```
**********************************************************
   56035 Mon Aug 26 18:30:10 2013
new/usr/src/uts/common/fs/zfs/metaslab.c
3954 metaslabs continue to load even after hitting zfs_mg_alloc_failure limit
4080 zpool clear fails to clear pool
4081 need zfs_mg_noalloc_threshold
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/dmu.h>
#include <sys/dmu_tx.h>
#include <sys/space_map.h>
#include <sys/metaslab_impl.h>
#include <sys/vdev_impl.h>
#include <sys/zio.h>

/*
 * Allow allocations to switch to gang blocks quickly. We do this to
 * avoid having to load lots of space_maps in a given txg. There are,
 * however, some cases where we want to avoid "fast" ganging and instead
 * we want to do an exhaustive search of all metaslabs on this device.
 * Currently we don't allow any gang, zil, or dump device related allocations
 * to "fast" gang.
 */
#define CAN_FASTGANG(flags) \
        (!((flags) & (METASLAB_GANG_CHILD | METASLAB_GANG_HEADER | \
        METASLAB_GANG_AVOID)))

uint64_t metaslab_aliquot = 512ULL << 10;
uint64_t metaslab_gang_bang = SPA_MAXBLOCKSIZE + 1;     /* force gang blocks */

/*
 * The in-core space map representation is more compact than its on-disk form.
 * The zfs_condense_pct determines how much more compact the in-core
 * space_map representation must be before we compact it on-disk.
 * Values should be greater than or equal to 100.
 */
int zfs_condense_pct = 200;
```



```
/*
 * This value defines the number of allowed allocation failures per vdev.
 * If a device reaches this threshold in a given txg then we consider skipping
 * allocations on that device. The value of zfs_mg_alloc_failures is computed
 * in zio_init() unless it has been overridden in /etc/system.
 */
int zfs_mg_alloc_failures = 0;
int zfs_mg_alloc_failures;

/*
 * The zfs_mg_noalloc_threshold defines which metaslab groups should
 * be eligible for allocation. The value is defined as a percentage of
 * a free space. Metaslab groups that have more free space than
 * zfs_mg_noalloc_threshold are always eligible for allocations. Once
 * a metaslab group's free space is less than or equal to the
 * zfs_mg_noalloc_threshold the allocator will avoid allocating to that
 * group unless all groups in the pool have reached zfs_mg_noalloc_threshold.
 * Once all groups in the pool reach zfs_mg_noalloc_threshold then all
 * groups are allowed to accept allocations. Gang blocks are always
 * eligible to allocate on any metaslab group. The default value of 0 means
 * no metaslab group will be excluded based on this criterion.
 */
int zfs_mg_noalloc_threshold = 0;

/*
 * Metaslab debugging: when set, keeps all space maps in core to verify frees.
 */
static int metaslab_debug = 0;

/*
 * Minimum size which forces the dynamic allocator to change
 * it's allocation strategy.  Once the space map cannot satisfy
 * an allocation of this size then it switches to using more
 * aggressive strategy (i.e search by size rather than offset).
 */
uint64_t metaslab_df_alloc_threshold = SPA_MAXBLOCKSIZE;

/*
 * The minimum free space, in percent, which must be available
 * in a space map to continue allocations in a first-fit fashion.
 * Once the space_map's free space drops below this level we dynamically
 * switch to using best-fit allocations.
 */
int metaslab_df_free_pct = 4;

/*
 * A metaslab is considered "free" if it contains a contiguous
 * segment which is greater than metaslab_min_alloc_size.
 */
uint64_t metaslab_min_alloc_size = DMU_MAX_ACCESS;

/*
 * Max number of space_maps to prefetch.
 */
int metaslab_prefetch_limit = SPA_DVAS_PER_BP;

/*
 * Percentage bonus multiplier for metaslabs that are in the bonus area.
 */
int metaslab_smo_bonus_pct = 150;

/*
 * Should we be willing to write data to degraded vdevs?
 */
boolean_t zfs_write_to_degraded = B_FALSE;
```

```
123 /*
124  * ==========================================================================
125  * Metaslab classes
126  * ==========================================================================
127  */
128 metaslab_class_t *
129 metaslab_class_create(spa_t *spa, space_map_ops_t *ops)
130 {
131         metaslab_class_t *mc;

133         mc = kmem_zalloc(sizeof (metaslab_class_t), KM_SLEEP);

135         mc->mc_spa = spa;
136         mc->mc_rotor = NULL;
137         mc->mc_ops = ops;

139         return (mc);
140 }
```

*unchanged_portion_omitted_*

```
243 /*
244  * Update the allocatable flag and the metaslab group's capacity.
245  * The allocatable flag is set to true if the capacity is below
246  * the zfs_mg_noalloc_threshold. If a metaslab group transitions
247  * from allocatable to non-allocatable or vice versa then the metaslab
248  * group's class is updated to reflect the transition.
249  */
250 static void
251 metaslab_group_alloc_update(metaslab_group_t *mg)
252 {
253         vdev_t *vd = mg->mg_vd;
254         metaslab_class_t *mc = mg->mg_class;
255         vdev_stat_t *vs = &vd->vdev_stat;
256         boolean_t was_allocatable;

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

260         mutex_enter(&mg->mg_lock);
261         was_allocatable = mg->mg_allocatable;

263         mg->mg_free_capacity = ((vs->vs_space - vs->vs_alloc) * 100) /
264             (vs->vs_space + 1);

266         mg->mg_allocatable = (mg->mg_free_capacity > zfs_mg_noalloc_threshold);

268         /*
269          * The mc_alloc_groups maintains a count of the number of
270          * groups in this metaslab class that are still above the
271          * zfs_mg_noalloc_threshold. This is used by the allocating
272          * threads to determine if they should avoid allocations to
273          * a given group. The allocator will avoid allocations to a group
274          * if that group has reached or is below the zfs_mg_noalloc_threshold
275          * and there are still other groups that are above the threshold.
276          * When a group transitions from allocatable to non-allocatable or
277          * vice versa we update the metaslab class to reflect that change.
278          * When the mc_alloc_groups value drops to 0 that means that all
279          * groups have reached the zfs_mg_noalloc_threshold making all groups
280          * eligible for allocations. This effectively means that all devices
281          * are balanced again.
282          */
283         if (was_allocatable && !mg->mg_allocatable)
284                 mc->mc_alloc_groups--;
285         else if (!was_allocatable && mg->mg_allocatable)
286                 mc->mc_alloc_groups++;
287         mutex_exit(&mg->mg_lock);
```



```
288 }

290 metaslab_group_t *
291 metaslab_group_create(metaslab_class_t *mc, vdev_t *vd)
292 {
293         metaslab_group_t *mg;

295         mg = kmem_zalloc(sizeof (metaslab_group_t), KM_SLEEP);
296         mutex_init(&mg->mg_lock, NULL, MUTEX_DEFAULT, NULL);
297         avl_create(&mg->mg_metaslab_tree, metaslab_compare,
298             sizeof (metaslab_t), offsetof(struct metaslab, ms_group_node));
299         mg->mg_vd = vd;
300         mg->mg_class = mc;
301         mg->mg_activation_count = 0;

303         return (mg);
304 }
```

*unchanged_portion_omitted_*

```
323 void
324 metaslab_group_activate(metaslab_group_t *mg)
325 {
326         metaslab_class_t *mc = mg->mg_class;
327         metaslab_group_t *mgprev, *mgnext;

329         ASSERT(spa_config_held(mc->mc_spa, SCL_ALLOC, RW_WRITER));

331         ASSERT(mc->mc_rotor != mg);
332         ASSERT(mg->mg_prev == NULL);
333         ASSERT(mg->mg_next == NULL);
334         ASSERT(mg->mg_activation_count <= 0);

336         if (++mg->mg_activation_count <= 0)
337                 return;

339         mg->mg_aliquot = metaslab_aliquot * MAX(1, mg->mg_vd->vdev_children);
340         metaslab_group_alloc_update(mg);

342         if ((mgprev = mc->mc_rotor) == NULL) {
343                 mg->mg_prev = mg;
344                 mg->mg_next = mg;
345         } else {
346                 mgnext = mgprev->mg_next;
347                 mg->mg_prev = mgprev;
348                 mg->mg_next = mgnext;
349                 mgprev->mg_next = mg;
350                 mgnext->mg_prev = mg;
351         }
352         mc->mc_rotor = mg;
353 }
```

*unchanged_portion_omitted_*

```
425 /*
426  * Determine if a given metaslab group should skip allocations. A metaslab
427  * group should avoid allocations if its used capacity has crossed the
428  * zfs_mg_noalloc_threshold and there is at least one metaslab group
429  * that can still handle allocations.
430  */
431 static boolean_t
432 metaslab_group_allocatable(metaslab_group_t *mg)
433 {
434         vdev_t *vd = mg->mg_vd;
435         spa_t *spa = vd->vdev_spa;
436         metaslab_class_t *mc = mg->mg_class;

438         /*
```

```
 439          * A metaslab group is considered allocatable if its free capacity
 440          * is greater than the set value of zfs_mg_noalloc_threshold, it's
 441          * associated with a slog, or there are no other metaslab groups
 442          * with free capacity greater than zfs_mg_noalloc_threshold.
 443          */
 444         return (mg->mg_free_capacity > zfs_mg_noalloc_threshold ||
 445             mc != spa_normal_class(spa) || mc->mc_alloc_groups == 0);
 446 }

 448 /*
 449  * ==========================================================================
 450  * Common allocator routines
 451  * ==========================================================================
 452  */
 453 static int
 454 metaslab_segsize_compare(const void *x1, const void *x2)
 455 {
 456         const space_seg_t *s1 = x1;
 457         const space_seg_t *s2 = x2;
 458         uint64_t ss_size1 = s1->ss_end - s1->ss_start;
 459         uint64_t ss_size2 = s2->ss_end - s2->ss_start;

 461         if (ss_size1 < ss_size2)
 462                 return (-1);
 463         if (ss_size1 > ss_size2)
 464                 return (1);

 466         if (s1->ss_start < s2->ss_start)
 467                 return (-1);
 468         if (s1->ss_start > s2->ss_start)
 469                 return (1);

 471         return (0);
 472 }
_____unchanged_portion_omitted_

1391 void
1392 metaslab_sync_reassess(metaslab_group_t *mg)
1393 {
1394         vdev_t *vd = mg->mg_vd;
1395         int64_t failures = mg->mg_alloc_failures;

1397         metaslab_group_alloc_update(mg);

1399         /*
1400          * Re-evaluate all metaslabs which have lower offsets than the
1401          * bonus area.
1402          */
1403         for (int m = 0; m < vd->vdev_ms_count; m++) {
1404                 metaslab_t *msp = vd->vdev_ms[m];

1406                 if (msp->ms_map->sm_start > mg->mg_bonus_area)
1407                         break;

1409                 mutex_enter(&msp->ms_lock);
1410                 metaslab_group_sort(mg, msp, metaslab_weight(msp));
1411                 mutex_exit(&msp->ms_lock);
1412         }

1414         atomic_add_64(&mg->mg_alloc_failures, -failures);

1416         /*
1417          * Prefetch the next potential metaslabs
1418          */
1419         metaslab_prefetch(mg);
1420 }
_____unchanged_portion_omitted_
```



```
1439 static uint64_t
1440 metaslab_group_alloc(metaslab_group_t *mg, uint64_t psize, uint64_t asize,
1441     uint64_t txg, uint64_t min_distance, dva_t *dva, int d, int flags)
1442 {
1443         spa_t *spa = mg->mg_vd->vdev_spa;
1444         metaslab_t *msp = NULL;
1445         uint64_t offset = -1ULL;
1446         avl_tree_t *t = &mg->mg_metaslab_tree;
1447         uint64_t activation_weight;
1448         uint64_t target_distance;
1449         int i;

1451         activation_weight = METASLAB_WEIGHT_PRIMARY;
1452         for (i = 0; i < d; i++) {
1453                 if (DVA_GET_VDEV(&dva[i]) == mg->mg_vd->vdev_id) {
1454                         activation_weight = METASLAB_WEIGHT_SECONDARY;
1455                         break;
1456                 }
1457         }

1459         for (;;) {
1460                 boolean_t was_active;

1462                 mutex_enter(&mg->mg_lock);
1463                 for (msp = avl_first(t); msp; msp = AVL_NEXT(t, msp)) {
1464                         if (msp->ms_weight < asize) {
1465                                 spa_dbgmsg(spa, "%s: failed to meet weight "
1466                                     "requirement: vdev %llu, txg %llu, mg %p, "
1467                                     "msp %p, psize %llu, asize %llu, "
1468                                     "failures %llu, weight %llu",
1469                                     spa_name(spa), mg->mg_vd->vdev_id, txg,
1470                                     mg, msp, psize, asize,
1471                                     mg->mg_alloc_failures, msp->ms_weight);
1472                                 mutex_exit(&mg->mg_lock);
1473                                 return (-1ULL);
1474                         }

1476                         /*
1477                          * If the selected metaslab is condensing, skip it.
1478                          */
1479                         if (msp->ms_map->sm_condensing)
1480                                 continue;

1482                         was_active = msp->ms_weight & METASLAB_ACTIVE_MASK;
1483                         if (activation_weight == METASLAB_WEIGHT_PRIMARY)
1484                                 break;

1486                         target_distance = min_distance +
1487                             (msp->ms_smo.smo_alloc ? 0 : min_distance >> 1);

1489                         for (i = 0; i < d; i++)
1490                                 if (metaslab_distance(msp, &dva[i]) <
1491                                     target_distance)
1492                                         break;
1493                         if (i == d)
1494                                 break;
1495                 }
1496                 mutex_exit(&mg->mg_lock);
1497                 if (msp == NULL)
1498                         return (-1ULL);

1500                 mutex_enter(&msp->ms_lock);

1502                 /*
1503                  * If we've already reached the allowable number of failed
```

```
1504                  * allocation attempts on this metaslab group then we
1505                  * consider skipping it. We skip it only if we're allowed
1506                  * to "fast" gang, the physical size is larger than
1507                  * a gang block, and we're attempting to allocate from
1508                  * the primary metaslab.
1509                  */
1510                 if (mg->mg_alloc_failures > zfs_mg_alloc_failures &&
1511                     CAN_FASTGANG(flags) && psize > SPA_GANGBLOCKSIZE &&
1512                     activation_weight == METASLAB_WEIGHT_PRIMARY) {
1513                         spa_dbgmsg(spa, "%s: skipping metaslab group: "
1514                             "vdev %llu, txg %llu, mg %p, psize %llu, "
1515                             "asize %llu, failures %llu", spa_name(spa),
1516                             mg->mg_vd->vdev_id, txg, mg, psize, asize,
1517                             mg->mg_alloc_failures);
1518                         mutex_exit(&msp->ms_lock);
1519                         return (-1ULL);
1520                 }

1430                 mutex_enter(&msp->ms_lock);

1522                 /*
1523                  * Ensure that the metaslab we have selected is still
1524                  * capable of handling our request. It's possible that
1525                  * another thread may have changed the weight while we
1526                  * were blocked on the metaslab lock.
1527                  */
1528                 if (msp->ms_weight < asize || (was_active &&
1529                     !(msp->ms_weight & METASLAB_ACTIVE_MASK) &&
1530                     activation_weight == METASLAB_WEIGHT_PRIMARY)) {
1531                         mutex_exit(&msp->ms_lock);
1532                         continue;
1533                 }

1535                 if ((msp->ms_weight & METASLAB_WEIGHT_SECONDARY) &&
1536                     activation_weight == METASLAB_WEIGHT_PRIMARY) {
1537                         metaslab_passivate(msp,
1538                             msp->ms_weight & ~METASLAB_ACTIVE_MASK);
1539                         mutex_exit(&msp->ms_lock);
1540                         continue;
1541                 }

1543                 if (metaslab_activate(msp, activation_weight) != 0) {
1544                         mutex_exit(&msp->ms_lock);
1545                         continue;
1546                 }

1548                 /*
1549                  * If this metaslab is currently condensing then pick again as
1550                  * we can't manipulate this metaslab until it's committed
1551                  * to disk.
1552                  */
1553                 if (msp->ms_map->sm_condensing) {
1554                         mutex_exit(&msp->ms_lock);
1555                         continue;
1556                 }

1558                 if ((offset = space_map_alloc(msp->ms_map, asize)) != -1ULL)
1559                         break;

1561                 atomic_inc_64(&mg->mg_alloc_failures);

1563                 metaslab_passivate(msp, space_map_maxsize(msp->ms_map));

1565                 mutex_exit(&msp->ms_lock);
1566         }
```



```
1568         if (msp->ms_allocmap[txg & TXG_MASK]->sm_space == 0)
1569                 vdev_dirty(mg->mg_vd, VDD_METASLAB, msp, txg);

1571         space_map_add(msp->ms_allocmap[txg & TXG_MASK], offset, asize);

1573         mutex_exit(&msp->ms_lock);

1575         return (offset);
1576 }

1578 /*
1579  * Allocate a block for the specified i/o.
1580  */
1581 static int
1582 metaslab_alloc_dva(spa_t *spa, metaslab_class_t *mc, uint64_t psize,
1583     dva_t *dva, int d, dva_t *hintdva, uint64_t txg, int flags)
1584 {
1585         metaslab_group_t *mg, *rotor;
1586         vdev_t *vd;
1587         int dshift = 3;
1588         int all_zero;
1589         int zio_lock = B_FALSE;
1590         boolean_t allocatable;
1591         uint64_t offset = -1ULL;
1592         uint64_t asize;
1593         uint64_t distance;

1595         ASSERT(!DVA_IS_VALID(&dva[d]));

1597         /*
1598          * For testing, make some blocks above a certain size be gang blocks.
1599          */
1600         if (psize >= metaslab_gang_bang && (ddi_get_lbolt() & 3) == 0)
1601                 return (SET_ERROR(ENOSPC));

1603         /*
1604          * Start at the rotor and loop through all mgs until we find something.
1605          * Note that there's no locking on mc_rotor or mc_aliquot because
1606          * nothing actually breaks if we miss a few updates -- we just won't
1607          * allocate quite as evenly.  It all balances out over time.
1608          *
1609          * If we are doing ditto or log blocks, try to spread them across
1610          * consecutive vdevs.  If we're forced to reuse a vdev before we've
1611          * allocated all of our ditto blocks, then try and spread them out on
1612          * that vdev as much as possible.  If it turns out to not be possible,
1613          * gradually lower our standards until anything becomes acceptable.
1614          * Also, allocating on consecutive vdevs (as opposed to random vdevs)
1615          * gives us hope of containing our fault domains to something we're
1616          * able to reason about.  Otherwise, any two top-level vdev failures
1617          * will guarantee the loss of data.  With consecutive allocation,
1618          * only two adjacent top-level vdev failures will result in data loss.
1619          *
1620          * If we are doing gang blocks (hintdva is non-NULL), try to keep
1621          * ourselves on the same vdev as our gang block header.  That
1622          * way, we can hope for locality in vdev_cache, plus it makes our
1623          * fault domains something tractable.
1624          */
1625         if (hintdva) {
1626                 vd = vdev_lookup_top(spa, DVA_GET_VDEV(&hintdva[d]));

1628                 /*
1629                  * It's possible the vdev we're using as the hint no
1630                  * longer exists (i.e. removed). Consult the rotor when
1631                  * all else fails.
1632                  */
1633                 if (vd != NULL) {
```

```
1634                        mg = vd->vdev_mg;

1636                        if (flags & METASLAB_HINTBP_AVOID &&
1637                            mg->mg_next != NULL)
1638                                mg = mg->mg_next;
1639                } else {
1640                        mg = mc->mc_rotor;
1641                }
1642        } else if (d != 0) {
1643                vd = vdev_lookup_top(spa, DVA_GET_VDEV(&dva[d - 1]));
1644                mg = vd->vdev_mg->mg_next;
1645        } else {
1646                mg = mc->mc_rotor;
1647        }

1649        /*
1650         * If the hint put us into the wrong metaslab class, or into a
1651         * metaslab group that has been passivated, just follow the rotor.
1652         */
1653        if (mg->mg_class != mc || mg->mg_activation_count <= 0)
1654                mg = mc->mc_rotor;

1656        rotor = mg;
1657 top:
1658        all_zero = B_TRUE;
1659        do {
1660                ASSERT(mg->mg_activation_count == 1);

1662                vd = mg->mg_vd;

1664                /*
1665                 * Don't allocate from faulted devices.
1666                 */
1667                if (zio_lock) {
1668                        spa_config_enter(spa, SCL_ZIO, FTAG, RW_READER);
1669                        allocatable = vdev_allocatable(vd);
1670                        spa_config_exit(spa, SCL_ZIO, FTAG);
1671                } else {
1672                        allocatable = vdev_allocatable(vd);
1673                }

1675                /*
1676                 * Determine if the selected metaslab group is eligible
1677                 * for allocations. If we're ganging or have requested
1678                 * an allocation for the smallest gang block size
1679                 * then we don't want to avoid allocating to the this
1680                 * metaslab group. If we're in this condition we should
1681                 * try to allocate from any device possible so that we
1682                 * don't inadvertently return ENOSPC and suspend the pool
1683                 * even though space is still available.
1684                 */
1685                if (allocatable && CAN_FASTGANG(flags) &&
1686                    psize > SPA_GANGBLOCKSIZE)
1687                        allocatable = metaslab_group_allocatable(mg);

1689                if (!allocatable)
1690                        goto next;

1692                /*
1693                 * Avoid writing single-copy data to a failing vdev
1694                 * unless the user instructs us that it is okay.
1695                 */
1696                if ((vd->vdev_stat.vs_write_errors > 0 ||
1697                    vd->vdev_state < VDEV_STATE_HEALTHY) &&
1698                    d == 0 && dshift == 3 &&
1699                    !(zfs_write_to_degraded && vd->vdev_state ==
```



```
1700                        VDEV_STATE_DEGRADED)) {
1701                            all_zero = B_FALSE;
1702                            goto next;
1703                }

1705                ASSERT(mg->mg_class == mc);

1707                distance = vd->vdev_asize >> dshift;
1708                if (distance <= (1ULL << vd->vdev_ms_shift))
1709                        distance = 0;
1710                else
1711                        all_zero = B_FALSE;

1713                asize = vdev_psize_to_asize(vd, psize);
1714                ASSERT(P2PHASE(asize, 1ULL << vd->vdev_ashift) == 0);

1716                offset = metaslab_group_alloc(mg, psize, asize, txg, distance,
1717                    dva, d, flags);
1718                if (offset != -1ULL) {
1719                        /*
1720                         * If we've just selected this metaslab group,
1721                         * figure out whether the corresponding vdev is
1722                         * over- or under-used relative to the pool,
1723                         * and set an allocation bias to even it out.
1724                         */
1725                        if (mc->mc_aliquot == 0) {
1726                                vdev_stat_t *vs = &vd->vdev_stat;
1727                                int64_t vu, cu;

1729                                vu = (vs->vs_alloc * 100) / (vs->vs_space + 1);
1730                                cu = (mc->mc_alloc * 100) / (mc->mc_space + 1);

1732                                /*
1733                                 * Calculate how much more or less we should
1734                                 * try to allocate from this device during
1735                                 * this iteration around the rotor.
1736                                 * For example, if a device is 80% full
1737                                 * and the pool is 20% full then we should
1738                                 * reduce allocations by 60% on this device.
1739                                 *
1740                                 * mg_bias = (20 - 80) * 512K / 100 = -307K
1741                                 *
1742                                 * This reduces allocations by 307K for this
1743                                 * iteration.
1744                                 */
1745                                mg->mg_bias = ((cu - vu) *
1746                                    (int64_t)mg->mg_aliquot) / 100;
1747                        }

1749                        if (atomic_add_64_nv(&mc->mc_aliquot, asize) >=
1750                            mg->mg_aliquot + mg->mg_bias) {
1751                                mc->mc_rotor = mg->mg_next;
1752                                mc->mc_aliquot = 0;
1753                        }

1755                        DVA_SET_VDEV(&dva[d], vd->vdev_id);
1756                        DVA_SET_OFFSET(&dva[d], offset);
1757                        DVA_SET_GANG(&dva[d], !!(flags & METASLAB_GANG_HEADER));
1758                        DVA_SET_ASIZE(&dva[d], asize);

1760                        return (0);
1761                }
1762 next:
1763                mc->mc_rotor = mg->mg_next;
1764                mc->mc_aliquot = 0;
1765        } while ((mg = mg->mg_next) != rotor);
```

```
1767         if (!all_zero) {
1768                 dshift++;
1769                 ASSERT(dshift < 64);
1770                 goto top;
1771         }

1773         if (!allocatable && !zio_lock) {
1774                 dshift = 3;
1775                 zio_lock = B_TRUE;
1776                 goto top;
1777         }

1779         bzero(&dva[d], sizeof (dva_t));

1781         return (SET_ERROR(ENOSPC));
1782 }
```

_____***unchanged_portion_omitted_***

****
4134 Mon Aug 26 18:30:11 2013
new/usr/src/uts/common/fs/zfs/sys/metaslab_impl.h
3954 metaslabs continue to load even after hitting zfs_mg_alloc_failure limit
4080 zpool clear fails to clear pool
4081 need zfs_mg_noalloc_threshold
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
****

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _SYS_METASLAB_IMPL_H
#define _SYS_METASLAB_IMPL_H

#include <sys/metaslab.h>
#include <sys/space_map.h>
#include <sys/vdev.h>
#include <sys/txg.h>
#include <sys/avl.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct metaslab_class {
        spa_t                   *mc_spa;
        metaslab_group_t        *mc_rotor;
        space_map_ops_t         *mc_ops;
        uint64_t                mc_aliquot;
        uint64_t                mc_alloc_groups; /* # of allocatable groups */
        uint64_t                mc_alloc;       /* total allocated space */
        uint64_t                mc_deferred;    /* total deferred frees */
        uint64_t                mc_space;       /* total space (alloc + free) */
        uint64_t                mc_dspace;      /* total deflated space */
};

struct metaslab_group {
        kmutex_t                mg_lock;
```



```
        avl_tree_t              mg_metaslab_tree;
        uint64_t                mg_aliquot;
        uint64_t                mg_bonus_area;
        uint64_t                mg_alloc_failures;
        boolean_t               mg_allocatable;         /* can we allocate? */
        uint64_t                mg_free_capacity;       /* percentage free */
        int64_t                 mg_bias;
        int64_t                 mg_activation_count;
        metaslab_class_t        *mg_class;
        vdev_t                  *mg_vd;
        metaslab_group_t        *mg_prev;
        metaslab_group_t        *mg_next;
};
```

*unchanged_portion_omitted_*

```
**********************************************************
  144473 Mon Aug 26 18:30:13 2013
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
3954 metaslabs continue to load even after hitting zfs_mg_alloc_failure limit
4080 zpool clear fails to clear pool
4081 need zfs_mg_noalloc_threshold
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_

5287 static void
5288 zfs_ioctl_init(void)
5289 {
5290         zfs_ioctl_register("snapshot", ZFS_IOC_SNAPSHOT,
5291             zfs_ioc_snapshot, zfs_secpolicy_snapshot, POOL_NAME,
5292             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5294         zfs_ioctl_register("log_history", ZFS_IOC_LOG_HISTORY,
5295             zfs_ioc_log_history, zfs_secpolicy_log_history, NO_NAME,
5296             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_FALSE, B_FALSE);

5298         zfs_ioctl_register("space_snaps", ZFS_IOC_SPACE_SNAPS,
5299             zfs_ioc_space_snaps, zfs_secpolicy_read, DATASET_NAME,
5300             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5302         zfs_ioctl_register("send", ZFS_IOC_SEND_NEW,
5303             zfs_ioc_send_new, zfs_secpolicy_send_new, DATASET_NAME,
5304             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5306         zfs_ioctl_register("send_space", ZFS_IOC_SEND_SPACE,
5307             zfs_ioc_send_space, zfs_secpolicy_read, DATASET_NAME,
5308             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5310         zfs_ioctl_register("create", ZFS_IOC_CREATE,
5311             zfs_ioc_create, zfs_secpolicy_create_clone, DATASET_NAME,
5312             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5314         zfs_ioctl_register("clone", ZFS_IOC_CLONE,
5315             zfs_ioc_clone, zfs_secpolicy_create_clone, DATASET_NAME,
5316             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5318         zfs_ioctl_register("destroy_snaps", ZFS_IOC_DESTROY_SNAPS,
5319             zfs_ioc_destroy_snaps, zfs_secpolicy_destroy_snaps, POOL_NAME,
5320             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5322         zfs_ioctl_register("hold", ZFS_IOC_HOLD,
5323             zfs_ioc_hold, zfs_secpolicy_hold, POOL_NAME,
5324             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);
5325         zfs_ioctl_register("release", ZFS_IOC_RELEASE,
5326             zfs_ioc_release, zfs_secpolicy_release, POOL_NAME,
5327             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5329         zfs_ioctl_register("get_holds", ZFS_IOC_GET_HOLDS,
5330             zfs_ioc_get_holds, zfs_secpolicy_read, DATASET_NAME,
5331             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5333         zfs_ioctl_register("rollback", ZFS_IOC_ROLLBACK,
5334             zfs_ioc_rollback, zfs_secpolicy_rollback, DATASET_NAME,
5335             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_FALSE, B_TRUE);

5337         /* IOCTLS that use the legacy function signature */

5339         zfs_ioctl_register_legacy(ZFS_IOC_POOL_FREEZE, zfs_ioc_pool_freeze,
5340             zfs_secpolicy_config, NO_NAME, B_FALSE, POOL_CHECK_READONLY);
```



```
5342         zfs_ioctl_register_pool(ZFS_IOC_POOL_CREATE, zfs_ioc_pool_create,
5343             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
5344         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SCAN,
5345             zfs_ioc_pool_scan);
5346         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_UPGRADE,
5347             zfs_ioc_pool_upgrade);
5348         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ADD,
5349             zfs_ioc_vdev_add);
5350         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_REMOVE,
5351             zfs_ioc_vdev_remove);
5352         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SET_STATE,
5353             zfs_ioc_vdev_set_state);
5354         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ATTACH,
5355             zfs_ioc_vdev_attach);
5356         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_DETACH,
5357             zfs_ioc_vdev_detach);
5358         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETPATH,
5359             zfs_ioc_vdev_setpath);
5360         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETFRU,
5361             zfs_ioc_vdev_setfru);
5362         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SET_PROPS,
5363             zfs_ioc_pool_set_props);
5364         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SPLIT,
5365             zfs_ioc_vdev_split);
5366         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_REGUID,
5367             zfs_ioc_pool_reguid);

5369         zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_CONFIGS,
5370             zfs_ioc_pool_configs, zfs_secpolicy_none);
5371         zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_TRYIMPORT,
5372             zfs_ioc_pool_tryimport, zfs_secpolicy_config);
5373         zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_FAULT,
5374             zfs_ioc_inject_fault, zfs_secpolicy_inject);
5375         zfs_ioctl_register_pool_meta(ZFS_IOC_CLEAR_FAULT,
5376             zfs_ioc_clear_fault, zfs_secpolicy_inject);
5377         zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_LIST_NEXT,
5378             zfs_ioc_inject_list_next, zfs_secpolicy_inject);

5380         /*
5381          * pool destroy, and export don't log the history as part of
5382          * zfsdev_ioctl, but rather zfs_ioc_pool_export
5383          * does the logging of those commands.
5384          */
5385         zfs_ioctl_register_pool(ZFS_IOC_POOL_DESTROY, zfs_ioc_pool_destroy,
5386             zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);
5387         zfs_ioctl_register_pool(ZFS_IOC_POOL_EXPORT, zfs_ioc_pool_export,
5388             zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);

5390         zfs_ioctl_register_pool(ZFS_IOC_POOL_STATS, zfs_ioc_pool_stats,
5391             zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);
5392         zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_PROPS, zfs_ioc_pool_get_props,
5393             zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);

5395         zfs_ioctl_register_pool(ZFS_IOC_ERROR_LOG, zfs_ioc_error_log,
5396             zfs_secpolicy_inject, B_FALSE, POOL_CHECK_SUSPENDED);
5397         zfs_ioctl_register_pool(ZFS_IOC_DSOBJ_TO_DSNAME,
5398             zfs_ioc_dsobj_to_dsname,
5399             zfs_secpolicy_diff, B_FALSE, POOL_CHECK_SUSPENDED);
5400         zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_HISTORY,
5401             zfs_ioc_pool_get_history,
5402             zfs_secpolicy_config, B_FALSE, POOL_CHECK_SUSPENDED);

5404         zfs_ioctl_register_pool(ZFS_IOC_POOL_IMPORT, zfs_ioc_pool_import,
5405             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);

5407         zfs_ioctl_register_pool(ZFS_IOC_CLEAR, zfs_ioc_clear,
```

```
5408             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
5408             zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);
5409         zfs_ioctl_register_pool(ZFS_IOC_POOL_REOPEN, zfs_ioc_pool_reopen,
5410             zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);

5412         zfs_ioctl_register_dataset_read(ZFS_IOC_SPACE_WRITTEN,
5413             zfs_ioc_space_written);
5414         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_RECVD_PROPS,
5415             zfs_ioc_objset_recvd_props);
5416         zfs_ioctl_register_dataset_read(ZFS_IOC_NEXT_OBJ,
5417             zfs_ioc_next_obj);
5418         zfs_ioctl_register_dataset_read(ZFS_IOC_GET_FSACL,
5419             zfs_ioc_get_fsacl);
5420         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_STATS,
5421             zfs_ioc_objset_stats);
5422         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_ZPLPROPS,
5423             zfs_ioc_objset_zplprops);
5424         zfs_ioctl_register_dataset_read(ZFS_IOC_DATASET_LIST_NEXT,
5425             zfs_ioc_dataset_list_next);
5426         zfs_ioctl_register_dataset_read(ZFS_IOC_SNAPSHOT_LIST_NEXT,
5427             zfs_ioc_snapshot_list_next);
5428         zfs_ioctl_register_dataset_read(ZFS_IOC_SEND_PROGRESS,
5429             zfs_ioc_send_progress);

5431         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_DIFF,
5432             zfs_ioc_diff, zfs_secpolicy_diff);
5433         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_STATS,
5434             zfs_ioc_obj_to_stats, zfs_secpolicy_diff);
5435         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_PATH,
5436             zfs_ioc_obj_to_path, zfs_secpolicy_diff);
5437         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_ONE,
5438             zfs_ioc_userspace_one, zfs_secpolicy_userspace_one);
5439         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_MANY,
5440             zfs_ioc_userspace_many, zfs_secpolicy_userspace_many);
5441         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_SEND,
5442             zfs_ioc_send, zfs_secpolicy_send);

5444         zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_PROP, zfs_ioc_set_prop,
5445             zfs_secpolicy_none);
5446         zfs_ioctl_register_dataset_modify(ZFS_IOC_DESTROY, zfs_ioc_destroy,
5447             zfs_secpolicy_destroy);
5448         zfs_ioctl_register_dataset_modify(ZFS_IOC_RENAME, zfs_ioc_rename,
5449             zfs_secpolicy_rename);
5450         zfs_ioctl_register_dataset_modify(ZFS_IOC_RECV, zfs_ioc_recv,
5451             zfs_secpolicy_recv);
5452         zfs_ioctl_register_dataset_modify(ZFS_IOC_PROMOTE, zfs_ioc_promote,
5453             zfs_secpolicy_promote);
5454         zfs_ioctl_register_dataset_modify(ZFS_IOC_INHERIT_PROP,
5455             zfs_ioc_inherit_prop, zfs_secpolicy_inherit_prop);
5456         zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_FSACL, zfs_ioc_set_fsacl,
5457             zfs_secpolicy_set_fsacl);

5459         zfs_ioctl_register_dataset_nolog(ZFS_IOC_SHARE, zfs_ioc_share,
5460             zfs_secpolicy_share, POOL_CHECK_NONE);
5461         zfs_ioctl_register_dataset_nolog(ZFS_IOC_SMB_ACL, zfs_ioc_smb_acl,
5462             zfs_secpolicy_smb_acl, POOL_CHECK_NONE);
5463         zfs_ioctl_register_dataset_nolog(ZFS_IOC_USERSPACE_UPGRADE,
5464             zfs_ioc_userspace_upgrade, zfs_secpolicy_userspace_upgrade,
5465             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5466         zfs_ioctl_register_dataset_nolog(ZFS_IOC_TMP_SNAPSHOT,
5467             zfs_ioc_tmp_snapshot, zfs_secpolicy_tmp_snapshot,
5468             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5469 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   90300 Mon Aug 26 18:30:14 2013
new/usr/src/uts/common/fs/zfs/zio.c
3954 metaslabs continue to load even after hitting zfs_mg_alloc_failure limit
4080 zpool clear fails to clear pool
4081 need zfs_mg_noalloc_threshold
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_

  50 /*
  51  * ==========================================================================
  52  * I/O kmem caches
  53  * ==========================================================================
  54  */
  55 kmem_cache_t *zio_cache;
  56 kmem_cache_t *zio_link_cache;
  57 kmem_cache_t *zio_buf_cache[SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT];
  58 kmem_cache_t *zio_data_buf_cache[SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT];

  60 #ifdef _KERNEL
  61 extern vmem_t *zio_alloc_arena;
  62 #endif
  63 extern int zfs_mg_alloc_failures;

  65 /*
  66  * The following actions directly effect the spa's sync-to-convergence logic.
  67  * The values below define the sync pass when we start performing the action.
  68  * Care should be taken when changing these values as they directly impact
  69  * spa_sync() performance. Tuning these values may introduce subtle performance
  70  * pathologies and should only be done in the context of performance analysis.
  71  * These tunables will eventually be removed and replaced with #defines once
  72  * enough analysis has been done to determine optimal values.
  73  *
  74  * The 'zfs_sync_pass_deferred_free' pass must be greater than 1 to ensure that
  75  * regular blocks are not deferred.
  76  */
  77 int zfs_sync_pass_deferred_free = 2; /* defer frees starting in this pass */
  78 int zfs_sync_pass_dont_compress = 5; /* don't compress starting in this pass */
  79 int zfs_sync_pass_rewrite = 2; /* rewrite new bps starting in this pass */

  81 /*
  82  * An allocating zio is one that either currently has the DVA allocate
  83  * stage set or will have it later in its lifetime.
  84  */
  85 #define IO_IS_ALLOCATING(zio) ((zio)->io_orig_pipeline & ZIO_STAGE_DVA_ALLOCATE)

  87 boolean_t       zio_requeue_io_start_cut_in_line = B_TRUE;

  89 #ifdef ZFS_DEBUG
  90 int zio_buf_debug_limit = 16384;
  91 #else
  92 int zio_buf_debug_limit = 0;
  93 #endif

  95 void
  96 zio_init(void)
  97 {
  98         size_t c;
  99         vmem_t *data_alloc_arena = NULL;

 101 #ifdef _KERNEL
 102         data_alloc_arena = zio_alloc_arena;
 103 #endif
 104         zio_cache = kmem_cache_create("zio_cache",
```



```
 105             sizeof (zio_t), 0, NULL, NULL, NULL, NULL, NULL, 0);
 106         zio_link_cache = kmem_cache_create("zio_link_cache",
 107             sizeof (zio_link_t), 0, NULL, NULL, NULL, NULL, NULL, 0);

 109         /*
 110          * For small buffers, we want a cache for each multiple of
 111          * SPA_MINBLOCKSIZE.  For medium-size buffers, we want a cache
 112          * for each quarter-power of 2.  For large buffers, we want
 113          * a cache for each multiple of PAGESIZE.
 114          */
 115         for (c = 0; c < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT; c++) {
 116                 size_t size = (c + 1) << SPA_MINBLOCKSHIFT;
 117                 size_t p2 = size;
 118                 size_t align = 0;
 119                 size_t cflags = (size > zio_buf_debug_limit) ? KMC_NODEBUG : 0;

 121                 while (p2 & (p2 - 1))
 122                         p2 &= p2 - 1;

 124 #ifndef _KERNEL
 125                 /*
 126                  * If we are using watchpoints, put each buffer on its own page,
 127                  * to eliminate the performance overhead of trapping to the
 128                  * kernel when modifying a non-watched buffer that shares the
 129                  * page with a watched buffer.
 130                  */
 131                 if (arc_watch && !IS_P2ALIGNED(size, PAGESIZE))
 132                         continue;
 133 #endif
 134                 if (size <= 4 * SPA_MINBLOCKSIZE) {
 135                         align = SPA_MINBLOCKSIZE;
 136                 } else if (IS_P2ALIGNED(size, PAGESIZE)) {
 137                         align = PAGESIZE;
 138                 } else if (IS_P2ALIGNED(size, p2 >> 2)) {
 139                         align = p2 >> 2;
 140                 }

 142                 if (align != 0) {
 143                         char name[36];
 144                         (void) sprintf(name, "zio_buf_%lu", (ulong_t)size);
 145                         zio_buf_cache[c] = kmem_cache_create(name, size,
 146                             align, NULL, NULL, NULL, NULL, NULL, cflags);

 148                         /*
 149                          * Since zio_data bufs do not appear in crash dumps, we
 150                          * pass KMC_NOTOUCH so that no allocator metadata is
 151                          * stored with the buffers.
 152                          */
 153                         (void) sprintf(name, "zio_data_buf_%lu", (ulong_t)size);
 154                         zio_data_buf_cache[c] = kmem_cache_create(name, size,
 155                             align, NULL, NULL, NULL, NULL, data_alloc_arena,
 156                             cflags | KMC_NOTOUCH);
 157                 }
 158         }

 160         while (--c != 0) {
 161                 ASSERT(zio_buf_cache[c] != NULL);
 162                 if (zio_buf_cache[c - 1] == NULL)
 163                         zio_buf_cache[c - 1] = zio_buf_cache[c];

 165                 ASSERT(zio_data_buf_cache[c] != NULL);
 166                 if (zio_data_buf_cache[c - 1] == NULL)
 167                         zio_data_buf_cache[c - 1] = zio_data_buf_cache[c];
 168         }

 170         /*
```

```
 171          * The zio write taskqs have 1 thread per cpu, allow 1/2 of the taskqs
 172          * to fail 3 times per txg or 8 failures, whichever is greater.
 173          */
 174         if (zfs_mg_alloc_failures == 0)
 175                 zfs_mg_alloc_failures = MAX((3 * max_ncpus / 2), 8);

 177         zio_inject_init();
 178 }
______unchanged_portion_omitted_

2344 /*
2345  * Try to allocate an intent log block.  Return 0 on success, errno on failure.
2346  */
2347 int
2348 zio_alloc_zil(spa_t *spa, uint64_t txg, blkptr_t *new_bp, blkptr_t *old_bp,
2349     uint64_t size, boolean_t use_slog)
2350 {
2351         int error = 1;

2353         ASSERT(txg > spa_syncing_txg(spa));

2355         /*
2356          * ZIL blocks are always contiguous (i.e. not gang blocks) so we
2357          * set the METASLAB_GANG_AVOID flag so that they don't "fast gang"
2358          * when allocating them.
2359          */
2360         if (use_slog) {
2361                 error = metaslab_alloc(spa, spa_log_class(spa), size,
2362                     new_bp, 1, txg, old_bp,
2363                     METASLAB_HINTBP_AVOID | METASLAB_GANG_AVOID);
2364         }

2366         if (error) {
2367                 error = metaslab_alloc(spa, spa_normal_class(spa), size,
2368                     new_bp, 1, txg, old_bp,
2369                     METASLAB_HINTBP_AVOID);
2368                     METASLAB_HINTBP_AVOID | METASLAB_GANG_AVOID);
2370         }

2372         if (error == 0) {
2373                 BP_SET_LSIZE(new_bp, size);
2374                 BP_SET_PSIZE(new_bp, size);
2375                 BP_SET_COMPRESS(new_bp, ZIO_COMPRESS_OFF);
2376                 BP_SET_CHECKSUM(new_bp,
2377                     spa_version(spa) >= SPA_VERSION_SLIM_ZIL
2378                     ? ZIO_CHECKSUM_ZILOG2 : ZIO_CHECKSUM_ZILOG);
2379                 BP_SET_TYPE(new_bp, DMU_OT_INTENT_LOG);
2380                 BP_SET_LEVEL(new_bp, 0);
2381                 BP_SET_DEDUP(new_bp, 0);
2382                 BP_SET_BYTEORDER(new_bp, ZFS_HOST_BYTEORDER);
2383         }

2385         return (error);
2386 }
______unchanged_portion_omitted_
```