```
**********************************************************
  160285 Thu Aug  1 22:39:34 2013
new/usr/src/cmd/ztest/ztest.c
3949 ztest fault injection should avoid resilvering devices
3950 ztest: deadman fires when we're doing a scan
3951 ztest hang when running dedup test
3952 ztest: ztest_reguid test and ztest_fault_inject don't place nice together
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 185 extern uint64_t metaslab_gang_bang;
 186 extern uint64_t metaslab_df_alloc_threshold;
 187 extern uint64_t zfs_deadman_synctime;

 189 static ztest_shared_opts_t *ztest_shared_opts;
 190 static ztest_shared_opts_t ztest_opts;

 192 typedef struct ztest_shared_ds {
 193         uint64_t        zd_seq;
 194 } ztest_shared_ds_t;
______unchanged_portion_omitted_

 302 static ztest_shared_callstate_t *ztest_shared_callstate;
 303 #define ZTEST_GET_SHARED_CALLSTATE(c) (&ztest_shared_callstate[c])

 305 /*
 306  * Note: these aren't static because we want dladdr() to work.
 307  */
 308 ztest_func_t ztest_dmu_read_write;
 309 ztest_func_t ztest_dmu_write_parallel;
 310 ztest_func_t ztest_dmu_object_alloc_free;
 311 ztest_func_t ztest_dmu_commit_callbacks;
 312 ztest_func_t ztest_zap;
 313 ztest_func_t ztest_zap_parallel;
 314 ztest_func_t ztest_zil_commit;
 315 ztest_func_t ztest_zil_remount;
 316 ztest_func_t ztest_dmu_read_write_zcopy;
 317 ztest_func_t ztest_dmu_objset_create_destroy;
 318 ztest_func_t ztest_dmu_prealloc;
 319 ztest_func_t ztest_fzap;
 320 ztest_func_t ztest_dmu_snapshot_create_destroy;
 321 ztest_func_t ztest_dsl_prop_get_set;
 322 ztest_func_t ztest_spa_prop_get_set;
 323 ztest_func_t ztest_spa_create_destroy;
 324 ztest_func_t ztest_fault_inject;
 325 ztest_func_t ztest_ddt_repair;
 326 ztest_func_t ztest_dmu_snapshot_hold;
 327 ztest_func_t ztest_spa_rename;
 328 ztest_func_t ztest_scrub;
 329 ztest_func_t ztest_dsl_dataset_promote_busy;
 330 ztest_func_t ztest_vdev_attach_detach;
 331 ztest_func_t ztest_vdev_LUN_growth;
 332 ztest_func_t ztest_vdev_add_remove;
 333 ztest_func_t ztest_vdev_aux_add_remove;
 334 ztest_func_t ztest_split_pool;
 335 ztest_func_t ztest_reguid;
 336 ztest_func_t ztest_spa_upgrade;

 338 uint64_t zopt_always = 0ULL * NANOSEC;          /* all the time */
 339 uint64_t zopt_incessant = 1ULL * NANOSEC / 10;  /* every 1/10 second */
 340 uint64_t zopt_often = 1ULL * NANOSEC;           /* every second */
 341 uint64_t zopt_sometimes = 10ULL * NANOSEC;      /* every 10 seconds */
 342 uint64_t zopt_rarely = 60ULL * NANOSEC;         /* every 60 seconds */
```



```
 344 ztest_info_t ztest_info[] = {
 345         { ztest_dmu_read_write,                 1,      &zopt_always    },
 346         { ztest_dmu_write_parallel,             10,     &zopt_always    },
 347         { ztest_dmu_object_alloc_free,          1,      &zopt_always    },
 348         { ztest_dmu_commit_callbacks,           1,      &zopt_always    },
 349         { ztest_zap,                            30,     &zopt_always    },
 350         { ztest_zap_parallel,                   100,    &zopt_always    },
 351         { ztest_split_pool,                     1,      &zopt_always    },
 352         { ztest_zil_commit,                     1,      &zopt_incessant },
 353         { ztest_zil_remount,                    1,      &zopt_sometimes },
 354         { ztest_dmu_read_write_zcopy,           1,      &zopt_often     },
 355         { ztest_dmu_objset_create_destroy,      1,      &zopt_often     },
 356         { ztest_dsl_prop_get_set,               1,      &zopt_often     },
 357         { ztest_spa_prop_get_set,               1,      &zopt_sometimes },
 358 #if 0
 359         { ztest_dmu_prealloc,                   1,      &zopt_sometimes },
 360 #endif
 361         { ztest_fzap,                           1,      &zopt_sometimes },
 362         { ztest_dmu_snapshot_create_destroy,    1,      &zopt_sometimes },
 363         { ztest_spa_create_destroy,             1,      &zopt_sometimes },
 364         { ztest_fault_inject,                   1,      &zopt_sometimes },
 365         { ztest_ddt_repair,                     1,      &zopt_sometimes },
 366         { ztest_dmu_snapshot_hold,              1,      &zopt_sometimes },
 367         { ztest_reguid,                         1,      &zopt_rarely    },
 366         { ztest_reguid,                         1,      &zopt_sometimes },
 368         { ztest_spa_rename,                     1,      &zopt_rarely    },
 369         { ztest_scrub,                          1,      &zopt_rarely    },
 370         { ztest_spa_upgrade,                    1,      &zopt_rarely    },
 371         { ztest_dsl_dataset_promote_busy,       1,      &zopt_rarely    },
 372         { ztest_vdev_attach_detach,             1,      &zopt_sometimes },
 373         { ztest_vdev_LUN_growth,                1,      &zopt_rarely    },
 374         { ztest_vdev_add_remove,                1,
 375           &ztest_opts.zo_vdevtime                                       },
 376         { ztest_vdev_aux_add_remove,            1,
 377           &ztest_opts.zo_vdevtime                                       },
 378 };
______unchanged_portion_omitted_

4724 /*
4725  * Inject random faults into the on-disk data.
4726  */
4727 /* ARGSUSED */
4728 void
4729 ztest_fault_inject(ztest_ds_t *zd, uint64_t id)
4730 {
4731         ztest_shared_t *zs = ztest_shared;
4732         spa_t *spa = ztest_spa;
4733         int fd;
4734         uint64_t offset;
4735         uint64_t leaves;
4736         uint64_t bad = 0x1990c0ffeedecade;
4737         uint64_t top, leaf;
4738         char path0[MAXPATHLEN];
4739         char pathrand[MAXPATHLEN];
4740         size_t fsize;
4741         int bshift = SPA_MAXBLOCKSHIFT + 2;     /* don't scrog all labels */
4742         int iters = 1000;
4743         int maxfaults;
4744         int mirror_save;
4745         vdev_t *vd0 = NULL;
4746         uint64_t guid0 = 0;
4747         boolean_t islog = B_FALSE;

4749         VERIFY(mutex_lock(&ztest_vdev_lock) == 0);
4750         maxfaults = MAXFAULTS();
4751         leaves = MAX(zs->zs_mirrors, 1) * ztest_opts.zo_raidz;
```

```
4752         mirror_save = zs->zs_mirrors;
4753         VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);

4755         ASSERT(leaves >= 1);

4757         /*
4758          * Grab the name lock as reader. There are some operations
4759          * which don't like to have their vdevs changed while
4760          * they are in progress (i.e. spa_change_guid). Those
4761          * operations will have grabbed the name lock as writer.
4762          */
4763         (void) rw_rdlock(&ztest_name_lock);

4765         /*
4766          * We need SCL_STATE here because we're going to look at vd0->vdev_tsd.
4767          */
4768         spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);

4770         if (ztest_random(2) == 0) {
4771                 /*
4772                  * Inject errors on a normal data device or slog device.
4773                  */
4774                 top = ztest_random_vdev_top(spa, B_TRUE);
4775                 leaf = ztest_random(leaves) + zs->zs_splits;

4777                 /*
4778                  * Generate paths to the first leaf in this top-level vdev,
4779                  * and to the random leaf we selected.  We'll induce transient
4780                  * write failures and random online/offline activity on leaf 0,
4781                  * and we'll write random garbage to the randomly chosen leaf.
4782                  */
4783                 (void) snprintf(path0, sizeof (path0), ztest_dev_template,
4784                     ztest_opts.zo_dir, ztest_opts.zo_pool,
4785                     top * leaves + zs->zs_splits);
4786                 (void) snprintf(pathrand, sizeof (pathrand), ztest_dev_template,
4787                     ztest_opts.zo_dir, ztest_opts.zo_pool,
4788                     top * leaves + leaf);

4790                 vd0 = vdev_lookup_by_path(spa->spa_root_vdev, path0);
4791                 if (vd0 != NULL && vd0->vdev_top->vdev_islog)
4792                         islog = B_TRUE;

4785                 if (vd0 != NULL && maxfaults != 1) {
4794                 /*
4795                  * If the top-level vdev needs to be resilvered
4796                  * then we only allow faults on the device that is
4797                  * resilvering.
4798                  */
4799                 if (vd0 != NULL && maxfaults != 1 &&
4800                     (!vdev_resilver_needed(vd0->vdev_top, NULL, NULL) ||
4801                     vd0->vdev_resilvering)) {
4802                         /*
4803                          * Make vd0 explicitly claim to be unreadable,
4804                          * or unwriteable, or reach behind its back
4805                          * and close the underlying fd.  We can do this if
4806                          * maxfaults == 0 because we'll fail and reexecute,
4807                          * and we can do it if maxfaults >= 2 because we'll
4808                          * have enough redundancy.  If maxfaults == 1, the
4809                          * combination of this with injection of random data
4810                          * corruption below exceeds the pool's fault tolerance.
4811                          */
4812                         vdev_file_t *vf = vd0->vdev_tsd;

4814                         if (vf != NULL && ztest_random(3) == 0) {
4815                                 (void) close(vf->vf_vnode->v_fd);
4816                                 vf->vf_vnode->v_fd = -1;
```



```
4817                         } else if (ztest_random(2) == 0) {
4818                                 vd0->vdev_cant_read = B_TRUE;
4819                         } else {
4820                                 vd0->vdev_cant_write = B_TRUE;
4821                         }
4822                         guid0 = vd0->vdev_guid;
4823                 }
4824         } else {
4825                 /*
4826                  * Inject errors on an l2cache device.
4827                  */
4828                 spa_aux_vdev_t *sav = &spa->spa_l2cache;

4830                 if (sav->sav_count == 0) {
4831                         spa_config_exit(spa, SCL_STATE, FTAG);
4832                         (void) rw_unlock(&ztest_name_lock);
4833                         return;
4834                 }
4835                 vd0 = sav->sav_vdevs[ztest_random(sav->sav_count)];
4836                 guid0 = vd0->vdev_guid;
4837                 (void) strcpy(path0, vd0->vdev_path);
4838                 (void) strcpy(pathrand, vd0->vdev_path);

4840                 leaf = 0;
4841                 leaves = 1;
4842                 maxfaults = INT_MAX;    /* no limit on cache devices */
4843         }

4845         spa_config_exit(spa, SCL_STATE, FTAG);
4846         (void) rw_unlock(&ztest_name_lock);

4848         /*
4849          * If we can tolerate two or more faults, or we're dealing
4850          * with a slog, randomly online/offline vd0.
4851          */
4852         if ((maxfaults >= 2 || islog) && guid0 != 0) {
4853                 if (ztest_random(10) < 6) {
4854                         int flags = (ztest_random(2) == 0 ?
4855                             ZFS_OFFLINE_TEMPORARY : 0);

4857                         /*
4858                          * We have to grab the zs_name_lock as writer to
4859                          * prevent a race between offlining a slog and
4860                          * destroying a dataset. Offlining the slog will
4861                          * grab a reference on the dataset which may cause
4862                          * dmu_objset_destroy() to fail with EBUSY thus
4863                          * leaving the dataset in an inconsistent state.
4864                          */
4865                         if (islog)
4866                                 (void) rw_wrlock(&ztest_name_lock);

4868                         VERIFY(vdev_offline(spa, guid0, flags) != EBUSY);

4870                         if (islog)
4871                                 (void) rw_unlock(&ztest_name_lock);
4872                 } else {
4873                         /*
4874                          * Ideally we would like to be able to randomly
4875                          * call vdev_[on|off]line without holding locks
4876                          * to force unpredictable failures but the side
4877                          * effects of vdev_[on|off]line prevent us from
4878                          * doing so. We grab the ztest_vdev_lock here to
4879                          * prevent a race between injection testing and
4880                          * aux_vdev removal.
4881                          */
4882                         VERIFY(mutex_lock(&ztest_vdev_lock) == 0);
```

```
4883                        (void) vdev_online(spa, guid0, 0, NULL);
4884                        VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);
4885                }
4886        }

4888        if (maxfaults == 0)
4889                return;

4891        /*
4892         * We have at least single-fault tolerance, so inject data corruption.
4893         */
4894        fd = open(pathrand, O_RDWR);

4896        if (fd == -1)   /* we hit a gap in the device namespace */
4897                return;

4899        fsize = lseek(fd, 0, SEEK_END);

4901        while (--iters != 0) {
4902                offset = ztest_random(fsize / (leaves << bshift)) *
4903                    (leaves << bshift) + (leaf << bshift) +
4904                    (ztest_random(1ULL << (bshift - 1)) & -8ULL);

4906                if (offset >= fsize)
4907                        continue;

4909                VERIFY(mutex_lock(&ztest_vdev_lock) == 0);
4910                if (mirror_save != zs->zs_mirrors) {
4911                        VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);
4912                        (void) close(fd);
4913                        return;
4914                }

4916                if (pwrite(fd, &bad, sizeof (bad), offset) != sizeof (bad))
4917                        fatal(1, "can't inject bad word at 0x%llx in %s",
4918                            offset, pathrand);

4920                VERIFY(mutex_unlock(&ztest_vdev_lock) == 0);

4922                if (ztest_opts.zo_verbose >= 7)
4923                        (void) printf("injected bad word into %s,"
4924                            " offset 0x%llx\n", pathrand, (u_longlong_t)offset);
4925        }

4927        (void) close(fd);
4928 }
```

*unchanged_portion_omitted_*

```
5307 static void *
5308 ztest_deadman_thread(void *arg)
5309 {
5310        ztest_shared_t *zs = arg;
5311        spa_t *spa = ztest_spa;
5312        hrtime_t delta, total = 0;
5293        int grace = 300;
5294        hrtime_t delta;

5314        for (;;) {
5315                delta = (zs->zs_thread_stop - zs->zs_thread_start) /
5316                    NANOSEC + zfs_deadman_synctime;
5296        delta = (zs->zs_thread_stop - zs->zs_thread_start) / NANOSEC + grace;

5318                (void) poll(NULL, 0, (int)(1000 * delta));

5320                /*
5321                 * If the pool is suspended then fail immediately. Otherwise,
```



```
5322                 * check to see if the pool is making any progress. If
5323                 * vdev_deadman() discovers that there hasn't been any recent
5324                 * I/Os then it will end up aborting the tests.
5325                 */
5326                if (spa_suspended(spa)) {
5327                        fatal(0, "aborting test after %llu seconds because "
5328                            "pool has transitioned to a suspended state.",
5329                            zfs_deadman_synctime);
5330                        return (NULL);
5331                }
5332                vdev_deadman(spa->spa_root_vdev);
5300        fatal(0, "failed to complete within %d seconds of deadline", grace);

5334                total += zfs_deadman_synctime;
5335                (void) printf("ztest has been running for %lld seconds\n",
5336                    total);
5337        }
5302        return (NULL);
5338 }
```

*unchanged_portion_omitted_*

```
6042 int
6043 main(int argc, char **argv)
6044 {
6045        int kills = 0;
6046        int iters = 0;
6047        int older = 0;
6048        int newer = 0;
6049        ztest_shared_t *zs;
6050        ztest_info_t *zi;
6051        ztest_shared_callstate_t *zc;
6052        char timebuf[100];
6053        char numbuf[6];
6054        spa_t *spa;
6055        char *cmd;
6056        boolean_t hasalt;
6057        char *fd_data_str = getenv("ZTEST_FD_DATA");

6059        (void) setvbuf(stdout, NULL, _IOLBF, 0);

6061        dprintf_setup(&argc, argv);
6062        zfs_deadman_synctime = 300;

6064        ztest_fd_rand = open("/dev/urandom", O_RDONLY);
6065        ASSERT3S(ztest_fd_rand, >=, 0);

6067        if (!fd_data_str) {
6068                process_options(argc, argv);

6070                setup_data_fd();
6071                setup_hdr();
6072                setup_data();
6073                bcopy(&ztest_opts, ztest_shared_opts,
6074                    sizeof (*ztest_shared_opts));
6075        } else {
6076                ztest_fd_data = atoi(fd_data_str);
6077                setup_data();
6078                bcopy(ztest_shared_opts, &ztest_opts, sizeof (ztest_opts));
6079        }
6080        ASSERT3U(ztest_opts.zo_datasets, ==, ztest_shared_hdr->zh_ds_count);

6082        /* Override location of zpool.cache */
6083        VERIFY3U(asprintf((char **)&spa_config_path, "%s/zpool.cache",
6084            ztest_opts.zo_dir), !=, -1);

6086        ztest_ds = umem_alloc(ztest_opts.zo_datasets * sizeof (ztest_ds_t),
```

```
6087             UMEM_NOFAIL);
6088         zs = ztest_shared;

6090         if (fd_data_str) {
6091                 metaslab_gang_bang = ztest_opts.zo_metaslab_gang_bang;
6092                 metaslab_df_alloc_threshold =
6093                     zs->zs_metaslab_df_alloc_threshold;

6095                 if (zs->zs_do_init)
6096                         ztest_run_init();
6097                 else
6098                         ztest_run(zs);
6099                 exit(0);
6100         }

6102         hasalt = (strlen(ztest_opts.zo_alt_ztest) != 0);

6104         if (ztest_opts.zo_verbose >= 1) {
6105                 (void) printf("%llu vdevs, %d datasets, %d threads,"
6106                     " %llu seconds...\n",
6107                     (u_longlong_t)ztest_opts.zo_vdevs,
6108                     ztest_opts.zo_datasets,
6109                     ztest_opts.zo_threads,
6110                     (u_longlong_t)ztest_opts.zo_time);
6111         }

6113         cmd = umem_alloc(MAXNAMELEN, UMEM_NOFAIL);
6114         (void) strlcpy(cmd, getexecname(), MAXNAMELEN);

6116         zs->zs_do_init = B_TRUE;
6117         if (strlen(ztest_opts.zo_alt_ztest) != 0) {
6118                 if (ztest_opts.zo_verbose >= 1) {
6119                         (void) printf("Executing older ztest for "
6120                             "initialization: %s\n", ztest_opts.zo_alt_ztest);
6121                 }
6122                 VERIFY(!exec_child(ztest_opts.zo_alt_ztest,
6123                     ztest_opts.zo_alt_libpath, B_FALSE, NULL));
6124         } else {
6125                 VERIFY(!exec_child(NULL, NULL, B_FALSE, NULL));
6126         }
6127         zs->zs_do_init = B_FALSE;

6129         zs->zs_proc_start = gethrtime();
6130         zs->zs_proc_stop = zs->zs_proc_start + ztest_opts.zo_time * NANOSEC;

6132         for (int f = 0; f < ZTEST_FUNCS; f++) {
6133                 zi = &ztest_info[f];
6134                 zc = ZTEST_GET_SHARED_CALLSTATE(f);
6135                 if (zs->zs_proc_start + zi->zi_interval[0] > zs->zs_proc_stop)
6136                         zc->zc_next = UINT64_MAX;
6137                 else
6138                         zc->zc_next = zs->zs_proc_start +
6139                             ztest_random(2 * zi->zi_interval[0] + 1);
6140         }

6142         /*
6143          * Run the tests in a loop.  These tests include fault injection
6144          * to verify that self-healing data works, and forced crashes
6145          * to verify that we never lose on-disk consistency.
6146          */
6147         while (gethrtime() < zs->zs_proc_stop) {
6148                 int status;
6149                 boolean_t killed;

6151                 /*
6152                  * Initialize the workload counters for each function.
```



```
6153                  */
6154                 for (int f = 0; f < ZTEST_FUNCS; f++) {
6155                         zc = ZTEST_GET_SHARED_CALLSTATE(f);
6156                         zc->zc_count = 0;
6157                         zc->zc_time = 0;
6158                 }

6160                 /* Set the allocation switch size */
6161                 zs->zs_metaslab_df_alloc_threshold =
6162                     ztest_random(zs->zs_metaslab_sz / 4) + 1;

6164                 if (!hasalt || ztest_random(2) == 0) {
6165                         if (hasalt && ztest_opts.zo_verbose >= 1) {
6166                                 (void) printf("Executing newer ztest: %s\n",
6167                                     cmd);
6168                         }
6169                         newer++;
6170                         killed = exec_child(cmd, NULL, B_TRUE, &status);
6171                 } else {
6172                         if (hasalt && ztest_opts.zo_verbose >= 1) {
6173                                 (void) printf("Executing older ztest: %s\n",
6174                                     ztest_opts.zo_alt_ztest);
6175                         }
6176                         older++;
6177                         killed = exec_child(ztest_opts.zo_alt_ztest,
6178                             ztest_opts.zo_alt_libpath, B_TRUE, &status);
6179                 }

6181                 if (killed)
6182                         kills++;
6183                 iters++;

6185                 if (ztest_opts.zo_verbose >= 1) {
6186                         hrtime_t now = gethrtime();

6188                         now = MIN(now, zs->zs_proc_stop);
6189                         print_time(zs->zs_proc_stop - now, timebuf);
6190                         nicenum(zs->zs_space, numbuf);

6192                         (void) printf("Pass %3d, %8s, %3llu ENOSPC, "
6193                             "%4.1f%% of %5s used, %3.0f%% done, %8s to go\n",
6194                             iters,
6195                             WIFEXITED(status) ? "Complete" : "SIGKILL",
6196                             (u_longlong_t)zs->zs_enospc_count,
6197                             100.0 * zs->zs_alloc / zs->zs_space,
6198                             numbuf,
6199                             100.0 * (now - zs->zs_proc_start) /
6200                             (ztest_opts.zo_time * NANOSEC), timebuf);
6201                 }

6203                 if (ztest_opts.zo_verbose >= 2) {
6204                         (void) printf("\nWorkload summary:\n\n");
6205                         (void) printf("%7s %9s   %s\n",
6206                             "Calls", "Time", "Function");
6207                         (void) printf("%7s %9s   %s\n",
6208                             "-----", "----", "--------");
6209                         for (int f = 0; f < ZTEST_FUNCS; f++) {
6210                                 Dl_info dli;

6212                                 zi = &ztest_info[f];
6213                                 zc = ZTEST_GET_SHARED_CALLSTATE(f);
6214                                 print_time(zc->zc_time, timebuf);
6215                                 (void) dladdr((void *)zi->zi_func, &dli);
6216                                 (void) printf("%7llu %9s   %s\n",
6217                                     (u_longlong_t)zc->zc_count, timebuf,
6218                                     dli.dli_sname);
```

```
6219                            }
6220                            (void) printf("\n");
6221                    }

6223                    /*
6224                     * It's possible that we killed a child during a rename test,
6225                     * in which case we'll have a 'ztest_tmp' pool lying around
6226                     * instead of 'ztest'.  Do a blind rename in case this happened.
6227                     */
6228                    kernel_init(FREAD);
6229                    if (spa_open(ztest_opts.zo_pool, &spa, FTAG) == 0) {
6230                            spa_close(spa, FTAG);
6231                    } else {
6232                            char tmpname[MAXNAMELEN];
6233                            kernel_fini();
6234                            kernel_init(FREAD | FWRITE);
6235                            (void) snprintf(tmpname, sizeof (tmpname), "%s_tmp",
6236                                ztest_opts.zo_pool);
6237                            (void) spa_rename(tmpname, ztest_opts.zo_pool);
6238                    }
6239                    kernel_fini();

6241                    ztest_run_zdb(ztest_opts.zo_pool);
6242            }

6244            if (ztest_opts.zo_verbose >= 1) {
6245                    if (hasalt) {
6246                            (void) printf("%d runs of older ztest: %s\n", older,
6247                                ztest_opts.zo_alt_ztest);
6248                            (void) printf("%d runs of newer ztest: %s\n", newer,
6249                                cmd);
6250                    }
6251                    (void) printf("%d killed, %d completed, %.0f%% kill rate\n",
6252                        kills, iters - kills, (100.0 * kills) / MAX(1, iters));
6253            }

6255            umem_free(cmd, MAXNAMELEN);

6257            return (0);
6258 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    1937 Thu Aug  1 22:39:37 2013
new/usr/src/lib/libzpool/common/llib-lzpool
3949 ztest fault injection should avoid resilvering devices
3950 ztest: deadman fires when we're doing a scan
3951 ztest hang when running dedup test
3952 ztest: ztest_reguid test and ztest_fault_inject don't place nice together
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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

/* LINTLIBRARY */
/* PROTOLIB1 */

#include <sys/zfs_context.h>
#include <sys/list.h>
#include <sys/list_impl.h>
#include <sys/sysmacros.h>
#include <sys/debug.h>
#include <sys/dmu_traverse.h>
#include <sys/dnode.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_dataset.h>
#include <sys/spa.h>
#include <sys/spa_impl.h>
#include <sys/space_map.h>
#include <sys/vdev.h>
#include <sys/vdev_impl.h>
#include <sys/zap.h>
#include <sys/zio.h>
#include <sys/zio_compress.h>
#include <sys/zil.h>
#include <sys/bplist.h>
#include <sys/zfs_znode.h>
#include <sys/arc.h>
#include <sys/dbuf.h>
#include <sys/zio_checksum.h>
#include <sys/ddt.h>
```



```
#include <sys/sa.h>
#include <sys/zfs_sa.h>
#include <sys/zfeature.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_destroy.h>
#include <sys/dsl_userhold.h>

extern uint64_t metaslab_gang_bang;
extern uint64_t metaslab_df_alloc_threshold;
extern boolean_t zfeature_checks_disable;
extern uint64_t zfs_deadman_synctime;
```

```
**********************************************************
  175636 Thu Aug  1 22:39:39 2013
new/usr/src/uts/common/fs/zfs/spa.c
3949 ztest fault injection should avoid resilvering devices
3950 ztest: deadman fires when we're doing a scan
3951 ztest hang when running dedup test
3952 ztest: ztest_reguid test and ztest_fault_inject don't place nice together
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 745 /*
 746  * Change the GUID for the pool.  This is done so that we can later
 747  * re-import a pool built from a clone of our own vdevs.  We will modify
 748  * the root vdev's guid, our own pool guid, and then mark all of our
 749  * vdevs dirty.  Note that we must make sure that all our vdevs are
 750  * online when we do this, or else any vdevs that weren't present
 751  * would be orphaned from our pool.  We are also going to issue a
 752  * sysevent to update any watchers.
 753  */
 754 int
 755 spa_change_guid(spa_t *spa)
 756 {
 757         int error;
 758         uint64_t guid;

 760         mutex_enter(&spa->spa_vdev_top_lock);
 761         mutex_enter(&spa_namespace_lock);
 762         guid = spa_generate_guid(NULL);

 764         error = dsl_sync_task(spa->spa_name, spa_change_guid_check,
 765             spa_change_guid_sync, &guid, 5);

 767         if (error == 0) {
 768                 spa_config_sync(spa, B_FALSE, B_TRUE);
 769                 spa_event_notify(spa, NULL, ESC_ZFS_POOL_REGUID);
 770         }

 772         mutex_exit(&spa_namespace_lock);
 773         mutex_exit(&spa->spa_vdev_top_lock);

 775         return (error);
 776 }
______unchanged_portion_omitted_

4525 /*
4526  * Detach a device from a mirror or replacing vdev.
4527  *
4528  * If 'replace_done' is specified, only detach if the parent
4529  * is a replacing vdev.
4530  */
4531 int
4532 spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid, int replace_done)
4533 {
4534         uint64_t txg;
4535         int error;
4536         vdev_t *rvd = spa->spa_root_vdev;
4537         vdev_t *vd, *pvd, *cvd, *tvd;
4538         boolean_t unspare = B_FALSE;
4539         uint64_t unspare_guid = 0;
4540         char *vdpath;

4542         ASSERT(spa_writeable(spa));

4544         txg = spa_vdev_enter(spa);
```



```
4546         vd = spa_lookup_by_guid(spa, guid, B_FALSE);

4548         if (vd == NULL)
4549                 return (spa_vdev_exit(spa, NULL, txg, ENODEV));

4551         if (!vd->vdev_ops->vdev_op_leaf)
4552                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4554         pvd = vd->vdev_parent;

4556         /*
4557          * If the parent/child relationship is not as expected, don't do it.
4558          * Consider M(A,R(B,C)) -- that is, a mirror of A with a replacing
4559          * vdev that's replacing B with C.  The user's intent in replacing
4560          * is to go from M(A,B) to M(A,C).  If the user decides to cancel
4561          * the replace by detaching C, the expected behavior is to end up
4562          * M(A,B).  But suppose that right after deciding to detach C,
4563          * the replacement of B completes.  We would have M(A,C), and then
4564          * ask to detach C, which would leave us with just A -- not what
4565          * the user wanted.  To prevent this, we make sure that the
4566          * parent/child relationship hasn't changed -- in this example,
4567          * that C's parent is still the replacing vdev R.
4568          */
4569         if (pvd->vdev_guid != pguid && pguid != 0)
4570                 return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4572         /*
4573          * Only 'replacing' or 'spare' vdevs can be replaced.
4574          */
4575         if (replace_done && pvd->vdev_ops != &vdev_replacing_ops &&
4576             pvd->vdev_ops != &vdev_spare_ops)
4577                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4579         ASSERT(pvd->vdev_ops != &vdev_spare_ops ||
4580             spa_version(spa) >= SPA_VERSION_SPARES);

4582         /*
4583          * Only mirror, replacing, and spare vdevs support detach.
4584          */
4585         if (pvd->vdev_ops != &vdev_replacing_ops &&
4586             pvd->vdev_ops != &vdev_mirror_ops &&
4587             pvd->vdev_ops != &vdev_spare_ops)
4588                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4590         /*
4591          * If this device has the only valid copy of some data,
4592          * we cannot safely detach it.
4593          */
4594         if (vdev_dtl_required(vd))
4595                 return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4597         ASSERT(pvd->vdev_children >= 2);

4599         /*
4600          * If we are detaching the second disk from a replacing vdev, then
4601          * check to see if we changed the original vdev's path to have "/old"
4602          * at the end in spa_vdev_attach().  If so, undo that change now.
4603          */
4604         if (pvd->vdev_ops == &vdev_replacing_ops && vd->vdev_id > 0 &&
4605             vd->vdev_path != NULL) {
4606                 size_t len = strlen(vd->vdev_path);

4608                 for (int c = 0; c < pvd->vdev_children; c++) {
4609                         cvd = pvd->vdev_child[c];
```

```
4611                        if (cvd == vd || cvd->vdev_path == NULL)
4612                                continue;
4613
4614                        if (strncmp(cvd->vdev_path, vd->vdev_path, len) == 0 &&
4615                            strcmp(cvd->vdev_path + len, "/old") == 0) {
4616                                spa_strfree(cvd->vdev_path);
4617                                cvd->vdev_path = spa_strdup(vd->vdev_path);
4618                                break;
4619                        }
4620                }
4621        }
4622
4623        /*
4624         * If we are detaching the original disk from a spare, then it implies
4625         * that the spare should become a real disk, and be removed from the
4626         * active spare list for the pool.
4627         */
4628        if (pvd->vdev_ops == &vdev_spare_ops &&
4629            vd->vdev_id == 0 &&
4630            pvd->vdev_child[pvd->vdev_children - 1]->vdev_isspare)
4631                unspare = B_TRUE;
4632
4633        /*
4634         * Erase the disk labels so the disk can be used for other things.
4635         * This must be done after all other error cases are handled,
4636         * but before we disembowel vd (so we can still do I/O to it).
4637         * But if we can't do it, don't treat the error as fatal --
4638         * it may be that the unwritability of the disk is the reason
4639         * it's being detached!
4640         */
4641        error = vdev_label_init(vd, 0, VDEV_LABEL_REMOVE);
4642
4643        /*
4644         * Remove vd from its parent and compact the parent's children.
4645         */
4646        vdev_remove_child(pvd, vd);
4647        vdev_compact_children(pvd);
4648
4649        /*
4650         * Remember one of the remaining children so we can get tvd below.
4651         */
4652        cvd = pvd->vdev_child[pvd->vdev_children - 1];
4653
4654        /*
4655         * If we need to remove the remaining child from the list of hot spares,
4656         * do it now, marking the vdev as no longer a spare in the process.
4657         * We must do this before vdev_remove_parent(), because that can
4658         * change the GUID if it creates a new toplevel GUID.  For a similar
4659         * reason, we must remove the spare now, in the same txg as the detach;
4660         * otherwise someone could attach a new sibling, change the GUID, and
4661         * the subsequent attempt to spa_vdev_remove(unspare_guid) would fail.
4662         */
4663        if (unspare) {
4664                ASSERT(cvd->vdev_isspare);
4665                spa_spare_remove(cvd);
4666                unspare_guid = cvd->vdev_guid;
4667                (void) spa_vdev_remove(spa, unspare_guid, B_TRUE);
4668                cvd->vdev_unspare = B_TRUE;
4669        }
4670
4671        /*
4672         * If the parent mirror/replacing vdev only has one child,
4673         * the parent is no longer needed.  Remove it from the tree.
4674         */
4675        if (pvd->vdev_children == 1) {
4676                if (pvd->vdev_ops == &vdev_spare_ops)
```



```
4677                        cvd->vdev_unspare = B_FALSE;
4678                vdev_remove_parent(cvd);
4677                cvd->vdev_resilvering = B_FALSE;
4679        }
4680
4681
4682        /*
4683         * We don't set tvd until now because the parent we just removed
4684         * may have been the previous top-level vdev.
4685         */
4686        tvd = cvd->vdev_top;
4687        ASSERT(tvd->vdev_parent == rvd);
4688
4689        /*
4690         * Reevaluate the parent vdev state.
4691         */
4692        vdev_propagate_state(cvd);
4693
4694        /*
4695         * If the 'autoexpand' property is set on the pool then automatically
4696         * try to expand the size of the pool. For example if the device we
4697         * just detached was smaller than the others, it may be possible to
4698         * add metaslabs (i.e. grow the pool). We need to reopen the vdev
4699         * first so that we can obtain the updated sizes of the leaf vdevs.
4700         */
4701        if (spa->spa_autoexpand) {
4702                vdev_reopen(tvd);
4703                vdev_expand(tvd, txg);
4704        }
4705
4706        vdev_config_dirty(tvd);
4707
4708        /*
4709         * Mark vd's DTL as dirty in this txg.  vdev_dtl_sync() will see that
4710         * vd->vdev_detached is set and free vd's DTL object in syncing context.
4711         * But first make sure we're not on any *other* txg's DTL list, to
4712         * prevent vd from being accessed after it's freed.
4713         */
4714        vdpath = spa_strdup(vd->vdev_path);
4715        for (int t = 0; t < TXG_SIZE; t++)
4716                (void) txg_list_remove_this(&tvd->vdev_dtl_list, vd, t);
4717        vd->vdev_detached = B_TRUE;
4718        vdev_dirty(tvd, VDD_DTL, vd, txg);
4719
4720        spa_event_notify(spa, vd, ESC_ZFS_VDEV_REMOVE);
4721
4722        /* hang on to the spa before we release the lock */
4723        spa_open_ref(spa, FTAG);
4724
4725        error = spa_vdev_exit(spa, vd, txg, 0);
4726
4727        spa_history_log_internal(spa, "detach", NULL,
4728            "vdev=%s", vdpath);
4729        spa_strfree(vdpath);
4730
4731        /*
4732         * If this was the removal of the original device in a hot spare vdev,
4733         * then we want to go through and remove the device from the hot spare
4734         * list of every other pool.
4735         */
4736        if (unspare) {
4737                spa_t *altspa = NULL;
4738
4739                mutex_enter(&spa_namespace_lock);
4740                while ((altspa = spa_next(altspa)) != NULL) {
4741                        if (altspa->spa_state != POOL_STATE_ACTIVE ||
```

```
4742                            altspa == spa)
4743                                continue;
4744
4745                        spa_open_ref(altspa, FTAG);
4746                        mutex_exit(&spa_namespace_lock);
4747                        (void) spa_vdev_remove(altspa, unspare_guid, B_TRUE);
4748                        mutex_enter(&spa_namespace_lock);
4749                        spa_close(altspa, FTAG);
4750                }
4751                mutex_exit(&spa_namespace_lock);
4752
4753                /* search the rest of the vdevs for spares to remove */
4754                spa_vdev_resilver_done(spa);
4755        }
4756
4757        /* all done with the spa; OK to release */
4758        mutex_enter(&spa_namespace_lock);
4759        spa_close(spa, FTAG);
4760        mutex_exit(&spa_namespace_lock);
4761
4762        return (error);
4763 }
_____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;
5299
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         }
5305
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         }
5312
5313         /*
5314          * Check for a completed replacement.  We always consider the first
5315          * vdev in the list to be the oldest vdev, and the last one to be
5316          * the newest (see spa_vdev_attach() for how that works).  In
5317          * the case where the newest vdev is faulted, we will not automatically
5318          * remove it after a resilver completes.  This is OK as it will require
5319          * user intervention to determine which disk the admin wishes to keep.
5320          */
5321         if (vd->vdev_ops == &vdev_replacing_ops) {
5322                 ASSERT(vd->vdev_children > 1);
5323
5324                 newvd = vd->vdev_child[vd->vdev_children - 1];
5325                 oldvd = vd->vdev_child[0];
5326
5327                 if (vdev_dtl_empty(newvd, DTL_MISSING) &&
5328                     vdev_dtl_empty(newvd, DTL_OUTAGE) &&
5329                     !vdev_dtl_required(oldvd))
5330                         return (oldvd);
5331         }
```



```
5332
5333         /*
5334          * Check for a completed resilver with the 'unspare' flag set.
5335          */
5336         if (vd->vdev_ops == &vdev_spare_ops) {
5337                 vdev_t *first = vd->vdev_child[0];
5338                 vdev_t *last = vd->vdev_child[vd->vdev_children - 1];
5339
5340                 if (last->vdev_unspare) {
5341                         oldvd = first;
5342                         newvd = last;
5343                 } else if (first->vdev_unspare) {
5344                         oldvd = last;
5345                         newvd = first;
5346                 } else {
5347                         oldvd = NULL;
5348                 }
5349
5350                 if (oldvd != NULL &&
5351                     vdev_dtl_empty(newvd, DTL_MISSING) &&
5352                     vdev_dtl_empty(newvd, DTL_OUTAGE) &&
5353                     !vdev_dtl_required(oldvd))
5354                         return (oldvd);
5355
5356                 /*
5357                  * If there are more than two spares attached to a disk,
5358                  * and those spares are not required, then we want to
5359                  * attempt to free them up now so that they can be used
5360                  * by other pools.  Once we're back down to a single
5361                  * disk+spare, we stop removing them.
5362                  */
5363                 if (vd->vdev_children > 2) {
5364                         newvd = vd->vdev_child[1];
5365
5366                         if (newvd->vdev_isspare && last->vdev_isspare &&
5367                             vdev_dtl_empty(last, DTL_MISSING) &&
5368                             vdev_dtl_empty(last, DTL_OUTAGE) &&
5369                             !vdev_dtl_required(newvd))
5370                                 return (newvd);
5371                 }
5372         }
5373
5374         return (NULL);
5375 }
_____unchanged_portion_omitted_
```

****
6126 Thu Aug 1 22:39:42 2013
new/usr/src/uts/common/fs/zfs/vdev_file.c
3949 ztest fault injection should avoid resilvering devices
3950 ztest: deadman fires when we're doing a scan
3951 ztest hang when running dedup test
3952 ztest: ztest_reguid test and ztest_fault_inject don't place nice together
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
****
_____***unchanged_portion_omitted_***

```
185 static int
186 vdev_file_io_start(zio_t *zio)
187 {
188         spa_t *spa = zio->io_spa;
189         vdev_t *vd = zio->io_vd;
190         vdev_file_t *vf = vd->vdev_tsd;
191         vdev_buf_t *vb;
192         buf_t *bp;
193
194         if (zio->io_type == ZIO_TYPE_IOCTL) {
195                 /* XXPOLICY */
196                 if (!vdev_readable(vd)) {
197                         zio->io_error = SET_ERROR(ENXIO);
198                         return (ZIO_PIPELINE_CONTINUE);
199                 }
200
201                 switch (zio->io_cmd) {
202                 case DKIOCFLUSHWRITECACHE:
203                         zio->io_error = VOP_FSYNC(vf->vf_vnode, FSYNC | FDSYNC,
204                             kcred, NULL);
205                         break;
206                 default:
207                         zio->io_error = SET_ERROR(ENOTSUP);
208                 }
209
210                 return (ZIO_PIPELINE_CONTINUE);
211         }
212
213         vb = kmem_alloc(sizeof (vdev_buf_t), KM_SLEEP);
214
215         vb->vb_io = zio;
216         bp = &vb->vb_buf;
217
218         bioinit(bp);
219         bp->b_flags = (zio->io_type == ZIO_TYPE_READ ? B_READ : B_WRITE);
220         bp->b_bcount = zio->io_size;
221         bp->b_un.b_addr = zio->io_data;
222         bp->b_lblkno = lbtodb(zio->io_offset);
223         bp->b_bufsize = zio->io_size;
224         bp->b_private = vf->vf_vnode;
225         bp->b_iodone = (int (*)())vdev_file_io_intr;
226
227         VERIFY3U(taskq_dispatch(system_taskq, vdev_file_io_strategy, bp,
228             TQ_SLEEP), !=, 0);
227         spa_taskq_dispatch_ent(spa, ZIO_TYPE_FREE, ZIO_TASKQ_ISSUE,
228             vdev_file_io_strategy, bp, 0, &zio->io_tqent);
229
230         return (ZIO_PIPELINE_STOP);
231 }
```

_____***unchanged_portion_omitted_***