;
  74         vdev_io_done_func_t             *vdev_op_io_done;
  75         vdev_state_change_func_t        *vdev_op_state_change;
  76         vdev_hold_func_t                *vdev_op_hold;
  77         vdev_rele_func_t                *vdev_op_rele;
  78         char                            vdev_op_type[16];
  79         boolean_t                       vdev_op_leaf;
  80 } vdev_ops_t;
_____unchanged_portion_omitted_

 111 /*
 112  * Virtual device descriptor
 113  */
 114 struct vdev {
 115         /*
 116          * Common to all vdev types.
 117          */
 118         uint64_t        vdev_id;        /* child number in vdev parent  */
 119         uint64_t        vdev_guid;      /* unique ID for this vdev      */
 120         uint64_t        vdev_guid_sum;  /* self guid + all child guids  */
 121         uint64_t        vdev_orig_guid; /* orig. guid prior to remove   */
 122         uint64_t        vdev_asize;     /* allocatable device capacity  */
 123         uint64_t        vdev_min_asize; /* min acceptable asize         */
 124         uint64_t        vdev_max_asize; /* max acceptable asize         */
 125         uint64_t        vdev_ashift;    /* block alignment shift        */
 126         uint64_t        vdev_state;     /* see VDEV_STATE_* #defines    */
 127         uint64_t        vdev_prevstate; /* used when reopening a vdev   */
 128         vdev_ops_t      *vdev_ops;      /* vdev operations              */
 129         spa_t           *vdev_spa;      /* spa for this vdev            */
 130         void            *vdev_tsd;      /* type-specific data           */
 131         vnode_t         *vdev_name_vp;  /* vnode for pathname           */
 132         vnode_t         *vdev_devid_vp; /* vnode for devid              */
 133         vdev_t          *vdev_top;      /* top-level vdev               */
 134         vdev_t          *vdev_parent;   /* parent vdev                  */
 135         vdev_t          **vdev_child;   /* array of children            */
 136         uint64_t        vdev_children;  /* number of children           */
 137         space_map_t     vdev_dtl[DTL_TYPES]; /* in-core dirty time logs */
 138         vdev_stat_t     vdev_stat;      /* virtual device statistics    */
 139         boolean_t       vdev_expanding; /* expand the vdev?             */
 140         boolean_t       vdev_reopening; /* reopen in progress?          */
 141         int             vdev_open_error; /* error on last open          */
 142         kthread_t       *vdev_open_thread; /* thread opening children   */
 143         uint64_t        vdev_crtxg;     /* txg when top-level was added */

 145         /*
 146          * Top-level vdev state.
```

```
147          */
148         uint64_t        vdev_ms_array;  /* metaslab array object        */
149         uint64_t        vdev_ms_shift;  /* metaslab size shift          */
150         uint64_t        vdev_ms_count;  /* number of metaslabs          */
151         metaslab_group_t *vdev_mg;      /* metaslab group               */
152         metaslab_t      **vdev_ms;      /* metaslab array               */
153         txg_list_t      vdev_ms_list;   /* per-txg dirty metaslab lists */
154         txg_list_t      vdev_dtl_list;  /* per-txg dirty DTL lists      */
155         txg_node_t      vdev_txg_node;  /* per-txg dirty vdev linkage   */
156         boolean_t       vdev_remove_wanted; /* async remove wanted?     */
157         boolean_t       vdev_probe_wanted; /* async probe wanted?       */
158         uint64_t        vdev_removing;  /* device is being removed?     */
159         list_node_t     vdev_config_dirty_node; /* config dirty list    */
160         list_node_t     vdev_state_dirty_node; /* state dirty list      */
161         uint64_t        vdev_deflate_ratio; /* deflation ratio (x512)   */
162         uint64_t        vdev_islog;     /* is an intent log device      */
163         uint64_t        vdev_ishole;    /* is a hole in the namespace   */

165         /*
166          * Leaf vdev state.
167          */
168         uint64_t        vdev_psize;     /* physical device capacity     */
169         space_map_obj_t vdev_dtl_smo;   /* dirty time log space map obj */
170         txg_node_t      vdev_dtl_node;  /* per-txg dirty DTL linkage    */
171         uint64_t        vdev_wholedisk; /* true if this is a whole disk */
172         uint64_t        vdev_offline;   /* persistent offline state     */
173         uint64_t        vdev_faulted;   /* persistent faulted state     */
174         uint64_t        vdev_degraded;  /* persistent degraded state    */
175         uint64_t        vdev_removed;   /* persistent removed state     */
176         uint64_t        vdev_resilver_txg; /* persistent resilvering state */
176         uint64_t        vdev_resilvering; /* persistent resilvering state */
177         uint64_t        vdev_nparity;   /* number of parity devices for raidz */
178         char            *vdev_path;     /* vdev path (if any)           */
179         char            *vdev_devid;    /* vdev devid (if any)          */
180         char            *vdev_physpath; /* vdev device path (if any)    */
181         char            *vdev_fru;      /* physical FRU location        */
182         uint64_t        vdev_not_present; /* not present during import  */
183         uint64_t        vdev_unspare;   /* unspare when resilvering done */
184         hrtime_t        vdev_last_try;  /* last reopen time             */
185         boolean_t       vdev_nowritecache; /* true if flushwritecache failed */
186         boolean_t       vdev_checkremove; /* temporary online test      */
187         boolean_t       vdev_forcefault; /* force online fault          */
188         boolean_t       vdev_splitting; /* split or repair in progress  */
189         boolean_t       vdev_delayed_close; /* delayed device close?    */
190         uint8_t         vdev_tmpoffline; /* device taken offline temporarily? */
191         uint8_t         vdev_detached;  /* device detached?             */
192         uint8_t         vdev_cant_read; /* vdev is failing all reads    */
193         uint8_t         vdev_cant_write; /* vdev is failing all writes  */
194         uint64_t        vdev_isspare;   /* was a hot spare              */
195         uint64_t        vdev_isl2cache; /* was a l2cache device         */
196         vdev_queue_t    vdev_queue;     /* I/O deadline schedule queue  */
197         vdev_cache_t    vdev_cache;     /* physical block cache         */
198         spa_aux_vdev_t  *vdev_aux;      /* for l2cache vdevs            */
199         zio_t           *vdev_probe_zio; /* root of current probe       */
200         vdev_aux_t      vdev_label_aux; /* on-disk aux state            */

202         /*
203          * For DTrace to work in userland (libzpool) context, these fields must
204          * remain at the end of the structure.  DTrace will use the kernel's
205          * CTF definition for 'struct vdev', and since the size of a kmutex_t is
206          * larger in userland, the offsets for the rest of the fields would be
207          * incorrect.
208          */
209         kmutex_t        vdev_dtl_lock;  /* vdev_dtl_{map,resilver}      */
210         kmutex_t        vdev_stat_lock; /* vdev_stat                    */
211         kmutex_t        vdev_probe_lock; /* protects vdev_probe_zio     */
```



```
212 };
______unchanged_portion_omitted_
```

```
**********************************************************
    2266 Thu Aug  1 22:44:40 2013
new/usr/src/uts/common/fs/zfs/sys/zfs_debug.h
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@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 (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _SYS_ZFS_DEBUG_H
#define _SYS_ZFS_DEBUG_H

#ifdef  __cplusplus
extern "C" {
#endif

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

/*
 * ZFS debugging
 */

#if defined(DEBUG) || !defined(_KERNEL)
#define ZFS_DEBUG
#endif

extern int zfs_flags;

#define ZFS_DEBUG_DPRINTF       (1<<0)
#define ZFS_DEBUG_DBUF_VERIFY   (1<<1)
```



```
#define ZFS_DEBUG_DNODE_VERIFY  (1<<2)
#define ZFS_DEBUG_SNAPNAMES     (1<<3)
#define ZFS_DEBUG_MODIFY        (1<<4)
#define ZFS_DEBUG_SPA           (1<<5)
#define ZFS_DEBUG_ZIO_FREE      (1<<6)

#ifdef ZFS_DEBUG
extern void __dprintf(const char *file, const char *func,
    int line, const char *fmt, ...);
#define dprintf(...) \
        if (zfs_flags & ZFS_DEBUG_DPRINTF) \
                __dprintf(__FILE__, __func__, __LINE__, __VA_ARGS__)
#else
#define dprintf(...) ((void)0)
#endif /* ZFS_DEBUG */

extern void zfs_panic_recover(const char *fmt, ...);

typedef struct zfs_dbgmsg {
        list_node_t zdm_node;
        time_t zdm_timestamp;
        char zdm_msg[1]; /* variable length allocation */
} zfs_dbgmsg_t;

extern void zfs_dbgmsg_init(void);
extern void zfs_dbgmsg_fini(void);
extern void zfs_dbgmsg(const char *fmt, ...);
extern void zfs_dbgmsg_print(const char *tag);

#ifndef _KERNEL
extern int dprintf_find_string(const char *string);
#endif

#ifdef  __cplusplus
}
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   88667 Thu Aug  1 22:44:41 2013
new/usr/src/uts/common/fs/zfs/vdev.c
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 336 /*
 337  * Allocate a new vdev.  The 'alloctype' is used to control whether we are
 338  * creating a new vdev or loading an existing one - the behavior is slightly
 339  * different for each case.
 340  */
 341 int
 342 vdev_alloc(spa_t *spa, vdev_t **vdp, nvlist_t *nv, vdev_t *parent, uint_t id,
 343     int alloctype)
 344 {
 345         vdev_ops_t *ops;
 346         char *type;
 347         uint64_t guid = 0, islog, nparity;
 348         vdev_t *vd;

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

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

 355         if ((ops = vdev_getops(type)) == NULL)
 356                 return (SET_ERROR(EINVAL));

 358         /*
 359          * If this is a load, get the vdev guid from the nvlist.
 360          * Otherwise, vdev_alloc_common() will generate one for us.
 361          */
 362         if (alloctype == VDEV_ALLOC_LOAD) {
 363                 uint64_t label_id;

 365                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ID, &label_id) ||
 366                     label_id != id)
 367                         return (SET_ERROR(EINVAL));

 369                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
 370                         return (SET_ERROR(EINVAL));
 371         } else if (alloctype == VDEV_ALLOC_SPARE) {
 372                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
 373                         return (SET_ERROR(EINVAL));
 374         } else if (alloctype == VDEV_ALLOC_L2CACHE) {
 375                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
 376                         return (SET_ERROR(EINVAL));
 377         } else if (alloctype == VDEV_ALLOC_ROOTPOOL) {
 378                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
 379                         return (SET_ERROR(EINVAL));
 380         }

 382         /*
 383          * The first allocated vdev must be of type 'root'.
 384          */
 385         if (ops != &vdev_root_ops && spa->spa_root_vdev == NULL)
 386                 return (SET_ERROR(EINVAL));
```



```
 388         /*
 389          * Determine whether we're a log vdev.
 390          */
 391         islog = 0;
 392         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_IS_LOG, &islog);
 393         if (islog && spa_version(spa) < SPA_VERSION_SLOGS)
 394                 return (SET_ERROR(ENOTSUP));

 396         if (ops == &vdev_hole_ops && spa_version(spa) < SPA_VERSION_HOLES)
 397                 return (SET_ERROR(ENOTSUP));

 399         /*
 400          * Set the nparity property for RAID-Z vdevs.
 401          */
 402         nparity = -1ULL;
 403         if (ops == &vdev_raidz_ops) {
 404                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_NPARITY,
 405                     &nparity) == 0) {
 406                         if (nparity == 0 || nparity > VDEV_RAIDZ_MAXPARITY)
 407                                 return (SET_ERROR(EINVAL));
 408                         /*
 409                          * Previous versions could only support 1 or 2 parity
 410                          * device.
 411                          */
 412                         if (nparity > 1 &&
 413                             spa_version(spa) < SPA_VERSION_RAIDZ2)
 414                                 return (SET_ERROR(ENOTSUP));
 415                         if (nparity > 2 &&
 416                             spa_version(spa) < SPA_VERSION_RAIDZ3)
 417                                 return (SET_ERROR(ENOTSUP));
 418                 } else {
 419                         /*
 420                          * We require the parity to be specified for SPAs that
 421                          * support multiple parity levels.
 422                          */
 423                         if (spa_version(spa) >= SPA_VERSION_RAIDZ2)
 424                                 return (SET_ERROR(EINVAL));
 425                         /*
 426                          * Otherwise, we default to 1 parity device for RAID-Z.
 427                          */
 428                         nparity = 1;
 429                 }
 430         } else {
 431                 nparity = 0;
 432         }
 433         ASSERT(nparity != -1ULL);

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

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

 440         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PATH, &vd->vdev_path) == 0)
 441                 vd->vdev_path = spa_strdup(vd->vdev_path);
 442         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_DEVID, &vd->vdev_devid) == 0)
 443                 vd->vdev_devid = spa_strdup(vd->vdev_devid);
 444         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PHYS_PATH,
 445             &vd->vdev_physpath) == 0)
 446                 vd->vdev_physpath = spa_strdup(vd->vdev_physpath);
 447         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_FRU, &vd->vdev_fru) == 0)
 448                 vd->vdev_fru = spa_strdup(vd->vdev_fru);

 450         /*
 451          * Set the whole_disk property.  If it's not specified, leave the value
 452          * as -1.
```

```
453          */
454         if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_WHOLE_DISK,
455             &vd->vdev_wholedisk) != 0)
456                 vd->vdev_wholedisk = -1ULL;
457
458         /*
459          * Look for the 'not present' flag.  This will only be set if the device
460          * was not present at the time of import.
461          */
462         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_NOT_PRESENT,
463             &vd->vdev_not_present);
464
465         /*
466          * Get the alignment requirement.
467          */
468         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ASHIFT, &vd->vdev_ashift);
469
470         /*
471          * Retrieve the vdev creation time.
472          */
473         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_CREATE_TXG,
474             &vd->vdev_crtxg);
475
476         /*
477          * If we're a top-level vdev, try to load the allocation parameters.
478          */
479         if (parent && !parent->vdev_parent &&
480             (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_SPLIT)) {
481                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_ARRAY,
482                     &vd->vdev_ms_array);
483                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_SHIFT,
484                     &vd->vdev_ms_shift);
485                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ASIZE,
486                     &vd->vdev_asize);
487                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_REMOVING,
488                     &vd->vdev_removing);
489         }
490
491         if (parent && !parent->vdev_parent && alloctype != VDEV_ALLOC_ATTACH) {
492                 ASSERT(alloctype == VDEV_ALLOC_LOAD ||
493                     alloctype == VDEV_ALLOC_ADD ||
494                     alloctype == VDEV_ALLOC_SPLIT ||
495                     alloctype == VDEV_ALLOC_ROOTPOOL);
496                 vd->vdev_mg = metaslab_group_create(islog ?
497                     spa_log_class(spa) : spa_normal_class(spa), vd);
498         }
499
500         /*
501          * If we're a leaf vdev, try to load the DTL object and other state.
502          */
503         if (vd->vdev_ops->vdev_op_leaf &&
504             (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_L2CACHE ||
505             alloctype == VDEV_ALLOC_ROOTPOOL)) {
506                 if (alloctype == VDEV_ALLOC_LOAD) {
507                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_DTL,
508                             &vd->vdev_dtl_smo.smo_object);
509                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_UNSPARE,
510                             &vd->vdev_unspare);
511                 }
512
513                 if (alloctype == VDEV_ALLOC_ROOTPOOL) {
514                         uint64_t spare = 0;
515
516                         if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_IS_SPARE,
517                             &spare) == 0 && spare)
518                                 spa_spare_add(vd);
```



```
519                 }
520
521                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_OFFLINE,
522                     &vd->vdev_offline);
523
524                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_RESILVER_TXG,
525                     &vd->vdev_resilver_txg);
524                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_RESILVERING,
525                     &vd->vdev_resilvering);
526
527                 /*
528                  * When importing a pool, we want to ignore the persistent fault
529                  * state, as the diagnosis made on another system may not be
530                  * valid in the current context.  Local vdevs will
531                  * remain in the faulted state.
532                  */
533                 if (spa_load_state(spa) == SPA_LOAD_OPEN) {
534                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_FAULTED,
535                             &vd->vdev_faulted);
536                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_DEGRADED,
537                             &vd->vdev_degraded);
538                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_REMOVED,
539                             &vd->vdev_removed);
540
541                         if (vd->vdev_faulted || vd->vdev_degraded) {
542                                 char *aux;
543
544                                 vd->vdev_label_aux =
545                                     VDEV_AUX_ERR_EXCEEDED;
546                                 if (nvlist_lookup_string(nv,
547                                     ZPOOL_CONFIG_AUX_STATE, &aux) == 0 &&
548                                     strcmp(aux, "external") == 0)
549                                         vd->vdev_label_aux = VDEV_AUX_EXTERNAL;
550                         }
551                 }
552         }
553
554         /*
555          * Add ourselves to the parent's list of children.
556          */
557         vdev_add_child(parent, vd);
558
559         *vdp = vd;
560
561         return (0);
562 }
_____unchanged_portion_omitted_

1665 /*
1666  * Returns the lowest txg in the DTL range.
1667  */
1668 static uint64_t
1669 vdev_dtl_min(vdev_t *vd)
1670 {
1671         space_seg_t *ss;
1672
1673         ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
1674         ASSERT3U(vd->vdev_dtl[DTL_MISSING].sm_space, !=, 0);
1675         ASSERT0(vd->vdev_children);
1676
1677         ss = avl_first(&vd->vdev_dtl[DTL_MISSING].sm_root);
1678         return (ss->ss_start - 1);
1679 }
1680
1681 /*
1682  * Returns the highest txg in the DTL.
```

```
1683  */
1684 static uint64_t
1685 vdev_dtl_max(vdev_t *vd)
1686 {
1687         space_seg_t *ss;
1688
1689         ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
1690         ASSERT3U(vd->vdev_dtl[DTL_MISSING].sm_space, !=, 0);
1691         ASSERT0(vd->vdev_children);
1692
1693         ss = avl_last(&vd->vdev_dtl[DTL_MISSING].sm_root);
1694         return (ss->ss_end);
1695 }
1696
1697 /*
1698  * Determine if a resilvering vdev should remove any DTL entries from
1699  * its range. If the vdev was resilvering for the entire duration of the
1700  * scan then it should excise that range from its DTLs. Otherwise, this
1701  * vdev is considered partially resilvered and should leave its DTL
1702  * entries intact. The comment in vdev_dtl_reassess() describes how we
1703  * excise the DTLs.
1704  */
1705 static boolean_t
1706 vdev_dtl_should_excise(vdev_t *vd)
1707 {
1708         spa_t *spa = vd->vdev_spa;
1709         dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;
1710
1711         ASSERT0(scn->scn_phys.scn_errors);
1712         ASSERT0(vd->vdev_children);
1713
1714         if (vd->vdev_resilver_txg == 0 ||
1715             vd->vdev_dtl[DTL_MISSING].sm_space == 0)
1716                 return (B_TRUE);
1717
1718         /*
1719          * When a resilver is initiated the scan will assign the scn_max_txg
1720          * value to the highest txg value that exists in all DTLs. If this
1721          * device's max DTL is not part of this scan (i.e. it is not in
1722          * the range (scn_min_txg, scn_max_txg] then it is not eligible
1723          * for excision.
1724          */
1725         if (vdev_dtl_max(vd) <= scn->scn_phys.scn_max_txg) {
1726                 ASSERT3U(scn->scn_phys.scn_min_txg, <=, vdev_dtl_min(vd));
1727                 ASSERT3U(scn->scn_phys.scn_min_txg, <, vd->vdev_resilver_txg);
1728                 ASSERT3U(vd->vdev_resilver_txg, <=, scn->scn_phys.scn_max_txg);
1729                 return (B_TRUE);
1730         }
1731         return (B_FALSE);
1732 }
1733
1734 /*
1735  * Reassess DTLs after a config change or scrub completion.
1736  */
1737 void
1738 vdev_dtl_reassess(vdev_t *vd, uint64_t txg, uint64_t scrub_txg, int scrub_done)
1739 {
1740         spa_t *spa = vd->vdev_spa;
1741         avl_tree_t reftree;
1742         int minref;
1743
1744         ASSERT(spa_config_held(spa, SCL_ALL, RW_READER) != 0);
1745
1746         for (int c = 0; c < vd->vdev_children; c++)
1747                 vdev_dtl_reassess(vd->vdev_child[c], txg,
1748                     scrub_txg, scrub_done);
```



```
1750         if (vd == spa->spa_root_vdev || vd->vdev_ishole || vd->vdev_aux)
1751                 return;
1752
1753         if (vd->vdev_ops->vdev_op_leaf) {
1754                 dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;
1755
1756                 mutex_enter(&vd->vdev_dtl_lock);
1757
1758                 /*
1759                  * If we've completed a scan cleanly then determine
1760                  * if this vdev should remove any DTLs. We only want to
1761                  * excise regions on vdevs that were available during
1762                  * the entire duration of this scan.
1763                  */
1764                 if (scrub_txg != 0 &&
1765                     (spa->spa_scrub_started ||
1766                     (scn != NULL && scn->scn_phys.scn_errors == 0)) &&
1767                     vdev_dtl_should_excise(vd)) {
1690                     (scn && scn->scn_phys.scn_errors == 0))) {
1768                         /*
1769                          * We completed a scrub up to scrub_txg.  If we
1770                          * did it without rebooting, then the scrub dtl
1771                          * will be valid, so excise the old region and
1772                          * fold in the scrub dtl.  Otherwise, leave the
1773                          * dtl as-is if there was an error.
1774                          *
1775                          * There's little trick here: to excise the beginning
1776                          * of the DTL_MISSING map, we put it into a reference
1777                          * tree and then add a segment with refcnt -1 that
1778                          * covers the range [0, scrub_txg).  This means
1779                          * that each txg in that range has refcnt -1 or 0.
1780                          * We then add DTL_SCRUB with a refcnt of 2, so that
1781                          * entries in the range [0, scrub_txg) will have a
1782                          * positive refcnt -- either 1 or 2.  We then convert
1783                          * the reference tree into the new DTL_MISSING map.
1784                          */
1785                         space_map_ref_create(&reftree);
1786                         space_map_ref_add_map(&reftree,
1787                             &vd->vdev_dtl[DTL_MISSING], 1);
1788                         space_map_ref_add_seg(&reftree, 0, scrub_txg, -1);
1789                         space_map_ref_add_map(&reftree,
1790                             &vd->vdev_dtl[DTL_SCRUB], 2);
1791                         space_map_ref_generate_map(&reftree,
1792                             &vd->vdev_dtl[DTL_MISSING], 1);
1793                         space_map_ref_destroy(&reftree);
1794                 }
1795                 space_map_vacate(&vd->vdev_dtl[DTL_PARTIAL], NULL, NULL);
1796                 space_map_walk(&vd->vdev_dtl[DTL_MISSING],
1797                     space_map_add, &vd->vdev_dtl[DTL_PARTIAL]);
1798                 if (scrub_done)
1799                         space_map_vacate(&vd->vdev_dtl[DTL_SCRUB], NULL, NULL);
1800                 space_map_vacate(&vd->vdev_dtl[DTL_OUTAGE], NULL, NULL);
1801                 if (!vdev_readable(vd))
1802                         space_map_add(&vd->vdev_dtl[DTL_OUTAGE], 0, -1ULL);
1803                 else
1804                         space_map_walk(&vd->vdev_dtl[DTL_MISSING],
1805                             space_map_add, &vd->vdev_dtl[DTL_OUTAGE]);
1806
1807                 /*
1808                  * If the vdev was resilvering and no longer has any
1809                  * DTLs then reset its resilvering flag.
1810                  */
1811                 if (vd->vdev_resilver_txg != 0 &&
1812                     vd->vdev_dtl[DTL_MISSING].sm_space == 0 &&
1813                     vd->vdev_dtl[DTL_OUTAGE].sm_space == 0)
```

```
1814                        vd->vdev_resilver_txg = 0;

1816                mutex_exit(&vd->vdev_dtl_lock);

1818                if (txg != 0)
1819                        vdev_dirty(vd->vdev_top, VDD_DTL, vd, txg);
1820                return;
1821        }

1823        mutex_enter(&vd->vdev_dtl_lock);
1824        for (int t = 0; t < DTL_TYPES; t++) {
1825                /* account for child's outage in parent's missing map */
1826                int s = (t == DTL_MISSING) ? DTL_OUTAGE: t;
1827                if (t == DTL_SCRUB)
1828                        continue;                       /* leaf vdevs only */
1829                if (t == DTL_PARTIAL)
1830                        minref = 1;                     /* i.e. non-zero */
1831                else if (vd->vdev_nparity != 0)
1832                        minref = vd->vdev_nparity + 1;  /* RAID-Z */
1833                else
1834                        minref = vd->vdev_children;     /* any kind of mirror */
1835                space_map_ref_create(&reftree);
1836                for (int c = 0; c < vd->vdev_children; c++) {
1837                        vdev_t *cvd = vd->vdev_child[c];
1838                        mutex_enter(&cvd->vdev_dtl_lock);
1839                        space_map_ref_add_map(&reftree, &cvd->vdev_dtl[s], 1);
1840                        mutex_exit(&cvd->vdev_dtl_lock);
1841                }
1842                space_map_ref_generate_map(&reftree, &vd->vdev_dtl[t], minref);
1843                space_map_ref_destroy(&reftree);
1844        }
1845        mutex_exit(&vd->vdev_dtl_lock);
1846 }
```

unchanged_portion_omitted_

```
1978 /*
1979  * Determine if resilver is needed, and if so the txg range.
1980  */
1981 boolean_t
1982 vdev_resilver_needed(vdev_t *vd, uint64_t *minp, uint64_t *maxp)
1983 {
1984        boolean_t needed = B_FALSE;
1985        uint64_t thismin = UINT64_MAX;
1986        uint64_t thismax = 0;

1988        if (vd->vdev_children == 0) {
1989                mutex_enter(&vd->vdev_dtl_lock);
1990                if (vd->vdev_dtl[DTL_MISSING].sm_space != 0 &&
1991                    vdev_writeable(vd)) {
1905                        space_seg_t *ss;

1993                        thismin = vdev_dtl_min(vd);
1994                        thismax = vdev_dtl_max(vd);
1907                        ss = avl_first(&vd->vdev_dtl[DTL_MISSING].sm_root);
1908                        thismin = ss->ss_start - 1;
1909                        ss = avl_last(&vd->vdev_dtl[DTL_MISSING].sm_root);
1910                        thismax = ss->ss_end;
1995                        needed = B_TRUE;
1996                }
1997                mutex_exit(&vd->vdev_dtl_lock);
1998        } else {
1999                for (int c = 0; c < vd->vdev_children; c++) {
2000                        vdev_t *cvd = vd->vdev_child[c];
2001                        uint64_t cmin, cmax;

2003                        if (vdev_resilver_needed(cvd, &cmin, &cmax)) {
```



```
2004                                thismin = MIN(thismin, cmin);
2005                                thismax = MAX(thismax, cmax);
2006                                needed = B_TRUE;
2007                        }
2008                }
2009        }

2011        if (needed && minp) {
2012                *minp = thismin;
2013                *maxp = thismax;
2014        }
2015        return (needed);
2016 }
```

unchanged_portion_omitted_

```
**********************************************************
   37490 Thu Aug  1 22:44:43 2013
new/usr/src/uts/common/fs/zfs/vdev_label.c
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 210 /*
 211  * Generate the nvlist representing this vdev's config.
 212  */
 213 nvlist_t *
 214 vdev_config_generate(spa_t *spa, vdev_t *vd, boolean_t getstats,
 215     vdev_config_flag_t flags)
 216 {
 217         nvlist_t *nv = NULL;

 219         nv = fnvlist_alloc();
 219         VERIFY(nvlist_alloc(&nv, NV_UNIQUE_NAME, KM_SLEEP) == 0);

 221         fnvlist_add_string(nv, ZPOOL_CONFIG_TYPE, vd->vdev_ops->vdev_op_type);
 221         VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_TYPE,
 222             vd->vdev_ops->vdev_op_type) == 0);
 222         if (!(flags & (VDEV_CONFIG_SPARE | VDEV_CONFIG_L2CACHE)))
 223                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_ID, vd->vdev_id);
 224         fnvlist_add_uint64(nv, ZPOOL_CONFIG_GUID, vd->vdev_guid);
 224                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_ID, vd->vdev_id)
 225                     == 0);
 226         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_GUID, vd->vdev_guid) == 0);

 226         if (vd->vdev_path != NULL)
 227                 fnvlist_add_string(nv, ZPOOL_CONFIG_PATH, vd->vdev_path);
 229                 VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_PATH,
 230                     vd->vdev_path) == 0);

 229         if (vd->vdev_devid != NULL)
 230                 fnvlist_add_string(nv, ZPOOL_CONFIG_DEVID, vd->vdev_devid);
 233                 VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_DEVID,
 234                     vd->vdev_devid) == 0);

 232         if (vd->vdev_physpath != NULL)
 233                 fnvlist_add_string(nv, ZPOOL_CONFIG_PHYS_PATH,
 234                     vd->vdev_physpath);
 237                 VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_PHYS_PATH,
 238                     vd->vdev_physpath) == 0);

 236         if (vd->vdev_fru != NULL)
 237                 fnvlist_add_string(nv, ZPOOL_CONFIG_FRU, vd->vdev_fru);
 241                 VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_FRU,
 242                     vd->vdev_fru) == 0);

 239         if (vd->vdev_nparity != 0) {
 240                 ASSERT(strcmp(vd->vdev_ops->vdev_op_type,
 241                     VDEV_TYPE_RAIDZ) == 0);

 243                 /*
 244                  * Make sure someone hasn't managed to sneak a fancy new vdev
 245                  * into a crufty old storage pool.
 246                  */
```



```
 247                 ASSERT(vd->vdev_nparity == 1 ||
 248                     (vd->vdev_nparity <= 2 &&
 249                     spa_version(spa) >= SPA_VERSION_RAIDZ2) ||
 250                     (vd->vdev_nparity <= 3 &&
 251                     spa_version(spa) >= SPA_VERSION_RAIDZ3));

 253                 /*
 254                  * Note that we'll add the nparity tag even on storage pools
 255                  * that only support a single parity device -- older software
 256                  * will just ignore it.
 257                  */
 258                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_NPARITY, vd->vdev_nparity);
 263                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_NPARITY,
 264                     vd->vdev_nparity) == 0);
 259         }

 261         if (vd->vdev_wholedisk != -1ULL)
 262                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_WHOLE_DISK,
 263                     vd->vdev_wholedisk);
 268                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_WHOLE_DISK,
 269                     vd->vdev_wholedisk) == 0);

 265         if (vd->vdev_not_present)
 266                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_NOT_PRESENT, 1);
 272                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_NOT_PRESENT, 1) == 0);

 268         if (vd->vdev_isspare)
 269                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_IS_SPARE, 1);
 275                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_IS_SPARE, 1) == 0);

 271         if (!(flags & (VDEV_CONFIG_SPARE | VDEV_CONFIG_L2CACHE)) &&
 272             vd == vd->vdev_top) {
 273                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_METASLAB_ARRAY,
 274                     vd->vdev_ms_array);
 275                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_METASLAB_SHIFT,
 276                     vd->vdev_ms_shift);
 277                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_ASHIFT, vd->vdev_ashift);
 278                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_ASIZE,
 279                     vd->vdev_asize);
 280                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_IS_LOG, vd->vdev_islog);
 279                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_METASLAB_ARRAY,
 280                     vd->vdev_ms_array) == 0);
 281                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_METASLAB_SHIFT,
 282                     vd->vdev_ms_shift) == 0);
 283                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_ASHIFT,
 284                     vd->vdev_ashift) == 0);
 285                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_ASIZE,
 286                     vd->vdev_asize) == 0);
 287                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_IS_LOG,
 288                     vd->vdev_islog) == 0);
 281                 if (vd->vdev_removing)
 282                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_REMOVING,
 283                             vd->vdev_removing);
 290                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_REMOVING,
 291                             vd->vdev_removing) == 0);
 284         }

 286         if (vd->vdev_dtl_smo.smo_object != 0)
 287                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_DTL,
 288                     vd->vdev_dtl_smo.smo_object);
 295                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_DTL,
 296                     vd->vdev_dtl_smo.smo_object) == 0);

 290         if (vd->vdev_crtxg)
 291                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_CREATE_TXG, vd->vdev_crtxg);
 299                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_CREATE_TXG,
```

```
                    vd->vdev_crtxg) == 0);

        if (getstats) {
                vdev_stat_t vs;
                pool_scan_stat_t ps;

                vdev_get_stats(vd, &vs);
                fnvlist_add_uint64_array(nv, ZPOOL_CONFIG_VDEV_STATS,
                    (uint64_t *)&vs, sizeof (vs) / sizeof (uint64_t));
                VERIFY(nvlist_add_uint64_array(nv, ZPOOL_CONFIG_VDEV_STATS,
                    (uint64_t *)&vs, sizeof (vs) / sizeof (uint64_t)) == 0);

                /* provide either current or previous scan information */
                if (spa_scan_get_stats(spa, &ps) == 0) {
                        fnvlist_add_uint64_array(nv,
                        VERIFY(nvlist_add_uint64_array(nv,
                            ZPOOL_CONFIG_SCAN_STATS, (uint64_t *)&ps,
                            sizeof (pool_scan_stat_t) / sizeof (uint64_t));
                            sizeof (pool_scan_stat_t) / sizeof (uint64_t))
                            == 0);
                }
        }

        if (!vd->vdev_ops->vdev_op_leaf) {
                nvlist_t **child;
                int c, idx;

                ASSERT(!vd->vdev_ishole);

                child = kmem_alloc(vd->vdev_children * sizeof (nvlist_t *),
                    KM_SLEEP);

                for (c = 0, idx = 0; c < vd->vdev_children; c++) {
                        vdev_t *cvd = vd->vdev_child[c];

                        /*
                         * If we're generating an nvlist of removing
                         * vdevs then skip over any device which is
                         * not being removed.
                         */
                        if ((flags & VDEV_CONFIG_REMOVING) &&
                            !cvd->vdev_removing)
                                continue;

                        child[idx++] = vdev_config_generate(spa, cvd,
                            getstats, flags);
                }

                if (idx) {
                        fnvlist_add_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN,
                            child, idx);
                        VERIFY(nvlist_add_nvlist_array(nv,
                            ZPOOL_CONFIG_CHILDREN, child, idx) == 0);
                }

                for (c = 0; c < idx; c++)
                        nvlist_free(child[c]);

                kmem_free(child, vd->vdev_children * sizeof (nvlist_t *));

        } else {
                const char *aux = NULL;

                if (vd->vdev_offline && !vd->vdev_tmpoffline)
                        fnvlist_add_uint64(nv, ZPOOL_CONFIG_OFFLINE, B_TRUE);
                if (vd->vdev_resilver_txg != 0)
```



```
                        fnvlist_add_uint64(nv, ZPOOL_CONFIG_RESILVER_TXG,
                            vd->vdev_resilver_txg);
                        VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_OFFLINE,
                            B_TRUE) == 0);
                if (vd->vdev_resilvering)
                        VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_RESILVERING,
                            B_TRUE) == 0);
                if (vd->vdev_faulted)
                        fnvlist_add_uint64(nv, ZPOOL_CONFIG_FAULTED, B_TRUE);
                        VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_FAULTED,
                            B_TRUE) == 0);
                if (vd->vdev_degraded)
                        fnvlist_add_uint64(nv, ZPOOL_CONFIG_DEGRADED, B_TRUE);
                        VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_DEGRADED,
                            B_TRUE) == 0);
                if (vd->vdev_removed)
                        fnvlist_add_uint64(nv, ZPOOL_CONFIG_REMOVED, B_TRUE);
                        VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_REMOVED,
                            B_TRUE) == 0);
                if (vd->vdev_unspare)
                        fnvlist_add_uint64(nv, ZPOOL_CONFIG_UNSPARE, B_TRUE);
                        VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_UNSPARE,
                            B_TRUE) == 0);
                if (vd->vdev_ishole)
                        fnvlist_add_uint64(nv, ZPOOL_CONFIG_IS_HOLE, B_TRUE);
                        VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_IS_HOLE,
                            B_TRUE) == 0);

                switch (vd->vdev_stat.vs_aux) {
                case VDEV_AUX_ERR_EXCEEDED:
                        aux = "err_exceeded";
                        break;

                case VDEV_AUX_EXTERNAL:
                        aux = "external";
                        break;
                }

                if (aux != NULL)
                        fnvlist_add_string(nv, ZPOOL_CONFIG_AUX_STATE, aux);
                        VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_AUX_STATE,
                            aux) == 0);

                if (vd->vdev_splitting && vd->vdev_orig_guid != 0LL) {
                        fnvlist_add_uint64(nv, ZPOOL_CONFIG_ORIG_GUID,
                            vd->vdev_orig_guid);
                        VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_ORIG_GUID,
                            vd->vdev_orig_guid) == 0);
                }
        }

        return (nv);
}
_____unchanged_portion_omitted_
```

****
2891 Thu Aug 1 22:44:44 2013
new/usr/src/uts/common/fs/zfs/zfs_debug.c
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@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) 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#include <sys/zfs_context.h>

list_t zfs_dbgmsgs;
int zfs_dbgmsg_size;
kmutex_t zfs_dbgmsgs_lock;
int zfs_dbgmsg_maxsize = 1<<20; /* 1MB */

void
zfs_dbgmsg_init(void)
{
        list_create(&zfs_dbgmsgs, sizeof (zfs_dbgmsg_t),
            offsetof(zfs_dbgmsg_t, zdm_node));
        mutex_init(&zfs_dbgmsgs_lock, NULL, MUTEX_DEFAULT, NULL);
}
```

*unchanged_portion_omitted_*

```
void
zfs_dbgmsg_print(const char *tag)
{
        zfs_dbgmsg_t *zdm;

        (void) printf("ZFS_DBGMSG(%s):\n", tag);
        mutex_enter(&zfs_dbgmsgs_lock);
        for (zdm = list_head(&zfs_dbgmsgs); zdm;
            zdm = list_next(&zfs_dbgmsgs, zdm))
                (void) printf("%s\n", zdm->zdm_msg);
        mutex_exit(&zfs_dbgmsgs_lock);
```



```
}
```

```
**********************************************************
   28913 Thu Aug  1 22:44:45 2013
new/usr/src/uts/common/sys/fs/zfs.h
3956 ::vdev -r should work with pipelines
3957 ztest should update the cachefile before killing itself
3958 multiple scans can lead to partial resilvering
3959 ddt entries are not always resilvered
3960 dsl_scan can skip over dedup-ed blocks if physical birth != logical birth
3961 freed gang blocks are not resilvered and can cause pool to suspend
3962 ztest should print out zfs debug buffer before exiting
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  24  * Copyright (c) 2013 by Delphix. All rights reserved.
  24  * Copyright (c) 2012 by Delphix. All rights reserved.
  25  * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
  26  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
  27  */

  29 /* Portions Copyright 2010 Robert Milkowski */

  31 #ifndef _SYS_FS_ZFS_H
  32 #define _SYS_FS_ZFS_H

  34 #include <sys/time.h>

  36 #ifdef  __cplusplus
  37 extern "C" {
  38 #endif

  40 /*
  41  * Types and constants shared between userland and the kernel.
  42  */

  44 /*
  45  * Each dataset can be one of the following types.  These constants can be
  46  * combined into masks that can be passed to various functions.
  47  */
  48 typedef enum {
  49         ZFS_TYPE_FILESYSTEM     = 0x1,
  50         ZFS_TYPE_SNAPSHOT       = 0x2,
  51         ZFS_TYPE_VOLUME         = 0x4,
  52         ZFS_TYPE_POOL           = 0x8
```



```
  53 } zfs_type_t;
_____unchanged_portion_omitted_

 476 /*
 477  * The following are configuration names used in the nvlist describing a pool's
 478  * configuration.
 479  */
 480 #define ZPOOL_CONFIG_VERSION            "version"
 481 #define ZPOOL_CONFIG_POOL_NAME          "name"
 482 #define ZPOOL_CONFIG_POOL_STATE         "state"
 483 #define ZPOOL_CONFIG_POOL_TXG           "txg"
 484 #define ZPOOL_CONFIG_POOL_GUID          "pool_guid"
 485 #define ZPOOL_CONFIG_CREATE_TXG         "create_txg"
 486 #define ZPOOL_CONFIG_TOP_GUID           "top_guid"
 487 #define ZPOOL_CONFIG_VDEV_TREE          "vdev_tree"
 488 #define ZPOOL_CONFIG_TYPE               "type"
 489 #define ZPOOL_CONFIG_CHILDREN           "children"
 490 #define ZPOOL_CONFIG_ID                 "id"
 491 #define ZPOOL_CONFIG_GUID               "guid"
 492 #define ZPOOL_CONFIG_PATH               "path"
 493 #define ZPOOL_CONFIG_DEVID              "devid"
 494 #define ZPOOL_CONFIG_METASLAB_ARRAY     "metaslab_array"
 495 #define ZPOOL_CONFIG_METASLAB_SHIFT     "metaslab_shift"
 496 #define ZPOOL_CONFIG_ASHIFT             "ashift"
 497 #define ZPOOL_CONFIG_ASIZE              "asize"
 498 #define ZPOOL_CONFIG_DTL                "DTL"
 499 #define ZPOOL_CONFIG_SCAN_STATS         "scan_stats"    /* not stored on disk */
 500 #define ZPOOL_CONFIG_VDEV_STATS         "vdev_stats"    /* not stored on disk */
 501 #define ZPOOL_CONFIG_WHOLE_DISK         "whole_disk"
 502 #define ZPOOL_CONFIG_ERRCOUNT           "error_count"
 503 #define ZPOOL_CONFIG_NOT_PRESENT        "not_present"
 504 #define ZPOOL_CONFIG_SPARES             "spares"
 505 #define ZPOOL_CONFIG_IS_SPARE           "is_spare"
 506 #define ZPOOL_CONFIG_NPARITY            "nparity"
 507 #define ZPOOL_CONFIG_HOSTID             "hostid"
 508 #define ZPOOL_CONFIG_HOSTNAME           "hostname"
 509 #define ZPOOL_CONFIG_LOADED_TIME        "initial_load_time"
 510 #define ZPOOL_CONFIG_UNSPARE            "unspare"
 511 #define ZPOOL_CONFIG_PHYS_PATH          "phys_path"
 512 #define ZPOOL_CONFIG_IS_LOG             "is_log"
 513 #define ZPOOL_CONFIG_L2CACHE            "l2cache"
 514 #define ZPOOL_CONFIG_HOLE_ARRAY         "hole_array"
 515 #define ZPOOL_CONFIG_VDEV_CHILDREN      "vdev_children"
 516 #define ZPOOL_CONFIG_IS_HOLE            "is_hole"
 517 #define ZPOOL_CONFIG_DDT_HISTOGRAM      "ddt_histogram"
 518 #define ZPOOL_CONFIG_DDT_OBJ_STATS      "ddt_object_stats"
 519 #define ZPOOL_CONFIG_DDT_STATS          "ddt_stats"
 520 #define ZPOOL_CONFIG_SPLIT              "splitcfg"
 521 #define ZPOOL_CONFIG_ORIG_GUID          "orig_guid"
 522 #define ZPOOL_CONFIG_SPLIT_GUID         "split_guid"
 523 #define ZPOOL_CONFIG_SPLIT_LIST         "guid_list"
 524 #define ZPOOL_CONFIG_REMOVING           "removing"
 525 #define ZPOOL_CONFIG_RESILVER_TXG       "resilver_txg"
 525 #define ZPOOL_CONFIG_RESILVERING        "resilvering"
 526 #define ZPOOL_CONFIG_COMMENT            "comment"
 527 #define ZPOOL_CONFIG_SUSPENDED          "suspended"     /* not stored on disk */
 528 #define ZPOOL_CONFIG_TIMESTAMP          "timestamp"     /* not stored on disk */
 529 #define ZPOOL_CONFIG_BOOTFS             "bootfs"        /* not stored on disk */
 530 #define ZPOOL_CONFIG_MISSING_DEVICES    "missing_vdevs" /* not stored on disk */
 531 #define ZPOOL_CONFIG_LOAD_INFO          "load_info"     /* not stored on disk */
 532 #define ZPOOL_CONFIG_REWIND_INFO        "rewind_info"   /* not stored on disk */
 533 #define ZPOOL_CONFIG_UNSUP_FEAT         "unsup_feat"    /* not stored on disk */
 534 #define ZPOOL_CONFIG_ENABLED_FEAT       "enabled_feat"  /* not stored on disk */
 535 #define ZPOOL_CONFIG_CAN_RDONLY         "can_rdonly"    /* not stored on disk */
 536 #define ZPOOL_CONFIG_FEATURES_FOR_READ  "features_for_read"
 537 #define ZPOOL_CONFIG_FEATURE_STATS      "feature_stats" /* not stored on disk */
```

```
538 /*
539  * The persistent vdev state is stored as separate values rather than a single
540  * 'vdev_state' entry.  This is because a device can be in multiple states, such
541  * as offline and degraded.
542  */
543 #define ZPOOL_CONFIG_OFFLINE            "offline"
544 #define ZPOOL_CONFIG_FAULTED            "faulted"
545 #define ZPOOL_CONFIG_DEGRADED           "degraded"
546 #define ZPOOL_CONFIG_REMOVED            "removed"
547 #define ZPOOL_CONFIG_FRU                "fru"
548 #define ZPOOL_CONFIG_AUX_STATE          "aux_state"

550 /* Rewind policy parameters */
551 #define ZPOOL_REWIND_POLICY             "rewind-policy"
552 #define ZPOOL_REWIND_REQUEST            "rewind-request"
553 #define ZPOOL_REWIND_REQUEST_TXG        "rewind-request-txg"
554 #define ZPOOL_REWIND_META_THRESH        "rewind-meta-thresh"
555 #define ZPOOL_REWIND_DATA_THRESH        "rewind-data-thresh"

557 /* Rewind data discovered */
558 #define ZPOOL_CONFIG_LOAD_TIME          "rewind_txg_ts"
559 #define ZPOOL_CONFIG_LOAD_DATA_ERRORS   "verify_data_errors"
560 #define ZPOOL_CONFIG_REWIND_TIME        "seconds_of_rewind"

562 #define VDEV_TYPE_ROOT                  "root"
563 #define VDEV_TYPE_MIRROR                "mirror"
564 #define VDEV_TYPE_REPLACING             "replacing"
565 #define VDEV_TYPE_RAIDZ                 "raidz"
566 #define VDEV_TYPE_DISK                  "disk"
567 #define VDEV_TYPE_FILE                  "file"
568 #define VDEV_TYPE_MISSING               "missing"
569 #define VDEV_TYPE_HOLE                  "hole"
570 #define VDEV_TYPE_SPARE                 "spare"
571 #define VDEV_TYPE_LOG                   "log"
572 #define VDEV_TYPE_L2CACHE               "l2cache"

574 /*
575  * This is needed in userland to report the minimum necessary device size.
576  */
577 #define SPA_MINDEVSIZE          (64ULL << 20)

579 /*
580  * The location of the pool configuration repository, shared between kernel and
581  * userland.
582  */
583 #define ZPOOL_CACHE             "/etc/zfs/zpool.cache"

585 /*
586  * vdev states are ordered from least to most healthy.
587  * A vdev that's CANT_OPEN or below is considered unusable.
588  */
589 typedef enum vdev_state {
590         VDEV_STATE_UNKNOWN = 0, /* Uninitialized vdev                   */
591         VDEV_STATE_CLOSED,      /* Not currently open                   */
592         VDEV_STATE_OFFLINE,     /* Not allowed to open                  */
593         VDEV_STATE_REMOVED,     /* Explicitly removed from system       */
594         VDEV_STATE_CANT_OPEN,   /* Tried to open, but failed            */
595         VDEV_STATE_FAULTED,     /* External request to fault device     */
596         VDEV_STATE_DEGRADED,    /* Replicated vdev with unhealthy kids  */
597         VDEV_STATE_HEALTHY      /* Presumed good                        */
598 } vdev_state_t;
```

_____***unchanged_portion_omitted_***