```
**********************************************************
   89516 Sat Jun 21 08:58:41 2014
new/usr/src/uts/common/fs/zfs/vdev.c
4932 vdev incorrectly expanding on last mirror child -> top-level vdev
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 759 /*
 760  * Remove a 1-way mirror/replacing vdev from the tree.
 761  */
 762 void
 763 vdev_remove_parent(vdev_t *cvd)
 764 {
 765         vdev_t *mvd = cvd->vdev_parent;
 766         vdev_t *pvd = mvd->vdev_parent;

 768         ASSERT(spa_config_held(cvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);

 770         ASSERT(mvd->vdev_children == 1);
 771         ASSERT(mvd->vdev_ops == &vdev_mirror_ops ||
 772             mvd->vdev_ops == &vdev_replacing_ops ||
 773             mvd->vdev_ops == &vdev_spare_ops);
 774         cvd->vdev_ashift = mvd->vdev_ashift;

 776         vdev_remove_child(mvd, cvd);
 777         vdev_remove_child(pvd, mvd);

 779         /*
 780          * If cvd will replace mvd as a top-level vdev, preserve mvd's guid.
 781          * Otherwise, we could have detached an offline device, and when we
 782          * go to import the pool we'll think we have two top-level vdevs,
 783          * instead of a different version of the same top-level vdev.
 784          */
 785         if (mvd->vdev_top == mvd) {
 786                 uint64_t guid_delta = mvd->vdev_guid - cvd->vdev_guid;
 787                 cvd->vdev_orig_guid = cvd->vdev_guid;
 788                 cvd->vdev_guid += guid_delta;
 789                 cvd->vdev_guid_sum += guid_delta;

 791                 /*
 792                  * If pool not set for autoexpand, we need to also preserve
 793                  * mvd's asize to prevent automatic expansion of cvd.
 794                  * Otherwise if we are adjusting the mirror by attaching and
 795                  * detaching children of non-uniform sizes, the mirror could
 796                  * autoexpand, unexpectedly requiring larger devices to
 797                  * re-establish the mirror.
 798                  */
 799                 if (!cvd->vdev_spa->spa_autoexpand)
 800                         cvd->vdev_asize = mvd->vdev_asize;
 801 #endif /* ! codereview */
 802         }
 803         cvd->vdev_id = mvd->vdev_id;
 804         vdev_add_child(pvd, cvd);
 805         vdev_top_update(cvd->vdev_top, cvd->vdev_top);

 807         if (cvd == cvd->vdev_top)
 808                 vdev_top_transfer(mvd, cvd);

 810         ASSERT(mvd->vdev_children == 0);
 811         vdev_free(mvd);
 812 }

 814 int
 815 vdev_metaslab_init(vdev_t *vd, uint64_t txg)
```



```
 816 {
 817         spa_t *spa = vd->vdev_spa;
 818         objset_t *mos = spa->spa_meta_objset;
 819         uint64_t m;
 820         uint64_t oldc = vd->vdev_ms_count;
 821         uint64_t newc = vd->vdev_asize >> vd->vdev_ms_shift;
 822         metaslab_t **mspp;
 823         int error;

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

 827         /*
 828          * This vdev is not being allocated from yet or is a hole.
 829          */
 830         if (vd->vdev_ms_shift == 0)
 831                 return (0);

 833         ASSERT(!vd->vdev_ishole);

 835         /*
 836          * Compute the raidz-deflation ratio.  Note, we hard-code
 837          * in 128k (1 << 17) because it is the current "typical" blocksize.
 838          * Even if SPA_MAXBLOCKSIZE changes, this algorithm must never change,
 839          * or we will inconsistently account for existing bp's.
 840          */
 841         vd->vdev_deflate_ratio = (1 << 17) /
 842             (vdev_psize_to_asize(vd, 1 << 17) >> SPA_MINBLOCKSHIFT);

 844         ASSERT(oldc <= newc);

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

 848         if (oldc != 0) {
 849                 bcopy(vd->vdev_ms, mspp, oldc * sizeof (*mspp));
 850                 kmem_free(vd->vdev_ms, oldc * sizeof (*mspp));
 851         }

 853         vd->vdev_ms = mspp;
 854         vd->vdev_ms_count = newc;

 856         for (m = oldc; m < newc; m++) {
 857                 uint64_t object = 0;

 859                 if (txg == 0) {
 860                         error = dmu_read(mos, vd->vdev_ms_array,
 861                             m * sizeof (uint64_t), sizeof (uint64_t), &object,
 862                             DMU_READ_PREFETCH);
 863                         if (error)
 864                                 return (error);
 865                 }
 866                 vd->vdev_ms[m] = metaslab_init(vd->vdev_mg, m, object, txg);
 867         }

 869         if (txg == 0)
 870                 spa_config_enter(spa, SCL_ALLOC, FTAG, RW_WRITER);

 872         /*
 873          * If the vdev is being removed we don't activate
 874          * the metaslabs since we want to ensure that no new
 875          * allocations are performed on this device.
 876          */
 877         if (oldc == 0 && !vd->vdev_removing)
 878                 metaslab_group_activate(vd->vdev_mg);

 880         if (txg == 0)
 881                 spa_config_exit(spa, SCL_ALLOC, FTAG);
```

```
883         return (0);
884 }

886 void
887 vdev_metaslab_fini(vdev_t *vd)
888 {
889         uint64_t m;
890         uint64_t count = vd->vdev_ms_count;

892         if (vd->vdev_ms != NULL) {
893                 metaslab_group_passivate(vd->vdev_mg);
894                 for (m = 0; m < count; m++) {
895                         metaslab_t *msp = vd->vdev_ms[m];

897                         if (msp != NULL)
898                                 metaslab_fini(msp);
899                 }
900                 kmem_free(vd->vdev_ms, count * sizeof (metaslab_t *));
901                 vd->vdev_ms = NULL;
902         }
903 }

905 typedef struct vdev_probe_stats {
906         boolean_t       vps_readable;
907         boolean_t       vps_writeable;
908         int             vps_flags;
909 } vdev_probe_stats_t;

911 static void
912 vdev_probe_done(zio_t *zio)
913 {
914         spa_t *spa = zio->io_spa;
915         vdev_t *vd = zio->io_vd;
916         vdev_probe_stats_t *vps = zio->io_private;

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

920         if (zio->io_type == ZIO_TYPE_READ) {
921                 if (zio->io_error == 0)
922                         vps->vps_readable = 1;
923                 if (zio->io_error == 0 && spa_writeable(spa)) {
924                         zio_nowait(zio_write_phys(vd->vdev_probe_zio, vd,
925                             zio->io_offset, zio->io_size, zio->io_data,
926                             ZIO_CHECKSUM_OFF, vdev_probe_done, vps,
927                             ZIO_PRIORITY_SYNC_WRITE, vps->vps_flags, B_TRUE));
928                 } else {
929                         zio_buf_free(zio->io_data, zio->io_size);
930                 }
931         } else if (zio->io_type == ZIO_TYPE_WRITE) {
932                 if (zio->io_error == 0)
933                         vps->vps_writeable = 1;
934                 zio_buf_free(zio->io_data, zio->io_size);
935         } else if (zio->io_type == ZIO_TYPE_NULL) {
936                 zio_t *pio;

938                 vd->vdev_cant_read |= !vps->vps_readable;
939                 vd->vdev_cant_write |= !vps->vps_writeable;

941                 if (vdev_readable(vd) &&
942                     (vdev_writeable(vd) || !spa_writeable(spa))) {
943                         zio->io_error = 0;
944                 } else {
945                         ASSERT(zio->io_error != 0);
946                         zfs_ereport_post(FM_EREPORT_ZFS_PROBE_FAILURE,
947                             spa, vd, NULL, 0, 0);
```



```
948                         zio->io_error = SET_ERROR(ENXIO);
949                 }

951                 mutex_enter(&vd->vdev_probe_lock);
952                 ASSERT(vd->vdev_probe_zio == zio);
953                 vd->vdev_probe_zio = NULL;
954                 mutex_exit(&vd->vdev_probe_lock);

956                 while ((pio = zio_walk_parents(zio)) != NULL)
957                         if (!vdev_accessible(vd, pio))
958                                 pio->io_error = SET_ERROR(ENXIO);

960                 kmem_free(vps, sizeof (*vps));
961         }
962 }

964 /*
965  * Determine whether this device is accessible.
966  *
967  * Read and write to several known locations: the pad regions of each
968  * vdev label but the first, which we leave alone in case it contains
969  * a VTOC.
970  */
971 zio_t *
972 vdev_probe(vdev_t *vd, zio_t *zio)
973 {
974         spa_t *spa = vd->vdev_spa;
975         vdev_probe_stats_t *vps = NULL;
976         zio_t *pio;

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

980         /*
981          * Don't probe the probe.
982          */
983         if (zio && (zio->io_flags & ZIO_FLAG_PROBE))
984                 return (NULL);

986         /*
987          * To prevent 'probe storms' when a device fails, we create
988          * just one probe i/o at a time.  All zios that want to probe
989          * this vdev will become parents of the probe io.
990          */
991         mutex_enter(&vd->vdev_probe_lock);

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

996                 vps->vps_flags = ZIO_FLAG_CANFAIL | ZIO_FLAG_PROBE |
997                     ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_AGGREGATE |
998                     ZIO_FLAG_TRYHARD;

1000                if (spa_config_held(spa, SCL_ZIO, RW_WRITER)) {
1001                        /*
1002                         * vdev_cant_read and vdev_cant_write can only
1003                         * transition from TRUE to FALSE when we have the
1004                         * SCL_ZIO lock as writer; otherwise they can only
1005                         * transition from FALSE to TRUE.  This ensures that
1006                         * any zio looking at these values can assume that
1007                         * failures persist for the life of the I/O.  That's
1008                         * important because when a device has intermittent
1009                         * connectivity problems, we want to ensure that
1010                         * they're ascribed to the device (ENXIO) and not
1011                         * the zio (EIO).
1012                         *
1013                         * Since we hold SCL_ZIO as writer here, clear both
```

```
1014                         * values so the probe can reevaluate from first
1015                         * principles.
1016                         */
1017                        vps->vps_flags |= ZIO_FLAG_CONFIG_WRITER;
1018                        vd->vdev_cant_read = B_FALSE;
1019                        vd->vdev_cant_write = B_FALSE;
1020                }

1022                vd->vdev_probe_zio = pio = zio_null(NULL, spa, vd,
1023                    vdev_probe_done, vps,
1024                    vps->vps_flags | ZIO_FLAG_DONT_PROPAGATE);

1026                /*
1027                 * We can't change the vdev state in this context, so we
1028                 * kick off an async task to do it on our behalf.
1029                 */
1030                if (zio != NULL) {
1031                        vd->vdev_probe_wanted = B_TRUE;
1032                        spa_async_request(spa, SPA_ASYNC_PROBE);
1033                }
1034        }

1036        if (zio != NULL)
1037                zio_add_child(zio, pio);

1039        mutex_exit(&vd->vdev_probe_lock);

1041        if (vps == NULL) {
1042                ASSERT(zio != NULL);
1043                return (NULL);
1044        }

1046        for (int l = 1; l < VDEV_LABELS; l++) {
1047                zio_nowait(zio_read_phys(pio, vd,
1048                    vdev_label_offset(vd->vdev_psize, l,
1049                    offsetof(vdev_label_t, vl_pad2)),
1050                    VDEV_PAD_SIZE, zio_buf_alloc(VDEV_PAD_SIZE),
1051                    ZIO_CHECKSUM_OFF, vdev_probe_done, vps,
1052                    ZIO_PRIORITY_SYNC_READ, vps->vps_flags, B_TRUE));
1053        }

1055        if (zio == NULL)
1056                return (pio);

1058        zio_nowait(pio);
1059        return (NULL);
1060 }

1062 static void
1063 vdev_open_child(void *arg)
1064 {
1065        vdev_t *vd = arg;

1067        vd->vdev_open_thread = curthread;
1068        vd->vdev_open_error = vdev_open(vd);
1069        vd->vdev_open_thread = NULL;
1070 }

1072 boolean_t
1073 vdev_uses_zvols(vdev_t *vd)
1074 {
1075        if (vd->vdev_path && strncmp(vd->vdev_path, ZVOL_DIR,
1076            strlen(ZVOL_DIR)) == 0)
1077                return (B_TRUE);
1078        for (int c = 0; c < vd->vdev_children; c++)
1079                if (vdev_uses_zvols(vd->vdev_child[c]))
```



```
1080                        return (B_TRUE);
1081        return (B_FALSE);
1082 }

1084 void
1085 vdev_open_children(vdev_t *vd)
1086 {
1087        taskq_t *tq;
1088        int children = vd->vdev_children;

1090        /*
1091         * in order to handle pools on top of zvols, do the opens
1092         * in a single thread so that the same thread holds the
1093         * spa_namespace_lock
1094         */
1095        if (vdev_uses_zvols(vd)) {
1096                for (int c = 0; c < children; c++)
1097                        vd->vdev_child[c]->vdev_open_error =
1098                            vdev_open(vd->vdev_child[c]);
1099                return;
1100        }
1101        tq = taskq_create("vdev_open", children, minclsyspri,
1102            children, children, TASKQ_PREPOPULATE);

1104        for (int c = 0; c < children; c++)
1105                VERIFY(taskq_dispatch(tq, vdev_open_child, vd->vdev_child[c],
1106                    TQ_SLEEP) != NULL);

1108        taskq_destroy(tq);
1109 }

1111 /*
1112  * Prepare a virtual device for access.
1113  */
1114 int
1115 vdev_open(vdev_t *vd)
1116 {
1117        spa_t *spa = vd->vdev_spa;
1118        int error;
1119        uint64_t osize = 0;
1120        uint64_t max_osize = 0;
1121        uint64_t asize, max_asize, psize;
1122        uint64_t ashift = 0;

1124        ASSERT(vd->vdev_open_thread == curthread ||
1125            spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
1126        ASSERT(vd->vdev_state == VDEV_STATE_CLOSED ||
1127            vd->vdev_state == VDEV_STATE_CANT_OPEN ||
1128            vd->vdev_state == VDEV_STATE_OFFLINE);

1130        vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
1131        vd->vdev_cant_read = B_FALSE;
1132        vd->vdev_cant_write = B_FALSE;
1133        vd->vdev_min_asize = vdev_get_min_asize(vd);

1135        /*
1136         * If this vdev is not removed, check its fault status.  If it's
1137         * faulted, bail out of the open.
1138         */
1139        if (!vd->vdev_removed && vd->vdev_faulted) {
1140                ASSERT(vd->vdev_children == 0);
1141                ASSERT(vd->vdev_label_aux == VDEV_AUX_ERR_EXCEEDED ||
1142                    vd->vdev_label_aux == VDEV_AUX_EXTERNAL);
1143                vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
1144                    vd->vdev_label_aux);
1145                return (SET_ERROR(ENXIO));
```

```
1146         } else if (vd->vdev_offline) {
1147                 ASSERT(vd->vdev_children == 0);
1148                 vdev_set_state(vd, B_TRUE, VDEV_STATE_OFFLINE, VDEV_AUX_NONE);
1149                 return (SET_ERROR(ENXIO));
1150         }
1151
1152         error = vd->vdev_ops->vdev_op_open(vd, &osize, &max_osize, &ashift);
1153
1154         /*
1155          * Reset the vdev_reopening flag so that we actually close
1156          * the vdev on error.
1157          */
1158         vd->vdev_reopening = B_FALSE;
1159         if (zio_injection_enabled && error == 0)
1160                 error = zio_handle_device_injection(vd, NULL, ENXIO);
1161
1162         if (error) {
1163                 if (vd->vdev_removed &&
1164                     vd->vdev_stat.vs_aux != VDEV_AUX_OPEN_FAILED)
1165                         vd->vdev_removed = B_FALSE;
1166
1167                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1168                     vd->vdev_stat.vs_aux);
1169                 return (error);
1170         }
1171
1172         vd->vdev_removed = B_FALSE;
1173
1174         /*
1175          * Recheck the faulted flag now that we have confirmed that
1176          * the vdev is accessible.  If we're faulted, bail.
1177          */
1178         if (vd->vdev_faulted) {
1179                 ASSERT(vd->vdev_children == 0);
1180                 ASSERT(vd->vdev_label_aux == VDEV_AUX_ERR_EXCEEDED ||
1181                     vd->vdev_label_aux == VDEV_AUX_EXTERNAL);
1182                 vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
1183                     vd->vdev_label_aux);
1184                 return (SET_ERROR(ENXIO));
1185         }
1186
1187         if (vd->vdev_degraded) {
1188                 ASSERT(vd->vdev_children == 0);
1189                 vdev_set_state(vd, B_TRUE, VDEV_STATE_DEGRADED,
1190                     VDEV_AUX_ERR_EXCEEDED);
1191         } else {
1192                 vdev_set_state(vd, B_TRUE, VDEV_STATE_HEALTHY, 0);
1193         }
1194
1195         /*
1196          * For hole or missing vdevs we just return success.
1197          */
1198         if (vd->vdev_ishole || vd->vdev_ops == &vdev_missing_ops)
1199                 return (0);
1200
1201         for (int c = 0; c < vd->vdev_children; c++) {
1202                 if (vd->vdev_child[c]->vdev_state != VDEV_STATE_HEALTHY) {
1203                         vdev_set_state(vd, B_TRUE, VDEV_STATE_DEGRADED,
1204                             VDEV_AUX_NONE);
1205                         break;
1206                 }
1207         }
1208
1209         osize = P2ALIGN(osize, (uint64_t)sizeof (vdev_label_t));
1210         max_osize = P2ALIGN(max_osize, (uint64_t)sizeof (vdev_label_t));
```



```
1212         if (vd->vdev_children == 0) {
1213                 if (osize < SPA_MINDEVSIZE) {
1214                         vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1215                             VDEV_AUX_TOO_SMALL);
1216                         return (SET_ERROR(EOVERFLOW));
1217                 }
1218                 psize = osize;
1219                 asize = osize - (VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE);
1220                 max_asize = max_osize - (VDEV_LABEL_START_SIZE +
1221                     VDEV_LABEL_END_SIZE);
1222         } else {
1223                 if (vd->vdev_parent != NULL && osize < SPA_MINDEVSIZE -
1224                     (VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE)) {
1225                         vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1226                             VDEV_AUX_TOO_SMALL);
1227                         return (SET_ERROR(EOVERFLOW));
1228                 }
1229                 psize = 0;
1230                 asize = osize;
1231                 max_asize = max_osize;
1232         }
1233
1234         vd->vdev_psize = psize;
1235
1236         /*
1237          * Make sure the allocatable size hasn't shrunk.
1238          */
1239         if (asize < vd->vdev_min_asize) {
1240                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1241                     VDEV_AUX_BAD_LABEL);
1242                 return (SET_ERROR(EINVAL));
1243         }
1244
1245         if (vd->vdev_asize == 0) {
1246                 /*
1247                  * This is the first-ever open, so use the computed values.
1248                  * For testing purposes, a higher ashift can be requested.
1249                  */
1250                 vd->vdev_asize = asize;
1251                 vd->vdev_max_asize = max_asize;
1252                 vd->vdev_ashift = MAX(ashift, vd->vdev_ashift);
1253         } else {
1254                 /*
1255                  * Detect if the alignment requirement has increased.
1256                  * We don't want to make the pool unavailable, just
1257                  * issue a warning instead.
1258                  */
1259                 if (ashift > vd->vdev_top->vdev_ashift &&
1260                     vd->vdev_ops->vdev_op_leaf) {
1261                         cmn_err(CE_WARN,
1262                             "Disk, '%s', has a block alignment that is "
1263                             "larger than the pool's alignment\n",
1264                             vd->vdev_path);
1265                 }
1266                 vd->vdev_max_asize = max_asize;
1267         }
1268
1269         /*
1270          * If all children are healthy and the asize has increased,
1271          * then we've experienced dynamic LUN growth.  If automatic
1272          * expansion is enabled then use the additional space.
1273          */
1274         if (vd->vdev_state == VDEV_STATE_HEALTHY && asize > vd->vdev_asize &&
1275             (vd->vdev_expanding || spa->spa_autoexpand))
1276                 vd->vdev_asize = asize;
```

```
1278         vdev_set_min_asize(vd);

1280         /*
1281          * Ensure we can issue some IO before declaring the
1282          * vdev open for business.
1283          */
1284         if (vd->vdev_ops->vdev_op_leaf &&
1285             (error = zio_wait(vdev_probe(vd, NULL))) != 0) {
1286                 vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
1287                     VDEV_AUX_ERR_EXCEEDED);
1288                 return (error);
1289         }

1291         /*
1292          * If a leaf vdev has a DTL, and seems healthy, then kick off a
1293          * resilver.  But don't do this if we are doing a reopen for a scrub,
1294          * since this would just restart the scrub we are already doing.
1295          */
1296         if (vd->vdev_ops->vdev_op_leaf && !spa->spa_scrub_reopen &&
1297             vdev_resilver_needed(vd, NULL, NULL))
1298                 spa_async_request(spa, SPA_ASYNC_RESILVER);

1300         return (0);
1301 }

1303 /*
1304  * Called once the vdevs are all opened, this routine validates the label
1305  * contents.  This needs to be done before vdev_load() so that we don't
1306  * inadvertently do repair I/Os to the wrong device.
1307  *
1308  * If 'strict' is false ignore the spa guid check. This is necessary because
1309  * if the machine crashed during a re-guid the new guid might have been written
1310  * to all of the vdev labels, but not the cached config. The strict check
1311  * will be performed when the pool is opened again using the mos config.
1312  *
1313  * This function will only return failure if one of the vdevs indicates that it
1314  * has since been destroyed or exported.  This is only possible if
1315  * /etc/zfs/zpool.cache was readonly at the time.  Otherwise, the vdev state
1316  * will be updated but the function will return 0.
1317  */
1318 int
1319 vdev_validate(vdev_t *vd, boolean_t strict)
1320 {
1321         spa_t *spa = vd->vdev_spa;
1322         nvlist_t *label;
1323         uint64_t guid = 0, top_guid;
1324         uint64_t state;

1326         for (int c = 0; c < vd->vdev_children; c++)
1327                 if (vdev_validate(vd->vdev_child[c], strict) != 0)
1328                         return (SET_ERROR(EBADF));

1330         /*
1331          * If the device has already failed, or was marked offline, don't do
1332          * any further validation.  Otherwise, label I/O will fail and we will
1333          * overwrite the previous state.
1334          */
1335         if (vd->vdev_ops->vdev_op_leaf && vdev_readable(vd)) {
1336                 uint64_t aux_guid = 0;
1337                 nvlist_t *nvl;
1338                 uint64_t txg = spa_last_synced_txg(spa) != 0 ?
1339                     spa_last_synced_txg(spa) : -1ULL;

1341                 if ((label = vdev_label_read_config(vd, txg)) == NULL) {
1342                         vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1343                             VDEV_AUX_BAD_LABEL);
```



```
1344                         return (0);
1345                 }

1347                 /*
1348                  * Determine if this vdev has been split off into another
1349                  * pool.  If so, then refuse to open it.
1350                  */
1351                 if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_SPLIT_GUID,
1352                     &aux_guid) == 0 && aux_guid == spa_guid(spa)) {
1353                         vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1354                             VDEV_AUX_SPLIT_POOL);
1355                         nvlist_free(label);
1356                         return (0);
1357                 }

1359                 if (strict && (nvlist_lookup_uint64(label,
1360                     ZPOOL_CONFIG_POOL_GUID, &guid) != 0 ||
1361                     guid != spa_guid(spa))) {
1362                         vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1363                             VDEV_AUX_CORRUPT_DATA);
1364                         nvlist_free(label);
1365                         return (0);
1366                 }

1368                 if (nvlist_lookup_nvlist(label, ZPOOL_CONFIG_VDEV_TREE, &nvl)
1369                     != 0 || nvlist_lookup_uint64(nvl, ZPOOL_CONFIG_ORIG_GUID,
1370                     &aux_guid) != 0)
1371                         aux_guid = 0;

1373                 /*
1374                  * If this vdev just became a top-level vdev because its
1375                  * sibling was detached, it will have adopted the parent's
1376                  * vdev guid -- but the label may or may not be on disk yet.
1377                  * Fortunately, either version of the label will have the
1378                  * same top guid, so if we're a top-level vdev, we can
1379                  * safely compare to that instead.
1380                  *
1381                  * If we split this vdev off instead, then we also check the
1382                  * original pool's guid.  We don't want to consider the vdev
1383                  * corrupt if it is partway through a split operation.
1384                  */
1385                 if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_GUID,
1386                     &guid) != 0 ||
1387                     nvlist_lookup_uint64(label, ZPOOL_CONFIG_TOP_GUID,
1388                     &top_guid) != 0 ||
1389                     ((vd->vdev_guid != guid && vd->vdev_guid != aux_guid) &&
1390                     (vd->vdev_guid != top_guid || vd != vd->vdev_top))) {
1391                         vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1392                             VDEV_AUX_CORRUPT_DATA);
1393                         nvlist_free(label);
1394                         return (0);
1395                 }

1397                 if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_STATE,
1398                     &state) != 0) {
1399                         vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1400                             VDEV_AUX_CORRUPT_DATA);
1401                         nvlist_free(label);
1402                         return (0);
1403                 }

1405                 nvlist_free(label);

1407                 /*
1408                  * If this is a verbatim import, no need to check the
1409                  * state of the pool.
```

```
1410                 */
1411                if (!(spa->spa_import_flags & ZFS_IMPORT_VERBATIM) &&
1412                    spa_load_state(spa) == SPA_LOAD_OPEN &&
1413                    state != POOL_STATE_ACTIVE)
1414                        return (SET_ERROR(EBADF));

1416                /*
1417                 * If we were able to open and validate a vdev that was
1418                 * previously marked permanently unavailable, clear that state
1419                 * now.
1420                 */
1421                if (vd->vdev_not_present)
1422                        vd->vdev_not_present = 0;
1423        }

1425        return (0);
1426 }

1428 /*
1429  * Close a virtual device.
1430  */
1431 void
1432 vdev_close(vdev_t *vd)
1433 {
1434        spa_t *spa = vd->vdev_spa;
1435        vdev_t *pvd = vd->vdev_parent;

1437        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

1439        /*
1440         * If our parent is reopening, then we are as well, unless we are
1441         * going offline.
1442         */
1443        if (pvd != NULL && pvd->vdev_reopening)
1444                vd->vdev_reopening = (pvd->vdev_reopening && !vd->vdev_offline);

1446        vd->vdev_ops->vdev_op_close(vd);

1448        vdev_cache_purge(vd);

1450        /*
1451         * We record the previous state before we close it, so that if we are
1452         * doing a reopen(), we don't generate FMA ereports if we notice that
1453         * it's still faulted.
1454         */
1455        vd->vdev_prevstate = vd->vdev_state;

1457        if (vd->vdev_offline)
1458                vd->vdev_state = VDEV_STATE_OFFLINE;
1459        else
1460                vd->vdev_state = VDEV_STATE_CLOSED;
1461        vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
1462 }

1464 void
1465 vdev_hold(vdev_t *vd)
1466 {
1467        spa_t *spa = vd->vdev_spa;

1469        ASSERT(spa_is_root(spa));
1470        if (spa->spa_state == POOL_STATE_UNINITIALIZED)
1471                return;

1473        for (int c = 0; c < vd->vdev_children; c++)
1474                vdev_hold(vd->vdev_child[c]);
```



```
1476        if (vd->vdev_ops->vdev_op_leaf)
1477                vd->vdev_ops->vdev_op_hold(vd);
1478 }

1480 void
1481 vdev_rele(vdev_t *vd)
1482 {
1483        spa_t *spa = vd->vdev_spa;

1485        ASSERT(spa_is_root(spa));
1486        for (int c = 0; c < vd->vdev_children; c++)
1487                vdev_rele(vd->vdev_child[c]);

1489        if (vd->vdev_ops->vdev_op_leaf)
1490                vd->vdev_ops->vdev_op_rele(vd);
1491 }

1493 /*
1494  * Reopen all interior vdevs and any unopened leaves.  We don't actually
1495  * reopen leaf vdevs which had previously been opened as they might deadlock
1496  * on the spa_config_lock.  Instead we only obtain the leaf's physical size.
1497  * If the leaf has never been opened then open it, as usual.
1498  */
1499 void
1500 vdev_reopen(vdev_t *vd)
1501 {
1502        spa_t *spa = vd->vdev_spa;

1504        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

1506        /* set the reopening flag unless we're taking the vdev offline */
1507        vd->vdev_reopening = !vd->vdev_offline;
1508        vdev_close(vd);
1509        (void) vdev_open(vd);

1511        /*
1512         * Call vdev_validate() here to make sure we have the same device.
1513         * Otherwise, a device with an invalid label could be successfully
1514         * opened in response to vdev_reopen().
1515         */
1516        if (vd->vdev_aux) {
1517                (void) vdev_validate_aux(vd);
1518                if (vdev_readable(vd) && vdev_writeable(vd) &&
1519                    vd->vdev_aux == &spa->spa_l2cache &&
1520                    !l2arc_vdev_present(vd))
1521                        l2arc_add_vdev(spa, vd);
1522        } else {
1523                (void) vdev_validate(vd, B_TRUE);
1524        }

1526        /*
1527         * Reassess parent vdev's health.
1528         */
1529        vdev_propagate_state(vd);
1530 }

1532 int
1533 vdev_create(vdev_t *vd, uint64_t txg, boolean_t isreplacing)
1534 {
1535        int error;

1537        /*
1538         * Normally, partial opens (e.g. of a mirror) are allowed.
1539         * For a create, however, we want to fail the request if
1540         * there are any components we can't open.
1541         */
```

```
1542         error = vdev_open(vd);

1544         if (error || vd->vdev_state != VDEV_STATE_HEALTHY) {
1545                 vdev_close(vd);
1546                 return (error ? error : ENXIO);
1547         }

1549         /*
1550          * Recursively load DTLs and initialize all labels.
1551          */
1552         if ((error = vdev_dtl_load(vd)) != 0 ||
1553             (error = vdev_label_init(vd, txg, isreplacing ?
1554             VDEV_LABEL_REPLACE : VDEV_LABEL_CREATE)) != 0) {
1555                 vdev_close(vd);
1556                 return (error);
1557         }

1559         return (0);
1560 }

1562 void
1563 vdev_metaslab_set_size(vdev_t *vd)
1564 {
1565         /*
1566          * Aim for roughly 200 metaslabs per vdev.
1567          */
1568         vd->vdev_ms_shift = highbit64(vd->vdev_asize / 200);
1569         vd->vdev_ms_shift = MAX(vd->vdev_ms_shift, SPA_MAXBLOCKSHIFT);
1570 }

1572 void
1573 vdev_dirty(vdev_t *vd, int flags, void *arg, uint64_t txg)
1574 {
1575         ASSERT(vd == vd->vdev_top);
1576         ASSERT(!vd->vdev_ishole);
1577         ASSERT(ISP2(flags));
1578         ASSERT(spa_writeable(vd->vdev_spa));

1580         if (flags & VDD_METASLAB)
1581                 (void) txg_list_add(&vd->vdev_ms_list, arg, txg);

1583         if (flags & VDD_DTL)
1584                 (void) txg_list_add(&vd->vdev_dtl_list, arg, txg);

1586         (void) txg_list_add(&vd->vdev_spa->spa_vdev_txg_list, vd, txg);
1587 }

1589 void
1590 vdev_dirty_leaves(vdev_t *vd, int flags, uint64_t txg)
1591 {
1592         for (int c = 0; c < vd->vdev_children; c++)
1593                 vdev_dirty_leaves(vd->vdev_child[c], flags, txg);

1595         if (vd->vdev_ops->vdev_op_leaf)
1596                 vdev_dirty(vd->vdev_top, flags, vd, txg);
1597 }

1599 /*
1600  * DTLs.
1601  *
1602  * A vdev's DTL (dirty time log) is the set of transaction groups for which
1603  * the vdev has less than perfect replication.  There are four kinds of DTL:
1604  *
1605  * DTL_MISSING: txgs for which the vdev has no valid copies of the data
1606  *
1607  * DTL_PARTIAL: txgs for which data is available, but not fully replicated
```



```
1608  *
1609  * DTL_SCRUB: the txgs that could not be repaired by the last scrub; upon
1610  *      scrub completion, DTL_SCRUB replaces DTL_MISSING in the range of
1611  *      txgs that was scrubbed.
1612  *
1613  * DTL_OUTAGE: txgs which cannot currently be read, whether due to
1614  *      persistent errors or just some device being offline.
1615  *      Unlike the other three, the DTL_OUTAGE map is not generally
1616  *      maintained; it's only computed when needed, typically to
1617  *      determine whether a device can be detached.
1618  *
1619  * For leaf vdevs, DTL_MISSING and DTL_PARTIAL are identical: the device
1620  * either has the data or it doesn't.
1621  *
1622  * For interior vdevs such as mirror and RAID-Z the picture is more complex.
1623  * A vdev's DTL_PARTIAL is the union of its children's DTL_PARTIALs, because
1624  * if any child is less than fully replicated, then so is its parent.
1625  * A vdev's DTL_MISSING is a modified union of its children's DTL_MISSINGs,
1626  * comprising only those txgs which appear in 'maxfaults' or more children;
1627  * those are the txgs we don't have enough replication to read.  For example,
1628  * double-parity RAID-Z can tolerate up to two missing devices (maxfaults == 2);
1629  * thus, its DTL_MISSING consists of the set of txgs that appear in more than
1630  * two child DTL_MISSING maps.
1631  *
1632  * It should be clear from the above that to compute the DTLs and outage maps
1633  * for all vdevs, it suffices to know just the leaf vdevs' DTL_MISSING maps.
1634  * Therefore, that is all we keep on disk.  When loading the pool, or after
1635  * a configuration change, we generate all other DTLs from first principles.
1636  */
1637 void
1638 vdev_dtl_dirty(vdev_t *vd, vdev_dtl_type_t t, uint64_t txg, uint64_t size)
1639 {
1640         range_tree_t *rt = vd->vdev_dtl[t];

1642         ASSERT(t < DTL_TYPES);
1643         ASSERT(vd != vd->vdev_spa->spa_root_vdev);
1644         ASSERT(spa_writeable(vd->vdev_spa));

1646         mutex_enter(rt->rt_lock);
1647         if (!range_tree_contains(rt, txg, size))
1648                 range_tree_add(rt, txg, size);
1649         mutex_exit(rt->rt_lock);
1650 }

1652 boolean_t
1653 vdev_dtl_contains(vdev_t *vd, vdev_dtl_type_t t, uint64_t txg, uint64_t size)
1654 {
1655         range_tree_t *rt = vd->vdev_dtl[t];
1656         boolean_t dirty = B_FALSE;

1658         ASSERT(t < DTL_TYPES);
1659         ASSERT(vd != vd->vdev_spa->spa_root_vdev);

1661         mutex_enter(rt->rt_lock);
1662         if (range_tree_space(rt) != 0)
1663                 dirty = range_tree_contains(rt, txg, size);
1664         mutex_exit(rt->rt_lock);

1666         return (dirty);
1667 }

1669 boolean_t
1670 vdev_dtl_empty(vdev_t *vd, vdev_dtl_type_t t)
1671 {
1672         range_tree_t *rt = vd->vdev_dtl[t];
1673         boolean_t empty;
```

```
1675         mutex_enter(rt->rt_lock);
1676         empty = (range_tree_space(rt) == 0);
1677         mutex_exit(rt->rt_lock);

1679         return (empty);
1680 }

1682 /*
1683  * Returns the lowest txg in the DTL range.
1684  */
1685 static uint64_t
1686 vdev_dtl_min(vdev_t *vd)
1687 {
1688         range_seg_t *rs;

1690         ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
1691         ASSERT3U(range_tree_space(vd->vdev_dtl[DTL_MISSING]), !=, 0);
1692         ASSERT0(vd->vdev_children);

1694         rs = avl_first(&vd->vdev_dtl[DTL_MISSING]->rt_root);
1695         return (rs->rs_start - 1);
1696 }

1698 /*
1699  * Returns the highest txg in the DTL.
1700  */
1701 static uint64_t
1702 vdev_dtl_max(vdev_t *vd)
1703 {
1704         range_seg_t *rs;

1706         ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
1707         ASSERT3U(range_tree_space(vd->vdev_dtl[DTL_MISSING]), !=, 0);
1708         ASSERT0(vd->vdev_children);

1710         rs = avl_last(&vd->vdev_dtl[DTL_MISSING]->rt_root);
1711         return (rs->rs_end);
1712 }

1714 /*
1715  * Determine if a resilvering vdev should remove any DTL entries from
1716  * its range. If the vdev was resilvering for the entire duration of the
1717  * scan then it should excise that range from its DTLs. Otherwise, this
1718  * vdev is considered partially resilvered and should leave its DTL
1719  * entries intact. The comment in vdev_dtl_reassess() describes how we
1720  * excise the DTLs.
1721  */
1722 static boolean_t
1723 vdev_dtl_should_excise(vdev_t *vd)
1724 {
1725         spa_t *spa = vd->vdev_spa;
1726         dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;

1728         ASSERT0(scn->scn_phys.scn_errors);
1729         ASSERT0(vd->vdev_children);

1731         if (vd->vdev_resilver_txg == 0 ||
1732             range_tree_space(vd->vdev_dtl[DTL_MISSING]) == 0)
1733                 return (B_TRUE);

1735         /*
1736          * When a resilver is initiated the scan will assign the scn_max_txg
1737          * value to the highest txg value that exists in all DTLs. If this
1738          * device's max DTL is not part of this scan (i.e. it is not in
1739          * the range (scn_min_txg, scn_max_txg] then it is not eligible
```



```
1740          * for excision.
1741          */
1742         if (vdev_dtl_max(vd) <= scn->scn_phys.scn_max_txg) {
1743                 ASSERT3U(scn->scn_phys.scn_min_txg, <=, vdev_dtl_min(vd));
1744                 ASSERT3U(scn->scn_phys.scn_min_txg, <, vd->vdev_resilver_txg);
1745                 ASSERT3U(vd->vdev_resilver_txg, <=, scn->scn_phys.scn_max_txg);
1746                 return (B_TRUE);
1747         }
1748         return (B_FALSE);
1749 }

1751 /*
1752  * Reassess DTLs after a config change or scrub completion.
1753  */
1754 void
1755 vdev_dtl_reassess(vdev_t *vd, uint64_t txg, uint64_t scrub_txg, int scrub_done)
1756 {
1757         spa_t *spa = vd->vdev_spa;
1758         avl_tree_t reftree;
1759         int minref;

1761         ASSERT(spa_config_held(spa, SCL_ALL, RW_READER) != 0);

1763         for (int c = 0; c < vd->vdev_children; c++)
1764                 vdev_dtl_reassess(vd->vdev_child[c], txg,
1765                     scrub_txg, scrub_done);

1767         if (vd == spa->spa_root_vdev || vd->vdev_ishole || vd->vdev_aux)
1768                 return;

1770         if (vd->vdev_ops->vdev_op_leaf) {
1771                 dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;

1773                 mutex_enter(&vd->vdev_dtl_lock);

1775                 /*
1776                  * If we've completed a scan cleanly then determine
1777                  * if this vdev should remove any DTLs. We only want to
1778                  * excise regions on vdevs that were available during
1779                  * the entire duration of this scan.
1780                  */
1781                 if (scrub_txg != 0 &&
1782                     (spa->spa_scrub_started ||
1783                     (scn != NULL && scn->scn_phys.scn_errors == 0)) &&
1784                     vdev_dtl_should_excise(vd)) {
1785                         /*
1786                          * We completed a scrub up to scrub_txg.  If we
1787                          * did it without rebooting, then the scrub dtl
1788                          * will be valid, so excise the old region and
1789                          * fold in the scrub dtl.  Otherwise, leave the
1790                          * dtl as-is if there was an error.
1791                          *
1792                          * There's little trick here: to excise the beginning
1793                          * of the DTL_MISSING map, we put it into a reference
1794                          * tree and then add a segment with refcnt -1 that
1795                          * covers the range [0, scrub_txg).  This means
1796                          * that each txg in that range has refcnt -1 or 0.
1797                          * We then add DTL_SCRUB with a refcnt of 2, so that
1798                          * entries in the range [0, scrub_txg) will have a
1799                          * positive refcnt -- either 1 or 2.  We then convert
1800                          * the reference tree into the new DTL_MISSING map.
1801                          */
1802                         space_reftree_create(&reftree);
1803                         space_reftree_add_map(&reftree,
1804                             vd->vdev_dtl[DTL_MISSING], 1);
1805                         space_reftree_add_seg(&reftree, 0, scrub_txg, -1);
```

```
1806                        space_reftree_add_map(&reftree,
1807                            vd->vdev_dtl[DTL_SCRUB], 2);
1808                        space_reftree_generate_map(&reftree,
1809                            vd->vdev_dtl[DTL_MISSING], 1);
1810                        space_reftree_destroy(&reftree);
1811                }
1812                range_tree_vacate(vd->vdev_dtl[DTL_PARTIAL], NULL, NULL);
1813                range_tree_walk(vd->vdev_dtl[DTL_MISSING],
1814                    range_tree_add, vd->vdev_dtl[DTL_PARTIAL]);
1815                if (scrub_done)
1816                        range_tree_vacate(vd->vdev_dtl[DTL_SCRUB], NULL, NULL);
1817                range_tree_vacate(vd->vdev_dtl[DTL_OUTAGE], NULL, NULL);
1818                if (!vdev_readable(vd))
1819                        range_tree_add(vd->vdev_dtl[DTL_OUTAGE], 0, -1ULL);
1820                else
1821                        range_tree_walk(vd->vdev_dtl[DTL_MISSING],
1822                            range_tree_add, vd->vdev_dtl[DTL_OUTAGE]);
1823
1824                /*
1825                 * If the vdev was resilvering and no longer has any
1826                 * DTLs then reset its resilvering flag.
1827                 */
1828                if (vd->vdev_resilver_txg != 0 &&
1829                    range_tree_space(vd->vdev_dtl[DTL_MISSING]) == 0 &&
1830                    range_tree_space(vd->vdev_dtl[DTL_OUTAGE]) == 0)
1831                        vd->vdev_resilver_txg = 0;
1832
1833                mutex_exit(&vd->vdev_dtl_lock);
1834
1835                if (txg != 0)
1836                        vdev_dirty(vd->vdev_top, VDD_DTL, vd, txg);
1837                return;
1838        }
1839
1840        mutex_enter(&vd->vdev_dtl_lock);
1841        for (int t = 0; t < DTL_TYPES; t++) {
1842                /* account for child's outage in parent's missing map */
1843                int s = (t == DTL_MISSING) ? DTL_OUTAGE: t;
1844                if (t == DTL_SCRUB)
1845                        continue;                       /* leaf vdevs only */
1846                if (t == DTL_PARTIAL)
1847                        minref = 1;                     /* i.e. non-zero */
1848                else if (vd->vdev_nparity != 0)
1849                        minref = vd->vdev_nparity + 1;  /* RAID-Z */
1850                else
1851                        minref = vd->vdev_children;     /* any kind of mirror */
1852                space_reftree_create(&reftree);
1853                for (int c = 0; c < vd->vdev_children; c++) {
1854                        vdev_t *cvd = vd->vdev_child[c];
1855                        mutex_enter(&cvd->vdev_dtl_lock);
1856                        space_reftree_add_map(&reftree, cvd->vdev_dtl[s], 1);
1857                        mutex_exit(&cvd->vdev_dtl_lock);
1858                }
1859                space_reftree_generate_map(&reftree, vd->vdev_dtl[t], minref);
1860                space_reftree_destroy(&reftree);
1861        }
1862        mutex_exit(&vd->vdev_dtl_lock);
1863 }
1864
1865 int
1866 vdev_dtl_load(vdev_t *vd)
1867 {
1868        spa_t *spa = vd->vdev_spa;
1869        objset_t *mos = spa->spa_meta_objset;
1870        int error = 0;
```



```
1872        if (vd->vdev_ops->vdev_op_leaf && vd->vdev_dtl_object != 0) {
1873                ASSERT(!vd->vdev_ishole);
1874
1875                error = space_map_open(&vd->vdev_dtl_sm, mos,
1876                    vd->vdev_dtl_object, 0, -1ULL, 0, &vd->vdev_dtl_lock);
1877                if (error)
1878                        return (error);
1879                ASSERT(vd->vdev_dtl_sm != NULL);
1880
1881                mutex_enter(&vd->vdev_dtl_lock);
1882
1883                /*
1884                 * Now that we've opened the space_map we need to update
1885                 * the in-core DTL.
1886                 */
1887                space_map_update(vd->vdev_dtl_sm);
1888
1889                error = space_map_load(vd->vdev_dtl_sm,
1890                    vd->vdev_dtl[DTL_MISSING], SM_ALLOC);
1891                mutex_exit(&vd->vdev_dtl_lock);
1892
1893                return (error);
1894        }
1895
1896        for (int c = 0; c < vd->vdev_children; c++) {
1897                error = vdev_dtl_load(vd->vdev_child[c]);
1898                if (error != 0)
1899                        break;
1900        }
1901
1902        return (error);
1903 }
1904
1905 void
1906 vdev_dtl_sync(vdev_t *vd, uint64_t txg)
1907 {
1908        spa_t *spa = vd->vdev_spa;
1909        range_tree_t *rt = vd->vdev_dtl[DTL_MISSING];
1910        objset_t *mos = spa->spa_meta_objset;
1911        range_tree_t *rtsync;
1912        kmutex_t rtlock;
1913        dmu_tx_t *tx;
1914        uint64_t object = space_map_object(vd->vdev_dtl_sm);
1915
1916        ASSERT(!vd->vdev_ishole);
1917        ASSERT(vd->vdev_ops->vdev_op_leaf);
1918
1919        tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);
1920
1921        if (vd->vdev_detached || vd->vdev_top->vdev_removing) {
1922                mutex_enter(&vd->vdev_dtl_lock);
1923                space_map_free(vd->vdev_dtl_sm, tx);
1924                space_map_close(vd->vdev_dtl_sm);
1925                vd->vdev_dtl_sm = NULL;
1926                mutex_exit(&vd->vdev_dtl_lock);
1927                dmu_tx_commit(tx);
1928                return;
1929        }
1930
1931        if (vd->vdev_dtl_sm == NULL) {
1932                uint64_t new_object;
1933
1934                new_object = space_map_alloc(mos, tx);
1935                VERIFY3U(new_object, !=, 0);
1936
1937                VERIFY0(space_map_open(&vd->vdev_dtl_sm, mos, new_object,
```

```
1938                    0, -1ULL, 0, &vd->vdev_dtl_lock));
1939                ASSERT(vd->vdev_dtl_sm != NULL);
1940        }

1942        mutex_init(&rtlock, NULL, MUTEX_DEFAULT, NULL);

1944        rtsync = range_tree_create(NULL, NULL, &rtlock);

1946        mutex_enter(&rtlock);

1948        mutex_enter(&vd->vdev_dtl_lock);
1949        range_tree_walk(rt, range_tree_add, rtsync);
1950        mutex_exit(&vd->vdev_dtl_lock);

1952        space_map_truncate(vd->vdev_dtl_sm, tx);
1953        space_map_write(vd->vdev_dtl_sm, rtsync, SM_ALLOC, tx);
1954        range_tree_vacate(rtsync, NULL, NULL);

1956        range_tree_destroy(rtsync);

1958        mutex_exit(&rtlock);
1959        mutex_destroy(&rtlock);

1961        /*
1962         * If the object for the space map has changed then dirty
1963         * the top level so that we update the config.
1964         */
1965        if (object != space_map_object(vd->vdev_dtl_sm)) {
1966                zfs_dbgmsg("txg %llu, spa %s, DTL old object %llu, "
1967                    "new object %llu", txg, spa_name(spa), object,
1968                    space_map_object(vd->vdev_dtl_sm));
1969                vdev_config_dirty(vd->vdev_top);
1970        }

1972        dmu_tx_commit(tx);

1974        mutex_enter(&vd->vdev_dtl_lock);
1975        space_map_update(vd->vdev_dtl_sm);
1976        mutex_exit(&vd->vdev_dtl_lock);
1977 }

1979 /*
1980  * Determine whether the specified vdev can be offlined/detached/removed
1981  * without losing data.
1982  */
1983 boolean_t
1984 vdev_dtl_required(vdev_t *vd)
1985 {
1986        spa_t *spa = vd->vdev_spa;
1987        vdev_t *tvd = vd->vdev_top;
1988        uint8_t cant_read = vd->vdev_cant_read;
1989        boolean_t required;

1991        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

1993        if (vd == spa->spa_root_vdev || vd == tvd)
1994                return (B_TRUE);

1996        /*
1997         * Temporarily mark the device as unreadable, and then determine
1998         * whether this results in any DTL outages in the top-level vdev.
1999         * If not, we can safely offline/detach/remove the device.
2000         */
2001        vd->vdev_cant_read = B_TRUE;
2002        vdev_dtl_reassess(tvd, 0, 0, B_FALSE);
2003        required = !vdev_dtl_empty(tvd, DTL_OUTAGE);
```



```
2004        vd->vdev_cant_read = cant_read;
2005        vdev_dtl_reassess(tvd, 0, 0, B_FALSE);

2007        if (!required && zio_injection_enabled)
2008                required = !!zio_handle_device_injection(vd, NULL, ECHILD);

2010        return (required);
2011 }

2013 /*
2014  * Determine if resilver is needed, and if so the txg range.
2015  */
2016 boolean_t
2017 vdev_resilver_needed(vdev_t *vd, uint64_t *minp, uint64_t *maxp)
2018 {
2019        boolean_t needed = B_FALSE;
2020        uint64_t thismin = UINT64_MAX;
2021        uint64_t thismax = 0;

2023        if (vd->vdev_children == 0) {
2024                mutex_enter(&vd->vdev_dtl_lock);
2025                if (range_tree_space(vd->vdev_dtl[DTL_MISSING]) != 0 &&
2026                    vdev_writeable(vd)) {

2028                        thismin = vdev_dtl_min(vd);
2029                        thismax = vdev_dtl_max(vd);
2030                        needed = B_TRUE;
2031                }
2032                mutex_exit(&vd->vdev_dtl_lock);
2033        } else {
2034                for (int c = 0; c < vd->vdev_children; c++) {
2035                        vdev_t *cvd = vd->vdev_child[c];
2036                        uint64_t cmin, cmax;

2038                        if (vdev_resilver_needed(cvd, &cmin, &cmax)) {
2039                                thismin = MIN(thismin, cmin);
2040                                thismax = MAX(thismax, cmax);
2041                                needed = B_TRUE;
2042                        }
2043                }
2044        }

2046        if (needed && minp) {
2047                *minp = thismin;
2048                *maxp = thismax;
2049        }
2050        return (needed);
2051 }

2053 void
2054 vdev_load(vdev_t *vd)
2055 {
2056        /*
2057         * Recursively load all children.
2058         */
2059        for (int c = 0; c < vd->vdev_children; c++)
2060                vdev_load(vd->vdev_child[c]);

2062        /*
2063         * If this is a top-level vdev, initialize its metaslabs.
2064         */
2065        if (vd == vd->vdev_top && !vd->vdev_ishole &&
2066            (vd->vdev_ashift == 0 || vd->vdev_asize == 0 ||
2067            vdev_metaslab_init(vd, 0) != 0))
2068                vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
2069                    VDEV_AUX_CORRUPT_DATA);
```

```
2071         /*
2072          * If this is a leaf vdev, load its DTL.
2073          */
2074         if (vd->vdev_ops->vdev_op_leaf && vdev_dtl_load(vd) != 0)
2075                 vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
2076                     VDEV_AUX_CORRUPT_DATA);
2077 }

2079 /*
2080  * The special vdev case is used for hot spares and l2cache devices.  Its
2081  * sole purpose it to set the vdev state for the associated vdev.  To do this,
2082  * we make sure that we can open the underlying device, then try to read the
2083  * label, and make sure that the label is sane and that it hasn't been
2084  * repurposed to another pool.
2085  */
2086 int
2087 vdev_validate_aux(vdev_t *vd)
2088 {
2089         nvlist_t *label;
2090         uint64_t guid, version;
2091         uint64_t state;

2093         if (!vdev_readable(vd))
2094                 return (0);

2096         if ((label = vdev_label_read_config(vd, -1ULL)) == NULL) {
2097                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
2098                     VDEV_AUX_CORRUPT_DATA);
2099                 return (-1);
2100         }

2102         if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_VERSION, &version) != 0 ||
2103             !SPA_VERSION_IS_SUPPORTED(version) ||
2104             nvlist_lookup_uint64(label, ZPOOL_CONFIG_GUID, &guid) != 0 ||
2105             guid != vd->vdev_guid ||
2106             nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_STATE, &state) != 0) {
2107                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
2108                     VDEV_AUX_CORRUPT_DATA);
2109                 nvlist_free(label);
2110                 return (-1);
2111         }

2113         /*
2114          * We don't actually check the pool state here.  If it's in fact in
2115          * use by another pool, we update this fact on the fly when requested.
2116          */
2117         nvlist_free(label);
2118         return (0);
2119 }

2121 void
2122 vdev_remove(vdev_t *vd, uint64_t txg)
2123 {
2124         spa_t *spa = vd->vdev_spa;
2125         objset_t *mos = spa->spa_meta_objset;
2126         dmu_tx_t *tx;

2128         tx = dmu_tx_create_assigned(spa_get_dsl(spa), txg);

2130         if (vd->vdev_ms != NULL) {
2131                 for (int m = 0; m < vd->vdev_ms_count; m++) {
2132                         metaslab_t *msp = vd->vdev_ms[m];

2134                         if (msp == NULL || msp->ms_sm == NULL)
2135                                 continue;
```



```
2137                         mutex_enter(&msp->ms_lock);
2138                         VERIFY0(space_map_allocated(msp->ms_sm));
2139                         space_map_free(msp->ms_sm, tx);
2140                         space_map_close(msp->ms_sm);
2141                         msp->ms_sm = NULL;
2142                         mutex_exit(&msp->ms_lock);
2143                 }
2144         }

2146         if (vd->vdev_ms_array) {
2147                 (void) dmu_object_free(mos, vd->vdev_ms_array, tx);
2148                 vd->vdev_ms_array = 0;
2149         }
2150         dmu_tx_commit(tx);
2151 }

2153 void
2154 vdev_sync_done(vdev_t *vd, uint64_t txg)
2155 {
2156         metaslab_t *msp;
2157         boolean_t reassess = !txg_list_empty(&vd->vdev_ms_list, TXG_CLEAN(txg));

2159         ASSERT(!vd->vdev_ishole);

2161         while (msp = txg_list_remove(&vd->vdev_ms_list, TXG_CLEAN(txg)))
2162                 metaslab_sync_done(msp, txg);

2164         if (reassess)
2165                 metaslab_sync_reassess(vd->vdev_mg);
2166 }

2168 void
2169 vdev_sync(vdev_t *vd, uint64_t txg)
2170 {
2171         spa_t *spa = vd->vdev_spa;
2172         vdev_t *lvd;
2173         metaslab_t *msp;
2174         dmu_tx_t *tx;

2176         ASSERT(!vd->vdev_ishole);

2178         if (vd->vdev_ms_array == 0 && vd->vdev_ms_shift != 0) {
2179                 ASSERT(vd == vd->vdev_top);
2180                 tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);
2181                 vd->vdev_ms_array = dmu_object_alloc(spa->spa_meta_objset,
2182                     DMU_OT_OBJECT_ARRAY, 0, DMU_OT_NONE, 0, tx);
2183                 ASSERT(vd->vdev_ms_array != 0);
2184                 vdev_config_dirty(vd);
2185                 dmu_tx_commit(tx);
2186         }

2188         /*
2189          * Remove the metadata associated with this vdev once it's empty.
2190          */
2191         if (vd->vdev_stat.vs_alloc == 0 && vd->vdev_removing)
2192                 vdev_remove(vd, txg);

2194         while ((msp = txg_list_remove(&vd->vdev_ms_list, txg)) != NULL) {
2195                 metaslab_sync(msp, txg);
2196                 (void) txg_list_add(&vd->vdev_ms_list, msp, TXG_CLEAN(txg));
2197         }

2199         while ((lvd = txg_list_remove(&vd->vdev_dtl_list, txg)) != NULL)
2200                 vdev_dtl_sync(lvd, txg);
```

```
2202         (void) txg_list_add(&spa->spa_vdev_txg_list, vd, TXG_CLEAN(txg));
2203 }

2205 uint64_t
2206 vdev_psize_to_asize(vdev_t *vd, uint64_t psize)
2207 {
2208         return (vd->vdev_ops->vdev_op_asize(vd, psize));
2209 }

2211 /*
2212  * Mark the given vdev faulted.  A faulted vdev behaves as if the device could
2213  * not be opened, and no I/O is attempted.
2214  */
2215 int
2216 vdev_fault(spa_t *spa, uint64_t guid, vdev_aux_t aux)
2217 {
2218         vdev_t *vd, *tvd;

2220         spa_vdev_state_enter(spa, SCL_NONE);

2222         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2223                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2225         if (!vd->vdev_ops->vdev_op_leaf)
2226                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2228         tvd = vd->vdev_top;

2230         /*
2231          * We don't directly use the aux state here, but if we do a
2232          * vdev_reopen(), we need this value to be present to remember why we
2233          * were faulted.
2234          */
2235         vd->vdev_label_aux = aux;

2237         /*
2238          * Faulted state takes precedence over degraded.
2239          */
2240         vd->vdev_delayed_close = B_FALSE;
2241         vd->vdev_faulted = 1ULL;
2242         vd->vdev_degraded = 0ULL;
2243         vdev_set_state(vd, B_FALSE, VDEV_STATE_FAULTED, aux);

2245         /*
2246          * If this device has the only valid copy of the data, then
2247          * back off and simply mark the vdev as degraded instead.
2248          */
2249         if (!tvd->vdev_islog && vd->vdev_aux == NULL && vdev_dtl_required(vd)) {
2250                 vd->vdev_degraded = 1ULL;
2251                 vd->vdev_faulted = 0ULL;

2253                 /*
2254                  * If we reopen the device and it's not dead, only then do we
2255                  * mark it degraded.
2256                  */
2257                 vdev_reopen(tvd);

2259                 if (vdev_readable(vd))
2260                         vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED, aux);
2261         }

2263         return (spa_vdev_state_exit(spa, vd, 0));
2264 }

2266 /*
2267  * Mark the given vdev degraded.  A degraded vdev is purely an indication to the
```



```
2268  * user that something is wrong.  The vdev continues to operate as normal as far
2269  * as I/O is concerned.
2270  */
2271 int
2272 vdev_degrade(spa_t *spa, uint64_t guid, vdev_aux_t aux)
2273 {
2274         vdev_t *vd;

2276         spa_vdev_state_enter(spa, SCL_NONE);

2278         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2279                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2281         if (!vd->vdev_ops->vdev_op_leaf)
2282                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2284         /*
2285          * If the vdev is already faulted, then don't do anything.
2286          */
2287         if (vd->vdev_faulted || vd->vdev_degraded)
2288                 return (spa_vdev_state_exit(spa, NULL, 0));

2290         vd->vdev_degraded = 1ULL;
2291         if (!vdev_is_dead(vd))
2292                 vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED,
2293                     aux);

2295         return (spa_vdev_state_exit(spa, vd, 0));
2296 }

2298 /*
2299  * Online the given vdev.
2300  *
2301  * If 'ZFS_ONLINE_UNSPARE' is set, it implies two things.  First, any attached
2302  * spare device should be detached when the device finishes resilvering.
2303  * Second, the online should be treated like a 'test' online case, so no FMA
2304  * events are generated if the device fails to open.
2305  */
2306 int
2307 vdev_online(spa_t *spa, uint64_t guid, uint64_t flags, vdev_state_t *newstate)
2308 {
2309         vdev_t *vd, *tvd, *pvd, *rvd = spa->spa_root_vdev;

2311         spa_vdev_state_enter(spa, SCL_NONE);

2313         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2314                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2316         if (!vd->vdev_ops->vdev_op_leaf)
2317                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2319         tvd = vd->vdev_top;
2320         vd->vdev_offline = B_FALSE;
2321         vd->vdev_tmpoffline = B_FALSE;
2322         vd->vdev_checkremove = !!(flags & ZFS_ONLINE_CHECKREMOVE);
2323         vd->vdev_forcefault = !!(flags & ZFS_ONLINE_FORCEFAULT);

2325         /* XXX - L2ARC 1.0 does not support expansion */
2326         if (!vd->vdev_aux) {
2327                 for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2328                         pvd->vdev_expanding = !!(flags & ZFS_ONLINE_EXPAND);
2329         }

2331         vdev_reopen(tvd);
2332         vd->vdev_checkremove = vd->vdev_forcefault = B_FALSE;
```

```
2334         if (!vd->vdev_aux) {
2335                 for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2336                         pvd->vdev_expanding = B_FALSE;
2337         }
2338
2339         if (newstate)
2340                 *newstate = vd->vdev_state;
2341         if ((flags & ZFS_ONLINE_UNSPARE) &&
2342             !vdev_is_dead(vd) && vd->vdev_parent &&
2343             vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
2344             vd->vdev_parent->vdev_child[0] == vd)
2345                 vd->vdev_unspare = B_TRUE;
2346
2347         if ((flags & ZFS_ONLINE_EXPAND) || spa->spa_autoexpand) {
2348
2349                 /* XXX - L2ARC 1.0 does not support expansion */
2350                 if (vd->vdev_aux)
2351                         return (spa_vdev_state_exit(spa, vd, ENOTSUP));
2352                 spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
2353         }
2354         return (spa_vdev_state_exit(spa, vd, 0));
2355 }
2356
2357 static int
2358 vdev_offline_locked(spa_t *spa, uint64_t guid, uint64_t flags)
2359 {
2360         vdev_t *vd, *tvd;
2361         int error = 0;
2362         uint64_t generation;
2363         metaslab_group_t *mg;
2364
2365 top:
2366         spa_vdev_state_enter(spa, SCL_ALLOC);
2367
2368         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2369                 return (spa_vdev_state_exit(spa, NULL, ENODEV));
2370
2371         if (!vd->vdev_ops->vdev_op_leaf)
2372                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));
2373
2374         tvd = vd->vdev_top;
2375         mg = tvd->vdev_mg;
2376         generation = spa->spa_config_generation + 1;
2377
2378         /*
2379          * If the device isn't already offline, try to offline it.
2380          */
2381         if (!vd->vdev_offline) {
2382                 /*
2383                  * If this device has the only valid copy of some data,
2384                  * don't allow it to be offlined. Log devices are always
2385                  * expendable.
2386                  */
2387                 if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
2388                     vdev_dtl_required(vd))
2389                         return (spa_vdev_state_exit(spa, NULL, EBUSY));
2390
2391                 /*
2392                  * If the top-level is a slog and it has had allocations
2393                  * then proceed.  We check that the vdev's metaslab group
2394                  * is not NULL since it's possible that we may have just
2395                  * added this vdev but not yet initialized its metaslabs.
2396                  */
2397                 if (tvd->vdev_islog && mg != NULL) {
2398                         /*
2399                          * Prevent any future allocations.
```



```
2400                          */
2401                         metaslab_group_passivate(mg);
2402                         (void) spa_vdev_state_exit(spa, vd, 0);
2403
2404                         error = spa_offline_log(spa);
2405
2406                         spa_vdev_state_enter(spa, SCL_ALLOC);
2407
2408                         /*
2409                          * Check to see if the config has changed.
2410                          */
2411                         if (error || generation != spa->spa_config_generation) {
2412                                 metaslab_group_activate(mg);
2413                                 if (error)
2414                                         return (spa_vdev_state_exit(spa,
2415                                             vd, error));
2416                                 (void) spa_vdev_state_exit(spa, vd, 0);
2417                                 goto top;
2418                         }
2419                         ASSERT0(tvd->vdev_stat.vs_alloc);
2420                 }
2421
2422                 /*
2423                  * Offline this device and reopen its top-level vdev.
2424                  * If the top-level vdev is a log device then just offline
2425                  * it. Otherwise, if this action results in the top-level
2426                  * vdev becoming unusable, undo it and fail the request.
2427                  */
2428                 vd->vdev_offline = B_TRUE;
2429                 vdev_reopen(tvd);
2430
2431                 if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
2432                     vdev_is_dead(tvd)) {
2433                         vd->vdev_offline = B_FALSE;
2434                         vdev_reopen(tvd);
2435                         return (spa_vdev_state_exit(spa, NULL, EBUSY));
2436                 }
2437
2438                 /*
2439                  * Add the device back into the metaslab rotor so that
2440                  * once we online the device it's open for business.
2441                  */
2442                 if (tvd->vdev_islog && mg != NULL)
2443                         metaslab_group_activate(mg);
2444         }
2445
2446         vd->vdev_tmpoffline = !!(flags & ZFS_OFFLINE_TEMPORARY);
2447
2448         return (spa_vdev_state_exit(spa, vd, 0));
2449 }
2450
2451 int
2452 vdev_offline(spa_t *spa, uint64_t guid, uint64_t flags)
2453 {
2454         int error;
2455
2456         mutex_enter(&spa->spa_vdev_top_lock);
2457         error = vdev_offline_locked(spa, guid, flags);
2458         mutex_exit(&spa->spa_vdev_top_lock);
2459
2460         return (error);
2461 }
2462
2463 /*
2464  * Clear the error counts associated with this vdev.  Unlike vdev_online() and
2465  * vdev_offline(), we assume the spa config is locked.  We also clear all
```

```
 * children.  If 'vd' is NULL, then the user wants to clear all vdevs.
 */
void
vdev_clear(spa_t *spa, vdev_t *vd)
{
        vdev_t *rvd = spa->spa_root_vdev;

        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

        if (vd == NULL)
                vd = rvd;

        vd->vdev_stat.vs_read_errors = 0;
        vd->vdev_stat.vs_write_errors = 0;
        vd->vdev_stat.vs_checksum_errors = 0;

        for (int c = 0; c < vd->vdev_children; c++)
                vdev_clear(spa, vd->vdev_child[c]);

        /*
         * If we're in the FAULTED state or have experienced failed I/O, then
         * clear the persistent state and attempt to reopen the device.  We
         * also mark the vdev config dirty, so that the new faulted state is
         * written out to disk.
         */
        if (vd->vdev_faulted || vd->vdev_degraded ||
            !vdev_readable(vd) || !vdev_writeable(vd)) {

                /*
                 * When reopening in reponse to a clear event, it may be due to
                 * a fmadm repair request.  In this case, if the device is
                 * still broken, we want to still post the ereport again.
                 */
                vd->vdev_forcefault = B_TRUE;

                vd->vdev_faulted = vd->vdev_degraded = 0ULL;
                vd->vdev_cant_read = B_FALSE;
                vd->vdev_cant_write = B_FALSE;

                vdev_reopen(vd == rvd ? rvd : vd->vdev_top);

                vd->vdev_forcefault = B_FALSE;

                if (vd != rvd && vdev_writeable(vd->vdev_top))
                        vdev_state_dirty(vd->vdev_top);

                if (vd->vdev_aux == NULL && !vdev_is_dead(vd))
                        spa_async_request(spa, SPA_ASYNC_RESILVER);

                spa_event_notify(spa, vd, ESC_ZFS_VDEV_CLEAR);
        }

        /*
         * When clearing a FMA-diagnosed fault, we always want to
         * unspare the device, as we assume that the original spare was
         * done in response to the FMA fault.
         */
        if (!vdev_is_dead(vd) && vd->vdev_parent != NULL &&
            vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
            vd->vdev_parent->vdev_child[0] == vd)
                vd->vdev_unspare = B_TRUE;
}

boolean_t
vdev_is_dead(vdev_t *vd)
{
```



```
        /*
         * Holes and missing devices are always considered "dead".
         * This simplifies the code since we don't have to check for
         * these types of devices in the various code paths.
         * Instead we rely on the fact that we skip over dead devices
         * before issuing I/O to them.
         */
        return (vd->vdev_state < VDEV_STATE_DEGRADED || vd->vdev_ishole ||
            vd->vdev_ops == &vdev_missing_ops);
}

boolean_t
vdev_readable(vdev_t *vd)
{
        return (!vdev_is_dead(vd) && !vd->vdev_cant_read);
}

boolean_t
vdev_writeable(vdev_t *vd)
{
        return (!vdev_is_dead(vd) && !vd->vdev_cant_write);
}

boolean_t
vdev_allocatable(vdev_t *vd)
{
        uint64_t state = vd->vdev_state;

        /*
         * We currently allow allocations from vdevs which may be in the
         * process of reopening (i.e. VDEV_STATE_CLOSED). If the device
         * fails to reopen then we'll catch it later when we're holding
         * the proper locks.  Note that we have to get the vdev state
         * in a local variable because although it changes atomically,
         * we're asking two separate questions about it.
         */
        return (!(state < VDEV_STATE_DEGRADED && state != VDEV_STATE_CLOSED) &&
            !vd->vdev_cant_write && !vd->vdev_ishole);
}

boolean_t
vdev_accessible(vdev_t *vd, zio_t *zio)
{
        ASSERT(zio->io_vd == vd);

        if (vdev_is_dead(vd) || vd->vdev_remove_wanted)
                return (B_FALSE);

        if (zio->io_type == ZIO_TYPE_READ)
                return (!vd->vdev_cant_read);

        if (zio->io_type == ZIO_TYPE_WRITE)
                return (!vd->vdev_cant_write);

        return (B_TRUE);
}

/*
 * Get statistics for the given vdev.
 */
void
vdev_get_stats(vdev_t *vd, vdev_stat_t *vs)
{
        vdev_t *rvd = vd->vdev_spa->spa_root_vdev;

        mutex_enter(&vd->vdev_stat_lock);
```

```
2598         bcopy(&vd->vdev_stat, vs, sizeof (*vs));
2599         vs->vs_timestamp = gethrtime() - vs->vs_timestamp;
2600         vs->vs_state = vd->vdev_state;
2601         vs->vs_rsize = vdev_get_min_asize(vd);
2602         if (vd->vdev_ops->vdev_op_leaf)
2603                 vs->vs_rsize += VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE;
2604         vs->vs_esize = vd->vdev_max_asize - vd->vdev_asize;
2605         mutex_exit(&vd->vdev_stat_lock);

2607         /*
2608          * If we're getting stats on the root vdev, aggregate the I/O counts
2609          * over all top-level vdevs (i.e. the direct children of the root).
2610          */
2611         if (vd == rvd) {
2612                 for (int c = 0; c < rvd->vdev_children; c++) {
2613                         vdev_t *cvd = rvd->vdev_child[c];
2614                         vdev_stat_t *cvs = &cvd->vdev_stat;

2616                         mutex_enter(&vd->vdev_stat_lock);
2617                         for (int t = 0; t < ZIO_TYPES; t++) {
2618                                 vs->vs_ops[t] += cvs->vs_ops[t];
2619                                 vs->vs_bytes[t] += cvs->vs_bytes[t];
2620                         }
2621                         cvs->vs_scan_removing = cvd->vdev_removing;
2622                         mutex_exit(&vd->vdev_stat_lock);
2623                 }
2624         }
2625 }

2627 void
2628 vdev_clear_stats(vdev_t *vd)
2629 {
2630         mutex_enter(&vd->vdev_stat_lock);
2631         vd->vdev_stat.vs_space = 0;
2632         vd->vdev_stat.vs_dspace = 0;
2633         vd->vdev_stat.vs_alloc = 0;
2634         mutex_exit(&vd->vdev_stat_lock);
2635 }

2637 void
2638 vdev_scan_stat_init(vdev_t *vd)
2639 {
2640         vdev_stat_t *vs = &vd->vdev_stat;

2642         for (int c = 0; c < vd->vdev_children; c++)
2643                 vdev_scan_stat_init(vd->vdev_child[c]);

2645         mutex_enter(&vd->vdev_stat_lock);
2646         vs->vs_scan_processed = 0;
2647         mutex_exit(&vd->vdev_stat_lock);
2648 }

2650 void
2651 vdev_stat_update(zio_t *zio, uint64_t psize)
2652 {
2653         spa_t *spa = zio->io_spa;
2654         vdev_t *rvd = spa->spa_root_vdev;
2655         vdev_t *vd = zio->io_vd ? zio->io_vd : rvd;
2656         vdev_t *pvd;
2657         uint64_t txg = zio->io_txg;
2658         vdev_stat_t *vs = &vd->vdev_stat;
2659         zio_type_t type = zio->io_type;
2660         int flags = zio->io_flags;

2662         /*
2663          * If this i/o is a gang leader, it didn't do any actual work.
```



```
2664          */
2665         if (zio->io_gang_tree)
2666                 return;

2668         if (zio->io_error == 0) {
2669                 /*
2670                  * If this is a root i/o, don't count it -- we've already
2671                  * counted the top-level vdevs, and vdev_get_stats() will
2672                  * aggregate them when asked.  This reduces contention on
2673                  * the root vdev_stat_lock and implicitly handles blocks
2674                  * that compress away to holes, for which there is no i/o.
2675                  * (Holes never create vdev children, so all the counters
2676                  * remain zero, which is what we want.)
2677                  *
2678                  * Note: this only applies to successful i/o (io_error == 0)
2679                  * because unlike i/o counts, errors are not additive.
2680                  * When reading a ditto block, for example, failure of
2681                  * one top-level vdev does not imply a root-level error.
2682                  */
2683                 if (vd == rvd)
2684                         return;

2686                 ASSERT(vd == zio->io_vd);

2688                 if (flags & ZIO_FLAG_IO_BYPASS)
2689                         return;

2691                 mutex_enter(&vd->vdev_stat_lock);

2693                 if (flags & ZIO_FLAG_IO_REPAIR) {
2694                         if (flags & ZIO_FLAG_SCAN_THREAD) {
2695                                 dsl_scan_phys_t *scn_phys =
2696                                     &spa->spa_dsl_pool->dp_scan->scn_phys;
2697                                 uint64_t *processed = &scn_phys->scn_processed;

2699                                 /* XXX cleanup? */
2700                                 if (vd->vdev_ops->vdev_op_leaf)
2701                                         atomic_add_64(processed, psize);
2702                                 vs->vs_scan_processed += psize;
2703                         }

2705                         if (flags & ZIO_FLAG_SELF_HEAL)
2706                                 vs->vs_self_healed += psize;
2707                 }

2709                 vs->vs_ops[type]++;
2710                 vs->vs_bytes[type] += psize;

2712                 mutex_exit(&vd->vdev_stat_lock);
2713                 return;
2714         }

2716         if (flags & ZIO_FLAG_SPECULATIVE)
2717                 return;

2719         /*
2720          * If this is an I/O error that is going to be retried, then ignore the
2721          * error.  Otherwise, the user may interpret B_FAILFAST I/O errors as
2722          * hard errors, when in reality they can happen for any number of
2723          * innocuous reasons (bus resets, MPxIO link failure, etc).
2724          */
2725         if (zio->io_error == EIO &&
2726             !(zio->io_flags & ZIO_FLAG_IO_RETRY))
2727                 return;

2729         /*
```

```
2730          * Intent logs writes won't propagate their error to the root
2731          * I/O so don't mark these types of failures as pool-level
2732          * errors.
2733          */
2734         if (zio->io_vd == NULL && (zio->io_flags & ZIO_FLAG_DONT_PROPAGATE))
2735                 return;

2737         mutex_enter(&vd->vdev_stat_lock);
2738         if (type == ZIO_TYPE_READ && !vdev_is_dead(vd)) {
2739                 if (zio->io_error == ECKSUM)
2740                         vs->vs_checksum_errors++;
2741                 else
2742                         vs->vs_read_errors++;
2743         }
2744         if (type == ZIO_TYPE_WRITE && !vdev_is_dead(vd))
2745                 vs->vs_write_errors++;
2746         mutex_exit(&vd->vdev_stat_lock);

2748         if (type == ZIO_TYPE_WRITE && txg != 0 &&
2749             (!(flags & ZIO_FLAG_IO_REPAIR) ||
2750             (flags & ZIO_FLAG_SCAN_THREAD) ||
2751             spa->spa_claiming)) {
2752                 /*
2753                  * This is either a normal write (not a repair), or it's
2754                  * a repair induced by the scrub thread, or it's a repair
2755                  * made by zil_claim() during spa_load() in the first txg.
2756                  * In the normal case, we commit the DTL change in the same
2757                  * txg as the block was born.  In the scrub-induced repair
2758                  * case, we know that scrubs run in first-pass syncing context,
2759                  * so we commit the DTL change in spa_syncing_txg(spa).
2760                  * In the zil_claim() case, we commit in spa_first_txg(spa).
2761                  *
2762                  * We currently do not make DTL entries for failed spontaneous
2763                  * self-healing writes triggered by normal (non-scrubbing)
2764                  * reads, because we have no transactional context in which to
2765                  * do so -- and it's not clear that it'd be desirable anyway.
2766                  */
2767                 if (vd->vdev_ops->vdev_op_leaf) {
2768                         uint64_t commit_txg = txg;
2769                         if (flags & ZIO_FLAG_SCAN_THREAD) {
2770                                 ASSERT(flags & ZIO_FLAG_IO_REPAIR);
2771                                 ASSERT(spa_sync_pass(spa) == 1);
2772                                 vdev_dtl_dirty(vd, DTL_SCRUB, txg, 1);
2773                                 commit_txg = spa_syncing_txg(spa);
2774                         } else if (spa->spa_claiming) {
2775                                 ASSERT(flags & ZIO_FLAG_IO_REPAIR);
2776                                 commit_txg = spa_first_txg(spa);
2777                         }
2778                         ASSERT(commit_txg >= spa_syncing_txg(spa));
2779                         if (vdev_dtl_contains(vd, DTL_MISSING, txg, 1))
2780                                 return;
2781                         for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2782                                 vdev_dtl_dirty(pvd, DTL_PARTIAL, txg, 1);
2783                         vdev_dirty(vd->vdev_top, VDD_DTL, vd, commit_txg);
2784                 }
2785                 if (vd != rvd)
2786                         vdev_dtl_dirty(vd, DTL_MISSING, txg, 1);
2787         }
2788 }

2790 /*
2791  * Update the in-core space usage stats for this vdev, its metaslab class,
2792  * and the root vdev.
2793  */
2794 void
2795 vdev_space_update(vdev_t *vd, int64_t alloc_delta, int64_t defer_delta,
```



```
2796     int64_t space_delta)
2797 {
2798         int64_t dspace_delta = space_delta;
2799         spa_t *spa = vd->vdev_spa;
2800         vdev_t *rvd = spa->spa_root_vdev;
2801         metaslab_group_t *mg = vd->vdev_mg;
2802         metaslab_class_t *mc = mg ? mg->mg_class : NULL;

2804         ASSERT(vd == vd->vdev_top);

2806         /*
2807          * Apply the inverse of the psize-to-asize (ie. RAID-Z) space-expansion
2808          * factor.  We must calculate this here and not at the root vdev
2809          * because the root vdev's psize-to-asize is simply the max of its
2810          * childrens', thus not accurate enough for us.
2811          */
2812         ASSERT((dspace_delta & (SPA_MINBLOCKSIZE-1)) == 0);
2813         ASSERT(vd->vdev_deflate_ratio != 0 || vd->vdev_isl2cache);
2814         dspace_delta = (dspace_delta >> SPA_MINBLOCKSHIFT) *
2815             vd->vdev_deflate_ratio;

2817         mutex_enter(&vd->vdev_stat_lock);
2818         vd->vdev_stat.vs_alloc += alloc_delta;
2819         vd->vdev_stat.vs_space += space_delta;
2820         vd->vdev_stat.vs_dspace += dspace_delta;
2821         mutex_exit(&vd->vdev_stat_lock);

2823         if (mc == spa_normal_class(spa)) {
2824                 mutex_enter(&rvd->vdev_stat_lock);
2825                 rvd->vdev_stat.vs_alloc += alloc_delta;
2826                 rvd->vdev_stat.vs_space += space_delta;
2827                 rvd->vdev_stat.vs_dspace += dspace_delta;
2828                 mutex_exit(&rvd->vdev_stat_lock);
2829         }

2831         if (mc != NULL) {
2832                 ASSERT(rvd == vd->vdev_parent);
2833                 ASSERT(vd->vdev_ms_count != 0);

2835                 metaslab_class_space_update(mc,
2836                     alloc_delta, defer_delta, space_delta, dspace_delta);
2837         }
2838 }

2840 /*
2841  * Mark a top-level vdev's config as dirty, placing it on the dirty list
2842  * so that it will be written out next time the vdev configuration is synced.
2843  * If the root vdev is specified (vdev_top == NULL), dirty all top-level vdevs.
2844  */
2845 void
2846 vdev_config_dirty(vdev_t *vd)
2847 {
2848         spa_t *spa = vd->vdev_spa;
2849         vdev_t *rvd = spa->spa_root_vdev;
2850         int c;

2852         ASSERT(spa_writeable(spa));

2854         /*
2855          * If this is an aux vdev (as with l2cache and spare devices), then we
2856          * update the vdev config manually and set the sync flag.
2857          */
2858         if (vd->vdev_aux != NULL) {
2859                 spa_aux_vdev_t *sav = vd->vdev_aux;
2860                 nvlist_t **aux;
2861                 uint_t naux;
```

```
2863                 for (c = 0; c < sav->sav_count; c++) {
2864                         if (sav->sav_vdevs[c] == vd)
2865                                 break;
2866                 }
2868                 if (c == sav->sav_count) {
2869                         /*
2870                          * We're being removed.  There's nothing more to do.
2871                          */
2872                         ASSERT(sav->sav_sync == B_TRUE);
2873                         return;
2874                 }
2876                 sav->sav_sync = B_TRUE;
2878                 if (nvlist_lookup_nvlist_array(sav->sav_config,
2879                     ZPOOL_CONFIG_L2CACHE, &aux, &naux) != 0) {
2880                         VERIFY(nvlist_lookup_nvlist_array(sav->sav_config,
2881                             ZPOOL_CONFIG_SPARES, &aux, &naux) == 0);
2882                 }
2884                 ASSERT(c < naux);
2886                 /*
2887                  * Setting the nvlist in the middle if the array is a little
2888                  * sketchy, but it will work.
2889                  */
2890                 nvlist_free(aux[c]);
2891                 aux[c] = vdev_config_generate(spa, vd, B_TRUE, 0);
2893                 return;
2894         }
2896         /*
2897          * The dirty list is protected by the SCL_CONFIG lock.  The caller
2898          * must either hold SCL_CONFIG as writer, or must be the sync thread
2899          * (which holds SCL_CONFIG as reader).  There's only one sync thread,
2900          * so this is sufficient to ensure mutual exclusion.
2901          */
2902         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_WRITER) ||
2903             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2904             spa_config_held(spa, SCL_CONFIG, RW_READER)));
2906         if (vd == rvd) {
2907                 for (c = 0; c < rvd->vdev_children; c++)
2908                         vdev_config_dirty(rvd->vdev_child[c]);
2909         } else {
2910                 ASSERT(vd == vd->vdev_top);
2912                 if (!list_link_active(&vd->vdev_config_dirty_node) &&
2913                     !vd->vdev_ishole)
2914                         list_insert_head(&spa->spa_config_dirty_list, vd);
2915         }
2916 }
2918 void
2919 vdev_config_clean(vdev_t *vd)
2920 {
2921         spa_t *spa = vd->vdev_spa;
2923         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_WRITER) ||
2924             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2925             spa_config_held(spa, SCL_CONFIG, RW_READER)));
2927         ASSERT(list_link_active(&vd->vdev_config_dirty_node));
```



```
2928         list_remove(&spa->spa_config_dirty_list, vd);
2929 }
2931 /*
2932  * Mark a top-level vdev's state as dirty, so that the next pass of
2933  * spa_sync() can convert this into vdev_config_dirty().  We distinguish
2934  * the state changes from larger config changes because they require
2935  * much less locking, and are often needed for administrative actions.
2936  */
2937 void
2938 vdev_state_dirty(vdev_t *vd)
2939 {
2940         spa_t *spa = vd->vdev_spa;
2942         ASSERT(spa_writeable(spa));
2943         ASSERT(vd == vd->vdev_top);
2945         /*
2946          * The state list is protected by the SCL_STATE lock.  The caller
2947          * must either hold SCL_STATE as writer, or must be the sync thread
2948          * (which holds SCL_STATE as reader).  There's only one sync thread,
2949          * so this is sufficient to ensure mutual exclusion.
2950          */
2951         ASSERT(spa_config_held(spa, SCL_STATE, RW_WRITER) ||
2952             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2953             spa_config_held(spa, SCL_STATE, RW_READER)));
2955         if (!list_link_active(&vd->vdev_state_dirty_node) && !vd->vdev_ishole)
2956                 list_insert_head(&spa->spa_state_dirty_list, vd);
2957 }
2959 void
2960 vdev_state_clean(vdev_t *vd)
2961 {
2962         spa_t *spa = vd->vdev_spa;
2964         ASSERT(spa_config_held(spa, SCL_STATE, RW_WRITER) ||
2965             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2966             spa_config_held(spa, SCL_STATE, RW_READER)));
2968         ASSERT(list_link_active(&vd->vdev_state_dirty_node));
2969         list_remove(&spa->spa_state_dirty_list, vd);
2970 }
2972 /*
2973  * Propagate vdev state up from children to parent.
2974  */
2975 void
2976 vdev_propagate_state(vdev_t *vd)
2977 {
2978         spa_t *spa = vd->vdev_spa;
2979         vdev_t *rvd = spa->spa_root_vdev;
2980         int degraded = 0, faulted = 0;
2981         int corrupted = 0;
2982         vdev_t *child;
2984         if (vd->vdev_children > 0) {
2985                 for (int c = 0; c < vd->vdev_children; c++) {
2986                         child = vd->vdev_child[c];
2988                         /*
2989                          * Don't factor holes into the decision.
2990                          */
2991                         if (child->vdev_ishole)
2992                                 continue;
```

```
2994                        if (!vdev_readable(child) ||
2995                            (!vdev_writeable(child) && spa_writeable(spa))) {
2996                                /*
2997                                 * Root special: if there is a top-level log
2998                                 * device, treat the root vdev as if it were
2999                                 * degraded.
3000                                 */
3001                                if (child->vdev_islog && vd == rvd)
3002                                        degraded++;
3003                                else
3004                                        faulted++;
3005                        } else if (child->vdev_state <= VDEV_STATE_DEGRADED) {
3006                                degraded++;
3007                        }
3008
3009                        if (child->vdev_stat.vs_aux == VDEV_AUX_CORRUPT_DATA)
3010                                corrupted++;
3011                }
3012
3013                vd->vdev_ops->vdev_op_state_change(vd, faulted, degraded);
3014
3015                /*
3016                 * Root special: if there is a top-level vdev that cannot be
3017                 * opened due to corrupted metadata, then propagate the root
3018                 * vdev's aux state as 'corrupt' rather than 'insufficient
3019                 * replicas'.
3020                 */
3021                if (corrupted && vd == rvd &&
3022                    rvd->vdev_state == VDEV_STATE_CANT_OPEN)
3023                        vdev_set_state(rvd, B_FALSE, VDEV_STATE_CANT_OPEN,
3024                            VDEV_AUX_CORRUPT_DATA);
3025        }
3026
3027        if (vd->vdev_parent)
3028                vdev_propagate_state(vd->vdev_parent);
3029 }
3030
3031 /*
3032  * Set a vdev's state.  If this is during an open, we don't update the parent
3033  * state, because we're in the process of opening children depth-first.
3034  * Otherwise, we propagate the change to the parent.
3035  *
3036  * If this routine places a device in a faulted state, an appropriate ereport is
3037  * generated.
3038  */
3039 void
3040 vdev_set_state(vdev_t *vd, boolean_t isopen, vdev_state_t state, vdev_aux_t aux)
3041 {
3042        uint64_t save_state;
3043        spa_t *spa = vd->vdev_spa;
3044
3045        if (state == vd->vdev_state) {
3046                vd->vdev_stat.vs_aux = aux;
3047                return;
3048        }
3049
3050        save_state = vd->vdev_state;
3051
3052        vd->vdev_state = state;
3053        vd->vdev_stat.vs_aux = aux;
3054
3055        /*
3056         * If we are setting the vdev state to anything but an open state, then
3057         * always close the underlying device unless the device has requested
3058         * a delayed close (i.e. we're about to remove or fault the device).
3059         * Otherwise, we keep accessible but invalid devices open forever.
```



```
3060         * We don't call vdev_close() itself, because that implies some extra
3061         * checks (offline, etc) that we don't want here.  This is limited to
3062         * leaf devices, because otherwise closing the device will affect other
3063         * children.
3064         */
3065        if (!vd->vdev_delayed_close && vdev_is_dead(vd) &&
3066            vd->vdev_ops->vdev_op_leaf)
3067                vd->vdev_ops->vdev_op_close(vd);
3068
3069        /*
3070         * If we have brought this vdev back into service, we need
3071         * to notify fmd so that it can gracefully repair any outstanding
3072         * cases due to a missing device.  We do this in all cases, even those
3073         * that probably don't correlate to a repaired fault.  This is sure to
3074         * catch all cases, and we let the zfs-retire agent sort it out.  If
3075         * this is a transient state it's OK, as the retire agent will
3076         * double-check the state of the vdev before repairing it.
3077         */
3078        if (state == VDEV_STATE_HEALTHY && vd->vdev_ops->vdev_op_leaf &&
3079            vd->vdev_prevstate != state)
3080                zfs_post_state_change(spa, vd);
3081
3082        if (vd->vdev_removed &&
3083            state == VDEV_STATE_CANT_OPEN &&
3084            (aux == VDEV_AUX_OPEN_FAILED || vd->vdev_checkremove)) {
3085                /*
3086                 * If the previous state is set to VDEV_STATE_REMOVED, then this
3087                 * device was previously marked removed and someone attempted to
3088                 * reopen it.  If this failed due to a nonexistent device, then
3089                 * keep the device in the REMOVED state.  We also let this be if
3090                 * it is one of our special test online cases, which is only
3091                 * attempting to online the device and shouldn't generate an FMA
3092                 * fault.
3093                 */
3094                vd->vdev_state = VDEV_STATE_REMOVED;
3095                vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
3096        } else if (state == VDEV_STATE_REMOVED) {
3097                vd->vdev_removed = B_TRUE;
3098        } else if (state == VDEV_STATE_CANT_OPEN) {
3099                /*
3100                 * If we fail to open a vdev during an import or recovery, we
3101                 * mark it as "not available", which signifies that it was
3102                 * never there to begin with.  Failure to open such a device
3103                 * is not considered an error.
3104                 */
3105                if ((spa_load_state(spa) == SPA_LOAD_IMPORT ||
3106                    spa_load_state(spa) == SPA_LOAD_RECOVER) &&
3107                    vd->vdev_ops->vdev_op_leaf)
3108                        vd->vdev_not_present = 1;
3109
3110                /*
3111                 * Post the appropriate ereport.  If the 'prevstate' field is
3112                 * set to something other than VDEV_STATE_UNKNOWN, it indicates
3113                 * that this is part of a vdev_reopen().  In this case, we don't
3114                 * want to post the ereport if the device was already in the
3115                 * CANT_OPEN state beforehand.
3116                 *
3117                 * If the 'checkremove' flag is set, then this is an attempt to
3118                 * online the device in response to an insertion event.  If we
3119                 * hit this case, then we have detected an insertion event for a
3120                 * faulted or offline device that wasn't in the removed state.
3121                 * In this scenario, we don't post an ereport because we are
3122                 * about to replace the device, or attempt an online with
3123                 * vdev_forcefault, which will generate the fault for us.
3124                 */
3125                if ((vd->vdev_prevstate != state || vd->vdev_forcefault) &&
```

```
3126                     !vd->vdev_not_present && !vd->vdev_checkremove &&
3127                     vd != spa->spa_root_vdev) {
3128                         const char *class;

3130                         switch (aux) {
3131                         case VDEV_AUX_OPEN_FAILED:
3132                                 class = FM_EREPORT_ZFS_DEVICE_OPEN_FAILED;
3133                                 break;
3134                         case VDEV_AUX_CORRUPT_DATA:
3135                                 class = FM_EREPORT_ZFS_DEVICE_CORRUPT_DATA;
3136                                 break;
3137                         case VDEV_AUX_NO_REPLICAS:
3138                                 class = FM_EREPORT_ZFS_DEVICE_NO_REPLICAS;
3139                                 break;
3140                         case VDEV_AUX_BAD_GUID_SUM:
3141                                 class = FM_EREPORT_ZFS_DEVICE_BAD_GUID_SUM;
3142                                 break;
3143                         case VDEV_AUX_TOO_SMALL:
3144                                 class = FM_EREPORT_ZFS_DEVICE_TOO_SMALL;
3145                                 break;
3146                         case VDEV_AUX_BAD_LABEL:
3147                                 class = FM_EREPORT_ZFS_DEVICE_BAD_LABEL;
3148                                 break;
3149                         default:
3150                                 class = FM_EREPORT_ZFS_DEVICE_UNKNOWN;
3151                         }

3153                         zfs_ereport_post(class, spa, vd, NULL, save_state, 0);
3154                 }

3156                 /* Erase any notion of persistent removed state */
3157                 vd->vdev_removed = B_FALSE;
3158         } else {
3159                 vd->vdev_removed = B_FALSE;
3160         }

3162         if (!isopen && vd->vdev_parent)
3163                 vdev_propagate_state(vd->vdev_parent);
3164 }

3166 /*
3167  * Check the vdev configuration to ensure that it's capable of supporting
3168  * a root pool. Currently, we do not support RAID-Z or partial configuration.
3169  * In addition, only a single top-level vdev is allowed and none of the leaves
3170  * can be wholedisks.
3171  */
3172 boolean_t
3173 vdev_is_bootable(vdev_t *vd)
3174 {
3175         if (!vd->vdev_ops->vdev_op_leaf) {
3176                 char *vdev_type = vd->vdev_ops->vdev_op_type;

3178                 if (strcmp(vdev_type, VDEV_TYPE_ROOT) == 0 &&
3179                     vd->vdev_children > 1) {
3180                         return (B_FALSE);
3181                 } else if (strcmp(vdev_type, VDEV_TYPE_RAIDZ) == 0 ||
3182                     strcmp(vdev_type, VDEV_TYPE_MISSING) == 0) {
3183                         return (B_FALSE);
3184                 }
3185         } else if (vd->vdev_wholedisk == 1) {
3186                 return (B_FALSE);
3187         }

3189         for (int c = 0; c < vd->vdev_children; c++) {
3190                 if (!vdev_is_bootable(vd->vdev_child[c]))
3191                         return (B_FALSE);
```



```
3192         }
3193         return (B_TRUE);
3194 }

3196 /*
3197  * Load the state from the original vdev tree (ovd) which
3198  * we've retrieved from the MOS config object. If the original
3199  * vdev was offline or faulted then we transfer that state to the
3200  * device in the current vdev tree (nvd).
3201  */
3202 void
3203 vdev_load_log_state(vdev_t *nvd, vdev_t *ovd)
3204 {
3205         spa_t *spa = nvd->vdev_spa;

3207         ASSERT(nvd->vdev_top->vdev_islog);
3208         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
3209         ASSERT3U(nvd->vdev_guid, ==, ovd->vdev_guid);

3211         for (int c = 0; c < nvd->vdev_children; c++)
3212                 vdev_load_log_state(nvd->vdev_child[c], ovd->vdev_child[c]);

3214         if (nvd->vdev_ops->vdev_op_leaf) {
3215                 /*
3216                  * Restore the persistent vdev state
3217                  */
3218                 nvd->vdev_offline = ovd->vdev_offline;
3219                 nvd->vdev_faulted = ovd->vdev_faulted;
3220                 nvd->vdev_degraded = ovd->vdev_degraded;
3221                 nvd->vdev_removed = ovd->vdev_removed;
3222         }
3223 }

3225 /*
3226  * Determine if a log device has valid content.  If the vdev was
3227  * removed or faulted in the MOS config then we know that
3228  * the content on the log device has already been written to the pool.
3229  */
3230 boolean_t
3231 vdev_log_state_valid(vdev_t *vd)
3232 {
3233         if (vd->vdev_ops->vdev_op_leaf && !vd->vdev_faulted &&
3234             !vd->vdev_removed)
3235                 return (B_TRUE);

3237         for (int c = 0; c < vd->vdev_children; c++)
3238                 if (vdev_log_state_valid(vd->vdev_child[c]))
3239                         return (B_TRUE);

3241         return (B_FALSE);
3242 }

3244 /*
3245  * Expand a vdev if possible.
3246  */
3247 void
3248 vdev_expand(vdev_t *vd, uint64_t txg)
3249 {
3250         ASSERT(vd->vdev_top == vd);
3251         ASSERT(spa_config_held(vd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);

3253         if ((vd->vdev_asize >> vd->vdev_ms_shift) > vd->vdev_ms_count) {
3254                 VERIFY(vdev_metaslab_init(vd, txg) == 0);
3255                 vdev_config_dirty(vd);
3256         }
3257 }
```

```
3259 /*
3260  * Split a vdev.
3261  */
3262 void
3263 vdev_split(vdev_t *vd)
3264 {
3265         vdev_t *cvd, *pvd = vd->vdev_parent;
3267         vdev_remove_child(pvd, vd);
3268         vdev_compact_children(pvd);
3270         cvd = pvd->vdev_child[0];
3271         if (pvd->vdev_children == 1) {
3272                 vdev_remove_parent(cvd);
3273                 cvd->vdev_splitting = B_TRUE;
3274         }
3275         vdev_propagate_state(cvd);
3276 }
3278 void
3279 vdev_deadman(vdev_t *vd)
3280 {
3281         for (int c = 0; c < vd->vdev_children; c++) {
3282                 vdev_t *cvd = vd->vdev_child[c];
3284                 vdev_deadman(cvd);
3285         }
3287         if (vd->vdev_ops->vdev_op_leaf) {
3288                 vdev_queue_t *vq = &vd->vdev_queue;
3290                 mutex_enter(&vq->vq_lock);
3291                 if (avl_numnodes(&vq->vq_active_tree) > 0) {
3292                         spa_t *spa = vd->vdev_spa;
3293                         zio_t *fio;
3294                         uint64_t delta;
3296                         /*
3297                          * Look at the head of all the pending queues,
3298                          * if any I/O has been outstanding for longer than
3299                          * the spa_deadman_synctime we panic the system.
3300                          */
3301                         fio = avl_first(&vq->vq_active_tree);
3302                         delta = gethrtime() - fio->io_timestamp;
3303                         if (delta > spa_deadman_synctime(spa)) {
3304                                 zfs_dbgmsg("SLOW IO: zio timestamp %lluns, "
3305                                     "delta %lluns, last io %lluns",
3306                                     fio->io_timestamp, delta,
3307                                     vq->vq_io_complete_ts);
3308                                 fm_panic("I/O to pool '%s' appears to be "
3309                                     "hung.", spa_name(spa));
3310                         }
3311                 }
3312                 mutex_exit(&vq->vq_lock);
3313         }
3314 }
```