```
**********************************************************
  194390 Fri Sep 11 10:32:14 2015
new/usr/src/uts/common/fs/zfs/arc.c
6220 memleak in l2arc on debug build
**********************************************************
______unchanged_portion_omitted_

1246 /*
1247  * Transition between the two allocation states for the arc_buf_hdr struct.
1248  * The arc_buf_hdr struct can be allocated with (hdr_full_cache) or without
1249  * (hdr_l2only_cache) the fields necessary for the L1 cache - the smaller
1250  * version is used when a cache buffer is only in the L2ARC in order to reduce
1251  * memory usage.
1252  */
1253 static arc_buf_hdr_t *
1254 arc_hdr_realloc(arc_buf_hdr_t *hdr, kmem_cache_t *old, kmem_cache_t *new)
1255 {
1256         ASSERT(HDR_HAS_L2HDR(hdr));

1258         arc_buf_hdr_t *nhdr;
1259         l2arc_dev_t *dev = hdr->b_l2hdr.b_dev;

1261         ASSERT((old == hdr_full_cache && new == hdr_l2only_cache) ||
1262             (old == hdr_l2only_cache && new == hdr_full_cache));

1264         nhdr = kmem_cache_alloc(new, KM_PUSHPAGE);

1266         ASSERT(MUTEX_HELD(HDR_LOCK(hdr)));
1267         buf_hash_remove(hdr);

1269         bcopy(hdr, nhdr, HDR_L2ONLY_SIZE);

1271         if (new == hdr_full_cache) {
1272                 nhdr->b_flags |= ARC_FLAG_HAS_L1HDR;
1273                 /*
1274                  * arc_access and arc_change_state need to be aware that a
1275                  * header has just come out of L2ARC, so we set its state to
1276                  * l2c_only even though it's about to change.
1277                  */
1278                 nhdr->b_l1hdr.b_state = arc_l2c_only;

1280                 /* Verify previous threads set to NULL before freeing */
1281                 ASSERT3P(nhdr->b_l1hdr.b_tmp_cdata, ==, NULL);
1282         } else {
1283                 ASSERT(hdr->b_l1hdr.b_buf == NULL);
1284                 ASSERT0(hdr->b_l1hdr.b_datacnt);

1286                 /*
1287                  * If we've reached here, We must have been called from
1288                  * arc_evict_hdr(), as such we should have already been
1289                  * removed from any ghost list we were previously on
1290                  * (which protects us from racing with arc_evict_state),
1291                  * thus no locking is needed during this check.
1292                  */
1293                 ASSERT(!multilist_link_active(&hdr->b_l1hdr.b_arc_node));

1295                 /*
1296                  * A buffer must not be moved into the arc_l2c_only
1297                  * state if it's not finished being written out to the
1298                  * l2arc device. Otherwise, the b_l1hdr.b_tmp_cdata field
1299                  * might try to be accessed, even though it was removed.
1300                  */
1301                 VERIFY(!HDR_L2_WRITING(hdr));
1302                 VERIFY3P(hdr->b_l1hdr.b_tmp_cdata, ==, NULL);

1304 #ifdef ZFS_DEBUG
```



```
1305                 if (hdr->b_l1hdr.b_thawed != NULL) {
1306                         kmem_free(hdr->b_l1hdr.b_thawed, 1);
1307                         hdr->b_l1hdr.b_thawed = NULL;
1308                 }
1309 #endif

1311 #endif /* ! codereview */
1312                 nhdr->b_flags &= ~ARC_FLAG_HAS_L1HDR;
1313         }
1314         /*
1315          * The header has been reallocated so we need to re-insert it into any
1316          * lists it was on.
1317          */
1318         (void) buf_hash_insert(nhdr, NULL);

1320         ASSERT(list_link_active(&hdr->b_l2hdr.b_l2node));

1322         mutex_enter(&dev->l2ad_mtx);

1324         /*
1325          * We must place the realloc'ed header back into the list at
1326          * the same spot. Otherwise, if it's placed earlier in the list,
1327          * l2arc_write_buffers() could find it during the function's
1328          * write phase, and try to write it out to the l2arc.
1329          */
1330         list_insert_after(&dev->l2ad_buflist, hdr, nhdr);
1331         list_remove(&dev->l2ad_buflist, hdr);

1333         mutex_exit(&dev->l2ad_mtx);

1335         /*
1336          * Since we're using the pointer address as the tag when
1337          * incrementing and decrementing the l2ad_alloc refcount, we
1338          * must remove the old pointer (that we're about to destroy) and
1339          * add the new pointer to the refcount. Otherwise we'd remove
1340          * the wrong pointer address when calling arc_hdr_destroy() later.
1341          */

1343         (void) refcount_remove_many(&dev->l2ad_alloc,
1344             hdr->b_l2hdr.b_asize, hdr);

1346         (void) refcount_add_many(&dev->l2ad_alloc,
1347             nhdr->b_l2hdr.b_asize, nhdr);

1349         buf_discard_identity(hdr);
1350         hdr->b_freeze_cksum = NULL;
1351         kmem_cache_free(old, hdr);

1353         return (nhdr);
1354 }


1357 #define ARC_MINTIME     (hz>>4) /* 62 ms */

1359 static void
1360 arc_cksum_verify(arc_buf_t *buf)
1361 {
1362         zio_cksum_t zc;

1364         if (!(zfs_flags & ZFS_DEBUG_MODIFY))
1365                 return;

1367         mutex_enter(&buf->b_hdr->b_l1hdr.b_freeze_lock);
1368         if (buf->b_hdr->b_freeze_cksum == NULL || HDR_IO_ERROR(buf->b_hdr)) {
1369                 mutex_exit(&buf->b_hdr->b_l1hdr.b_freeze_lock);
1370                 return;
```

```
1371         }
1372         fletcher_2_native(buf->b_data, buf->b_hdr->b_size, &zc);
1373         if (!ZIO_CHECKSUM_EQUAL(*buf->b_hdr->b_freeze_cksum, zc))
1374                 panic("buffer modified while frozen!");
1375         mutex_exit(&buf->b_hdr->b_l1hdr.b_freeze_lock);
1376 }

1378 static int
1379 arc_cksum_equal(arc_buf_t *buf)
1380 {
1381         zio_cksum_t zc;
1382         int equal;

1384         mutex_enter(&buf->b_hdr->b_l1hdr.b_freeze_lock);
1385         fletcher_2_native(buf->b_data, buf->b_hdr->b_size, &zc);
1386         equal = ZIO_CHECKSUM_EQUAL(*buf->b_hdr->b_freeze_cksum, zc);
1387         mutex_exit(&buf->b_hdr->b_l1hdr.b_freeze_lock);

1389         return (equal);
1390 }

1392 static void
1393 arc_cksum_compute(arc_buf_t *buf, boolean_t force)
1394 {
1395         if (!force && !(zfs_flags & ZFS_DEBUG_MODIFY))
1396                 return;

1398         mutex_enter(&buf->b_hdr->b_l1hdr.b_freeze_lock);
1399         if (buf->b_hdr->b_freeze_cksum != NULL) {
1400                 mutex_exit(&buf->b_hdr->b_l1hdr.b_freeze_lock);
1401                 return;
1402         }
1403         buf->b_hdr->b_freeze_cksum = kmem_alloc(sizeof (zio_cksum_t), KM_SLEEP);
1404         fletcher_2_native(buf->b_data, buf->b_hdr->b_size,
1405             buf->b_hdr->b_freeze_cksum);
1406         mutex_exit(&buf->b_hdr->b_l1hdr.b_freeze_lock);
1407         arc_buf_watch(buf);
1408 }

1410 #ifndef _KERNEL
1411 typedef struct procctl {
1412         long cmd;
1413         prwatch_t prwatch;
1414 } procctl_t;
1415 #endif

1417 /* ARGSUSED */
1418 static void
1419 arc_buf_unwatch(arc_buf_t *buf)
1420 {
1421 #ifndef _KERNEL
1422         if (arc_watch) {
1423                 int result;
1424                 procctl_t ctl;
1425                 ctl.cmd = PCWATCH;
1426                 ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
1427                 ctl.prwatch.pr_size = 0;
1428                 ctl.prwatch.pr_wflags = 0;
1429                 result = write(arc_procfd, &ctl, sizeof (ctl));
1430                 ASSERT3U(result, ==, sizeof (ctl));
1431         }
1432 #endif
1433 }

1435 /* ARGSUSED */
1436 static void
```



```
1437 arc_buf_watch(arc_buf_t *buf)
1438 {
1439 #ifndef _KERNEL
1440         if (arc_watch) {
1441                 int result;
1442                 procctl_t ctl;
1443                 ctl.cmd = PCWATCH;
1444                 ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
1445                 ctl.prwatch.pr_size = buf->b_hdr->b_size;
1446                 ctl.prwatch.pr_wflags = WA_WRITE;
1447                 result = write(arc_procfd, &ctl, sizeof (ctl));
1448                 ASSERT3U(result, ==, sizeof (ctl));
1449         }
1450 #endif
1451 }

1453 static arc_buf_contents_t
1454 arc_buf_type(arc_buf_hdr_t *hdr)
1455 {
1456         if (HDR_ISTYPE_METADATA(hdr)) {
1457                 return (ARC_BUFC_METADATA);
1458         } else {
1459                 return (ARC_BUFC_DATA);
1460         }
1461 }

1463 static uint32_t
1464 arc_bufc_to_flags(arc_buf_contents_t type)
1465 {
1466         switch (type) {
1467         case ARC_BUFC_DATA:
1468                 /* metadata field is 0 if buffer contains normal data */
1469                 return (0);
1470         case ARC_BUFC_METADATA:
1471                 return (ARC_FLAG_BUFC_METADATA);
1472         default:
1473                 break;
1474         }
1475         panic("undefined ARC buffer type!");
1476         return ((uint32_t)-1);
1477 }

1479 void
1480 arc_buf_thaw(arc_buf_t *buf)
1481 {
1482         if (zfs_flags & ZFS_DEBUG_MODIFY) {
1483                 if (buf->b_hdr->b_l1hdr.b_state != arc_anon)
1484                         panic("modifying non-anon buffer!");
1485                 if (HDR_IO_IN_PROGRESS(buf->b_hdr))
1486                         panic("modifying buffer while i/o in progress!");
1487                 arc_cksum_verify(buf);
1488         }

1490         mutex_enter(&buf->b_hdr->b_l1hdr.b_freeze_lock);
1491         if (buf->b_hdr->b_freeze_cksum != NULL) {
1492                 kmem_free(buf->b_hdr->b_freeze_cksum, sizeof (zio_cksum_t));
1493                 buf->b_hdr->b_freeze_cksum = NULL;
1494         }

1496 #ifdef ZFS_DEBUG
1497         if (zfs_flags & ZFS_DEBUG_MODIFY) {
1498                 if (buf->b_hdr->b_l1hdr.b_thawed != NULL)
1499                         kmem_free(buf->b_hdr->b_l1hdr.b_thawed, 1);
1500                 buf->b_hdr->b_l1hdr.b_thawed = kmem_alloc(1, KM_SLEEP);
1501         }
1502 #endif
```

```
1504         mutex_exit(&buf->b_hdr->b_l1hdr.b_freeze_lock);

1506         arc_buf_unwatch(buf);
1507 }

1509 void
1510 arc_buf_freeze(arc_buf_t *buf)
1511 {
1512         kmutex_t *hash_lock;

1514         if (!(zfs_flags & ZFS_DEBUG_MODIFY))
1515                 return;

1517         hash_lock = HDR_LOCK(buf->b_hdr);
1518         mutex_enter(hash_lock);

1520         ASSERT(buf->b_hdr->b_freeze_cksum != NULL ||
1521             buf->b_hdr->b_l1hdr.b_state == arc_anon);
1522         arc_cksum_compute(buf, B_FALSE);
1523         mutex_exit(hash_lock);

1525 }

1527 static void
1528 add_reference(arc_buf_hdr_t *hdr, kmutex_t *hash_lock, void *tag)
1529 {
1530         ASSERT(HDR_HAS_L1HDR(hdr));
1531         ASSERT(MUTEX_HELD(hash_lock));
1532         arc_state_t *state = hdr->b_l1hdr.b_state;

1534         if ((refcount_add(&hdr->b_l1hdr.b_refcnt, tag) == 1) &&
1535             (state != arc_anon)) {
1536                 /* We don't use the L2-only state list. */
1537                 if (state != arc_l2c_only) {
1538                         arc_buf_contents_t type = arc_buf_type(hdr);
1539                         uint64_t delta = hdr->b_size * hdr->b_l1hdr.b_datacnt;
1540                         multilist_t *list = &state->arcs_list[type];
1541                         uint64_t *size = &state->arcs_lsize[type];

1543                         multilist_remove(list, hdr);

1545                         if (GHOST_STATE(state)) {
1546                                 ASSERT0(hdr->b_l1hdr.b_datacnt);
1547                                 ASSERT3P(hdr->b_l1hdr.b_buf, ==, NULL);
1548                                 delta = hdr->b_size;
1549                         }
1550                         ASSERT(delta > 0);
1551                         ASSERT3U(*size, >=, delta);
1552                         atomic_add_64(size, -delta);
1553                 }
1554                 /* remove the prefetch flag if we get a reference */
1555                 hdr->b_flags &= ~ARC_FLAG_PREFETCH;
1556         }
1557 }

1559 static int
1560 remove_reference(arc_buf_hdr_t *hdr, kmutex_t *hash_lock, void *tag)
1561 {
1562         int cnt;
1563         arc_state_t *state = hdr->b_l1hdr.b_state;

1565         ASSERT(HDR_HAS_L1HDR(hdr));
1566         ASSERT(state == arc_anon || MUTEX_HELD(hash_lock));
1567         ASSERT(!GHOST_STATE(state));
```



```
1569         /*
1570          * arc_l2c_only counts as a ghost state so we don't need to explicitly
1571          * check to prevent usage of the arc_l2c_only list.
1572          */
1573         if (((cnt = refcount_remove(&hdr->b_l1hdr.b_refcnt, tag)) == 0) &&
1574             (state != arc_anon)) {
1575                 arc_buf_contents_t type = arc_buf_type(hdr);
1576                 multilist_t *list = &state->arcs_list[type];
1577                 uint64_t *size = &state->arcs_lsize[type];

1579                 multilist_insert(list, hdr);

1581                 ASSERT(hdr->b_l1hdr.b_datacnt > 0);
1582                 atomic_add_64(size, hdr->b_size *
1583                     hdr->b_l1hdr.b_datacnt);
1584         }
1585         return (cnt);
1586 }

1588 /*
1589  * Move the supplied buffer to the indicated state. The hash lock
1590  * for the buffer must be held by the caller.
1591  */
1592 static void
1593 arc_change_state(arc_state_t *new_state, arc_buf_hdr_t *hdr,
1594     kmutex_t *hash_lock)
1595 {
1596         arc_state_t *old_state;
1597         int64_t refcnt;
1598         uint32_t datacnt;
1599         uint64_t from_delta, to_delta;
1600         arc_buf_contents_t buftype = arc_buf_type(hdr);

1602         /*
1603          * We almost always have an L1 hdr here, since we call arc_hdr_realloc()
1604          * in arc_read() when bringing a buffer out of the L2ARC.  However, the
1605          * L1 hdr doesn't always exist when we change state to arc_anon before
1606          * destroying a header, in which case reallocating to add the L1 hdr is
1607          * pointless.
1608          */
1609         if (HDR_HAS_L1HDR(hdr)) {
1610                 old_state = hdr->b_l1hdr.b_state;
1611                 refcnt = refcount_count(&hdr->b_l1hdr.b_refcnt);
1612                 datacnt = hdr->b_l1hdr.b_datacnt;
1613         } else {
1614                 old_state = arc_l2c_only;
1615                 refcnt = 0;
1616                 datacnt = 0;
1617         }

1619         ASSERT(MUTEX_HELD(hash_lock));
1620         ASSERT3P(new_state, !=, old_state);
1621         ASSERT(refcnt == 0 || datacnt > 0);
1622         ASSERT(!GHOST_STATE(new_state) || datacnt == 0);
1623         ASSERT(old_state != arc_anon || datacnt <= 1);

1625         from_delta = to_delta = datacnt * hdr->b_size;

1627         /*
1628          * If this buffer is evictable, transfer it from the
1629          * old state list to the new state list.
1630          */
1631         if (refcnt == 0) {
1632                 if (old_state != arc_anon && old_state != arc_l2c_only) {
1633                         uint64_t *size = &old_state->arcs_lsize[buftype];
```

```
1635                        ASSERT(HDR_HAS_L1HDR(hdr));
1636                        multilist_remove(&old_state->arcs_list[buftype], hdr);
1638                        /*
1639                         * If prefetching out of the ghost cache,
1640                         * we will have a non-zero datacnt.
1641                         */
1642                        if (GHOST_STATE(old_state) && datacnt == 0) {
1643                                /* ghost elements have a ghost size */
1644                                ASSERT(hdr->b_l1hdr.b_buf == NULL);
1645                                from_delta = hdr->b_size;
1646                        }
1647                        ASSERT3U(*size, >=, from_delta);
1648                        atomic_add_64(size, -from_delta);
1649                }
1650                if (new_state != arc_anon && new_state != arc_l2c_only) {
1651                        uint64_t *size = &new_state->arcs_lsize[buftype];
1653                        /*
1654                         * An L1 header always exists here, since if we're
1655                         * moving to some L1-cached state (i.e. not l2c_only or
1656                         * anonymous), we realloc the header to add an L1hdr
1657                         * beforehand.
1658                         */
1659                        ASSERT(HDR_HAS_L1HDR(hdr));
1660                        multilist_insert(&new_state->arcs_list[buftype], hdr);
1662                        /* ghost elements have a ghost size */
1663                        if (GHOST_STATE(new_state)) {
1664                                ASSERT0(datacnt);
1665                                ASSERT(hdr->b_l1hdr.b_buf == NULL);
1666                                to_delta = hdr->b_size;
1667                        }
1668                        atomic_add_64(size, to_delta);
1669                }
1670        }
1672        ASSERT(!BUF_EMPTY(hdr));
1673        if (new_state == arc_anon && HDR_IN_HASH_TABLE(hdr))
1674                buf_hash_remove(hdr);
1676        /* adjust state sizes (ignore arc_l2c_only) */
1678        if (to_delta && new_state != arc_l2c_only) {
1679                ASSERT(HDR_HAS_L1HDR(hdr));
1680                if (GHOST_STATE(new_state)) {
1681                        ASSERT0(datacnt);
1683                        /*
1684                         * We moving a header to a ghost state, we first
1685                         * remove all arc buffers. Thus, we'll have a
1686                         * datacnt of zero, and no arc buffer to use for
1687                         * the reference. As a result, we use the arc
1688                         * header pointer for the reference.
1689                         */
1690                        (void) refcount_add_many(&new_state->arcs_size,
1691                            hdr->b_size, hdr);
1692                } else {
1693                        ASSERT3U(datacnt, !=, 0);
1695                        /*
1696                         * Each individual buffer holds a unique reference,
1697                         * thus we must remove each of these references one
1698                         * at a time.
1699                         */
1700                        for (arc_buf_t *buf = hdr->b_l1hdr.b_buf; buf != NULL;
```



```
1701                            buf = buf->b_next) {
1702                                (void) refcount_add_many(&new_state->arcs_size,
1703                                    hdr->b_size, buf);
1704                        }
1705                }
1706        }
1708        if (from_delta && old_state != arc_l2c_only) {
1709                ASSERT(HDR_HAS_L1HDR(hdr));
1710                if (GHOST_STATE(old_state)) {
1711                        /*
1712                         * When moving a header off of a ghost state,
1713                         * there's the possibility for datacnt to be
1714                         * non-zero. This is because we first add the
1715                         * arc buffer to the header prior to changing
1716                         * the header's state. Since we used the header
1717                         * for the reference when putting the header on
1718                         * the ghost state, we must balance that and use
1719                         * the header when removing off the ghost state
1720                         * (even though datacnt is non zero).
1721                         */
1723                        IMPLY(datacnt == 0, new_state == arc_anon ||
1724                            new_state == arc_l2c_only);
1726                        (void) refcount_remove_many(&old_state->arcs_size,
1727                            hdr->b_size, hdr);
1728                } else {
1729                        ASSERT3P(datacnt, !=, 0);
1731                        /*
1732                         * Each individual buffer holds a unique reference,
1733                         * thus we must remove each of these references one
1734                         * at a time.
1735                         */
1736                        for (arc_buf_t *buf = hdr->b_l1hdr.b_buf; buf != NULL;
1737                            buf = buf->b_next) {
1738                                (void) refcount_remove_many(
1739                                    &old_state->arcs_size, hdr->b_size, buf);
1740                        }
1741                }
1742        }
1744        if (HDR_HAS_L1HDR(hdr))
1745                hdr->b_l1hdr.b_state = new_state;
1747        /*
1748         * L2 headers should never be on the L2 state list since they don't
1749         * have L1 headers allocated.
1750         */
1751        ASSERT(multilist_is_empty(&arc_l2c_only->arcs_list[ARC_BUFC_DATA]) &&
1752            multilist_is_empty(&arc_l2c_only->arcs_list[ARC_BUFC_METADATA]));
1753 }
1755 void
1756 arc_space_consume(uint64_t space, arc_space_type_t type)
1757 {
1758        ASSERT(type >= 0 && type < ARC_SPACE_NUMTYPES);
1760        switch (type) {
1761        case ARC_SPACE_DATA:
1762                ARCSTAT_INCR(arcstat_data_size, space);
1763                break;
1764        case ARC_SPACE_META:
1765                ARCSTAT_INCR(arcstat_metadata_size, space);
1766                break;
```

```
1767         case ARC_SPACE_OTHER:
1768                 ARCSTAT_INCR(arcstat_other_size, space);
1769                 break;
1770         case ARC_SPACE_HDRS:
1771                 ARCSTAT_INCR(arcstat_hdr_size, space);
1772                 break;
1773         case ARC_SPACE_L2HDRS:
1774                 ARCSTAT_INCR(arcstat_l2_hdr_size, space);
1775                 break;
1776         }

1778         if (type != ARC_SPACE_DATA)
1779                 ARCSTAT_INCR(arcstat_meta_used, space);

1781         atomic_add_64(&arc_size, space);
1782 }

1784 void
1785 arc_space_return(uint64_t space, arc_space_type_t type)
1786 {
1787         ASSERT(type >= 0 && type < ARC_SPACE_NUMTYPES);

1789         switch (type) {
1790         case ARC_SPACE_DATA:
1791                 ARCSTAT_INCR(arcstat_data_size, -space);
1792                 break;
1793         case ARC_SPACE_META:
1794                 ARCSTAT_INCR(arcstat_metadata_size, -space);
1795                 break;
1796         case ARC_SPACE_OTHER:
1797                 ARCSTAT_INCR(arcstat_other_size, -space);
1798                 break;
1799         case ARC_SPACE_HDRS:
1800                 ARCSTAT_INCR(arcstat_hdr_size, -space);
1801                 break;
1802         case ARC_SPACE_L2HDRS:
1803                 ARCSTAT_INCR(arcstat_l2_hdr_size, -space);
1804                 break;
1805         }

1807         if (type != ARC_SPACE_DATA) {
1808                 ASSERT(arc_meta_used >= space);
1809                 if (arc_meta_max < arc_meta_used)
1810                         arc_meta_max = arc_meta_used;
1811                 ARCSTAT_INCR(arcstat_meta_used, -space);
1812         }

1814         ASSERT(arc_size >= space);
1815         atomic_add_64(&arc_size, -space);
1816 }

1818 arc_buf_t *
1819 arc_buf_alloc(spa_t *spa, int32_t size, void *tag, arc_buf_contents_t type)
1820 {
1821         arc_buf_hdr_t *hdr;
1822         arc_buf_t *buf;

1824         ASSERT3U(size, >, 0);
1825         hdr = kmem_cache_alloc(hdr_full_cache, KM_PUSHPAGE);
1826         ASSERT(BUF_EMPTY(hdr));
1827         ASSERT3P(hdr->b_freeze_cksum, ==, NULL);
1828         hdr->b_size = size;
1829         hdr->b_spa = spa_load_guid(spa);

1831         buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
1832         buf->b_hdr = hdr;
```



```
1833         buf->b_data = NULL;
1834         buf->b_efunc = NULL;
1835         buf->b_private = NULL;
1836         buf->b_next = NULL;

1838         hdr->b_flags = arc_bufc_to_flags(type);
1839         hdr->b_flags |= ARC_FLAG_HAS_L1HDR;

1841         hdr->b_l1hdr.b_buf = buf;
1842         hdr->b_l1hdr.b_state = arc_anon;
1843         hdr->b_l1hdr.b_arc_access = 0;
1844         hdr->b_l1hdr.b_datacnt = 1;
1845         hdr->b_l1hdr.b_tmp_cdata = NULL;

1847         arc_get_data_buf(buf);
1848         ASSERT(refcount_is_zero(&hdr->b_l1hdr.b_refcnt));
1849         (void) refcount_add(&hdr->b_l1hdr.b_refcnt, tag);

1851         return (buf);
1852 }

1854 static char *arc_onloan_tag = "onloan";

1856 /*
1857  * Loan out an anonymous arc buffer. Loaned buffers are not counted as in
1858  * flight data by arc_tempreserve_space() until they are "returned". Loaned
1859  * buffers must be returned to the arc before they can be used by the DMU or
1860  * freed.
1861  */
1862 arc_buf_t *
1863 arc_loan_buf(spa_t *spa, int size)
1864 {
1865         arc_buf_t *buf;

1867         buf = arc_buf_alloc(spa, size, arc_onloan_tag, ARC_BUFC_DATA);

1869         atomic_add_64(&arc_loaned_bytes, size);
1870         return (buf);
1871 }

1873 /*
1874  * Return a loaned arc buffer to the arc.
1875  */
1876 void
1877 arc_return_buf(arc_buf_t *buf, void *tag)
1878 {
1879         arc_buf_hdr_t *hdr = buf->b_hdr;

1881         ASSERT(buf->b_data != NULL);
1882         ASSERT(HDR_HAS_L1HDR(hdr));
1883         (void) refcount_add(&hdr->b_l1hdr.b_refcnt, tag);
1884         (void) refcount_remove(&hdr->b_l1hdr.b_refcnt, arc_onloan_tag);

1886         atomic_add_64(&arc_loaned_bytes, -hdr->b_size);
1887 }

1889 /* Detach an arc_buf from a dbuf (tag) */
1890 void
1891 arc_loan_inuse_buf(arc_buf_t *buf, void *tag)
1892 {
1893         arc_buf_hdr_t *hdr = buf->b_hdr;

1895         ASSERT(buf->b_data != NULL);
1896         ASSERT(HDR_HAS_L1HDR(hdr));
1897         (void) refcount_add(&hdr->b_l1hdr.b_refcnt, arc_onloan_tag);
1898         (void) refcount_remove(&hdr->b_l1hdr.b_refcnt, tag);
```

```
1899         buf->b_efunc = NULL;
1900         buf->b_private = NULL;

1902         atomic_add_64(&arc_loaned_bytes, hdr->b_size);
1903 }

1905 static arc_buf_t *
1906 arc_buf_clone(arc_buf_t *from)
1907 {
1908         arc_buf_t *buf;
1909         arc_buf_hdr_t *hdr = from->b_hdr;
1910         uint64_t size = hdr->b_size;

1912         ASSERT(HDR_HAS_L1HDR(hdr));
1913         ASSERT(hdr->b_l1hdr.b_state != arc_anon);

1915         buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
1916         buf->b_hdr = hdr;
1917         buf->b_data = NULL;
1918         buf->b_efunc = NULL;
1919         buf->b_private = NULL;
1920         buf->b_next = hdr->b_l1hdr.b_buf;
1921         hdr->b_l1hdr.b_buf = buf;
1922         arc_get_data_buf(buf);
1923         bcopy(from->b_data, buf->b_data, size);

1925         /*
1926          * This buffer already exists in the arc so create a duplicate
1927          * copy for the caller.  If the buffer is associated with user data
1928          * then track the size and number of duplicates.  These stats will be
1929          * updated as duplicate buffers are created and destroyed.
1930          */
1931         if (HDR_ISTYPE_DATA(hdr)) {
1932                 ARCSTAT_BUMP(arcstat_duplicate_buffers);
1933                 ARCSTAT_INCR(arcstat_duplicate_buffers_size, size);
1934         }
1935         hdr->b_l1hdr.b_datacnt += 1;
1936         return (buf);
1937 }

1939 void
1940 arc_buf_add_ref(arc_buf_t *buf, void* tag)
1941 {
1942         arc_buf_hdr_t *hdr;
1943         kmutex_t *hash_lock;

1945         /*
1946          * Check to see if this buffer is evicted.  Callers
1947          * must verify b_data != NULL to know if the add_ref
1948          * was successful.
1949          */
1950         mutex_enter(&buf->b_evict_lock);
1951         if (buf->b_data == NULL) {
1952                 mutex_exit(&buf->b_evict_lock);
1953                 return;
1954         }
1955         hash_lock = HDR_LOCK(buf->b_hdr);
1956         mutex_enter(hash_lock);
1957         hdr = buf->b_hdr;
1958         ASSERT(HDR_HAS_L1HDR(hdr));
1959         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
1960         mutex_exit(&buf->b_evict_lock);

1962         ASSERT(hdr->b_l1hdr.b_state == arc_mru ||
1963             hdr->b_l1hdr.b_state == arc_mfu);
```



```
1965         add_reference(hdr, hash_lock, tag);
1966         DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
1967         arc_access(hdr, hash_lock);
1968         mutex_exit(hash_lock);
1969         ARCSTAT_BUMP(arcstat_hits);
1970         ARCSTAT_CONDSTAT(!HDR_PREFETCH(hdr),
1971             demand, prefetch, !HDR_ISTYPE_METADATA(hdr),
1972             data, metadata, hits);
1973 }

1975 static void
1976 arc_buf_free_on_write(void *data, size_t size,
1977     void (*free_func)(void *, size_t))
1978 {
1979         l2arc_data_free_t *df;

1981         df = kmem_alloc(sizeof (*df), KM_SLEEP);
1982         df->l2df_data = data;
1983         df->l2df_size = size;
1984         df->l2df_func = free_func;
1985         mutex_enter(&l2arc_free_on_write_mtx);
1986         list_insert_head(l2arc_free_on_write, df);
1987         mutex_exit(&l2arc_free_on_write_mtx);
1988 }

1990 /*
1991  * Free the arc data buffer.  If it is an l2arc write in progress,
1992  * the buffer is placed on l2arc_free_on_write to be freed later.
1993  */
1994 static void
1995 arc_buf_data_free(arc_buf_t *buf, void (*free_func)(void *, size_t))
1996 {
1997         arc_buf_hdr_t *hdr = buf->b_hdr;

1999         if (HDR_L2_WRITING(hdr)) {
2000                 arc_buf_free_on_write(buf->b_data, hdr->b_size, free_func);
2001                 ARCSTAT_BUMP(arcstat_l2_free_on_write);
2002         } else {
2003                 free_func(buf->b_data, hdr->b_size);
2004         }
2005 }

2007 static void
2008 arc_buf_l2_cdata_free(arc_buf_hdr_t *hdr)
2009 {
2010         ASSERT(HDR_HAS_L2HDR(hdr));
2011         ASSERT(MUTEX_HELD(&hdr->b_l2hdr.b_dev->l2ad_mtx));

2013         /*
2014          * The b_tmp_cdata field is linked off of the b_l1hdr, so if
2015          * that doesn't exist, the header is in the arc_l2c_only state,
2016          * and there isn't anything to free (it's already been freed).
2017          */
2018         if (!HDR_HAS_L1HDR(hdr))
2019                 return;

2021         /*
2022          * The header isn't being written to the l2arc device, thus it
2023          * shouldn't have a b_tmp_cdata to free.
2024          */
2025         if (!HDR_L2_WRITING(hdr)) {
2026                 ASSERT3P(hdr->b_l1hdr.b_tmp_cdata, ==, NULL);
2027                 return;
2028         }

2030         /*
```

```
2031          * The header does not have compression enabled. This can be due
2032          * to the buffer not being compressible, or because we're
2033          * freeing the buffer before the second phase of
2034          * l2arc_write_buffer() has started (which does the compression
2035          * step). In either case, b_tmp_cdata does not point to a
2036          * separately compressed buffer, so there's nothing to free (it
2037          * points to the same buffer as the arc_buf_t's b_data field).
2038          */
2039         if (hdr->b_l2hdr.b_compress == ZIO_COMPRESS_OFF) {
2040                 hdr->b_l1hdr.b_tmp_cdata = NULL;
2041                 return;
2042         }

2044         /*
2045          * There's nothing to free since the buffer was all zero's and
2046          * compressed to a zero length buffer.
2047          */
2048         if (hdr->b_l2hdr.b_compress == ZIO_COMPRESS_EMPTY) {
2049                 ASSERT3P(hdr->b_l1hdr.b_tmp_cdata, ==, NULL);
2050                 return;
2051         }

2053         ASSERT(L2ARC_IS_VALID_COMPRESS(hdr->b_l2hdr.b_compress));

2055         arc_buf_free_on_write(hdr->b_l1hdr.b_tmp_cdata,
2056             hdr->b_size, zio_data_buf_free);

2058         ARCSTAT_BUMP(arcstat_l2_cdata_free_on_write);
2059         hdr->b_l1hdr.b_tmp_cdata = NULL;
2060 }

2062 /*
2063  * Free up buf->b_data and if 'remove' is set, then pull the
2064  * arc_buf_t off of the the arc_buf_hdr_t's list and free it.
2065  */
2066 static void
2067 arc_buf_destroy(arc_buf_t *buf, boolean_t remove)
2068 {
2069         arc_buf_t **bufp;

2071         /* free up data associated with the buf */
2072         if (buf->b_data != NULL) {
2073                 arc_state_t *state = buf->b_hdr->b_l1hdr.b_state;
2074                 uint64_t size = buf->b_hdr->b_size;
2075                 arc_buf_contents_t type = arc_buf_type(buf->b_hdr);

2077                 arc_cksum_verify(buf);
2078                 arc_buf_unwatch(buf);

2080                 if (type == ARC_BUFC_METADATA) {
2081                         arc_buf_data_free(buf, zio_buf_free);
2082                         arc_space_return(size, ARC_SPACE_META);
2083                 } else {
2084                         ASSERT(type == ARC_BUFC_DATA);
2085                         arc_buf_data_free(buf, zio_data_buf_free);
2086                         arc_space_return(size, ARC_SPACE_DATA);
2087                 }

2089                 /* protected by hash lock, if in the hash table */
2090                 if (multilist_link_active(&buf->b_hdr->b_l1hdr.b_arc_node)) {
2091                         uint64_t *cnt = &state->arcs_lsize[type];

2093                         ASSERT(refcount_is_zero(
2094                             &buf->b_hdr->b_l1hdr.b_refcnt));
2095                         ASSERT(state != arc_anon && state != arc_l2c_only);
```



```
2097                         ASSERT3U(*cnt, >=, size);
2098                         atomic_add_64(cnt, -size);
2099                 }

2101                 (void) refcount_remove_many(&state->arcs_size, size, buf);
2102                 buf->b_data = NULL;

2104                 /*
2105                  * If we're destroying a duplicate buffer make sure
2106                  * that the appropriate statistics are updated.
2107                  */
2108                 if (buf->b_hdr->b_l1hdr.b_datacnt > 1 &&
2109                     HDR_ISTYPE_DATA(buf->b_hdr)) {
2110                         ARCSTAT_BUMPDOWN(arcstat_duplicate_buffers);
2111                         ARCSTAT_INCR(arcstat_duplicate_buffers_size, -size);
2112                 }
2113                 ASSERT(buf->b_hdr->b_l1hdr.b_datacnt > 0);
2114                 buf->b_hdr->b_l1hdr.b_datacnt -= 1;
2115         }

2117         /* only remove the buf if requested */
2118         if (!remove)
2119                 return;

2121         /* remove the buf from the hdr list */
2122         for (bufp = &buf->b_hdr->b_l1hdr.b_buf; *bufp != buf;
2123             bufp = &(*bufp)->b_next)
2124                 continue;
2125         *bufp = buf->b_next;
2126         buf->b_next = NULL;

2128         ASSERT(buf->b_efunc == NULL);

2130         /* clean up the buf */
2131         buf->b_hdr = NULL;
2132         kmem_cache_free(buf_cache, buf);
2133 }

2135 static void
2136 arc_hdr_l2hdr_destroy(arc_buf_hdr_t *hdr)
2137 {
2138         l2arc_buf_hdr_t *l2hdr = &hdr->b_l2hdr;
2139         l2arc_dev_t *dev = l2hdr->b_dev;

2141         ASSERT(MUTEX_HELD(&dev->l2ad_mtx));
2142         ASSERT(HDR_HAS_L2HDR(hdr));

2144         list_remove(&dev->l2ad_buflist, hdr);

2146         /*
2147          * We don't want to leak the b_tmp_cdata buffer that was
2148          * allocated in l2arc_write_buffers()
2149          */
2150         arc_buf_l2_cdata_free(hdr);

2152         /*
2153          * If the l2hdr's b_daddr is equal to L2ARC_ADDR_UNSET, then
2154          * this header is being processed by l2arc_write_buffers() (i.e.
2155          * it's in the first stage of l2arc_write_buffers()).
2156          * Re-affirming that truth here, just to serve as a reminder. If
2157          * b_daddr does not equal L2ARC_ADDR_UNSET, then the header may or
2158          * may not have its HDR_L2_WRITING flag set. (the write may have
2159          * completed, in which case HDR_L2_WRITING will be false and the
2160          * b_daddr field will point to the address of the buffer on disk).
2161          */
2162         IMPLY(l2hdr->b_daddr == L2ARC_ADDR_UNSET, HDR_L2_WRITING(hdr));
```

```
2164         /*
2165          * If b_daddr is equal to L2ARC_ADDR_UNSET, we're racing with
2166          * l2arc_write_buffers(). Since we've just removed this header
2167          * from the l2arc buffer list, this header will never reach the
2168          * second stage of l2arc_write_buffers(), which increments the
2169          * accounting stats for this header. Thus, we must be careful
2170          * not to decrement them for this header either.
2171          */
2172         if (l2hdr->b_daddr != L2ARC_ADDR_UNSET) {
2173                 ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
2174                 ARCSTAT_INCR(arcstat_l2_size, -hdr->b_size);

2176                 vdev_space_update(dev->l2ad_vdev,
2177                     -l2hdr->b_asize, 0, 0);

2179                 (void) refcount_remove_many(&dev->l2ad_alloc,
2180                     l2hdr->b_asize, hdr);
2181         }

2183         hdr->b_flags &= ~ARC_FLAG_HAS_L2HDR;
2184 }

2186 static void
2187 arc_hdr_destroy(arc_buf_hdr_t *hdr)
2188 {
2189         if (HDR_HAS_L1HDR(hdr)) {
2190                 ASSERT(hdr->b_l1hdr.b_buf == NULL ||
2191                     hdr->b_l1hdr.b_datacnt > 0);
2192                 ASSERT(refcount_is_zero(&hdr->b_l1hdr.b_refcnt));
2193                 ASSERT3P(hdr->b_l1hdr.b_state, ==, arc_anon);
2194         }
2195         ASSERT(!HDR_IO_IN_PROGRESS(hdr));
2196         ASSERT(!HDR_IN_HASH_TABLE(hdr));

2198         if (HDR_HAS_L2HDR(hdr)) {
2199                 l2arc_dev_t *dev = hdr->b_l2hdr.b_dev;
2200                 boolean_t buflist_held = MUTEX_HELD(&dev->l2ad_mtx);

2202                 if (!buflist_held)
2203                         mutex_enter(&dev->l2ad_mtx);

2205                 /*
2206                  * Even though we checked this conditional above, we
2207                  * need to check this again now that we have the
2208                  * l2ad_mtx. This is because we could be racing with
2209                  * another thread calling l2arc_evict() which might have
2210                  * destroyed this header's L2 portion as we were waiting
2211                  * to acquire the l2ad_mtx. If that happens, we don't
2212                  * want to re-destroy the header's L2 portion.
2213                  */
2214                 if (HDR_HAS_L2HDR(hdr))
2215                         arc_hdr_l2hdr_destroy(hdr);

2217                 if (!buflist_held)
2218                         mutex_exit(&dev->l2ad_mtx);
2219         }

2221         if (!BUF_EMPTY(hdr))
2222                 buf_discard_identity(hdr);

2224         if (hdr->b_freeze_cksum != NULL) {
2225                 kmem_free(hdr->b_freeze_cksum, sizeof (zio_cksum_t));
2226                 hdr->b_freeze_cksum = NULL;
2227         }
```



```
2229         if (HDR_HAS_L1HDR(hdr)) {
2230                 while (hdr->b_l1hdr.b_buf) {
2231                         arc_buf_t *buf = hdr->b_l1hdr.b_buf;

2233                         if (buf->b_efunc != NULL) {
2234                                 mutex_enter(&arc_user_evicts_lock);
2235                                 mutex_enter(&buf->b_evict_lock);
2236                                 ASSERT(buf->b_hdr != NULL);
2237                                 arc_buf_destroy(hdr->b_l1hdr.b_buf, FALSE);
2238                                 hdr->b_l1hdr.b_buf = buf->b_next;
2239                                 buf->b_hdr = &arc_eviction_hdr;
2240                                 buf->b_next = arc_eviction_list;
2241                                 arc_eviction_list = buf;
2242                                 mutex_exit(&buf->b_evict_lock);
2243                                 cv_signal(&arc_user_evicts_cv);
2244                                 mutex_exit(&arc_user_evicts_lock);
2245                         } else {
2246                                 arc_buf_destroy(hdr->b_l1hdr.b_buf, TRUE);
2247                         }
2248                 }
2249 #ifdef ZFS_DEBUG
2250                 if (hdr->b_l1hdr.b_thawed != NULL) {
2251                         kmem_free(hdr->b_l1hdr.b_thawed, 1);
2252                         hdr->b_l1hdr.b_thawed = NULL;
2253                 }
2254 #endif
2255         }

2257         ASSERT3P(hdr->b_hash_next, ==, NULL);
2258         if (HDR_HAS_L1HDR(hdr)) {
2259                 ASSERT(!multilist_link_active(&hdr->b_l1hdr.b_arc_node));
2260                 ASSERT3P(hdr->b_l1hdr.b_acb, ==, NULL);
2261                 kmem_cache_free(hdr_full_cache, hdr);
2262         } else {
2263                 kmem_cache_free(hdr_l2only_cache, hdr);
2264         }
2265 }

2267 void
2268 arc_buf_free(arc_buf_t *buf, void *tag)
2269 {
2270         arc_buf_hdr_t *hdr = buf->b_hdr;
2271         int hashed = hdr->b_l1hdr.b_state != arc_anon;

2273         ASSERT(buf->b_efunc == NULL);
2274         ASSERT(buf->b_data != NULL);

2276         if (hashed) {
2277                 kmutex_t *hash_lock = HDR_LOCK(hdr);

2279                 mutex_enter(hash_lock);
2280                 hdr = buf->b_hdr;
2281                 ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));

2283                 (void) remove_reference(hdr, hash_lock, tag);
2284                 if (hdr->b_l1hdr.b_datacnt > 1) {
2285                         arc_buf_destroy(buf, TRUE);
2286                 } else {
2287                         ASSERT(buf == hdr->b_l1hdr.b_buf);
2288                         ASSERT(buf->b_efunc == NULL);
2289                         hdr->b_flags |= ARC_FLAG_BUF_AVAILABLE;
2290                 }
2291                 mutex_exit(hash_lock);
2292         } else if (HDR_IO_IN_PROGRESS(hdr)) {
2293                 int destroy_hdr;
2294                 /*
```

```
2295                  * We are in the middle of an async write.  Don't destroy
2296                  * this buffer unless the write completes before we finish
2297                  * decrementing the reference count.
2298                  */
2299                 mutex_enter(&arc_user_evicts_lock);
2300                 (void) remove_reference(hdr, NULL, tag);
2301                 ASSERT(refcount_is_zero(&hdr->b_l1hdr.b_refcnt));
2302                 destroy_hdr = !HDR_IO_IN_PROGRESS(hdr);
2303                 mutex_exit(&arc_user_evicts_lock);
2304                 if (destroy_hdr)
2305                         arc_hdr_destroy(hdr);
2306         } else {
2307                 if (remove_reference(hdr, NULL, tag) > 0)
2308                         arc_buf_destroy(buf, TRUE);
2309                 else
2310                         arc_hdr_destroy(hdr);
2311         }
2312 }

2314 boolean_t
2315 arc_buf_remove_ref(arc_buf_t *buf, void* tag)
2316 {
2317         arc_buf_hdr_t *hdr = buf->b_hdr;
2318         kmutex_t *hash_lock = HDR_LOCK(hdr);
2319         boolean_t no_callback = (buf->b_efunc == NULL);

2321         if (hdr->b_l1hdr.b_state == arc_anon) {
2322                 ASSERT(hdr->b_l1hdr.b_datacnt == 1);
2323                 arc_buf_free(buf, tag);
2324                 return (no_callback);
2325         }

2327         mutex_enter(hash_lock);
2328         hdr = buf->b_hdr;
2329         ASSERT(hdr->b_l1hdr.b_datacnt > 0);
2330         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
2331         ASSERT(hdr->b_l1hdr.b_state != arc_anon);
2332         ASSERT(buf->b_data != NULL);

2334         (void) remove_reference(hdr, hash_lock, tag);
2335         if (hdr->b_l1hdr.b_datacnt > 1) {
2336                 if (no_callback)
2337                         arc_buf_destroy(buf, TRUE);
2338         } else if (no_callback) {
2339                 ASSERT(hdr->b_l1hdr.b_buf == buf && buf->b_next == NULL);
2340                 ASSERT(buf->b_efunc == NULL);
2341                 hdr->b_flags |= ARC_FLAG_BUF_AVAILABLE;
2342         }
2343         ASSERT(no_callback || hdr->b_l1hdr.b_datacnt > 1 ||
2344             refcount_is_zero(&hdr->b_l1hdr.b_refcnt));
2345         mutex_exit(hash_lock);
2346         return (no_callback);
2347 }

2349 int32_t
2350 arc_buf_size(arc_buf_t *buf)
2351 {
2352         return (buf->b_hdr->b_size);
2353 }

2355 /*
2356  * Called from the DMU to determine if the current buffer should be
2357  * evicted. In order to ensure proper locking, the eviction must be initiated
2358  * from the DMU. Return true if the buffer is associated with user data and
2359  * duplicate buffers still exist.
2360  */
```



```
2361 boolean_t
2362 arc_buf_eviction_needed(arc_buf_t *buf)
2363 {
2364         arc_buf_hdr_t *hdr;
2365         boolean_t evict_needed = B_FALSE;

2367         if (zfs_disable_dup_eviction)
2368                 return (B_FALSE);

2370         mutex_enter(&buf->b_evict_lock);
2371         hdr = buf->b_hdr;
2372         if (hdr == NULL) {
2373                 /*
2374                  * We are in arc_do_user_evicts(); let that function
2375                  * perform the eviction.
2376                  */
2377                 ASSERT(buf->b_data == NULL);
2378                 mutex_exit(&buf->b_evict_lock);
2379                 return (B_FALSE);
2380         } else if (buf->b_data == NULL) {
2381                 /*
2382                  * We have already been added to the arc eviction list;
2383                  * recommend eviction.
2384                  */
2385                 ASSERT3P(hdr, ==, &arc_eviction_hdr);
2386                 mutex_exit(&buf->b_evict_lock);
2387                 return (B_TRUE);
2388         }

2390         if (hdr->b_l1hdr.b_datacnt > 1 && HDR_ISTYPE_DATA(hdr))
2391                 evict_needed = B_TRUE;

2393         mutex_exit(&buf->b_evict_lock);
2394         return (evict_needed);
2395 }

2397 /*
2398  * Evict the arc_buf_hdr that is provided as a parameter. The resultant
2399  * state of the header is dependent on it's state prior to entering this
2400  * function. The following transitions are possible:
2401  *
2402  *    - arc_mru -> arc_mru_ghost
2403  *    - arc_mfu -> arc_mfu_ghost
2404  *    - arc_mru_ghost -> arc_l2c_only
2405  *    - arc_mru_ghost -> deleted
2406  *    - arc_mfu_ghost -> arc_l2c_only
2407  *    - arc_mfu_ghost -> deleted
2408  */
2409 static int64_t
2410 arc_evict_hdr(arc_buf_hdr_t *hdr, kmutex_t *hash_lock)
2411 {
2412         arc_state_t *evicted_state, *state;
2413         int64_t bytes_evicted = 0;

2415         ASSERT(MUTEX_HELD(hash_lock));
2416         ASSERT(HDR_HAS_L1HDR(hdr));

2418         state = hdr->b_l1hdr.b_state;
2419         if (GHOST_STATE(state)) {
2420                 ASSERT(!HDR_IO_IN_PROGRESS(hdr));
2421                 ASSERT(hdr->b_l1hdr.b_buf == NULL);

2423                 /*
2424                  * l2arc_write_buffers() relies on a header's L1 portion
2425                  * (i.e. it's b_tmp_cdata field) during it's write phase.
2426                  * Thus, we cannot push a header onto the arc_l2c_only
```

```
2427                 * state (removing it's L1 piece) until the header is
2428                 * done being written to the l2arc.
2429                 */
2430                if (HDR_HAS_L2HDR(hdr) && HDR_L2_WRITING(hdr)) {
2431                        ARCSTAT_BUMP(arcstat_evict_l2_skip);
2432                        return (bytes_evicted);
2433                }

2435                ARCSTAT_BUMP(arcstat_deleted);
2436                bytes_evicted += hdr->b_size;

2438                DTRACE_PROBE1(arc__delete, arc_buf_hdr_t *, hdr);

2440                if (HDR_HAS_L2HDR(hdr)) {
2441                        /*
2442                         * This buffer is cached on the 2nd Level ARC;
2443                         * don't destroy the header.
2444                         */
2445                        arc_change_state(arc_l2c_only, hdr, hash_lock);
2446                        /*
2447                         * dropping from L1+L2 cached to L2-only,
2448                         * realloc to remove the L1 header.
2449                         */
2450                        hdr = arc_hdr_realloc(hdr, hdr_full_cache,
2451                            hdr_l2only_cache);
2452                } else {
2453                        arc_change_state(arc_anon, hdr, hash_lock);
2454                        arc_hdr_destroy(hdr);
2455                }
2456                return (bytes_evicted);
2457        }

2459        ASSERT(state == arc_mru || state == arc_mfu);
2460        evicted_state = (state == arc_mru) ? arc_mru_ghost : arc_mfu_ghost;

2462        /* prefetch buffers have a minimum lifespan */
2463        if (HDR_IO_IN_PROGRESS(hdr) ||
2464            ((hdr->b_flags & (ARC_FLAG_PREFETCH | ARC_FLAG_INDIRECT)) &&
2465            ddi_get_lbolt() - hdr->b_l1hdr.b_arc_access <
2466            arc_min_prefetch_lifespan)) {
2467                ARCSTAT_BUMP(arcstat_evict_skip);
2468                return (bytes_evicted);
2469        }

2471        ASSERT0(refcount_count(&hdr->b_l1hdr.b_refcnt));
2472        ASSERT3U(hdr->b_l1hdr.b_datacnt, >, 0);
2473        while (hdr->b_l1hdr.b_buf) {
2474                arc_buf_t *buf = hdr->b_l1hdr.b_buf;
2475                if (!mutex_tryenter(&buf->b_evict_lock)) {
2476                        ARCSTAT_BUMP(arcstat_mutex_miss);
2477                        break;
2478                }
2479                if (buf->b_data != NULL)
2480                        bytes_evicted += hdr->b_size;
2481                if (buf->b_efunc != NULL) {
2482                        mutex_enter(&arc_user_evicts_lock);
2483                        arc_buf_destroy(buf, FALSE);
2484                        hdr->b_l1hdr.b_buf = buf->b_next;
2485                        buf->b_hdr = &arc_eviction_hdr;
2486                        buf->b_next = arc_eviction_list;
2487                        arc_eviction_list = buf;
2488                        cv_signal(&arc_user_evicts_cv);
2489                        mutex_exit(&arc_user_evicts_lock);
2490                        mutex_exit(&buf->b_evict_lock);
2491                } else {
2492                        mutex_exit(&buf->b_evict_lock);
```



```
2493                        arc_buf_destroy(buf, TRUE);
2494                }
2495        }

2497        if (HDR_HAS_L2HDR(hdr)) {
2498                ARCSTAT_INCR(arcstat_evict_l2_cached, hdr->b_size);
2499        } else {
2500                if (l2arc_write_eligible(hdr->b_spa, hdr))
2501                        ARCSTAT_INCR(arcstat_evict_l2_eligible, hdr->b_size);
2502                else
2503                        ARCSTAT_INCR(arcstat_evict_l2_ineligible, hdr->b_size);
2504        }

2506        if (hdr->b_l1hdr.b_datacnt == 0) {
2507                arc_change_state(evicted_state, hdr, hash_lock);
2508                ASSERT(HDR_IN_HASH_TABLE(hdr));
2509                hdr->b_flags |= ARC_FLAG_IN_HASH_TABLE;
2510                hdr->b_flags &= ~ARC_FLAG_BUF_AVAILABLE;
2511                DTRACE_PROBE1(arc__evict, arc_buf_hdr_t *, hdr);
2512        }

2514        return (bytes_evicted);
2515 }

2517 static uint64_t
2518 arc_evict_state_impl(multilist_t *ml, int idx, arc_buf_hdr_t *marker,
2519     uint64_t spa, int64_t bytes)
2520 {
2521        multilist_sublist_t *mls;
2522        uint64_t bytes_evicted = 0;
2523        arc_buf_hdr_t *hdr;
2524        kmutex_t *hash_lock;
2525        int evict_count = 0;

2527        ASSERT3P(marker, !=, NULL);
2528        IMPLY(bytes < 0, bytes == ARC_EVICT_ALL);

2530        mls = multilist_sublist_lock(ml, idx);

2532        for (hdr = multilist_sublist_prev(mls, marker); hdr != NULL;
2533            hdr = multilist_sublist_prev(mls, marker)) {
2534                if ((bytes != ARC_EVICT_ALL && bytes_evicted >= bytes) ||
2535                    (evict_count >= zfs_arc_evict_batch_limit))
2536                        break;

2538                /*
2539                 * To keep our iteration location, move the marker
2540                 * forward. Since we're not holding hdr's hash lock, we
2541                 * must be very careful and not remove 'hdr' from the
2542                 * sublist. Otherwise, other consumers might mistake the
2543                 * 'hdr' as not being on a sublist when they call the
2544                 * multilist_link_active() function (they all rely on
2545                 * the hash lock protecting concurrent insertions and
2546                 * removals). multilist_sublist_move_forward() was
2547                 * specifically implemented to ensure this is the case
2548                 * (only 'marker' will be removed and re-inserted).
2549                 */
2550                multilist_sublist_move_forward(mls, marker);

2552                /*
2553                 * The only case where the b_spa field should ever be
2554                 * zero, is the marker headers inserted by
2555                 * arc_evict_state(). It's possible for multiple threads
2556                 * to be calling arc_evict_state() concurrently (e.g.
2557                 * dsl_pool_close() and zio_inject_fault()), so we must
2558                 * skip any markers we see from these other threads.
```

```
2559                 */
2560                if (hdr->b_spa == 0)
2561                        continue;

2563                /* we're only interested in evicting buffers of a certain spa */
2564                if (spa != 0 && hdr->b_spa != spa) {
2565                        ARCSTAT_BUMP(arcstat_evict_skip);
2566                        continue;
2567                }

2569                hash_lock = HDR_LOCK(hdr);

2571                /*
2572                 * We aren't calling this function from any code path
2573                 * that would already be holding a hash lock, so we're
2574                 * asserting on this assumption to be defensive in case
2575                 * this ever changes. Without this check, it would be
2576                 * possible to incorrectly increment arcstat_mutex_miss
2577                 * below (e.g. if the code changed such that we called
2578                 * this function with a hash lock held).
2579                 */
2580                ASSERT(!MUTEX_HELD(hash_lock));

2582                if (mutex_tryenter(hash_lock)) {
2583                        uint64_t evicted = arc_evict_hdr(hdr, hash_lock);
2584                        mutex_exit(hash_lock);

2586                        bytes_evicted += evicted;

2588                        /*
2589                         * If evicted is zero, arc_evict_hdr() must have
2590                         * decided to skip this header, don't increment
2591                         * evict_count in this case.
2592                         */
2593                        if (evicted != 0)
2594                                evict_count++;

2596                        /*
2597                         * If arc_size isn't overflowing, signal any
2598                         * threads that might happen to be waiting.
2599                         *
2600                         * For each header evicted, we wake up a single
2601                         * thread. If we used cv_broadcast, we could
2602                         * wake up "too many" threads causing arc_size
2603                         * to significantly overflow arc_c; since
2604                         * arc_get_data_buf() doesn't check for overflow
2605                         * when it's woken up (it doesn't because it's
2606                         * possible for the ARC to be overflowing while
2607                         * full of un-evictable buffers, and the
2608                         * function should proceed in this case).
2609                         *
2610                         * If threads are left sleeping, due to not
2611                         * using cv_broadcast, they will be woken up
2612                         * just before arc_reclaim_thread() sleeps.
2613                         */
2614                        mutex_enter(&arc_reclaim_lock);
2615                        if (!arc_is_overflowing())
2616                                cv_signal(&arc_reclaim_waiters_cv);
2617                        mutex_exit(&arc_reclaim_lock);
2618                } else {
2619                        ARCSTAT_BUMP(arcstat_mutex_miss);
2620                }
2621        }

2623        multilist_sublist_unlock(mls);
```



```
2625        return (bytes_evicted);
2626 }

2628 /*
2629  * Evict buffers from the given arc state, until we've removed the
2630  * specified number of bytes. Move the removed buffers to the
2631  * appropriate evict state.
2632  *
2633  * This function makes a "best effort". It skips over any buffers
2634  * it can't get a hash_lock on, and so, may not catch all candidates.
2635  * It may also return without evicting as much space as requested.
2636  *
2637  * If bytes is specified using the special value ARC_EVICT_ALL, this
2638  * will evict all available (i.e. unlocked and evictable) buffers from
2639  * the given arc state; which is used by arc_flush().
2640  */
2641 static uint64_t
2642 arc_evict_state(arc_state_t *state, uint64_t spa, int64_t bytes,
2643     arc_buf_contents_t type)
2644 {
2645        uint64_t total_evicted = 0;
2646        multilist_t *ml = &state->arcs_list[type];
2647        int num_sublists;
2648        arc_buf_hdr_t **markers;

2650        IMPLY(bytes < 0, bytes == ARC_EVICT_ALL);

2652        num_sublists = multilist_get_num_sublists(ml);

2654        /*
2655         * If we've tried to evict from each sublist, made some
2656         * progress, but still have not hit the target number of bytes
2657         * to evict, we want to keep trying. The markers allow us to
2658         * pick up where we left off for each individual sublist, rather
2659         * than starting from the tail each time.
2660         */
2661        markers = kmem_zalloc(sizeof (*markers) * num_sublists, KM_SLEEP);
2662        for (int i = 0; i < num_sublists; i++) {
2663                markers[i] = kmem_cache_alloc(hdr_full_cache, KM_SLEEP);

2665                /*
2666                 * A b_spa of 0 is used to indicate that this header is
2667                 * a marker. This fact is used in arc_adjust_type() and
2668                 * arc_evict_state_impl().
2669                 */
2670                markers[i]->b_spa = 0;

2672                multilist_sublist_t *mls = multilist_sublist_lock(ml, i);
2673                multilist_sublist_insert_tail(mls, markers[i]);
2674                multilist_sublist_unlock(mls);
2675        }

2677        /*
2678         * While we haven't hit our target number of bytes to evict, or
2679         * we're evicting all available buffers.
2680         */
2681        while (total_evicted < bytes || bytes == ARC_EVICT_ALL) {
2682                /*
2683                 * Start eviction using a randomly selected sublist,
2684                 * this is to try and evenly balance eviction across all
2685                 * sublists. Always starting at the same sublist
2686                 * (e.g. index 0) would cause evictions to favor certain
2687                 * sublists over others.
2688                 */
2689                int sublist_idx = multilist_get_random_index(ml);
2690                uint64_t scan_evicted = 0;
```

```
2692                 for (int i = 0; i < num_sublists; i++) {
2693                         uint64_t bytes_remaining;
2694                         uint64_t bytes_evicted;
2695
2696                         if (bytes == ARC_EVICT_ALL)
2697                                 bytes_remaining = ARC_EVICT_ALL;
2698                         else if (total_evicted < bytes)
2699                                 bytes_remaining = bytes - total_evicted;
2700                         else
2701                                 break;
2702
2703                         bytes_evicted = arc_evict_state_impl(ml, sublist_idx,
2704                             markers[sublist_idx], spa, bytes_remaining);
2705
2706                         scan_evicted += bytes_evicted;
2707                         total_evicted += bytes_evicted;
2708
2709                         /* we've reached the end, wrap to the beginning */
2710                         if (++sublist_idx >= num_sublists)
2711                                 sublist_idx = 0;
2712                 }
2713
2714                 /*
2715                  * If we didn't evict anything during this scan, we have
2716                  * no reason to believe we'll evict more during another
2717                  * scan, so break the loop.
2718                  */
2719                 if (scan_evicted == 0) {
2720                         /* This isn't possible, let's make that obvious */
2721                         ASSERT3S(bytes, !=, 0);
2722
2723                         /*
2724                          * When bytes is ARC_EVICT_ALL, the only way to
2725                          * break the loop is when scan_evicted is zero.
2726                          * In that case, we actually have evicted enough,
2727                          * so we don't want to increment the kstat.
2728                          */
2729                         if (bytes != ARC_EVICT_ALL) {
2730                                 ASSERT3S(total_evicted, <, bytes);
2731                                 ARCSTAT_BUMP(arcstat_evict_not_enough);
2732                         }
2733
2734                         break;
2735                 }
2736         }
2737
2738         for (int i = 0; i < num_sublists; i++) {
2739                 multilist_sublist_t *mls = multilist_sublist_lock(ml, i);
2740                 multilist_sublist_remove(mls, markers[i]);
2741                 multilist_sublist_unlock(mls);
2742
2743                 kmem_cache_free(hdr_full_cache, markers[i]);
2744         }
2745         kmem_free(markers, sizeof (*markers) * num_sublists);
2746
2747         return (total_evicted);
2748 }
2749
2750 /*
2751  * Flush all "evictable" data of the given type from the arc state
2752  * specified. This will not evict any "active" buffers (i.e. referenced).
2753  *
2754  * When 'retry' is set to FALSE, the function will make a single pass
2755  * over the state and evict any buffers that it can. Since it doesn't
2756  * continually retry the eviction, it might end up leaving some buffers
```



```
2757  * in the ARC due to lock misses.
2758  *
2759  * When 'retry' is set to TRUE, the function will continually retry the
2760  * eviction until *all* evictable buffers have been removed from the
2761  * state. As a result, if concurrent insertions into the state are
2762  * allowed (e.g. if the ARC isn't shutting down), this function might
2763  * wind up in an infinite loop, continually trying to evict buffers.
2764  */
2765 static uint64_t
2766 arc_flush_state(arc_state_t *state, uint64_t spa, arc_buf_contents_t type,
2767     boolean_t retry)
2768 {
2769         uint64_t evicted = 0;
2770
2771         while (state->arcs_lsize[type] != 0) {
2772                 evicted += arc_evict_state(state, spa, ARC_EVICT_ALL, type);
2773
2774                 if (!retry)
2775                         break;
2776         }
2777
2778         return (evicted);
2779 }
2780
2781 /*
2782  * Evict the specified number of bytes from the state specified,
2783  * restricting eviction to the spa and type given. This function
2784  * prevents us from trying to evict more from a state's list than
2785  * is "evictable", and to skip evicting altogether when passed a
2786  * negative value for "bytes". In contrast, arc_evict_state() will
2787  * evict everything it can, when passed a negative value for "bytes".
2788  */
2789 static uint64_t
2790 arc_adjust_impl(arc_state_t *state, uint64_t spa, int64_t bytes,
2791     arc_buf_contents_t type)
2792 {
2793         int64_t delta;
2794
2795         if (bytes > 0 && state->arcs_lsize[type] > 0) {
2796                 delta = MIN(state->arcs_lsize[type], bytes);
2797                 return (arc_evict_state(state, spa, delta, type));
2798         }
2799
2800         return (0);
2801 }
2802
2803 /*
2804  * Evict metadata buffers from the cache, such that arc_meta_used is
2805  * capped by the arc_meta_limit tunable.
2806  */
2807 static uint64_t
2808 arc_adjust_meta(void)
2809 {
2810         uint64_t total_evicted = 0;
2811         int64_t target;
2812
2813         /*
2814          * If we're over the meta limit, we want to evict enough
2815          * metadata to get back under the meta limit. We don't want to
2816          * evict so much that we drop the MRU below arc_p, though. If
2817          * we're over the meta limit more than we're over arc_p, we
2818          * evict some from the MRU here, and some from the MFU below.
2819          */
2820         target = MIN((int64_t)(arc_meta_used - arc_meta_limit),
2821             (int64_t)(refcount_count(&arc_anon->arcs_size) +
2822             refcount_count(&arc_mru->arcs_size) - arc_p));
```

```
2824         total_evicted += arc_adjust_impl(arc_mru, 0, target, ARC_BUFC_METADATA);

2826         /*
2827          * Similar to the above, we want to evict enough bytes to get us
2828          * below the meta limit, but not so much as to drop us below the
2829          * space alloted to the MFU (which is defined as arc_c - arc_p).
2830          */
2831         target = MIN((int64_t)(arc_meta_used - arc_meta_limit),
2832             (int64_t)(refcount_count(&arc_mfu->arcs_size) - (arc_c - arc_p)));

2834         total_evicted += arc_adjust_impl(arc_mfu, 0, target, ARC_BUFC_METADATA);

2836         return (total_evicted);
2837 }

2839 /*
2840  * Return the type of the oldest buffer in the given arc state
2841  *
2842  * This function will select a random sublist of type ARC_BUFC_DATA and
2843  * a random sublist of type ARC_BUFC_METADATA. The tail of each sublist
2844  * is compared, and the type which contains the "older" buffer will be
2845  * returned.
2846  */
2847 static arc_buf_contents_t
2848 arc_adjust_type(arc_state_t *state)
2849 {
2850         multilist_t *data_ml = &state->arcs_list[ARC_BUFC_DATA];
2851         multilist_t *meta_ml = &state->arcs_list[ARC_BUFC_METADATA];
2852         int data_idx = multilist_get_random_index(data_ml);
2853         int meta_idx = multilist_get_random_index(meta_ml);
2854         multilist_sublist_t *data_mls;
2855         multilist_sublist_t *meta_mls;
2856         arc_buf_contents_t type;
2857         arc_buf_hdr_t *data_hdr;
2858         arc_buf_hdr_t *meta_hdr;

2860         /*
2861          * We keep the sublist lock until we're finished, to prevent
2862          * the headers from being destroyed via arc_evict_state().
2863          */
2864         data_mls = multilist_sublist_lock(data_ml, data_idx);
2865         meta_mls = multilist_sublist_lock(meta_ml, meta_idx);

2867         /*
2868          * These two loops are to ensure we skip any markers that
2869          * might be at the tail of the lists due to arc_evict_state().
2870          */

2872         for (data_hdr = multilist_sublist_tail(data_mls); data_hdr != NULL;
2873             data_hdr = multilist_sublist_prev(data_mls, data_hdr)) {
2874                 if (data_hdr->b_spa != 0)
2875                         break;
2876         }

2878         for (meta_hdr = multilist_sublist_tail(meta_mls); meta_hdr != NULL;
2879             meta_hdr = multilist_sublist_prev(meta_mls, meta_hdr)) {
2880                 if (meta_hdr->b_spa != 0)
2881                         break;
2882         }

2884         if (data_hdr == NULL && meta_hdr == NULL) {
2885                 type = ARC_BUFC_DATA;
2886         } else if (data_hdr == NULL) {
2887                 ASSERT3P(meta_hdr, !=, NULL);
2888                 type = ARC_BUFC_METADATA;
```



```
2889         } else if (meta_hdr == NULL) {
2890                 ASSERT3P(data_hdr, !=, NULL);
2891                 type = ARC_BUFC_DATA;
2892         } else {
2893                 ASSERT3P(data_hdr, !=, NULL);
2894                 ASSERT3P(meta_hdr, !=, NULL);

2896                 /* The headers can't be on the sublist without an L1 header */
2897                 ASSERT(HDR_HAS_L1HDR(data_hdr));
2898                 ASSERT(HDR_HAS_L1HDR(meta_hdr));

2900                 if (data_hdr->b_l1hdr.b_arc_access <
2901                     meta_hdr->b_l1hdr.b_arc_access) {
2902                         type = ARC_BUFC_DATA;
2903                 } else {
2904                         type = ARC_BUFC_METADATA;
2905                 }
2906         }

2908         multilist_sublist_unlock(meta_mls);
2909         multilist_sublist_unlock(data_mls);

2911         return (type);
2912 }

2914 /*
2915  * Evict buffers from the cache, such that arc_size is capped by arc_c.
2916  */
2917 static uint64_t
2918 arc_adjust(void)
2919 {
2920         uint64_t total_evicted = 0;
2921         uint64_t bytes;
2922         int64_t target;

2924         /*
2925          * If we're over arc_meta_limit, we want to correct that before
2926          * potentially evicting data buffers below.
2927          */
2928         total_evicted += arc_adjust_meta();

2930         /*
2931          * Adjust MRU size
2932          *
2933          * If we're over the target cache size, we want to evict enough
2934          * from the list to get back to our target size. We don't want
2935          * to evict too much from the MRU, such that it drops below
2936          * arc_p. So, if we're over our target cache size more than
2937          * the MRU is over arc_p, we'll evict enough to get back to
2938          * arc_p here, and then evict more from the MFU below.
2939          */
2940         target = MIN((int64_t)(arc_size - arc_c),
2941             (int64_t)(refcount_count(&arc_anon->arcs_size) +
2942             refcount_count(&arc_mru->arcs_size) + arc_meta_used - arc_p));

2944         /*
2945          * If we're below arc_meta_min, always prefer to evict data.
2946          * Otherwise, try to satisfy the requested number of bytes to
2947          * evict from the type which contains older buffers; in an
2948          * effort to keep newer buffers in the cache regardless of their
2949          * type. If we cannot satisfy the number of bytes from this
2950          * type, spill over into the next type.
2951          */
2952         if (arc_adjust_type(arc_mru) == ARC_BUFC_METADATA &&
2953             arc_meta_used > arc_meta_min) {
2954                 bytes = arc_adjust_impl(arc_mru, 0, target, ARC_BUFC_METADATA);
```

```
2955                 total_evicted += bytes;
2956
2957                 /*
2958                  * If we couldn't evict our target number of bytes from
2959                  * metadata, we try to get the rest from data.
2960                  */
2961                 target -= bytes;
2962
2963                 total_evicted +=
2964                     arc_adjust_impl(arc_mru, 0, target, ARC_BUFC_DATA);
2965         } else {
2966                 bytes = arc_adjust_impl(arc_mru, 0, target, ARC_BUFC_DATA);
2967                 total_evicted += bytes;
2968
2969                 /*
2970                  * If we couldn't evict our target number of bytes from
2971                  * data, we try to get the rest from metadata.
2972                  */
2973                 target -= bytes;
2974
2975                 total_evicted +=
2976                     arc_adjust_impl(arc_mru, 0, target, ARC_BUFC_METADATA);
2977         }
2978
2979         /*
2980          * Adjust MFU size
2981          *
2982          * Now that we've tried to evict enough from the MRU to get its
2983          * size back to arc_p, if we're still above the target cache
2984          * size, we evict the rest from the MFU.
2985          */
2986         target = arc_size - arc_c;
2987
2988         if (arc_adjust_type(arc_mfu) == ARC_BUFC_METADATA &&
2989             arc_meta_used > arc_meta_min) {
2990                 bytes = arc_adjust_impl(arc_mfu, 0, target, ARC_BUFC_METADATA);
2991                 total_evicted += bytes;
2992
2993                 /*
2994                  * If we couldn't evict our target number of bytes from
2995                  * metadata, we try to get the rest from data.
2996                  */
2997                 target -= bytes;
2998
2999                 total_evicted +=
3000                     arc_adjust_impl(arc_mfu, 0, target, ARC_BUFC_DATA);
3001         } else {
3002                 bytes = arc_adjust_impl(arc_mfu, 0, target, ARC_BUFC_DATA);
3003                 total_evicted += bytes;
3004
3005                 /*
3006                  * If we couldn't evict our target number of bytes from
3007                  * data, we try to get the rest from data.
3008                  */
3009                 target -= bytes;
3010
3011                 total_evicted +=
3012                     arc_adjust_impl(arc_mfu, 0, target, ARC_BUFC_METADATA);
3013         }
3014
3015         /*
3016          * Adjust ghost lists
3017          *
3018          * In addition to the above, the ARC also defines target values
3019          * for the ghost lists. The sum of the mru list and mru ghost
3020          * list should never exceed the target size of the cache, and
```



```
3021          * the sum of the mru list, mfu list, mru ghost list, and mfu
3022          * ghost list should never exceed twice the target size of the
3023          * cache. The following logic enforces these limits on the ghost
3024          * caches, and evicts from them as needed.
3025          */
3026         target = refcount_count(&arc_mru->arcs_size) +
3027             refcount_count(&arc_mru_ghost->arcs_size) - arc_c;
3028
3029         bytes = arc_adjust_impl(arc_mru_ghost, 0, target, ARC_BUFC_DATA);
3030         total_evicted += bytes;
3031
3032         target -= bytes;
3033
3034         total_evicted +=
3035             arc_adjust_impl(arc_mru_ghost, 0, target, ARC_BUFC_METADATA);
3036
3037         /*
3038          * We assume the sum of the mru list and mfu list is less than
3039          * or equal to arc_c (we enforced this above), which means we
3040          * can use the simpler of the two equations below:
3041          *
3042          *      mru + mfu + mru ghost + mfu ghost <= 2 * arc_c
3043          *                  mru ghost + mfu ghost <= arc_c
3044          */
3045         target = refcount_count(&arc_mru_ghost->arcs_size) +
3046             refcount_count(&arc_mfu_ghost->arcs_size) - arc_c;
3047
3048         bytes = arc_adjust_impl(arc_mfu_ghost, 0, target, ARC_BUFC_DATA);
3049         total_evicted += bytes;
3050
3051         target -= bytes;
3052
3053         total_evicted +=
3054             arc_adjust_impl(arc_mfu_ghost, 0, target, ARC_BUFC_METADATA);
3055
3056         return (total_evicted);
3057 }
3058
3059 static void
3060 arc_do_user_evicts(void)
3061 {
3062         mutex_enter(&arc_user_evicts_lock);
3063         while (arc_eviction_list != NULL) {
3064                 arc_buf_t *buf = arc_eviction_list;
3065                 arc_eviction_list = buf->b_next;
3066                 mutex_enter(&buf->b_evict_lock);
3067                 buf->b_hdr = NULL;
3068                 mutex_exit(&buf->b_evict_lock);
3069                 mutex_exit(&arc_user_evicts_lock);
3070
3071                 if (buf->b_efunc != NULL)
3072                         VERIFY0(buf->b_efunc(buf->b_private));
3073
3074                 buf->b_efunc = NULL;
3075                 buf->b_private = NULL;
3076                 kmem_cache_free(buf_cache, buf);
3077                 mutex_enter(&arc_user_evicts_lock);
3078         }
3079         mutex_exit(&arc_user_evicts_lock);
3080 }
3081
3082 void
3083 arc_flush(spa_t *spa, boolean_t retry)
3084 {
3085         uint64_t guid = 0;
```

```
3087         /*
3088          * If retry is TRUE, a spa must not be specified since we have
3089          * no good way to determine if all of a spa's buffers have been
3090          * evicted from an arc state.
3091          */
3092         ASSERT(!retry || spa == 0);

3094         if (spa != NULL)
3095                 guid = spa_load_guid(spa);

3097         (void) arc_flush_state(arc_mru, guid, ARC_BUFC_DATA, retry);
3098         (void) arc_flush_state(arc_mru, guid, ARC_BUFC_METADATA, retry);

3100         (void) arc_flush_state(arc_mfu, guid, ARC_BUFC_DATA, retry);
3101         (void) arc_flush_state(arc_mfu, guid, ARC_BUFC_METADATA, retry);

3103         (void) arc_flush_state(arc_mru_ghost, guid, ARC_BUFC_DATA, retry);
3104         (void) arc_flush_state(arc_mru_ghost, guid, ARC_BUFC_METADATA, retry);

3106         (void) arc_flush_state(arc_mfu_ghost, guid, ARC_BUFC_DATA, retry);
3107         (void) arc_flush_state(arc_mfu_ghost, guid, ARC_BUFC_METADATA, retry);

3109         arc_do_user_evicts();
3110         ASSERT(spa || arc_eviction_list == NULL);
3111 }

3113 void
3114 arc_shrink(int64_t to_free)
3115 {
3116         if (arc_c > arc_c_min) {

3118                 if (arc_c > arc_c_min + to_free)
3119                         atomic_add_64(&arc_c, -to_free);
3120                 else
3121                         arc_c = arc_c_min;

3123                 atomic_add_64(&arc_p, -(arc_p >> arc_shrink_shift));
3124                 if (arc_c > arc_size)
3125                         arc_c = MAX(arc_size, arc_c_min);
3126                 if (arc_p > arc_c)
3127                         arc_p = (arc_c >> 1);
3128                 ASSERT(arc_c >= arc_c_min);
3129                 ASSERT((int64_t)arc_p >= 0);
3130         }

3132         if (arc_size > arc_c)
3133                 (void) arc_adjust();
3134 }

3136 typedef enum free_memory_reason_t {
3137         FMR_UNKNOWN,
3138         FMR_NEEDFREE,
3139         FMR_LOTSFREE,
3140         FMR_SWAPFS_MINFREE,
3141         FMR_PAGES_PP_MAXIMUM,
3142         FMR_HEAP_ARENA,
3143         FMR_ZIO_ARENA,
3144 } free_memory_reason_t;

3146 int64_t last_free_memory;
3147 free_memory_reason_t last_free_reason;

3149 /*
3150  * Additional reserve of pages for pp_reserve.
3151  */
3152 int64_t arc_pages_pp_reserve = 64;
```



```
3154 /*
3155  * Additional reserve of pages for swapfs.
3156  */
3157 int64_t arc_swapfs_reserve = 64;

3159 /*
3160  * Return the amount of memory that can be consumed before reclaim will be
3161  * needed.  Positive if there is sufficient free memory, negative indicates
3162  * the amount of memory that needs to be freed up.
3163  */
3164 static int64_t
3165 arc_available_memory(void)
3166 {
3167         int64_t lowest = INT64_MAX;
3168         int64_t n;
3169         free_memory_reason_t r = FMR_UNKNOWN;

3171 #ifdef _KERNEL
3172         if (needfree > 0) {
3173                 n = PAGESIZE * (-needfree);
3174                 if (n < lowest) {
3175                         lowest = n;
3176                         r = FMR_NEEDFREE;
3177                 }
3178         }

3180         /*
3181          * check that we're out of range of the pageout scanner.  It starts to
3182          * schedule paging if freemem is less than lotsfree and needfree.
3183          * lotsfree is the high-water mark for pageout, and needfree is the
3184          * number of needed free pages.  We add extra pages here to make sure
3185          * the scanner doesn't start up while we're freeing memory.
3186          */
3187         n = PAGESIZE * (freemem - lotsfree - needfree - desfree);
3188         if (n < lowest) {
3189                 lowest = n;
3190                 r = FMR_LOTSFREE;
3191         }

3193         /*
3194          * check to make sure that swapfs has enough space so that anon
3195          * reservations can still succeed. anon_resvmem() checks that the
3196          * availrmem is greater than swapfs_minfree, and the number of reserved
3197          * swap pages.  We also add a bit of extra here just to prevent
3198          * circumstances from getting really dire.
3199          */
3200         n = PAGESIZE * (availrmem - swapfs_minfree - swapfs_reserve -
3201             desfree - arc_swapfs_reserve);
3202         if (n < lowest) {
3203                 lowest = n;
3204                 r = FMR_SWAPFS_MINFREE;
3205         }


3208         /*
3209          * Check that we have enough availrmem that memory locking (e.g., via
3210          * mlock(3C) or memcntl(2)) can still succeed.  (pages_pp_maximum
3211          * stores the number of pages that cannot be locked; when availrmem
3212          * drops below pages_pp_maximum, page locking mechanisms such as
3213          * page_pp_lock() will fail.)
3214          */
3215         n = PAGESIZE * (availrmem - pages_pp_maximum -
3216             arc_pages_pp_reserve);
3217         if (n < lowest) {
3218                 lowest = n;
```

```
3219                 r = FMR_PAGES_PP_MAXIMUM;
3220         }

3222 #if defined(__i386)
3223         /*
3224          * If we're on an i386 platform, it's possible that we'll exhaust the
3225          * kernel heap space before we ever run out of available physical
3226          * memory.  Most checks of the size of the heap_area compare against
3227          * tune.t_minarmem, which is the minimum available real memory that we
3228          * can have in the system.  However, this is generally fixed at 25 pages
3229          * which is so low that it's useless.  In this comparison, we seek to
3230          * calculate the total heap-size, and reclaim if more than 3/4ths of the
3231          * heap is allocated.  (Or, in the calculation, if less than 1/4th is
3232          * free)
3233          */
3234         n = vmem_size(heap_arena, VMEM_FREE) -
3235             (vmem_size(heap_arena, VMEM_FREE | VMEM_ALLOC) >> 2);
3236         if (n < lowest) {
3237                 lowest = n;
3238                 r = FMR_HEAP_ARENA;
3239         }
3240 #endif

3242         /*
3243          * If zio data pages are being allocated out of a separate heap segment,
3244          * then enforce that the size of available vmem for this arena remains
3245          * above about 1/16th free.
3246          *
3247          * Note: The 1/16th arena free requirement was put in place
3248          * to aggressively evict memory from the arc in order to avoid
3249          * memory fragmentation issues.
3250          */
3251         if (zio_arena != NULL) {
3252                 n = vmem_size(zio_arena, VMEM_FREE) -
3253                     (vmem_size(zio_arena, VMEM_ALLOC) >> 4);
3254                 if (n < lowest) {
3255                         lowest = n;
3256                         r = FMR_ZIO_ARENA;
3257                 }
3258         }
3259 #else
3260         /* Every 100 calls, free a small amount */
3261         if (spa_get_random(100) == 0)
3262                 lowest = -1024;
3263 #endif

3265         last_free_memory = lowest;
3266         last_free_reason = r;

3268         return (lowest);
3269 }


3272 /*
3273  * Determine if the system is under memory pressure and is asking
3274  * to reclaim memory. A return value of TRUE indicates that the system
3275  * is under memory pressure and that the arc should adjust accordingly.
3276  */
3277 static boolean_t
3278 arc_reclaim_needed(void)
3279 {
3280         return (arc_available_memory() < 0);
3281 }

3283 static void
3284 arc_kmem_reap_now(void)
```



```
3285 {
3286         size_t                  i;
3287         kmem_cache_t            *prev_cache = NULL;
3288         kmem_cache_t            *prev_data_cache = NULL;
3289         extern kmem_cache_t     *zio_buf_cache[];
3290         extern kmem_cache_t     *zio_data_buf_cache[];
3291         extern kmem_cache_t     *range_seg_cache;

3293 #ifdef _KERNEL
3294         if (arc_meta_used >= arc_meta_limit) {
3295                 /*
3296                  * We are exceeding our meta-data cache limit.
3297                  * Purge some DNLC entries to release holds on meta-data.
3298                  */
3299                 dnlc_reduce_cache((void *)(uintptr_t)arc_reduce_dnlc_percent);
3300         }
3301 #if defined(__i386)
3302         /*
3303          * Reclaim unused memory from all kmem caches.
3304          */
3305         kmem_reap();
3306 #endif
3307 #endif

3309         for (i = 0; i < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT; i++) {
3310                 if (zio_buf_cache[i] != prev_cache) {
3311                         prev_cache = zio_buf_cache[i];
3312                         kmem_cache_reap_now(zio_buf_cache[i]);
3313                 }
3314                 if (zio_data_buf_cache[i] != prev_data_cache) {
3315                         prev_data_cache = zio_data_buf_cache[i];
3316                         kmem_cache_reap_now(zio_data_buf_cache[i]);
3317                 }
3318         }
3319         kmem_cache_reap_now(buf_cache);
3320         kmem_cache_reap_now(hdr_full_cache);
3321         kmem_cache_reap_now(hdr_l2only_cache);
3322         kmem_cache_reap_now(range_seg_cache);

3324         if (zio_arena != NULL) {
3325                 /*
3326                  * Ask the vmem arena to reclaim unused memory from its
3327                  * quantum caches.
3328                  */
3329                 vmem_qcache_reap(zio_arena);
3330         }
3331 }

3333 /*
3334  * Threads can block in arc_get_data_buf() waiting for this thread to evict
3335  * enough data and signal them to proceed. When this happens, the threads in
3336  * arc_get_data_buf() are sleeping while holding the hash lock for their
3337  * particular arc header. Thus, we must be careful to never sleep on a
3338  * hash lock in this thread. This is to prevent the following deadlock:
3339  *
3340  *  - Thread A sleeps on CV in arc_get_data_buf() holding hash lock "L",
3341  *    waiting for the reclaim thread to signal it.
3342  *
3343  *  - arc_reclaim_thread() tries to acquire hash lock "L" using mutex_enter,
3344  *    fails, and goes to sleep forever.
3345  *
3346  * This possible deadlock is avoided by always acquiring a hash lock
3347  * using mutex_tryenter() from arc_reclaim_thread().
3348  */
3349 static void
3350 arc_reclaim_thread(void)
```

```
3351 {
3352         clock_t                 growtime = 0;
3353         callb_cpr_t             cpr;

3355         CALLB_CPR_INIT(&cpr, &arc_reclaim_lock, callb_generic_cpr, FTAG);

3357         mutex_enter(&arc_reclaim_lock);
3358         while (!arc_reclaim_thread_exit) {
3359                 int64_t free_memory = arc_available_memory();
3360                 uint64_t evicted = 0;

3362                 mutex_exit(&arc_reclaim_lock);

3364                 if (free_memory < 0) {

3366                         arc_no_grow = B_TRUE;
3367                         arc_warm = B_TRUE;

3369                         /*
3370                          * Wait at least zfs_grow_retry (default 60) seconds
3371                          * before considering growing.
3372                          */
3373                         growtime = ddi_get_lbolt() + (arc_grow_retry * hz);

3375                         arc_kmem_reap_now();

3377                         /*
3378                          * If we are still low on memory, shrink the ARC
3379                          * so that we have arc_shrink_min free space.
3380                          */
3381                         free_memory = arc_available_memory();

3383                         int64_t to_free =
3384                             (arc_c >> arc_shrink_shift) - free_memory;
3385                         if (to_free > 0) {
3386 #ifdef _KERNEL
3387                                 to_free = MAX(to_free, ptob(needfree));
3388 #endif
3389                                 arc_shrink(to_free);
3390                         }
3391                 } else if (free_memory < arc_c >> arc_no_grow_shift) {
3392                         arc_no_grow = B_TRUE;
3393                 } else if (ddi_get_lbolt() >= growtime) {
3394                         arc_no_grow = B_FALSE;
3395                 }

3397                 evicted = arc_adjust();

3399                 mutex_enter(&arc_reclaim_lock);

3401                 /*
3402                  * If evicted is zero, we couldn't evict anything via
3403                  * arc_adjust(). This could be due to hash lock
3404                  * collisions, but more likely due to the majority of
3405                  * arc buffers being unevictable. Therefore, even if
3406                  * arc_size is above arc_c, another pass is unlikely to
3407                  * be helpful and could potentially cause us to enter an
3408                  * infinite loop.
3409                  */
3410                 if (arc_size <= arc_c || evicted == 0) {
3411                         /*
3412                          * We're either no longer overflowing, or we
3413                          * can't evict anything more, so we should wake
3414                          * up any threads before we go to sleep.
3415                          */
3416                         cv_broadcast(&arc_reclaim_waiters_cv);
```



```
3418                         /*
3419                          * Block until signaled, or after one second (we
3420                          * might need to perform arc_kmem_reap_now()
3421                          * even if we aren't being signalled)
3422                          */
3423                         CALLB_CPR_SAFE_BEGIN(&cpr);
3424                         (void) cv_timedwait(&arc_reclaim_thread_cv,
3425                             &arc_reclaim_lock, ddi_get_lbolt() + hz);
3426                         CALLB_CPR_SAFE_END(&cpr, &arc_reclaim_lock);
3427                 }
3428         }

3430         arc_reclaim_thread_exit = FALSE;
3431         cv_broadcast(&arc_reclaim_thread_cv);
3432         CALLB_CPR_EXIT(&cpr);           /* drops arc_reclaim_lock */
3433         thread_exit();
3434 }

3436 static void
3437 arc_user_evicts_thread(void)
3438 {
3439         callb_cpr_t cpr;

3441         CALLB_CPR_INIT(&cpr, &arc_user_evicts_lock, callb_generic_cpr, FTAG);

3443         mutex_enter(&arc_user_evicts_lock);
3444         while (!arc_user_evicts_thread_exit) {
3445                 mutex_exit(&arc_user_evicts_lock);

3447                 arc_do_user_evicts();

3449                 /*
3450                  * This is necessary in order for the mdb ::arc dcmd to
3451                  * show up to date information. Since the ::arc command
3452                  * does not call the kstat's update function, without
3453                  * this call, the command may show stale stats for the
3454                  * anon, mru, mru_ghost, mfu, and mfu_ghost lists. Even
3455                  * with this change, the data might be up to 1 second
3456                  * out of date; but that should suffice. The arc_state_t
3457                  * structures can be queried directly if more accurate
3458                  * information is needed.
3459                  */
3460                 if (arc_ksp != NULL)
3461                         arc_ksp->ks_update(arc_ksp, KSTAT_READ);

3463                 mutex_enter(&arc_user_evicts_lock);

3465                 /*
3466                  * Block until signaled, or after one second (we need to
3467                  * call the arc's kstat update function regularly).
3468                  */
3469                 CALLB_CPR_SAFE_BEGIN(&cpr);
3470                 (void) cv_timedwait(&arc_user_evicts_cv,
3471                     &arc_user_evicts_lock, ddi_get_lbolt() + hz);
3472                 CALLB_CPR_SAFE_END(&cpr, &arc_user_evicts_lock);
3473         }

3475         arc_user_evicts_thread_exit = FALSE;
3476         cv_broadcast(&arc_user_evicts_cv);
3477         CALLB_CPR_EXIT(&cpr);           /* drops arc_user_evicts_lock */
3478         thread_exit();
3479 }

3481 /*
3482  * Adapt arc info given the number of bytes we are trying to add and
```

```
3483  * the state that we are comming from.  This function is only called
3484  * when we are adding new content to the cache.
3485  */
3486 static void
3487 arc_adapt(int bytes, arc_state_t *state)
3488 {
3489         int mult;
3490         uint64_t arc_p_min = (arc_c >> arc_p_min_shift);
3491         int64_t mrug_size = refcount_count(&arc_mru_ghost->arcs_size);
3492         int64_t mfug_size = refcount_count(&arc_mfu_ghost->arcs_size);
3493
3494         if (state == arc_l2c_only)
3495                 return;
3496
3497         ASSERT(bytes > 0);
3498         /*
3499          * Adapt the target size of the MRU list:
3500          *      - if we just hit in the MRU ghost list, then increase
3501          *        the target size of the MRU list.
3502          *      - if we just hit in the MFU ghost list, then increase
3503          *        the target size of the MFU list by decreasing the
3504          *        target size of the MRU list.
3505          */
3506         if (state == arc_mru_ghost) {
3507                 mult = (mrug_size >= mfug_size) ? 1 : (mfug_size / mrug_size);
3508                 mult = MIN(mult, 10); /* avoid wild arc_p adjustment */
3509
3510                 arc_p = MIN(arc_c - arc_p_min, arc_p + bytes * mult);
3511         } else if (state == arc_mfu_ghost) {
3512                 uint64_t delta;
3513
3514                 mult = (mfug_size >= mrug_size) ? 1 : (mrug_size / mfug_size);
3515                 mult = MIN(mult, 10);
3516
3517                 delta = MIN(bytes * mult, arc_p);
3518                 arc_p = MAX(arc_p_min, arc_p - delta);
3519         }
3520         ASSERT((int64_t)arc_p >= 0);
3521
3522         if (arc_reclaim_needed()) {
3523                 cv_signal(&arc_reclaim_thread_cv);
3524                 return;
3525         }
3526
3527         if (arc_no_grow)
3528                 return;
3529
3530         if (arc_c >= arc_c_max)
3531                 return;
3532
3533         /*
3534          * If we're within (2 * maxblocksize) bytes of the target
3535          * cache size, increment the target cache size
3536          */
3537         if (arc_size > arc_c - (2ULL << SPA_MAXBLOCKSHIFT)) {
3538                 atomic_add_64(&arc_c, (int64_t)bytes);
3539                 if (arc_c > arc_c_max)
3540                         arc_c = arc_c_max;
3541                 else if (state == arc_anon)
3542                         atomic_add_64(&arc_p, (int64_t)bytes);
3543                 if (arc_p > arc_c)
3544                         arc_p = arc_c;
3545         }
3546         ASSERT((int64_t)arc_p >= 0);
3547 }
```



```
3549 /*
3550  * Check if arc_size has grown past our upper threshold, determined by
3551  * zfs_arc_overflow_shift.
3552  */
3553 static boolean_t
3554 arc_is_overflowing(void)
3555 {
3556         /* Always allow at least one block of overflow */
3557         uint64_t overflow = MAX(SPA_MAXBLOCKSIZE,
3558             arc_c >> zfs_arc_overflow_shift);
3559
3560         return (arc_size >= arc_c + overflow);
3561 }
3562
3563 /*
3564  * The buffer, supplied as the first argument, needs a data block. If we
3565  * are hitting the hard limit for the cache size, we must sleep, waiting
3566  * for the eviction thread to catch up. If we're past the target size
3567  * but below the hard limit, we'll only signal the reclaim thread and
3568  * continue on.
3569  */
3570 static void
3571 arc_get_data_buf(arc_buf_t *buf)
3572 {
3573         arc_state_t             *state = buf->b_hdr->b_l1hdr.b_state;
3574         uint64_t                size = buf->b_hdr->b_size;
3575         arc_buf_contents_t      type = arc_buf_type(buf->b_hdr);
3576
3577         arc_adapt(size, state);
3578
3579         /*
3580          * If arc_size is currently overflowing, and has grown past our
3581          * upper limit, we must be adding data faster than the evict
3582          * thread can evict. Thus, to ensure we don't compound the
3583          * problem by adding more data and forcing arc_size to grow even
3584          * further past it's target size, we halt and wait for the
3585          * eviction thread to catch up.
3586          *
3587          * It's also possible that the reclaim thread is unable to evict
3588          * enough buffers to get arc_size below the overflow limit (e.g.
3589          * due to buffers being un-evictable, or hash lock collisions).
3590          * In this case, we want to proceed regardless if we're
3591          * overflowing; thus we don't use a while loop here.
3592          */
3593         if (arc_is_overflowing()) {
3594                 mutex_enter(&arc_reclaim_lock);
3595
3596                 /*
3597                  * Now that we've acquired the lock, we may no longer be
3598                  * over the overflow limit, lets check.
3599                  *
3600                  * We're ignoring the case of spurious wake ups. If that
3601                  * were to happen, it'd let this thread consume an ARC
3602                  * buffer before it should have (i.e. before we're under
3603                  * the overflow limit and were signalled by the reclaim
3604                  * thread). As long as that is a rare occurrence, it
3605                  * shouldn't cause any harm.
3606                  */
3607                 if (arc_is_overflowing()) {
3608                         cv_signal(&arc_reclaim_thread_cv);
3609                         cv_wait(&arc_reclaim_waiters_cv, &arc_reclaim_lock);
3610                 }
3611
3612                 mutex_exit(&arc_reclaim_lock);
3613         }
```

```
3615         if (type == ARC_BUFC_METADATA) {
3616                 buf->b_data = zio_buf_alloc(size);
3617                 arc_space_consume(size, ARC_SPACE_META);
3618         } else {
3619                 ASSERT(type == ARC_BUFC_DATA);
3620                 buf->b_data = zio_data_buf_alloc(size);
3621                 arc_space_consume(size, ARC_SPACE_DATA);
3622         }

3624         /*
3625          * Update the state size.  Note that ghost states have a
3626          * "ghost size" and so don't need to be updated.
3627          */
3628         if (!GHOST_STATE(buf->b_hdr->b_l1hdr.b_state)) {
3629                 arc_buf_hdr_t *hdr = buf->b_hdr;
3630                 arc_state_t *state = hdr->b_l1hdr.b_state;

3632                 (void) refcount_add_many(&state->arcs_size, size, buf);

3634                 /*
3635                  * If this is reached via arc_read, the link is
3636                  * protected by the hash lock. If reached via
3637                  * arc_buf_alloc, the header should not be accessed by
3638                  * any other thread. And, if reached via arc_read_done,
3639                  * the hash lock will protect it if it's found in the
3640                  * hash table; otherwise no other thread should be
3641                  * trying to [add|remove]_reference it.
3642                  */
3643                 if (multilist_link_active(&hdr->b_l1hdr.b_arc_node)) {
3644                         ASSERT(refcount_is_zero(&hdr->b_l1hdr.b_refcnt));
3645                         atomic_add_64(&hdr->b_l1hdr.b_state->arcs_lsize[type],
3646                             size);
3647                 }
3648                 /*
3649                  * If we are growing the cache, and we are adding anonymous
3650                  * data, and we have outgrown arc_p, update arc_p
3651                  */
3652                 if (arc_size < arc_c && hdr->b_l1hdr.b_state == arc_anon &&
3653                     (refcount_count(&arc_anon->arcs_size) +
3654                     refcount_count(&arc_mru->arcs_size) > arc_p))
3655                         arc_p = MIN(arc_c, arc_p + size);
3656         }
3657 }

3659 /*
3660  * This routine is called whenever a buffer is accessed.
3661  * NOTE: the hash lock is dropped in this function.
3662  */
3663 static void
3664 arc_access(arc_buf_hdr_t *hdr, kmutex_t *hash_lock)
3665 {
3666         clock_t now;

3668         ASSERT(MUTEX_HELD(hash_lock));
3669         ASSERT(HDR_HAS_L1HDR(hdr));

3671         if (hdr->b_l1hdr.b_state == arc_anon) {
3672                 /*
3673                  * This buffer is not in the cache, and does not
3674                  * appear in our "ghost" list.  Add the new buffer
3675                  * to the MRU state.
3676                  */

3678                 ASSERT0(hdr->b_l1hdr.b_arc_access);
3679                 hdr->b_l1hdr.b_arc_access = ddi_get_lbolt();
3680                 DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, hdr);
```



```
3681                 arc_change_state(arc_mru, hdr, hash_lock);

3683         } else if (hdr->b_l1hdr.b_state == arc_mru) {
3684                 now = ddi_get_lbolt();

3686                 /*
3687                  * If this buffer is here because of a prefetch, then either:
3688                  * - clear the flag if this is a "referencing" read
3689                  *   (any subsequent access will bump this into the MFU state).
3690                  * or
3691                  * - move the buffer to the head of the list if this is
3692                  *   another prefetch (to make it less likely to be evicted).
3693                  */
3694                 if (HDR_PREFETCH(hdr)) {
3695                         if (refcount_count(&hdr->b_l1hdr.b_refcnt) == 0) {
3696                                 /* link protected by hash lock */
3697                                 ASSERT(multilist_link_active(
3698                                     &hdr->b_l1hdr.b_arc_node));
3699                         } else {
3700                                 hdr->b_flags &= ~ARC_FLAG_PREFETCH;
3701                                 ARCSTAT_BUMP(arcstat_mru_hits);
3702                         }
3703                         hdr->b_l1hdr.b_arc_access = now;
3704                         return;
3705                 }

3707                 /*
3708                  * This buffer has been "accessed" only once so far,
3709                  * but it is still in the cache. Move it to the MFU
3710                  * state.
3711                  */
3712                 if (now > hdr->b_l1hdr.b_arc_access + ARC_MINTIME) {
3713                         /*
3714                          * More than 125ms have passed since we
3715                          * instantiated this buffer.  Move it to the
3716                          * most frequently used state.
3717                          */
3718                         hdr->b_l1hdr.b_arc_access = now;
3719                         DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, hdr);
3720                         arc_change_state(arc_mfu, hdr, hash_lock);
3721                 }
3722                 ARCSTAT_BUMP(arcstat_mru_hits);
3723         } else if (hdr->b_l1hdr.b_state == arc_mru_ghost) {
3724                 arc_state_t     *new_state;
3725                 /*
3726                  * This buffer has been "accessed" recently, but
3727                  * was evicted from the cache.  Move it to the
3728                  * MFU state.
3729                  */

3731                 if (HDR_PREFETCH(hdr)) {
3732                         new_state = arc_mru;
3733                         if (refcount_count(&hdr->b_l1hdr.b_refcnt) > 0)
3734                                 hdr->b_flags &= ~ARC_FLAG_PREFETCH;
3735                         DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, hdr);
3736                 } else {
3737                         new_state = arc_mfu;
3738                         DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, hdr);
3739                 }

3741                 hdr->b_l1hdr.b_arc_access = ddi_get_lbolt();
3742                 arc_change_state(new_state, hdr, hash_lock);

3744                 ARCSTAT_BUMP(arcstat_mru_ghost_hits);
3745         } else if (hdr->b_l1hdr.b_state == arc_mfu) {
3746                 /*
```

```
3747                  * This buffer has been accessed more than once and is
3748                  * still in the cache.  Keep it in the MFU state.
3749                  *
3750                  * NOTE: an add_reference() that occurred when we did
3751                  * the arc_read() will have kicked this off the list.
3752                  * If it was a prefetch, we will explicitly move it to
3753                  * the head of the list now.
3754                  */
3755                 if ((HDR_PREFETCH(hdr)) != 0) {
3756                         ASSERT(refcount_is_zero(&hdr->b_l1hdr.b_refcnt));
3757                         /* link protected by hash_lock */
3758                         ASSERT(multilist_link_active(&hdr->b_l1hdr.b_arc_node));
3759                 }
3760                 ARCSTAT_BUMP(arcstat_mfu_hits);
3761                 hdr->b_l1hdr.b_arc_access = ddi_get_lbolt();
3762         } else if (hdr->b_l1hdr.b_state == arc_mfu_ghost) {
3763                 arc_state_t     *new_state = arc_mfu;
3764                 /*
3765                  * This buffer has been accessed more than once but has
3766                  * been evicted from the cache.  Move it back to the
3767                  * MFU state.
3768                  */

3770                 if (HDR_PREFETCH(hdr)) {
3771                         /*
3772                          * This is a prefetch access...
3773                          * move this block back to the MRU state.
3774                          */
3775                         ASSERT0(refcount_count(&hdr->b_l1hdr.b_refcnt));
3776                         new_state = arc_mru;
3777                 }

3779                 hdr->b_l1hdr.b_arc_access = ddi_get_lbolt();
3780                 DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, hdr);
3781                 arc_change_state(new_state, hdr, hash_lock);

3783                 ARCSTAT_BUMP(arcstat_mfu_ghost_hits);
3784         } else if (hdr->b_l1hdr.b_state == arc_l2c_only) {
3785                 /*
3786                  * This buffer is on the 2nd Level ARC.
3787                  */

3789                 hdr->b_l1hdr.b_arc_access = ddi_get_lbolt();
3790                 DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, hdr);
3791                 arc_change_state(arc_mfu, hdr, hash_lock);
3792         } else {
3793                 ASSERT(!"invalid arc state");
3794         }
3795 }

3797 /* a generic arc_done_func_t which you can use */
3798 /* ARGSUSED */
3799 void
3800 arc_bcopy_func(zio_t *zio, arc_buf_t *buf, void *arg)
3801 {
3802         if (zio == NULL || zio->io_error == 0)
3803                 bcopy(buf->b_data, arg, buf->b_hdr->b_size);
3804         VERIFY(arc_buf_remove_ref(buf, arg));
3805 }

3807 /* a generic arc_done_func_t */
3808 void
3809 arc_getbuf_func(zio_t *zio, arc_buf_t *buf, void *arg)
3810 {
3811         arc_buf_t **bufp = arg;
3812         if (zio && zio->io_error) {
```



```
3813                 VERIFY(arc_buf_remove_ref(buf, arg));
3814                 *bufp = NULL;
3815         } else {
3816                 *bufp = buf;
3817                 ASSERT(buf->b_data);
3818         }
3819 }

3821 static void
3822 arc_read_done(zio_t *zio)
3823 {
3824         arc_buf_hdr_t   *hdr;
3825         arc_buf_t       *buf;
3826         arc_buf_t       *abuf;  /* buffer we're assigning to callback */
3827         kmutex_t        *hash_lock = NULL;
3828         arc_callback_t  *callback_list, *acb;
3829         int             freeable = FALSE;

3831         buf = zio->io_private;
3832         hdr = buf->b_hdr;

3834         /*
3835          * The hdr was inserted into hash-table and removed from lists
3836          * prior to starting I/O.  We should find this header, since
3837          * it's in the hash table, and it should be legit since it's
3838          * not possible to evict it during the I/O.  The only possible
3839          * reason for it not to be found is if we were freed during the
3840          * read.
3841          */
3842         if (HDR_IN_HASH_TABLE(hdr)) {
3843                 ASSERT3U(hdr->b_birth, ==, BP_PHYSICAL_BIRTH(zio->io_bp));
3844                 ASSERT3U(hdr->b_dva.dva_word[0], ==,
3845                     BP_IDENTITY(zio->io_bp)->dva_word[0]);
3846                 ASSERT3U(hdr->b_dva.dva_word[1], ==,
3847                     BP_IDENTITY(zio->io_bp)->dva_word[1]);

3849                 arc_buf_hdr_t *found = buf_hash_find(hdr->b_spa, zio->io_bp,
3850                     &hash_lock);

3852                 ASSERT((found == NULL && HDR_FREED_IN_READ(hdr) &&
3853                     hash_lock == NULL) ||
3854                     (found == hdr &&
3855                     DVA_EQUAL(&hdr->b_dva, BP_IDENTITY(zio->io_bp))) ||
3856                     (found == hdr && HDR_L2_READING(hdr)));
3857         }

3859         hdr->b_flags &= ~ARC_FLAG_L2_EVICTED;
3860         if (l2arc_noprefetch && HDR_PREFETCH(hdr))
3861                 hdr->b_flags &= ~ARC_FLAG_L2CACHE;

3863         /* byteswap if necessary */
3864         callback_list = hdr->b_l1hdr.b_acb;
3865         ASSERT(callback_list != NULL);
3866         if (BP_SHOULD_BYTESWAP(zio->io_bp) && zio->io_error == 0) {
3867                 dmu_object_byteswap_t bswap =
3868                     DMU_OT_BYTESWAP(BP_GET_TYPE(zio->io_bp));
3869                 arc_byteswap_func_t *func = BP_GET_LEVEL(zio->io_bp) > 0 ?
3870                     byteswap_uint64_array :
3871                     dmu_ot_byteswap[bswap].ob_func;
3872                 func(buf->b_data, hdr->b_size);
3873         }

3875         arc_cksum_compute(buf, B_FALSE);
3876         arc_buf_watch(buf);

3878         if (hash_lock && zio->io_error == 0 &&
```

```
3879             hdr->b_l1hdr.b_state == arc_anon) {
3880                 /*
3881                  * Only call arc_access on anonymous buffers.  This is because
3882                  * if we've issued an I/O for an evicted buffer, we've already
3883                  * called arc_access (to prevent any simultaneous readers from
3884                  * getting confused).
3885                  */
3886                 arc_access(hdr, hash_lock);
3887         }
3888
3889         /* create copies of the data buffer for the callers */
3890         abuf = buf;
3891         for (acb = callback_list; acb; acb = acb->acb_next) {
3892                 if (acb->acb_done) {
3893                         if (abuf == NULL) {
3894                                 ARCSTAT_BUMP(arcstat_duplicate_reads);
3895                                 abuf = arc_buf_clone(buf);
3896                         }
3897                         acb->acb_buf = abuf;
3898                         abuf = NULL;
3899                 }
3900         }
3901         hdr->b_l1hdr.b_acb = NULL;
3902         hdr->b_flags &= ~ARC_FLAG_IO_IN_PROGRESS;
3903         ASSERT(!HDR_BUF_AVAILABLE(hdr));
3904         if (abuf == buf) {
3905                 ASSERT(buf->b_efunc == NULL);
3906                 ASSERT(hdr->b_l1hdr.b_datacnt == 1);
3907                 hdr->b_flags |= ARC_FLAG_BUF_AVAILABLE;
3908         }
3909
3910         ASSERT(refcount_is_zero(&hdr->b_l1hdr.b_refcnt) ||
3911             callback_list != NULL);
3912
3913         if (zio->io_error != 0) {
3914                 hdr->b_flags |= ARC_FLAG_IO_ERROR;
3915                 if (hdr->b_l1hdr.b_state != arc_anon)
3916                         arc_change_state(arc_anon, hdr, hash_lock);
3917                 if (HDR_IN_HASH_TABLE(hdr))
3918                         buf_hash_remove(hdr);
3919                 freeable = refcount_is_zero(&hdr->b_l1hdr.b_refcnt);
3920         }
3921
3922         /*
3923          * Broadcast before we drop the hash_lock to avoid the possibility
3924          * that the hdr (and hence the cv) might be freed before we get to
3925          * the cv_broadcast().
3926          */
3927         cv_broadcast(&hdr->b_l1hdr.b_cv);
3928
3929         if (hash_lock != NULL) {
3930                 mutex_exit(hash_lock);
3931         } else {
3932                 /*
3933                  * This block was freed while we waited for the read to
3934                  * complete.  It has been removed from the hash table and
3935                  * moved to the anonymous state (so that it won't show up
3936                  * in the cache).
3937                  */
3938                 ASSERT3P(hdr->b_l1hdr.b_state, ==, arc_anon);
3939                 freeable = refcount_is_zero(&hdr->b_l1hdr.b_refcnt);
3940         }
3941
3942         /* execute each callback and free its structure */
3943         while ((acb = callback_list) != NULL) {
3944                 if (acb->acb_done)
```



```
3945                         acb->acb_done(zio, acb->acb_buf, acb->acb_private);
3946
3947                 if (acb->acb_zio_dummy != NULL) {
3948                         acb->acb_zio_dummy->io_error = zio->io_error;
3949                         zio_nowait(acb->acb_zio_dummy);
3950                 }
3951
3952                 callback_list = acb->acb_next;
3953                 kmem_free(acb, sizeof (arc_callback_t));
3954         }
3955
3956         if (freeable)
3957                 arc_hdr_destroy(hdr);
3958 }
3959
3960 /*
3961  * "Read" the block at the specified DVA (in bp) via the
3962  * cache.  If the block is found in the cache, invoke the provided
3963  * callback immediately and return.  Note that the 'zio' parameter
3964  * in the callback will be NULL in this case, since no IO was
3965  * required.  If the block is not in the cache pass the read request
3966  * on to the spa with a substitute callback function, so that the
3967  * requested block will be added to the cache.
3968  *
3969  * If a read request arrives for a block that has a read in-progress,
3970  * either wait for the in-progress read to complete (and return the
3971  * results); or, if this is a read with a "done" func, add a record
3972  * to the read to invoke the "done" func when the read completes,
3973  * and return; or just return.
3974  *
3975  * arc_read_done() will invoke all the requested "done" functions
3976  * for readers of this block.
3977  */
3978 int
3979 arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, arc_done_func_t *done,
3980     void *private, zio_priority_t priority, int zio_flags,
3981     arc_flags_t *arc_flags, const zbookmark_phys_t *zb)
3982 {
3983         arc_buf_hdr_t *hdr = NULL;
3984         arc_buf_t *buf = NULL;
3985         kmutex_t *hash_lock = NULL;
3986         zio_t *rzio;
3987         uint64_t guid = spa_load_guid(spa);
3988
3989         ASSERT(!BP_IS_EMBEDDED(bp) ||
3990             BPE_GET_ETYPE(bp) == BP_EMBEDDED_TYPE_DATA);
3991
3992 top:
3993         if (!BP_IS_EMBEDDED(bp)) {
3994                 /*
3995                  * Embedded BP's have no DVA and require no I/O to "read".
3996                  * Create an anonymous arc buf to back it.
3997                  */
3998                 hdr = buf_hash_find(guid, bp, &hash_lock);
3999         }
4000
4001         if (hdr != NULL && HDR_HAS_L1HDR(hdr) && hdr->b_l1hdr.b_datacnt > 0) {
4002
4003                 *arc_flags |= ARC_FLAG_CACHED;
4004
4005                 if (HDR_IO_IN_PROGRESS(hdr)) {
4006
4007                         if (*arc_flags & ARC_FLAG_WAIT) {
4008                                 cv_wait(&hdr->b_l1hdr.b_cv, hash_lock);
4009                                 mutex_exit(hash_lock);
4010                                 goto top;
```

```
4011                        }
4012                        ASSERT(*arc_flags & ARC_FLAG_NOWAIT);

4014                        if (done) {
4015                                arc_callback_t  *acb = NULL;

4017                                acb = kmem_zalloc(sizeof (arc_callback_t),
4018                                    KM_SLEEP);
4019                                acb->acb_done = done;
4020                                acb->acb_private = private;
4021                                if (pio != NULL)
4022                                        acb->acb_zio_dummy = zio_null(pio,
4023                                            spa, NULL, NULL, NULL, zio_flags);

4025                                ASSERT(acb->acb_done != NULL);
4026                                acb->acb_next = hdr->b_l1hdr.b_acb;
4027                                hdr->b_l1hdr.b_acb = acb;
4028                                add_reference(hdr, hash_lock, private);
4029                                mutex_exit(hash_lock);
4030                                return (0);
4031                        }
4032                        mutex_exit(hash_lock);
4033                        return (0);
4034                }

4036                ASSERT(hdr->b_l1hdr.b_state == arc_mru ||
4037                    hdr->b_l1hdr.b_state == arc_mfu);

4039                if (done) {
4040                        add_reference(hdr, hash_lock, private);
4041                        /*
4042                         * If this block is already in use, create a new
4043                         * copy of the data so that we will be guaranteed
4044                         * that arc_release() will always succeed.
4045                         */
4046                        buf = hdr->b_l1hdr.b_buf;
4047                        ASSERT(buf);
4048                        ASSERT(buf->b_data);
4049                        if (HDR_BUF_AVAILABLE(hdr)) {
4050                                ASSERT(buf->b_efunc == NULL);
4051                                hdr->b_flags &= ~ARC_FLAG_BUF_AVAILABLE;
4052                        } else {
4053                                buf = arc_buf_clone(buf);
4054                        }

4056                } else if (*arc_flags & ARC_FLAG_PREFETCH &&
4057                    refcount_count(&hdr->b_l1hdr.b_refcnt) == 0) {
4058                        hdr->b_flags |= ARC_FLAG_PREFETCH;
4059                }
4060                DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
4061                arc_access(hdr, hash_lock);
4062                if (*arc_flags & ARC_FLAG_L2CACHE)
4063                        hdr->b_flags |= ARC_FLAG_L2CACHE;
4064                if (*arc_flags & ARC_FLAG_L2COMPRESS)
4065                        hdr->b_flags |= ARC_FLAG_L2COMPRESS;
4066                mutex_exit(hash_lock);
4067                ARCSTAT_BUMP(arcstat_hits);
4068                ARCSTAT_CONDSTAT(!HDR_PREFETCH(hdr),
4069                    demand, prefetch, !HDR_ISTYPE_METADATA(hdr),
4070                    data, metadata, hits);

4072                if (done)
4073                        done(NULL, buf, private);
4074        } else {
4075                uint64_t size = BP_GET_LSIZE(bp);
4076                arc_callback_t *acb;
```



```
4077                vdev_t *vd = NULL;
4078                uint64_t addr = 0;
4079                boolean_t devw = B_FALSE;
4080                enum zio_compress b_compress = ZIO_COMPRESS_OFF;
4081                int32_t b_asize = 0;

4083                if (hdr == NULL) {
4084                        /* this block is not in the cache */
4085                        arc_buf_hdr_t *exists = NULL;
4086                        arc_buf_contents_t type = BP_GET_BUFC_TYPE(bp);
4087                        buf = arc_buf_alloc(spa, size, private, type);
4088                        hdr = buf->b_hdr;
4089                        if (!BP_IS_EMBEDDED(bp)) {
4090                                hdr->b_dva = *BP_IDENTITY(bp);
4091                                hdr->b_birth = BP_PHYSICAL_BIRTH(bp);
4092                                exists = buf_hash_insert(hdr, &hash_lock);
4093                        }
4094                        if (exists != NULL) {
4095                                /* somebody beat us to the hash insert */
4096                                mutex_exit(hash_lock);
4097                                buf_discard_identity(hdr);
4098                                (void) arc_buf_remove_ref(buf, private);
4099                                goto top; /* restart the IO request */
4100                        }

4102                        /* if this is a prefetch, we don't have a reference */
4103                        if (*arc_flags & ARC_FLAG_PREFETCH) {
4104                                (void) remove_reference(hdr, hash_lock,
4105                                    private);
4106                                hdr->b_flags |= ARC_FLAG_PREFETCH;
4107                        }
4108                        if (*arc_flags & ARC_FLAG_L2CACHE)
4109                                hdr->b_flags |= ARC_FLAG_L2CACHE;
4110                        if (*arc_flags & ARC_FLAG_L2COMPRESS)
4111                                hdr->b_flags |= ARC_FLAG_L2COMPRESS;
4112                        if (BP_GET_LEVEL(bp) > 0)
4113                                hdr->b_flags |= ARC_FLAG_INDIRECT;
4114                } else {
4115                        /*
4116                         * This block is in the ghost cache. If it was L2-only
4117                         * (and thus didn't have an L1 hdr), we realloc the
4118                         * header to add an L1 hdr.
4119                         */
4120                        if (!HDR_HAS_L1HDR(hdr)) {
4121                                hdr = arc_hdr_realloc(hdr, hdr_l2only_cache,
4122                                    hdr_full_cache);
4123                        }

4125                        ASSERT(GHOST_STATE(hdr->b_l1hdr.b_state));
4126                        ASSERT(!HDR_IO_IN_PROGRESS(hdr));
4127                        ASSERT(refcount_is_zero(&hdr->b_l1hdr.b_refcnt));
4128                        ASSERT3P(hdr->b_l1hdr.b_buf, ==, NULL);

4130                        /* if this is a prefetch, we don't have a reference */
4131                        if (*arc_flags & ARC_FLAG_PREFETCH)
4132                                hdr->b_flags |= ARC_FLAG_PREFETCH;
4133                        else
4134                                add_reference(hdr, hash_lock, private);
4135                        if (*arc_flags & ARC_FLAG_L2CACHE)
4136                                hdr->b_flags |= ARC_FLAG_L2CACHE;
4137                        if (*arc_flags & ARC_FLAG_L2COMPRESS)
4138                                hdr->b_flags |= ARC_FLAG_L2COMPRESS;
4139                        buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
4140                        buf->b_hdr = hdr;
4141                        buf->b_data = NULL;
4142                        buf->b_efunc = NULL;
```

```
4143                        buf->b_private = NULL;
4144                        buf->b_next = NULL;
4145                        hdr->b_l1hdr.b_buf = buf;
4146                        ASSERT0(hdr->b_l1hdr.b_datacnt);
4147                        hdr->b_l1hdr.b_datacnt = 1;
4148                        arc_get_data_buf(buf);
4149                        arc_access(hdr, hash_lock);
4150                }

4152                ASSERT(!GHOST_STATE(hdr->b_l1hdr.b_state));

4154                acb = kmem_zalloc(sizeof (arc_callback_t), KM_SLEEP);
4155                acb->acb_done = done;
4156                acb->acb_private = private;

4158                ASSERT(hdr->b_l1hdr.b_acb == NULL);
4159                hdr->b_l1hdr.b_acb = acb;
4160                hdr->b_flags |= ARC_FLAG_IO_IN_PROGRESS;

4162                if (HDR_HAS_L2HDR(hdr) &&
4163                    (vd = hdr->b_l2hdr.b_dev->l2ad_vdev) != NULL) {
4164                        devw = hdr->b_l2hdr.b_dev->l2ad_writing;
4165                        addr = hdr->b_l2hdr.b_daddr;
4166                        b_compress = hdr->b_l2hdr.b_compress;
4167                        b_asize = hdr->b_l2hdr.b_asize;
4168                        /*
4169                         * Lock out device removal.
4170                         */
4171                        if (vdev_is_dead(vd) ||
4172                            !spa_config_tryenter(spa, SCL_L2ARC, vd, RW_READER))
4173                                vd = NULL;
4174                }

4176                if (hash_lock != NULL)
4177                        mutex_exit(hash_lock);

4179                /*
4180                 * At this point, we have a level 1 cache miss.  Try again in
4181                 * L2ARC if possible.
4182                 */
4183                ASSERT3U(hdr->b_size, ==, size);
4184                DTRACE_PROBE4(arc__miss, arc_buf_hdr_t *, hdr, blkptr_t *, bp,
4185                    uint64_t, size, zbookmark_phys_t *, zb);
4186                ARCSTAT_BUMP(arcstat_misses);
4187                ARCSTAT_CONDSTAT(!HDR_PREFETCH(hdr),
4188                    demand, prefetch, !HDR_ISTYPE_METADATA(hdr),
4189                    data, metadata, misses);

4191                if (vd != NULL && l2arc_ndev != 0 && !(l2arc_norw && devw)) {
4192                        /*
4193                         * Read from the L2ARC if the following are true:
4194                         * 1. The L2ARC vdev was previously cached.
4195                         * 2. This buffer still has L2ARC metadata.
4196                         * 3. This buffer isn't currently writing to the L2ARC.
4197                         * 4. The L2ARC entry wasn't evicted, which may
4198                         *    also have invalidated the vdev.
4199                         * 5. This isn't prefetch and l2arc_noprefetch is set.
4200                         */
4201                        if (HDR_HAS_L2HDR(hdr) &&
4202                            !HDR_L2_WRITING(hdr) && !HDR_L2_EVICTED(hdr) &&
4203                            !(l2arc_noprefetch && HDR_PREFETCH(hdr))) {
4204                                l2arc_read_callback_t *cb;

4206                                DTRACE_PROBE1(l2arc__hit, arc_buf_hdr_t *, hdr);
4207                                ARCSTAT_BUMP(arcstat_l2_hits);
```



```
4209                                cb = kmem_zalloc(sizeof (l2arc_read_callback_t),
4210                                    KM_SLEEP);
4211                                cb->l2rcb_buf = buf;
4212                                cb->l2rcb_spa = spa;
4213                                cb->l2rcb_bp = *bp;
4214                                cb->l2rcb_zb = *zb;
4215                                cb->l2rcb_flags = zio_flags;
4216                                cb->l2rcb_compress = b_compress;

4218                                ASSERT(addr >= VDEV_LABEL_START_SIZE &&
4219                                    addr + size < vd->vdev_psize -
4220                                    VDEV_LABEL_END_SIZE);

4222                                /*
4223                                 * l2arc read.  The SCL_L2ARC lock will be
4224                                 * released by l2arc_read_done().
4225                                 * Issue a null zio if the underlying buffer
4226                                 * was squashed to zero size by compression.
4227                                 */
4228                                if (b_compress == ZIO_COMPRESS_EMPTY) {
4229                                        rzio = zio_null(pio, spa, vd,
4230                                            l2arc_read_done, cb,
4231                                            zio_flags | ZIO_FLAG_DONT_CACHE |
4232                                            ZIO_FLAG_CANFAIL |
4233                                            ZIO_FLAG_DONT_PROPAGATE |
4234                                            ZIO_FLAG_DONT_RETRY);
4235                                } else {
4236                                        rzio = zio_read_phys(pio, vd, addr,
4237                                            b_asize, buf->b_data,
4238                                            ZIO_CHECKSUM_OFF,
4239                                            l2arc_read_done, cb, priority,
4240                                            zio_flags | ZIO_FLAG_DONT_CACHE |
4241                                            ZIO_FLAG_CANFAIL |
4242                                            ZIO_FLAG_DONT_PROPAGATE |
4243                                            ZIO_FLAG_DONT_RETRY, B_FALSE);
4244                                }
4245                                DTRACE_PROBE2(l2arc__read, vdev_t *, vd,
4246                                    zio_t *, rzio);
4247                                ARCSTAT_INCR(arcstat_l2_read_bytes, b_asize);

4249                                if (*arc_flags & ARC_FLAG_NOWAIT) {
4250                                        zio_nowait(rzio);
4251                                        return (0);
4252                                }

4254                                ASSERT(*arc_flags & ARC_FLAG_WAIT);
4255                                if (zio_wait(rzio) == 0)
4256                                        return (0);

4258                                /* l2arc read error; goto zio_read() */
4259                        } else {
4260                                DTRACE_PROBE1(l2arc__miss,
4261                                    arc_buf_hdr_t *, hdr);
4262                                ARCSTAT_BUMP(arcstat_l2_misses);
4263                                if (HDR_L2_WRITING(hdr))
4264                                        ARCSTAT_BUMP(arcstat_l2_rw_clash);
4265                                spa_config_exit(spa, SCL_L2ARC, vd);
4266                        }
4267                } else {
4268                        if (vd != NULL)
4269                                spa_config_exit(spa, SCL_L2ARC, vd);
4270                        if (l2arc_ndev != 0) {
4271                                DTRACE_PROBE1(l2arc__miss,
4272                                    arc_buf_hdr_t *, hdr);
4273                                ARCSTAT_BUMP(arcstat_l2_misses);
4274                        }
```

```
4275                 }
4276
4277                 rzio = zio_read(pio, spa, bp, buf->b_data, size,
4278                     arc_read_done, buf, priority, zio_flags, zb);
4279
4280                 if (*arc_flags & ARC_FLAG_WAIT)
4281                         return (zio_wait(rzio));
4282
4283                 ASSERT(*arc_flags & ARC_FLAG_NOWAIT);
4284                 zio_nowait(rzio);
4285         }
4286         return (0);
4287 }
4288
4289 void
4290 arc_set_callback(arc_buf_t *buf, arc_evict_func_t *func, void *private)
4291 {
4292         ASSERT(buf->b_hdr != NULL);
4293         ASSERT(buf->b_hdr->b_l1hdr.b_state != arc_anon);
4294         ASSERT(!refcount_is_zero(&buf->b_hdr->b_l1hdr.b_refcnt) ||
4295             func == NULL);
4296         ASSERT(buf->b_efunc == NULL);
4297         ASSERT(!HDR_BUF_AVAILABLE(buf->b_hdr));
4298
4299         buf->b_efunc = func;
4300         buf->b_private = private;
4301 }
4302
4303 /*
4304  * Notify the arc that a block was freed, and thus will never be used again.
4305  */
4306 void
4307 arc_freed(spa_t *spa, const blkptr_t *bp)
4308 {
4309         arc_buf_hdr_t *hdr;
4310         kmutex_t *hash_lock;
4311         uint64_t guid = spa_load_guid(spa);
4312
4313         ASSERT(!BP_IS_EMBEDDED(bp));
4314
4315         hdr = buf_hash_find(guid, bp, &hash_lock);
4316         if (hdr == NULL)
4317                 return;
4318         if (HDR_BUF_AVAILABLE(hdr)) {
4319                 arc_buf_t *buf = hdr->b_l1hdr.b_buf;
4320                 add_reference(hdr, hash_lock, FTAG);
4321                 hdr->b_flags &= ~ARC_FLAG_BUF_AVAILABLE;
4322                 mutex_exit(hash_lock);
4323
4324                 arc_release(buf, FTAG);
4325                 (void) arc_buf_remove_ref(buf, FTAG);
4326         } else {
4327                 mutex_exit(hash_lock);
4328         }
4329
4330 }
4331
4332 /*
4333  * Clear the user eviction callback set by arc_set_callback(), first calling
4334  * it if it exists.  Because the presence of a callback keeps an arc_buf cached
4335  * clearing the callback may result in the arc_buf being destroyed.  However,
4336  * it will not result in the *last* arc_buf being destroyed, hence the data
4337  * will remain cached in the ARC. We make a copy of the arc buffer here so
4338  * that we can process the callback without holding any locks.
4339  *
4340  * It's possible that the callback is already in the process of being cleared
```



```
4341  * by another thread.  In this case we can not clear the callback.
4342  *
4343  * Returns B_TRUE if the callback was successfully called and cleared.
4344  */
4345 boolean_t
4346 arc_clear_callback(arc_buf_t *buf)
4347 {
4348         arc_buf_hdr_t *hdr;
4349         kmutex_t *hash_lock;
4350         arc_evict_func_t *efunc = buf->b_efunc;
4351         void *private = buf->b_private;
4352
4353         mutex_enter(&buf->b_evict_lock);
4354         hdr = buf->b_hdr;
4355         if (hdr == NULL) {
4356                 /*
4357                  * We are in arc_do_user_evicts().
4358                  */
4359                 ASSERT(buf->b_data == NULL);
4360                 mutex_exit(&buf->b_evict_lock);
4361                 return (B_FALSE);
4362         } else if (buf->b_data == NULL) {
4363                 /*
4364                  * We are on the eviction list; process this buffer now
4365                  * but let arc_do_user_evicts() do the reaping.
4366                  */
4367                 buf->b_efunc = NULL;
4368                 mutex_exit(&buf->b_evict_lock);
4369                 VERIFY0(efunc(private));
4370                 return (B_TRUE);
4371         }
4372         hash_lock = HDR_LOCK(hdr);
4373         mutex_enter(hash_lock);
4374         hdr = buf->b_hdr;
4375         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
4376
4377         ASSERT3U(refcount_count(&hdr->b_l1hdr.b_refcnt), <,
4378             hdr->b_l1hdr.b_datacnt);
4379         ASSERT(hdr->b_l1hdr.b_state == arc_mru ||
4380             hdr->b_l1hdr.b_state == arc_mfu);
4381
4382         buf->b_efunc = NULL;
4383         buf->b_private = NULL;
4384
4385         if (hdr->b_l1hdr.b_datacnt > 1) {
4386                 mutex_exit(&buf->b_evict_lock);
4387                 arc_buf_destroy(buf, TRUE);
4388         } else {
4389                 ASSERT(buf == hdr->b_l1hdr.b_buf);
4390                 hdr->b_flags |= ARC_FLAG_BUF_AVAILABLE;
4391                 mutex_exit(&buf->b_evict_lock);
4392         }
4393
4394         mutex_exit(hash_lock);
4395         VERIFY0(efunc(private));
4396         return (B_TRUE);
4397 }
4398
4399 /*
4400  * Release this buffer from the cache, making it an anonymous buffer.  This
4401  * must be done after a read and prior to modifying the buffer contents.
4402  * If the buffer has more than one reference, we must make
4403  * a new hdr for the buffer.
4404  */
4405 void
4406 arc_release(arc_buf_t *buf, void *tag)
```

```
4407 {
4408         arc_buf_hdr_t *hdr = buf->b_hdr;
4409
4410         /*
4411          * It would be nice to assert that if it's DMU metadata (level >
4412          * 0 || it's the dnode file), then it must be syncing context.
4413          * But we don't know that information at this level.
4414          */
4415
4416         mutex_enter(&buf->b_evict_lock);
4417
4418         ASSERT(HDR_HAS_L1HDR(hdr));
4419
4420         /*
4421          * We don't grab the hash lock prior to this check, because if
4422          * the buffer's header is in the arc_anon state, it won't be
4423          * linked into the hash table.
4424          */
4425         if (hdr->b_l1hdr.b_state == arc_anon) {
4426                 mutex_exit(&buf->b_evict_lock);
4427                 ASSERT(!HDR_IO_IN_PROGRESS(hdr));
4428                 ASSERT(!HDR_IN_HASH_TABLE(hdr));
4429                 ASSERT(!HDR_HAS_L2HDR(hdr));
4430                 ASSERT(BUF_EMPTY(hdr));
4431
4432                 ASSERT3U(hdr->b_l1hdr.b_datacnt, ==, 1);
4433                 ASSERT3S(refcount_count(&hdr->b_l1hdr.b_refcnt), ==, 1);
4434                 ASSERT(!list_link_active(&hdr->b_l1hdr.b_arc_node));
4435
4436                 ASSERT3P(buf->b_efunc, ==, NULL);
4437                 ASSERT3P(buf->b_private, ==, NULL);
4438
4439                 hdr->b_l1hdr.b_arc_access = 0;
4440                 arc_buf_thaw(buf);
4441
4442                 return;
4443         }
4444
4445         kmutex_t *hash_lock = HDR_LOCK(hdr);
4446         mutex_enter(hash_lock);
4447
4448         /*
4449          * This assignment is only valid as long as the hash_lock is
4450          * held, we must be careful not to reference state or the
4451          * b_state field after dropping the lock.
4452          */
4453         arc_state_t *state = hdr->b_l1hdr.b_state;
4454         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
4455         ASSERT3P(state, !=, arc_anon);
4456
4457         /* this buffer is not on any list */
4458         ASSERT(refcount_count(&hdr->b_l1hdr.b_refcnt) > 0);
4459
4460         if (HDR_HAS_L2HDR(hdr)) {
4461                 mutex_enter(&hdr->b_l2hdr.b_dev->l2ad_mtx);
4462
4463                 /*
4464                  * We have to recheck this conditional again now that
4465                  * we're holding the l2ad_mtx to prevent a race with
4466                  * another thread which might be concurrently calling
4467                  * l2arc_evict(). In that case, l2arc_evict() might have
4468                  * destroyed the header's L2 portion as we were waiting
4469                  * to acquire the l2ad_mtx.
4470                  */
4471                 if (HDR_HAS_L2HDR(hdr))
4472                         arc_hdr_l2hdr_destroy(hdr);
```



```
4473
4474                 mutex_exit(&hdr->b_l2hdr.b_dev->l2ad_mtx);
4475         }
4476
4477         /*
4478          * Do we have more than one buf?
4479          */
4480         if (hdr->b_l1hdr.b_datacnt > 1) {
4481                 arc_buf_hdr_t *nhdr;
4482                 arc_buf_t **bufp;
4483                 uint64_t blksz = hdr->b_size;
4484                 uint64_t spa = hdr->b_spa;
4485                 arc_buf_contents_t type = arc_buf_type(hdr);
4486                 uint32_t flags = hdr->b_flags;
4487
4488                 ASSERT(hdr->b_l1hdr.b_buf != buf || buf->b_next != NULL);
4489                 /*
4490                  * Pull the data off of this hdr and attach it to
4491                  * a new anonymous hdr.
4492                  */
4493                 (void) remove_reference(hdr, hash_lock, tag);
4494                 bufp = &hdr->b_l1hdr.b_buf;
4495                 while (*bufp != buf)
4496                         bufp = &(*bufp)->b_next;
4497                 *bufp = buf->b_next;
4498                 buf->b_next = NULL;
4499
4500                 ASSERT3P(state, !=, arc_l2c_only);
4501
4502                 (void) refcount_remove_many(
4503                     &state->arcs_size, hdr->b_size, buf);
4504
4505                 if (refcount_is_zero(&hdr->b_l1hdr.b_refcnt)) {
4506                         ASSERT3P(state, !=, arc_l2c_only);
4507                         uint64_t *size = &state->arcs_lsize[type];
4508                         ASSERT3U(*size, >=, hdr->b_size);
4509                         atomic_add_64(size, -hdr->b_size);
4510                 }
4511
4512                 /*
4513                  * We're releasing a duplicate user data buffer, update
4514                  * our statistics accordingly.
4515                  */
4516                 if (HDR_ISTYPE_DATA(hdr)) {
4517                         ARCSTAT_BUMPDOWN(arcstat_duplicate_buffers);
4518                         ARCSTAT_INCR(arcstat_duplicate_buffers_size,
4519                             -hdr->b_size);
4520                 }
4521                 hdr->b_l1hdr.b_datacnt -= 1;
4522                 arc_cksum_verify(buf);
4523                 arc_buf_unwatch(buf);
4524
4525                 mutex_exit(hash_lock);
4526
4527                 nhdr = kmem_cache_alloc(hdr_full_cache, KM_PUSHPAGE);
4528                 nhdr->b_size = blksz;
4529                 nhdr->b_spa = spa;
4530
4531                 nhdr->b_flags = flags & ARC_FLAG_L2_WRITING;
4532                 nhdr->b_flags |= arc_bufc_to_flags(type);
4533                 nhdr->b_flags |= ARC_FLAG_HAS_L1HDR;
4534
4535                 nhdr->b_l1hdr.b_buf = buf;
4536                 nhdr->b_l1hdr.b_datacnt = 1;
4537                 nhdr->b_l1hdr.b_state = arc_anon;
4538                 nhdr->b_l1hdr.b_arc_access = 0;
```

```
4539                 nhdr->b_l1hdr.b_tmp_cdata = NULL;
4540                 nhdr->b_freeze_cksum = NULL;

4542                 (void) refcount_add(&nhdr->b_l1hdr.b_refcnt, tag);
4543                 buf->b_hdr = nhdr;
4544                 mutex_exit(&buf->b_evict_lock);
4545                 (void) refcount_add_many(&arc_anon->arcs_size, blksz, buf);
4546         } else {
4547                 mutex_exit(&buf->b_evict_lock);
4548                 ASSERT(refcount_count(&hdr->b_l1hdr.b_refcnt) == 1);
4549                 /* protected by hash lock, or hdr is on arc_anon */
4550                 ASSERT(!multilist_link_active(&hdr->b_l1hdr.b_arc_node));
4551                 ASSERT(!HDR_IO_IN_PROGRESS(hdr));
4552                 arc_change_state(arc_anon, hdr, hash_lock);
4553                 hdr->b_l1hdr.b_arc_access = 0;
4554                 mutex_exit(hash_lock);

4556                 buf_discard_identity(hdr);
4557                 arc_buf_thaw(buf);
4558         }
4559         buf->b_efunc = NULL;
4560         buf->b_private = NULL;
4561 }

4563 int
4564 arc_released(arc_buf_t *buf)
4565 {
4566         int released;

4568         mutex_enter(&buf->b_evict_lock);
4569         released = (buf->b_data != NULL &&
4570             buf->b_hdr->b_l1hdr.b_state == arc_anon);
4571         mutex_exit(&buf->b_evict_lock);
4572         return (released);
4573 }

4575 #ifdef ZFS_DEBUG
4576 int
4577 arc_referenced(arc_buf_t *buf)
4578 {
4579         int referenced;

4581         mutex_enter(&buf->b_evict_lock);
4582         referenced = (refcount_count(&buf->b_hdr->b_l1hdr.b_refcnt));
4583         mutex_exit(&buf->b_evict_lock);
4584         return (referenced);
4585 }
4586 #endif

4588 static void
4589 arc_write_ready(zio_t *zio)
4590 {
4591         arc_write_callback_t *callback = zio->io_private;
4592         arc_buf_t *buf = callback->awcb_buf;
4593         arc_buf_hdr_t *hdr = buf->b_hdr;

4595         ASSERT(HDR_HAS_L1HDR(hdr));
4596         ASSERT(!refcount_is_zero(&buf->b_hdr->b_l1hdr.b_refcnt));
4597         ASSERT(hdr->b_l1hdr.b_datacnt > 0);
4598         callback->awcb_ready(zio, buf, callback->awcb_private);

4600         /*
4601          * If the IO is already in progress, then this is a re-write
4602          * attempt, so we need to thaw and re-compute the cksum.
4603          * It is the responsibility of the callback to handle the
4604          * accounting for any re-write attempt.
```



```
4605          */
4606         if (HDR_IO_IN_PROGRESS(hdr)) {
4607                 mutex_enter(&hdr->b_l1hdr.b_freeze_lock);
4608                 if (hdr->b_freeze_cksum != NULL) {
4609                         kmem_free(hdr->b_freeze_cksum, sizeof (zio_cksum_t));
4610                         hdr->b_freeze_cksum = NULL;
4611                 }
4612                 mutex_exit(&hdr->b_l1hdr.b_freeze_lock);
4613         }
4614         arc_cksum_compute(buf, B_FALSE);
4615         hdr->b_flags |= ARC_FLAG_IO_IN_PROGRESS;
4616 }

4618 /*
4619  * The SPA calls this callback for each physical write that happens on behalf
4620  * of a logical write.  See the comment in dbuf_write_physdone() for details.
4621  */
4622 static void
4623 arc_write_physdone(zio_t *zio)
4624 {
4625         arc_write_callback_t *cb = zio->io_private;
4626         if (cb->awcb_physdone != NULL)
4627                 cb->awcb_physdone(zio, cb->awcb_buf, cb->awcb_private);
4628 }

4630 static void
4631 arc_write_done(zio_t *zio)
4632 {
4633         arc_write_callback_t *callback = zio->io_private;
4634         arc_buf_t *buf = callback->awcb_buf;
4635         arc_buf_hdr_t *hdr = buf->b_hdr;

4637         ASSERT(hdr->b_l1hdr.b_acb == NULL);

4639         if (zio->io_error == 0) {
4640                 if (BP_IS_HOLE(zio->io_bp) || BP_IS_EMBEDDED(zio->io_bp)) {
4641                         buf_discard_identity(hdr);
4642                 } else {
4643                         hdr->b_dva = *BP_IDENTITY(zio->io_bp);
4644                         hdr->b_birth = BP_PHYSICAL_BIRTH(zio->io_bp);
4645                 }
4646         } else {
4647                 ASSERT(BUF_EMPTY(hdr));
4648         }

4650         /*
4651          * If the block to be written was all-zero or compressed enough to be
4652          * embedded in the BP, no write was performed so there will be no
4653          * dva/birth/checksum.  The buffer must therefore remain anonymous
4654          * (and uncached).
4655          */
4656         if (!BUF_EMPTY(hdr)) {
4657                 arc_buf_hdr_t *exists;
4658                 kmutex_t *hash_lock;

4660                 ASSERT(zio->io_error == 0);

4662                 arc_cksum_verify(buf);

4664                 exists = buf_hash_insert(hdr, &hash_lock);
4665                 if (exists != NULL) {
4666                         /*
4667                          * This can only happen if we overwrite for
4668                          * sync-to-convergence, because we remove
4669                          * buffers from the hash table when we arc_free().
4670                          */
```

```
4671                        if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {
4672                                if (!BP_EQUAL(&zio->io_bp_orig, zio->io_bp))
4673                                        panic("bad overwrite, hdr=%p exists=%p",
4674                                            (void *)hdr, (void *)exists);
4675                                ASSERT(refcount_is_zero(
4676                                    &exists->b_l1hdr.b_refcnt));
4677                                arc_change_state(arc_anon, exists, hash_lock);
4678                                mutex_exit(hash_lock);
4679                                arc_hdr_destroy(exists);
4680                                exists = buf_hash_insert(hdr, &hash_lock);
4681                                ASSERT3P(exists, ==, NULL);
4682                        } else if (zio->io_flags & ZIO_FLAG_NOPWRITE) {
4683                                /* nopwrite */
4684                                ASSERT(zio->io_prop.zp_nopwrite);
4685                                if (!BP_EQUAL(&zio->io_bp_orig, zio->io_bp))
4686                                        panic("bad nopwrite, hdr=%p exists=%p",
4687                                            (void *)hdr, (void *)exists);
4688                        } else {
4689                                /* Dedup */
4690                                ASSERT(hdr->b_l1hdr.b_datacnt == 1);
4691                                ASSERT(hdr->b_l1hdr.b_state == arc_anon);
4692                                ASSERT(BP_GET_DEDUP(zio->io_bp));
4693                                ASSERT(BP_GET_LEVEL(zio->io_bp) == 0);
4694                        }
4695                }
4696                hdr->b_flags &= ~ARC_FLAG_IO_IN_PROGRESS;
4697                /* if it's not anon, we are doing a scrub */
4698                if (exists == NULL && hdr->b_l1hdr.b_state == arc_anon)
4699                        arc_access(hdr, hash_lock);
4700                mutex_exit(hash_lock);
4701        } else {
4702                hdr->b_flags &= ~ARC_FLAG_IO_IN_PROGRESS;
4703        }

4705        ASSERT(!refcount_is_zero(&hdr->b_l1hdr.b_refcnt));
4706        callback->awcb_done(zio, buf, callback->awcb_private);

4708        kmem_free(callback, sizeof (arc_write_callback_t));
4709 }

4711 zio_t *
4712 arc_write(zio_t *pio, spa_t *spa, uint64_t txg,
4713     blkptr_t *bp, arc_buf_t *buf, boolean_t l2arc, boolean_t l2arc_compress,
4714     const zio_prop_t *zp, arc_done_func_t *ready, arc_done_func_t *physdone,
4715     arc_done_func_t *done, void *private, zio_priority_t priority,
4716     int zio_flags, const zbookmark_phys_t *zb)
4717 {
4718        arc_buf_hdr_t *hdr = buf->b_hdr;
4719        arc_write_callback_t *callback;
4720        zio_t *zio;

4722        ASSERT(ready != NULL);
4723        ASSERT(done != NULL);
4724        ASSERT(!HDR_IO_ERROR(hdr));
4725        ASSERT(!HDR_IO_IN_PROGRESS(hdr));
4726        ASSERT(hdr->b_l1hdr.b_acb == NULL);
4727        ASSERT(hdr->b_l1hdr.b_datacnt > 0);
4728        if (l2arc)
4729                hdr->b_flags |= ARC_FLAG_L2CACHE;
4730        if (l2arc_compress)
4731                hdr->b_flags |= ARC_FLAG_L2COMPRESS;
4732        callback = kmem_zalloc(sizeof (arc_write_callback_t), KM_SLEEP);
4733        callback->awcb_ready = ready;
4734        callback->awcb_physdone = physdone;
4735        callback->awcb_done = done;
4736        callback->awcb_private = private;
```



```
4737        callback->awcb_buf = buf;

4739        zio = zio_write(pio, spa, txg, bp, buf->b_data, hdr->b_size, zp,
4740            arc_write_ready, arc_write_physdone, arc_write_done, callback,
4741            priority, zio_flags, zb);

4743        return (zio);
4744 }

4746 static int
4747 arc_memory_throttle(uint64_t reserve, uint64_t txg)
4748 {
4749 #ifdef _KERNEL
4750        uint64_t available_memory = ptob(freemem);
4751        static uint64_t page_load = 0;
4752        static uint64_t last_txg = 0;

4754 #if defined(__i386)
4755        available_memory =
4756            MIN(available_memory, vmem_size(heap_arena, VMEM_FREE));
4757 #endif

4759        if (freemem > physmem * arc_lotsfree_percent / 100)
4760                return (0);

4762        if (txg > last_txg) {
4763                last_txg = txg;
4764                page_load = 0;
4765        }
4766        /*
4767         * If we are in pageout, we know that memory is already tight,
4768         * the arc is already going to be evicting, so we just want to
4769         * continue to let page writes occur as quickly as possible.
4770         */
4771        if (curproc == proc_pageout) {
4772                if (page_load > MAX(ptob(minfree), available_memory) / 4)
4773                        return (SET_ERROR(ERESTART));
4774                /* Note: reserve is inflated, so we deflate */
4775                page_load += reserve / 8;
4776                return (0);
4777        } else if (page_load > 0 && arc_reclaim_needed()) {
4778                /* memory is low, delay before restarting */
4779                ARCSTAT_INCR(arcstat_memory_throttle_count, 1);
4780                return (SET_ERROR(EAGAIN));
4781        }
4782        page_load = 0;
4783 #endif
4784        return (0);
4785 }

4787 void
4788 arc_tempreserve_clear(uint64_t reserve)
4789 {
4790        atomic_add_64(&arc_tempreserve, -reserve);
4791        ASSERT((int64_t)arc_tempreserve >= 0);
4792 }

4794 int
4795 arc_tempreserve_space(uint64_t reserve, uint64_t txg)
4796 {
4797        int error;
4798        uint64_t anon_size;

4800        if (reserve > arc_c/4 && !arc_no_grow)
4801                arc_c = MIN(arc_c_max, reserve * 4);
4802        if (reserve > arc_c)
```

```
4803                 return (SET_ERROR(ENOMEM));

4805         /*
4806          * Don't count loaned bufs as in flight dirty data to prevent long
4807          * network delays from blocking transactions that are ready to be
4808          * assigned to a txg.
4809          */
4810         anon_size = MAX((int64_t)(refcount_count(&arc_anon->arcs_size) -
4811             arc_loaned_bytes), 0);

4813         /*
4814          * Writes will, almost always, require additional memory allocations
4815          * in order to compress/encrypt/etc the data.  We therefore need to
4816          * make sure that there is sufficient available memory for this.
4817          */
4818         error = arc_memory_throttle(reserve, txg);
4819         if (error != 0)
4820                 return (error);

4822         /*
4823          * Throttle writes when the amount of dirty data in the cache
4824          * gets too large.  We try to keep the cache less than half full
4825          * of dirty blocks so that our sync times don't grow too large.
4826          * Note: if two requests come in concurrently, we might let them
4827          * both succeed, when one of them should fail.  Not a huge deal.
4828          */

4830         if (reserve + arc_tempreserve + anon_size > arc_c / 2 &&
4831             anon_size > arc_c / 4) {
4832                 dprintf("failing, arc_tempreserve=%lluK anon_meta=%lluK "
4833                     "anon_data=%lluK tempreserve=%lluK arc_c=%lluK\n",
4834                     arc_tempreserve>>10,
4835                     arc_anon->arcs_lsize[ARC_BUFC_METADATA]>>10,
4836                     arc_anon->arcs_lsize[ARC_BUFC_DATA]>>10,
4837                     reserve>>10, arc_c>>10);
4838                 return (SET_ERROR(ERESTART));
4839         }
4840         atomic_add_64(&arc_tempreserve, reserve);
4841         return (0);
4842 }

4844 static void
4845 arc_kstat_update_state(arc_state_t *state, kstat_named_t *size,
4846     kstat_named_t *evict_data, kstat_named_t *evict_metadata)
4847 {
4848         size->value.ui64 = refcount_count(&state->arcs_size);
4849         evict_data->value.ui64 = state->arcs_lsize[ARC_BUFC_DATA];
4850         evict_metadata->value.ui64 = state->arcs_lsize[ARC_BUFC_METADATA];
4851 }

4853 static int
4854 arc_kstat_update(kstat_t *ksp, int rw)
4855 {
4856         arc_stats_t *as = ksp->ks_data;

4858         if (rw == KSTAT_WRITE) {
4859                 return (EACCES);
4860         } else {
4861                 arc_kstat_update_state(arc_anon,
4862                     &as->arcstat_anon_size,
4863                     &as->arcstat_anon_evictable_data,
4864                     &as->arcstat_anon_evictable_metadata);
4865                 arc_kstat_update_state(arc_mru,
4866                     &as->arcstat_mru_size,
4867                     &as->arcstat_mru_evictable_data,
4868                     &as->arcstat_mru_evictable_metadata);
```



```
4869                 arc_kstat_update_state(arc_mru_ghost,
4870                     &as->arcstat_mru_ghost_size,
4871                     &as->arcstat_mru_ghost_evictable_data,
4872                     &as->arcstat_mru_ghost_evictable_metadata);
4873                 arc_kstat_update_state(arc_mfu,
4874                     &as->arcstat_mfu_size,
4875                     &as->arcstat_mfu_evictable_data,
4876                     &as->arcstat_mfu_evictable_metadata);
4877                 arc_kstat_update_state(arc_mfu_ghost,
4878                     &as->arcstat_mfu_ghost_size,
4879                     &as->arcstat_mfu_ghost_evictable_data,
4880                     &as->arcstat_mfu_ghost_evictable_metadata);
4881         }

4883         return (0);
4884 }

4886 /*
4887  * This function *must* return indices evenly distributed between all
4888  * sublists of the multilist. This is needed due to how the ARC eviction
4889  * code is laid out; arc_evict_state() assumes ARC buffers are evenly
4890  * distributed between all sublists and uses this assumption when
4891  * deciding which sublist to evict from and how much to evict from it.
4892  */
4893 unsigned int
4894 arc_state_multilist_index_func(multilist_t *ml, void *obj)
4895 {
4896         arc_buf_hdr_t *hdr = obj;

4898         /*
4899          * We rely on b_dva to generate evenly distributed index
4900          * numbers using buf_hash below. So, as an added precaution,
4901          * let's make sure we never add empty buffers to the arc lists.
4902          */
4903         ASSERT(!BUF_EMPTY(hdr));

4905         /*
4906          * The assumption here, is the hash value for a given
4907          * arc_buf_hdr_t will remain constant throughout it's lifetime
4908          * (i.e. it's b_spa, b_dva, and b_birth fields don't change).
4909          * Thus, we don't need to store the header's sublist index
4910          * on insertion, as this index can be recalculated on removal.
4911          *
4912          * Also, the low order bits of the hash value are thought to be
4913          * distributed evenly. Otherwise, in the case that the multilist
4914          * has a power of two number of sublists, each sublists' usage
4915          * would not be evenly distributed.
4916          */
4917         return (buf_hash(hdr->b_spa, &hdr->b_dva, hdr->b_birth) %
4918             multilist_get_num_sublists(ml));
4919 }

4921 void
4922 arc_init(void)
4923 {
4924         /*
4925          * allmem is "all memory that we could possibly use".
4926          */
4927 #ifdef _KERNEL
4928         uint64_t allmem = ptob(physmem - swapfs_minfree);
4929 #else
4930         uint64_t allmem = (physmem * PAGESIZE) / 2;
4931 #endif

4933         mutex_init(&arc_reclaim_lock, NULL, MUTEX_DEFAULT, NULL);
4934         cv_init(&arc_reclaim_thread_cv, NULL, CV_DEFAULT, NULL);
```

```
4935         cv_init(&arc_reclaim_waiters_cv, NULL, CV_DEFAULT, NULL);

4937         mutex_init(&arc_user_evicts_lock, NULL, MUTEX_DEFAULT, NULL);
4938         cv_init(&arc_user_evicts_cv, NULL, CV_DEFAULT, NULL);

4940         /* Convert seconds to clock ticks */
4941         arc_min_prefetch_lifespan = 1 * hz;

4943         /* Start out with 1/8 of all memory */
4944         arc_c = allmem / 8;

4946 #ifdef _KERNEL
4947         /*
4948          * On architectures where the physical memory can be larger
4949          * than the addressable space (intel in 32-bit mode), we may
4950          * need to limit the cache to 1/8 of VM size.
4951          */
4952         arc_c = MIN(arc_c, vmem_size(heap_arena, VMEM_ALLOC | VMEM_FREE) / 8);
4953 #endif

4955         /* set min cache to 1/32 of all memory, or 64MB, whichever is more */
4956         arc_c_min = MAX(allmem / 32, 64 << 20);
4957         /* set max to 3/4 of all memory, or all but 1GB, whichever is more */
4958         if (allmem >= 1 << 30)
4959                 arc_c_max = allmem - (1 << 30);
4960         else
4961                 arc_c_max = arc_c_min;
4962         arc_c_max = MAX(allmem * 3 / 4, arc_c_max);

4964         /*
4965          * Allow the tunables to override our calculations if they are
4966          * reasonable (ie. over 64MB)
4967          */
4968         if (zfs_arc_max > 64 << 20 && zfs_arc_max < allmem)
4969                 arc_c_max = zfs_arc_max;
4970         if (zfs_arc_min > 64 << 20 && zfs_arc_min <= arc_c_max)
4971                 arc_c_min = zfs_arc_min;

4973         arc_c = arc_c_max;
4974         arc_p = (arc_c >> 1);

4976         /* limit meta-data to 1/4 of the arc capacity */
4977         arc_meta_limit = arc_c_max / 4;

4979         /* Allow the tunable to override if it is reasonable */
4980         if (zfs_arc_meta_limit > 0 && zfs_arc_meta_limit <= arc_c_max)
4981                 arc_meta_limit = zfs_arc_meta_limit;

4983         if (arc_c_min < arc_meta_limit / 2 && zfs_arc_min == 0)
4984                 arc_c_min = arc_meta_limit / 2;

4986         if (zfs_arc_meta_min > 0) {
4987                 arc_meta_min = zfs_arc_meta_min;
4988         } else {
4989                 arc_meta_min = arc_c_min / 2;
4990         }

4992         if (zfs_arc_grow_retry > 0)
4993                 arc_grow_retry = zfs_arc_grow_retry;

4995         if (zfs_arc_shrink_shift > 0)
4996                 arc_shrink_shift = zfs_arc_shrink_shift;

4998         /*
4999          * Ensure that arc_no_grow_shift is less than arc_shrink_shift.
5000          */
```



```
5001         if (arc_no_grow_shift >= arc_shrink_shift)
5002                 arc_no_grow_shift = arc_shrink_shift - 1;

5004         if (zfs_arc_p_min_shift > 0)
5005                 arc_p_min_shift = zfs_arc_p_min_shift;

5007         if (zfs_arc_num_sublists_per_state < 1)
5008                 zfs_arc_num_sublists_per_state = MAX(boot_ncpus, 1);

5010         /* if kmem_flags are set, lets try to use less memory */
5011         if (kmem_debugging())
5012                 arc_c = arc_c / 2;
5013         if (arc_c < arc_c_min)
5014                 arc_c = arc_c_min;

5016         arc_anon = &ARC_anon;
5017         arc_mru = &ARC_mru;
5018         arc_mru_ghost = &ARC_mru_ghost;
5019         arc_mfu = &ARC_mfu;
5020         arc_mfu_ghost = &ARC_mfu_ghost;
5021         arc_l2c_only = &ARC_l2c_only;
5022         arc_size = 0;

5024         multilist_create(&arc_mru->arcs_list[ARC_BUFC_METADATA],
5025             sizeof (arc_buf_hdr_t),
5026             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5027             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);
5028         multilist_create(&arc_mru->arcs_list[ARC_BUFC_DATA],
5029             sizeof (arc_buf_hdr_t),
5030             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5031             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);
5032         multilist_create(&arc_mru_ghost->arcs_list[ARC_BUFC_METADATA],
5033             sizeof (arc_buf_hdr_t),
5034             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5035             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);
5036         multilist_create(&arc_mru_ghost->arcs_list[ARC_BUFC_DATA],
5037             sizeof (arc_buf_hdr_t),
5038             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5039             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);
5040         multilist_create(&arc_mfu->arcs_list[ARC_BUFC_METADATA],
5041             sizeof (arc_buf_hdr_t),
5042             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5043             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);
5044         multilist_create(&arc_mfu->arcs_list[ARC_BUFC_DATA],
5045             sizeof (arc_buf_hdr_t),
5046             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5047             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);
5048         multilist_create(&arc_mfu_ghost->arcs_list[ARC_BUFC_METADATA],
5049             sizeof (arc_buf_hdr_t),
5050             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5051             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);
5052         multilist_create(&arc_mfu_ghost->arcs_list[ARC_BUFC_DATA],
5053             sizeof (arc_buf_hdr_t),
5054             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5055             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);
5056         multilist_create(&arc_l2c_only->arcs_list[ARC_BUFC_METADATA],
5057             sizeof (arc_buf_hdr_t),
5058             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5059             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);
5060         multilist_create(&arc_l2c_only->arcs_list[ARC_BUFC_DATA],
5061             sizeof (arc_buf_hdr_t),
5062             offsetof(arc_buf_hdr_t, b_l1hdr.b_arc_node),
5063             zfs_arc_num_sublists_per_state, arc_state_multilist_index_func);

5065         refcount_create(&arc_anon->arcs_size);
5066         refcount_create(&arc_mru->arcs_size);
```

```
5067         refcount_create(&arc_mru_ghost->arcs_size);
5068         refcount_create(&arc_mfu->arcs_size);
5069         refcount_create(&arc_mfu_ghost->arcs_size);
5070         refcount_create(&arc_l2c_only->arcs_size);

5072         buf_init();

5074         arc_reclaim_thread_exit = FALSE;
5075         arc_user_evicts_thread_exit = FALSE;
5076         arc_eviction_list = NULL;
5077         bzero(&arc_eviction_hdr, sizeof (arc_buf_hdr_t));

5079         arc_ksp = kstat_create("zfs", 0, "arcstats", "misc", KSTAT_TYPE_NAMED,
5080             sizeof (arc_stats) / sizeof (kstat_named_t), KSTAT_FLAG_VIRTUAL);

5082         if (arc_ksp != NULL) {
5083                 arc_ksp->ks_data = &arc_stats;
5084                 arc_ksp->ks_update = arc_kstat_update;
5085                 kstat_install(arc_ksp);
5086         }

5088         (void) thread_create(NULL, 0, arc_reclaim_thread, NULL, 0, &p0,
5089             TS_RUN, minclsyspri);

5091         (void) thread_create(NULL, 0, arc_user_evicts_thread, NULL, 0, &p0,
5092             TS_RUN, minclsyspri);

5094         arc_dead = FALSE;
5095         arc_warm = B_FALSE;

5097         /*
5098          * Calculate maximum amount of dirty data per pool.
5099          *
5100          * If it has been set by /etc/system, take that.
5101          * Otherwise, use a percentage of physical memory defined by
5102          * zfs_dirty_data_max_percent (default 10%) with a cap at
5103          * zfs_dirty_data_max_max (default 4GB).
5104          */
5105         if (zfs_dirty_data_max == 0) {
5106                 zfs_dirty_data_max = physmem * PAGESIZE *
5107                     zfs_dirty_data_max_percent / 100;
5108                 zfs_dirty_data_max = MIN(zfs_dirty_data_max,
5109                     zfs_dirty_data_max_max);
5110         }
5111 }

5113 void
5114 arc_fini(void)
5115 {
5116         mutex_enter(&arc_reclaim_lock);
5117         arc_reclaim_thread_exit = TRUE;
5118         /*
5119          * The reclaim thread will set arc_reclaim_thread_exit back to
5120          * FALSE when it is finished exiting; we're waiting for that.
5121          */
5122         while (arc_reclaim_thread_exit) {
5123                 cv_signal(&arc_reclaim_thread_cv);
5124                 cv_wait(&arc_reclaim_thread_cv, &arc_reclaim_lock);
5125         }
5126         mutex_exit(&arc_reclaim_lock);

5128         mutex_enter(&arc_user_evicts_lock);
5129         arc_user_evicts_thread_exit = TRUE;
5130         /*
5131          * The user evicts thread will set arc_user_evicts_thread_exit
5132          * to FALSE when it is finished exiting; we're waiting for that.
```



```
5133          */
5134         while (arc_user_evicts_thread_exit) {
5135                 cv_signal(&arc_user_evicts_cv);
5136                 cv_wait(&arc_user_evicts_cv, &arc_user_evicts_lock);
5137         }
5138         mutex_exit(&arc_user_evicts_lock);

5140         /* Use TRUE to ensure *all* buffers are evicted */
5141         arc_flush(NULL, TRUE);

5143         arc_dead = TRUE;

5145         if (arc_ksp != NULL) {
5146                 kstat_delete(arc_ksp);
5147                 arc_ksp = NULL;
5148         }

5150         mutex_destroy(&arc_reclaim_lock);
5151         cv_destroy(&arc_reclaim_thread_cv);
5152         cv_destroy(&arc_reclaim_waiters_cv);

5154         mutex_destroy(&arc_user_evicts_lock);
5155         cv_destroy(&arc_user_evicts_cv);

5157         refcount_destroy(&arc_anon->arcs_size);
5158         refcount_destroy(&arc_mru->arcs_size);
5159         refcount_destroy(&arc_mru_ghost->arcs_size);
5160         refcount_destroy(&arc_mfu->arcs_size);
5161         refcount_destroy(&arc_mfu_ghost->arcs_size);
5162         refcount_destroy(&arc_l2c_only->arcs_size);

5164         multilist_destroy(&arc_mru->arcs_list[ARC_BUFC_METADATA]);
5165         multilist_destroy(&arc_mru_ghost->arcs_list[ARC_BUFC_METADATA]);
5166         multilist_destroy(&arc_mfu->arcs_list[ARC_BUFC_METADATA]);
5167         multilist_destroy(&arc_mfu_ghost->arcs_list[ARC_BUFC_METADATA]);
5168         multilist_destroy(&arc_mru->arcs_list[ARC_BUFC_DATA]);
5169         multilist_destroy(&arc_mru_ghost->arcs_list[ARC_BUFC_DATA]);
5170         multilist_destroy(&arc_mfu->arcs_list[ARC_BUFC_DATA]);
5171         multilist_destroy(&arc_mfu_ghost->arcs_list[ARC_BUFC_DATA]);

5173         buf_fini();

5175         ASSERT0(arc_loaned_bytes);
5176 }

5178 /*
5179  * Level 2 ARC
5180  *
5181  * The level 2 ARC (L2ARC) is a cache layer in-between main memory and disk.
5182  * It uses dedicated storage devices to hold cached data, which are populated
5183  * using large infrequent writes.  The main role of this cache is to boost
5184  * the performance of random read workloads.  The intended L2ARC devices
5185  * include short-stroked disks, solid state disks, and other media with
5186  * substantially faster read latency than disk.
5187  *
5188  *                 +-----------------------+
5189  *                 |         ARC           |
5190  *                 +-----------------------+
5191  *                    |         ^     ^
5192  *                    |         |     |
5193  *      l2arc_feed_thread()    arc_read()
5194  *                    |         |     |
5195  *                    |  l2arc read   |
5196  *                    V         |     |
5197  *               +---------------+    |
5198  *               |     L2ARC     |    |
```

```
5199  *               +---------------+    |
5200  *                   |    ^           |
5201  *            l2arc_write() |           |
5202  *                   |    |           |
5203  *                   V    |           |
5204  *                 +-------+      +-------+
5205  *                 | vdev  |      | vdev  |
5206  *                 | cache |      | cache |
5207  *                 +-------+      +-------+
5208  *                 +=========+     .-----.
5209  *                 :  L2ARC  :    |-_____-|
5210  *                 : devices :    | Disks |
5211  *                 +=========+    `-_____-'
5212  *
5213  * Read requests are satisfied from the following sources, in order:
5214  *
5215  *      1) ARC
5216  *      2) vdev cache of L2ARC devices
5217  *      3) L2ARC devices
5218  *      4) vdev cache of disks
5219  *      5) disks
5220  *
5221  * Some L2ARC device types exhibit extremely slow write performance.
5222  * To accommodate for this there are some significant differences between
5223  * the L2ARC and traditional cache design:
5224  *
5225  * 1. There is no eviction path from the ARC to the L2ARC.  Evictions from
5226  * the ARC behave as usual, freeing buffers and placing headers on ghost
5227  * lists.  The ARC does not send buffers to the L2ARC during eviction as
5228  * this would add inflated write latencies for all ARC memory pressure.
5229  *
5230  * 2. The L2ARC attempts to cache data from the ARC before it is evicted.
5231  * It does this by periodically scanning buffers from the eviction-end of
5232  * the MFU and MRU ARC lists, copying them to the L2ARC devices if they are
5233  * not already there. It scans until a headroom of buffers is satisfied,
5234  * which itself is a buffer for ARC eviction. If a compressible buffer is
5235  * found during scanning and selected for writing to an L2ARC device, we
5236  * temporarily boost scanning headroom during the next scan cycle to make
5237  * sure we adapt to compression effects (which might significantly reduce
5238  * the data volume we write to L2ARC). The thread that does this is
5239  * l2arc_feed_thread(), illustrated below; example sizes are included to
5240  * provide a better sense of ratio than this diagram:
5241  *
5242  *             head -->                        tail
5243  *              +---------------------+----------+
5244  *      ARC_mfu |:::::#:::::::::::::::|o#o###o###|-->.   # already on L2ARC
5245  *              +---------------------+----------+   |   o L2ARC eligible
5246  *      ARC_mru |:#::::::::::::::::::|#o#ooo####|-->|   : ARC buffer
5247  *              +---------------------+----------+   |
5248  *                   15.9 Gbytes      ^ 32 Mbytes    |
5249  *                                  headroom         |
5250  *                                           l2arc_feed_thread()
5251  *                                                   |
5252  *                       l2arc write hand <--[oooo]--'
5253  *                               |           8 Mbyte
5254  *                               |          write max
5255  *                               V
5256  *                +==============================+
5257  *      L2ARC dev |####|#|###|###|    |####| ... |
5258  *                +==============================+
5259  *                               32 Gbytes
5260  *
5261  * 3. If an ARC buffer is copied to the L2ARC but then hit instead of
5262  * evicted, then the L2ARC has cached a buffer much sooner than it probably
5263  * needed to, potentially wasting L2ARC device bandwidth and storage.  It is
5264  * safe to say that this is an uncommon case, since buffers at the end of
```



```
5265  * the ARC lists have moved there due to inactivity.
5266  *
5267  * 4. If the ARC evicts faster than the L2ARC can maintain a headroom,
5268  * then the L2ARC simply misses copying some buffers.  This serves as a
5269  * pressure valve to prevent heavy read workloads from both stalling the ARC
5270  * with waits and clogging the L2ARC with writes.  This also helps prevent
5271  * the potential for the L2ARC to churn if it attempts to cache content too
5272  * quickly, such as during backups of the entire pool.
5273  *
5274  * 5. After system boot and before the ARC has filled main memory, there are
5275  * no evictions from the ARC and so the tails of the ARC_mfu and ARC_mru
5276  * lists can remain mostly static.  Instead of searching from tail of these
5277  * lists as pictured, the l2arc_feed_thread() will search from the list heads
5278  * for eligible buffers, greatly increasing its chance of finding them.
5279  *
5280  * The L2ARC device write speed is also boosted during this time so that
5281  * the L2ARC warms up faster.  Since there have been no ARC evictions yet,
5282  * there are no L2ARC reads, and no fear of degrading read performance
5283  * through increased writes.
5284  *
5285  * 6. Writes to the L2ARC devices are grouped and sent in-sequence, so that
5286  * the vdev queue can aggregate them into larger and fewer writes.  Each
5287  * device is written to in a rotor fashion, sweeping writes through
5288  * available space then repeating.
5289  *
5290  * 7. The L2ARC does not store dirty content.  It never needs to flush
5291  * write buffers back to disk based storage.
5292  *
5293  * 8. If an ARC buffer is written (and dirtied) which also exists in the
5294  * L2ARC, the now stale L2ARC buffer is immediately dropped.
5295  *
5296  * The performance of the L2ARC can be tweaked by a number of tunables, which
5297  * may be necessary for different workloads:
5298  *
5299  *      l2arc_write_max         max write bytes per interval
5300  *      l2arc_write_boost       extra write bytes during device warmup
5301  *      l2arc_noprefetch        skip caching prefetched buffers
5302  *      l2arc_headroom          number of max device writes to precache
5303  *      l2arc_headroom_boost    when we find compressed buffers during ARC
5304  *                              scanning, we multiply headroom by this
5305  *                              percentage factor for the next scan cycle,
5306  *                              since more compressed buffers are likely to
5307  *                              be present
5308  *      l2arc_feed_secs         seconds between L2ARC writing
5309  *
5310  * Tunables may be removed or added as future performance improvements are
5311  * integrated, and also may become zpool properties.
5312  *
5313  * There are three key functions that control how the L2ARC warms up:
5314  *
5315  *      l2arc_write_eligible()  check if a buffer is eligible to cache
5316  *      l2arc_write_size()      calculate how much to write
5317  *      l2arc_write_interval()  calculate sleep delay between writes
5318  *
5319  * These three functions determine what to write, how much, and how quickly
5320  * to send writes.
5321  */

5323 static boolean_t
5324 l2arc_write_eligible(uint64_t spa_guid, arc_buf_hdr_t *hdr)
5325 {
5326         /*
5327          * A buffer is *not* eligible for the L2ARC if it:
5328          * 1. belongs to a different spa.
5329          * 2. is already cached on the L2ARC.
5330          * 3. has an I/O in progress (it may be an incomplete read).
```

```
5331          * 4. is flagged not eligible (zfs property).
5332          */
5333         if (hdr->b_spa != spa_guid || HDR_HAS_L2HDR(hdr) ||
5334             HDR_IO_IN_PROGRESS(hdr) || !HDR_L2CACHE(hdr))
5335                 return (B_FALSE);
5336
5337         return (B_TRUE);
5338 }
5339
5340 static uint64_t
5341 l2arc_write_size(void)
5342 {
5343         uint64_t size;
5344
5345         /*
5346          * Make sure our globals have meaningful values in case the user
5347          * altered them.
5348          */
5349         size = l2arc_write_max;
5350         if (size == 0) {
5351                 cmn_err(CE_NOTE, "Bad value for l2arc_write_max, value must "
5352                     "be greater than zero, resetting it to the default (%d)",
5353                     L2ARC_WRITE_SIZE);
5354                 size = l2arc_write_max = L2ARC_WRITE_SIZE;
5355         }
5356
5357         if (arc_warm == B_FALSE)
5358                 size += l2arc_write_boost;
5359
5360         return (size);
5361
5362 }
5363
5364 static clock_t
5365 l2arc_write_interval(clock_t began, uint64_t wanted, uint64_t wrote)
5366 {
5367         clock_t interval, next, now;
5368
5369         /*
5370          * If the ARC lists are busy, increase our write rate; if the
5371          * lists are stale, idle back.  This is achieved by checking
5372          * how much we previously wrote - if it was more than half of
5373          * what we wanted, schedule the next write much sooner.
5374          */
5375         if (l2arc_feed_again && wrote > (wanted / 2))
5376                 interval = (hz * l2arc_feed_min_ms) / 1000;
5377         else
5378                 interval = hz * l2arc_feed_secs;
5379
5380         now = ddi_get_lbolt();
5381         next = MAX(now, MIN(now + interval, began + interval));
5382
5383         return (next);
5384 }
5385
5386 /*
5387  * Cycle through L2ARC devices.  This is how L2ARC load balances.
5388  * If a device is returned, this also returns holding the spa config lock.
5389  */
5390 static l2arc_dev_t *
5391 l2arc_dev_get_next(void)
5392 {
5393         l2arc_dev_t *first, *next = NULL;
5394
5395         /*
5396          * Lock out the removal of spas (spa_namespace_lock), then removal
```



```
5397          * of cache devices (l2arc_dev_mtx).  Once a device has been selected,
5398          * both locks will be dropped and a spa config lock held instead.
5399          */
5400         mutex_enter(&spa_namespace_lock);
5401         mutex_enter(&l2arc_dev_mtx);
5402
5403         /* if there are no vdevs, there is nothing to do */
5404         if (l2arc_ndev == 0)
5405                 goto out;
5406
5407         first = NULL;
5408         next = l2arc_dev_last;
5409         do {
5410                 /* loop around the list looking for a non-faulted vdev */
5411                 if (next == NULL) {
5412                         next = list_head(l2arc_dev_list);
5413                 } else {
5414                         next = list_next(l2arc_dev_list, next);
5415                         if (next == NULL)
5416                                 next = list_head(l2arc_dev_list);
5417                 }
5418
5419                 /* if we have come back to the start, bail out */
5420                 if (first == NULL)
5421                         first = next;
5422                 else if (next == first)
5423                         break;
5424
5425         } while (vdev_is_dead(next->l2ad_vdev));
5426
5427         /* if we were unable to find any usable vdevs, return NULL */
5428         if (vdev_is_dead(next->l2ad_vdev))
5429                 next = NULL;
5430
5431         l2arc_dev_last = next;
5432
5433 out:
5434         mutex_exit(&l2arc_dev_mtx);
5435
5436         /*
5437          * Grab the config lock to prevent the 'next' device from being
5438          * removed while we are writing to it.
5439          */
5440         if (next != NULL)
5441                 spa_config_enter(next->l2ad_spa, SCL_L2ARC, next, RW_READER);
5442         mutex_exit(&spa_namespace_lock);
5443
5444         return (next);
5445 }
5446
5447 /*
5448  * Free buffers that were tagged for destruction.
5449  */
5450 static void
5451 l2arc_do_free_on_write()
5452 {
5453         list_t *buflist;
5454         l2arc_data_free_t *df, *df_prev;
5455
5456         mutex_enter(&l2arc_free_on_write_mtx);
5457         buflist = l2arc_free_on_write;
5458
5459         for (df = list_tail(buflist); df; df = df_prev) {
5460                 df_prev = list_prev(buflist, df);
5461                 ASSERT(df->l2df_data != NULL);
5462                 ASSERT(df->l2df_func != NULL);
```

```
5463                 df->l2df_func(df->l2df_data, df->l2df_size);
5464                 list_remove(buflist, df);
5465                 kmem_free(df, sizeof (l2arc_data_free_t));
5466         }

5468         mutex_exit(&l2arc_free_on_write_mtx);
5469 }

5471 /*
5472  * A write to a cache device has completed.  Update all headers to allow
5473  * reads from these buffers to begin.
5474  */
5475 static void
5476 l2arc_write_done(zio_t *zio)
5477 {
5478         l2arc_write_callback_t *cb;
5479         l2arc_dev_t *dev;
5480         list_t *buflist;
5481         arc_buf_hdr_t *head, *hdr, *hdr_prev;
5482         kmutex_t *hash_lock;
5483         int64_t bytes_dropped = 0;

5485         cb = zio->io_private;
5486         ASSERT(cb != NULL);
5487         dev = cb->l2wcb_dev;
5488         ASSERT(dev != NULL);
5489         head = cb->l2wcb_head;
5490         ASSERT(head != NULL);
5491         buflist = &dev->l2ad_buflist;
5492         ASSERT(buflist != NULL);
5493         DTRACE_PROBE2(l2arc__iodone, zio_t *, zio,
5494             l2arc_write_callback_t *, cb);

5496         if (zio->io_error != 0)
5497                 ARCSTAT_BUMP(arcstat_l2_writes_error);

5499         /*
5500          * All writes completed, or an error was hit.
5501          */
5502 top:
5503         mutex_enter(&dev->l2ad_mtx);
5504         for (hdr = list_prev(buflist, head); hdr; hdr = hdr_prev) {
5505                 hdr_prev = list_prev(buflist, hdr);

5507                 hash_lock = HDR_LOCK(hdr);

5509                 /*
5510                  * We cannot use mutex_enter or else we can deadlock
5511                  * with l2arc_write_buffers (due to swapping the order
5512                  * the hash lock and l2ad_mtx are taken).
5513                  */
5514                 if (!mutex_tryenter(hash_lock)) {
5515                         /*
5516                          * Missed the hash lock. We must retry so we
5517                          * don't leave the ARC_FLAG_L2_WRITING bit set.
5518                          */
5519                         ARCSTAT_BUMP(arcstat_l2_writes_lock_retry);

5521                         /*
5522                          * We don't want to rescan the headers we've
5523                          * already marked as having been written out, so
5524                          * we reinsert the head node so we can pick up
5525                          * where we left off.
5526                          */
5527                         list_remove(buflist, head);
5528                         list_insert_after(buflist, hdr, head);
```



```
5530                         mutex_exit(&dev->l2ad_mtx);

5532                         /*
5533                          * We wait for the hash lock to become available
5534                          * to try and prevent busy waiting, and increase
5535                          * the chance we'll be able to acquire the lock
5536                          * the next time around.
5537                          */
5538                         mutex_enter(hash_lock);
5539                         mutex_exit(hash_lock);
5540                         goto top;
5541                 }

5543                 /*
5544                  * We could not have been moved into the arc_l2c_only
5545                  * state while in-flight due to our ARC_FLAG_L2_WRITING
5546                  * bit being set. Let's just ensure that's being enforced.
5547                  */
5548                 ASSERT(HDR_HAS_L1HDR(hdr));

5550                 /*
5551                  * We may have allocated a buffer for L2ARC compression,
5552                  * we must release it to avoid leaking this data.
5553                  */
5554                 l2arc_release_cdata_buf(hdr);

5556                 if (zio->io_error != 0) {
5557                         /*
5558                          * Error - drop L2ARC entry.
5559                          */
5560                         list_remove(buflist, hdr);
5561                         hdr->b_flags &= ~ARC_FLAG_HAS_L2HDR;

5563                         ARCSTAT_INCR(arcstat_l2_asize, -hdr->b_l2hdr.b_asize);
5564                         ARCSTAT_INCR(arcstat_l2_size, -hdr->b_size);

5566                         bytes_dropped += hdr->b_l2hdr.b_asize;
5567                         (void) refcount_remove_many(&dev->l2ad_alloc,
5568                             hdr->b_l2hdr.b_asize, hdr);
5569                 }

5571                 /*
5572                  * Allow ARC to begin reads and ghost list evictions to
5573                  * this L2ARC entry.
5574                  */
5575                 hdr->b_flags &= ~ARC_FLAG_L2_WRITING;

5577                 mutex_exit(hash_lock);
5578         }

5580         atomic_inc_64(&l2arc_writes_done);
5581         list_remove(buflist, head);
5582         ASSERT(!HDR_HAS_L1HDR(head));
5583         kmem_cache_free(hdr_l2only_cache, head);
5584         mutex_exit(&dev->l2ad_mtx);

5586         vdev_space_update(dev->l2ad_vdev, -bytes_dropped, 0, 0);

5588         l2arc_do_free_on_write();

5590         kmem_free(cb, sizeof (l2arc_write_callback_t));
5591 }

5593 /*
5594  * A read to a cache device completed.  Validate buffer contents before
```

```
5595  * handing over to the regular ARC routines.
5596  */
5597 static void
5598 l2arc_read_done(zio_t *zio)
5599 {
5600         l2arc_read_callback_t *cb;
5601         arc_buf_hdr_t *hdr;
5602         arc_buf_t *buf;
5603         kmutex_t *hash_lock;
5604         int equal;

5606         ASSERT(zio->io_vd != NULL);
5607         ASSERT(zio->io_flags & ZIO_FLAG_DONT_PROPAGATE);

5609         spa_config_exit(zio->io_spa, SCL_L2ARC, zio->io_vd);

5611         cb = zio->io_private;
5612         ASSERT(cb != NULL);
5613         buf = cb->l2rcb_buf;
5614         ASSERT(buf != NULL);

5616         hash_lock = HDR_LOCK(buf->b_hdr);
5617         mutex_enter(hash_lock);
5618         hdr = buf->b_hdr;
5619         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));

5621         /*
5622          * If the buffer was compressed, decompress it first.
5623          */
5624         if (cb->l2rcb_compress != ZIO_COMPRESS_OFF)
5625                 l2arc_decompress_zio(zio, hdr, cb->l2rcb_compress);
5626         ASSERT(zio->io_data != NULL);
5627         ASSERT3U(zio->io_size, ==, hdr->b_size);
5628         ASSERT3U(BP_GET_LSIZE(&cb->l2rcb_bp), ==, hdr->b_size);

5630         /*
5631          * Check this survived the L2ARC journey.
5632          */
5633         equal = arc_cksum_equal(buf);
5634         if (equal && zio->io_error == 0 && !HDR_L2_EVICTED(hdr)) {
5635                 mutex_exit(hash_lock);
5636                 zio->io_private = buf;
5637                 zio->io_bp_copy = cb->l2rcb_bp; /* XXX fix in L2ARC 2.0 */
5638                 zio->io_bp = &zio->io_bp_copy;  /* XXX fix in L2ARC 2.0 */
5639                 arc_read_done(zio);
5640         } else {
5641                 mutex_exit(hash_lock);
5642                 /*
5643                  * Buffer didn't survive caching.  Increment stats and
5644                  * reissue to the original storage device.
5645                  */
5646                 if (zio->io_error != 0) {
5647                         ARCSTAT_BUMP(arcstat_l2_io_error);
5648                 } else {
5649                         zio->io_error = SET_ERROR(EIO);
5650                 }
5651                 if (!equal)
5652                         ARCSTAT_BUMP(arcstat_l2_cksum_bad);

5654                 /*
5655                  * If there's no waiter, issue an async i/o to the primary
5656                  * storage now.  If there *is* a waiter, the caller must
5657                  * issue the i/o in a context where it's OK to block.
5658                  */
5659                 if (zio->io_waiter == NULL) {
5660                         zio_t *pio = zio_unique_parent(zio);
```



```
5662                         ASSERT(!pio || pio->io_child_type == ZIO_CHILD_LOGICAL);

5664                         zio_nowait(zio_read(pio, cb->l2rcb_spa, &cb->l2rcb_bp,
5665                             buf->b_data, hdr->b_size, arc_read_done, buf,
5666                             zio->io_priority, cb->l2rcb_flags, &cb->l2rcb_zb));
5667                 }
5668         }

5670         kmem_free(cb, sizeof (l2arc_read_callback_t));
5671 }

5673 /*
5674  * This is the list priority from which the L2ARC will search for pages to
5675  * cache.  This is used within loops (0..3) to cycle through lists in the
5676  * desired order.  This order can have a significant effect on cache
5677  * performance.
5678  *
5679  * Currently the metadata lists are hit first, MFU then MRU, followed by
5680  * the data lists.  This function returns a locked list, and also returns
5681  * the lock pointer.
5682  */
5683 static multilist_sublist_t *
5684 l2arc_sublist_lock(int list_num)
5685 {
5686         multilist_t *ml = NULL;
5687         unsigned int idx;

5689         ASSERT(list_num >= 0 && list_num <= 3);

5691         switch (list_num) {
5692         case 0:
5693                 ml = &arc_mfu->arcs_list[ARC_BUFC_METADATA];
5694                 break;
5695         case 1:
5696                 ml = &arc_mru->arcs_list[ARC_BUFC_METADATA];
5697                 break;
5698         case 2:
5699                 ml = &arc_mfu->arcs_list[ARC_BUFC_DATA];
5700                 break;
5701         case 3:
5702                 ml = &arc_mru->arcs_list[ARC_BUFC_DATA];
5703                 break;
5704         }

5706         /*
5707          * Return a randomly-selected sublist. This is acceptable
5708          * because the caller feeds only a little bit of data for each
5709          * call (8MB). Subsequent calls will result in different
5710          * sublists being selected.
5711          */
5712         idx = multilist_get_random_index(ml);
5713         return (multilist_sublist_lock(ml, idx));
5714 }

5716 /*
5717  * Evict buffers from the device write hand to the distance specified in
5718  * bytes.  This distance may span populated buffers, it may span nothing.
5719  * This is clearing a region on the L2ARC device ready for writing.
5720  * If the 'all' boolean is set, every buffer is evicted.
5721  */
5722 static void
5723 l2arc_evict(l2arc_dev_t *dev, uint64_t distance, boolean_t all)
5724 {
5725         list_t *buflist;
5726         arc_buf_hdr_t *hdr, *hdr_prev;
```

```
5727         kmutex_t *hash_lock;
5728         uint64_t taddr;

5730         buflist = &dev->l2ad_buflist;

5732         if (!all && dev->l2ad_first) {
5733                 /*
5734                  * This is the first sweep through the device.  There is
5735                  * nothing to evict.
5736                  */
5737                 return;
5738         }

5740         if (dev->l2ad_hand >= (dev->l2ad_end - (2 * distance))) {
5741                 /*
5742                  * When nearing the end of the device, evict to the end
5743                  * before the device write hand jumps to the start.
5744                  */
5745                 taddr = dev->l2ad_end;
5746         } else {
5747                 taddr = dev->l2ad_hand + distance;
5748         }
5749         DTRACE_PROBE4(l2arc__evict, l2arc_dev_t *, dev, list_t *, buflist,
5750             uint64_t, taddr, boolean_t, all);

5752 top:
5753         mutex_enter(&dev->l2ad_mtx);
5754         for (hdr = list_tail(buflist); hdr; hdr = hdr_prev) {
5755                 hdr_prev = list_prev(buflist, hdr);

5757                 hash_lock = HDR_LOCK(hdr);

5759                 /*
5760                  * We cannot use mutex_enter or else we can deadlock
5761                  * with l2arc_write_buffers (due to swapping the order
5762                  * the hash lock and l2ad_mtx are taken).
5763                  */
5764                 if (!mutex_tryenter(hash_lock)) {
5765                         /*
5766                          * Missed the hash lock.  Retry.
5767                          */
5768                         ARCSTAT_BUMP(arcstat_l2_evict_lock_retry);
5769                         mutex_exit(&dev->l2ad_mtx);
5770                         mutex_enter(hash_lock);
5771                         mutex_exit(hash_lock);
5772                         goto top;
5773                 }

5775                 if (HDR_L2_WRITE_HEAD(hdr)) {
5776                         /*
5777                          * We hit a write head node.  Leave it for
5778                          * l2arc_write_done().
5779                          */
5780                         list_remove(buflist, hdr);
5781                         mutex_exit(hash_lock);
5782                         continue;
5783                 }

5785                 if (!all && HDR_HAS_L2HDR(hdr) &&
5786                     (hdr->b_l2hdr.b_daddr > taddr ||
5787                     hdr->b_l2hdr.b_daddr < dev->l2ad_hand)) {
5788                         /*
5789                          * We've evicted to the target address,
5790                          * or the end of the device.
5791                          */
5792                         mutex_exit(hash_lock);
```



```
5793                         break;
5794                 }

5796                 ASSERT(HDR_HAS_L2HDR(hdr));
5797                 if (!HDR_HAS_L1HDR(hdr)) {
5798                         ASSERT(!HDR_L2_READING(hdr));
5799                         /*
5800                          * This doesn't exist in the ARC.  Destroy.
5801                          * arc_hdr_destroy() will call list_remove()
5802                          * and decrement arcstat_l2_size.
5803                          */
5804                         arc_change_state(arc_anon, hdr, hash_lock);
5805                         arc_hdr_destroy(hdr);
5806                 } else {
5807                         ASSERT(hdr->b_l1hdr.b_state != arc_l2c_only);
5808                         ARCSTAT_BUMP(arcstat_l2_evict_l1cached);
5809                         /*
5810                          * Invalidate issued or about to be issued
5811                          * reads, since we may be about to write
5812                          * over this location.
5813                          */
5814                         if (HDR_L2_READING(hdr)) {
5815                                 ARCSTAT_BUMP(arcstat_l2_evict_reading);
5816                                 hdr->b_flags |= ARC_FLAG_L2_EVICTED;
5817                         }

5819                         /* Ensure this header has finished being written */
5820                         ASSERT(!HDR_L2_WRITING(hdr));
5821                         ASSERT3P(hdr->b_l1hdr.b_tmp_cdata, ==, NULL);

5823                         arc_hdr_l2hdr_destroy(hdr);
5824                 }
5825                 mutex_exit(hash_lock);
5826         }
5827         mutex_exit(&dev->l2ad_mtx);
5828 }

5830 /*
5831  * Find and write ARC buffers to the L2ARC device.
5832  *
5833  * An ARC_FLAG_L2_WRITING flag is set so that the L2ARC buffers are not valid
5834  * for reading until they have completed writing.
5835  * The headroom_boost is an in-out parameter used to maintain headroom boost
5836  * state between calls to this function.
5837  *
5838  * Returns the number of bytes actually written (which may be smaller than
5839  * the delta by which the device hand has changed due to alignment).
5840  */
5841 static uint64_t
5842 l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz,
5843     boolean_t *headroom_boost)
5844 {
5845         arc_buf_hdr_t *hdr, *hdr_prev, *head;
5846         uint64_t write_asize, write_psize, write_sz, headroom,
5847             buf_compress_minsz;
5848         void *buf_data;
5849         boolean_t full;
5850         l2arc_write_callback_t *cb;
5851         zio_t *pio, *wzio;
5852         uint64_t guid = spa_load_guid(spa);
5853         const boolean_t do_headroom_boost = *headroom_boost;

5855         ASSERT(dev->l2ad_vdev != NULL);

5857         /* Lower the flag now, we might want to raise it again later. */
5858         *headroom_boost = B_FALSE;
```

```
5860         pio = NULL;
5861         write_sz = write_asize = write_psize = 0;
5862         full = B_FALSE;
5863         head = kmem_cache_alloc(hdr_l2only_cache, KM_PUSHPAGE);
5864         head->b_flags |= ARC_FLAG_L2_WRITE_HEAD;
5865         head->b_flags |= ARC_FLAG_HAS_L2HDR;

5867         /*
5868          * We will want to try to compress buffers that are at least 2x the
5869          * device sector size.
5870          */
5871         buf_compress_minsz = 2 << dev->l2ad_vdev->vdev_ashift;

5873         /*
5874          * Copy buffers for L2ARC writing.
5875          */
5876         for (int try = 0; try <= 3; try++) {
5877                 multilist_sublist_t *mls = l2arc_sublist_lock(try);
5878                 uint64_t passed_sz = 0;

5880                 /*
5881                  * L2ARC fast warmup.
5882                  *
5883                  * Until the ARC is warm and starts to evict, read from the
5884                  * head of the ARC lists rather than the tail.
5885                  */
5886                 if (arc_warm == B_FALSE)
5887                         hdr = multilist_sublist_head(mls);
5888                 else
5889                         hdr = multilist_sublist_tail(mls);

5891                 headroom = target_sz * l2arc_headroom;
5892                 if (do_headroom_boost)
5893                         headroom = (headroom * l2arc_headroom_boost) / 100;

5895                 for (; hdr; hdr = hdr_prev) {
5896                         kmutex_t *hash_lock;
5897                         uint64_t buf_sz;

5899                         if (arc_warm == B_FALSE)
5900                                 hdr_prev = multilist_sublist_next(mls, hdr);
5901                         else
5902                                 hdr_prev = multilist_sublist_prev(mls, hdr);

5904                         hash_lock = HDR_LOCK(hdr);
5905                         if (!mutex_tryenter(hash_lock)) {
5906                                 /*
5907                                  * Skip this buffer rather than waiting.
5908                                  */
5909                                 continue;
5910                         }

5912                         passed_sz += hdr->b_size;
5913                         if (passed_sz > headroom) {
5914                                 /*
5915                                  * Searched too far.
5916                                  */
5917                                 mutex_exit(hash_lock);
5918                                 break;
5919                         }

5921                         if (!l2arc_write_eligible(guid, hdr)) {
5922                                 mutex_exit(hash_lock);
5923                                 continue;
5924                         }
```



```
5926                         if ((write_sz + hdr->b_size) > target_sz) {
5927                                 full = B_TRUE;
5928                                 mutex_exit(hash_lock);
5929                                 break;
5930                         }

5932                         if (pio == NULL) {
5933                                 /*
5934                                  * Insert a dummy header on the buflist so
5935                                  * l2arc_write_done() can find where the
5936                                  * write buffers begin without searching.
5937                                  */
5938                                 mutex_enter(&dev->l2ad_mtx);
5939                                 list_insert_head(&dev->l2ad_buflist, head);
5940                                 mutex_exit(&dev->l2ad_mtx);

5942                                 cb = kmem_alloc(
5943                                     sizeof (l2arc_write_callback_t), KM_SLEEP);
5944                                 cb->l2wcb_dev = dev;
5945                                 cb->l2wcb_head = head;
5946                                 pio = zio_root(spa, l2arc_write_done, cb,
5947                                     ZIO_FLAG_CANFAIL);
5948                         }

5950                         /*
5951                          * Create and add a new L2ARC header.
5952                          */
5953                         hdr->b_l2hdr.b_dev = dev;
5954                         hdr->b_flags |= ARC_FLAG_L2_WRITING;
5955                         /*
5956                          * Temporarily stash the data buffer in b_tmp_cdata.
5957                          * The subsequent write step will pick it up from
5958                          * there. This is because can't access b_l1hdr.b_buf
5959                          * without holding the hash_lock, which we in turn
5960                          * can't access without holding the ARC list locks
5961                          * (which we want to avoid during compression/writing).
5962                          */
5963                         hdr->b_l2hdr.b_compress = ZIO_COMPRESS_OFF;
5964                         hdr->b_l2hdr.b_asize = hdr->b_size;
5965                         hdr->b_l1hdr.b_tmp_cdata = hdr->b_l1hdr.b_buf->b_data;

5967                         /*
5968                          * Explicitly set the b_daddr field to a known
5969                          * value which means "invalid address". This
5970                          * enables us to differentiate which stage of
5971                          * l2arc_write_buffers() the particular header
5972                          * is in (e.g. this loop, or the one below).
5973                          * ARC_FLAG_L2_WRITING is not enough to make
5974                          * this distinction, and we need to know in
5975                          * order to do proper l2arc vdev accounting in
5976                          * arc_release() and arc_hdr_destroy().
5977                          *
5978                          * Note, we can't use a new flag to distinguish
5979                          * the two stages because we don't hold the
5980                          * header's hash_lock below, in the second stage
5981                          * of this function. Thus, we can't simply
5982                          * change the b_flags field to denote that the
5983                          * IO has been sent. We can change the b_daddr
5984                          * field of the L2 portion, though, since we'll
5985                          * be holding the l2ad_mtx; which is why we're
5986                          * using it to denote the header's state change.
5987                          */
5988                         hdr->b_l2hdr.b_daddr = L2ARC_ADDR_UNSET;

5990                         buf_sz = hdr->b_size;
```

```
5991                        hdr->b_flags |= ARC_FLAG_HAS_L2HDR;

5993                        mutex_enter(&dev->l2ad_mtx);
5994                        list_insert_head(&dev->l2ad_buflist, hdr);
5995                        mutex_exit(&dev->l2ad_mtx);

5997                        /*
5998                         * Compute and store the buffer cksum before
5999                         * writing.  On debug the cksum is verified first.
6000                         */
6001                        arc_cksum_verify(hdr->b_l1hdr.b_buf);
6002                        arc_cksum_compute(hdr->b_l1hdr.b_buf, B_TRUE);

6004                        mutex_exit(hash_lock);

6006                        write_sz += buf_sz;
6007                }

6009                multilist_sublist_unlock(mls);

6011                if (full == B_TRUE)
6012                        break;
6013        }

6015        /* No buffers selected for writing? */
6016        if (pio == NULL) {
6017                ASSERT0(write_sz);
6018                ASSERT(!HDR_HAS_L1HDR(head));
6019                kmem_cache_free(hdr_l2only_cache, head);
6020                return (0);
6021        }

6023        mutex_enter(&dev->l2ad_mtx);

6025        /*
6026         * Now start writing the buffers. We're starting at the write head
6027         * and work backwards, retracing the course of the buffer selector
6028         * loop above.
6029         */
6030        for (hdr = list_prev(&dev->l2ad_buflist, head); hdr;
6031            hdr = list_prev(&dev->l2ad_buflist, hdr)) {
6032                uint64_t buf_sz;

6034                /*
6035                 * We rely on the L1 portion of the header below, so
6036                 * it's invalid for this header to have been evicted out
6037                 * of the ghost cache, prior to being written out. The
6038                 * ARC_FLAG_L2_WRITING bit ensures this won't happen.
6039                 */
6040                ASSERT(HDR_HAS_L1HDR(hdr));

6042                /*
6043                 * We shouldn't need to lock the buffer here, since we flagged
6044                 * it as ARC_FLAG_L2_WRITING in the previous step, but we must
6045                 * take care to only access its L2 cache parameters. In
6046                 * particular, hdr->l1hdr.b_buf may be invalid by now due to
6047                 * ARC eviction.
6048                 */
6049                hdr->b_l2hdr.b_daddr = dev->l2ad_hand;

6051                if ((HDR_L2COMPRESS(hdr)) &&
6052                    hdr->b_l2hdr.b_asize >= buf_compress_minsz) {
6053                        if (l2arc_compress_buf(hdr)) {
6054                                /*
6055                                 * If compression succeeded, enable headroom
6056                                 * boost on the next scan cycle.
```



```
6057                                 */
6058                                *headroom_boost = B_TRUE;
6059                        }
6060                }

6062                /*
6063                 * Pick up the buffer data we had previously stashed away
6064                 * (and now potentially also compressed).
6065                 */
6066                buf_data = hdr->b_l1hdr.b_tmp_cdata;
6067                buf_sz = hdr->b_l2hdr.b_asize;

6069                /*
6070                 * We need to do this regardless if buf_sz is zero or
6071                 * not, otherwise, when this l2hdr is evicted we'll
6072                 * remove a reference that was never added.
6073                 */
6074                (void) refcount_add_many(&dev->l2ad_alloc, buf_sz, hdr);

6076                /* Compression may have squashed the buffer to zero length. */
6077                if (buf_sz != 0) {
6078                        uint64_t buf_p_sz;

6080                        wzio = zio_write_phys(pio, dev->l2ad_vdev,
6081                            dev->l2ad_hand, buf_sz, buf_data, ZIO_CHECKSUM_OFF,
6082                            NULL, NULL, ZIO_PRIORITY_ASYNC_WRITE,
6083                            ZIO_FLAG_CANFAIL, B_FALSE);

6085                        DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
6086                            zio_t *, wzio);
6087                        (void) zio_nowait(wzio);

6089                        write_asize += buf_sz;

6091                        /*
6092                         * Keep the clock hand suitably device-aligned.
6093                         */
6094                        buf_p_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
6095                        write_psize += buf_p_sz;
6096                        dev->l2ad_hand += buf_p_sz;
6097                }
6098        }

6100        mutex_exit(&dev->l2ad_mtx);

6102        ASSERT3U(write_asize, <=, target_sz);
6103        ARCSTAT_BUMP(arcstat_l2_writes_sent);
6104        ARCSTAT_INCR(arcstat_l2_write_bytes, write_asize);
6105        ARCSTAT_INCR(arcstat_l2_size, write_sz);
6106        ARCSTAT_INCR(arcstat_l2_asize, write_asize);
6107        vdev_space_update(dev->l2ad_vdev, write_asize, 0, 0);

6109        /*
6110         * Bump device hand to the device start if it is approaching the end.
6111         * l2arc_evict() will already have evicted ahead for this case.
6112         */
6113        if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
6114                dev->l2ad_hand = dev->l2ad_start;
6115                dev->l2ad_first = B_FALSE;
6116        }

6118        dev->l2ad_writing = B_TRUE;
6119        (void) zio_wait(pio);
6120        dev->l2ad_writing = B_FALSE;

6122        return (write_asize);
```

```
6123 }
6124
6125 /*
6126  * Compresses an L2ARC buffer.
6127  * The data to be compressed must be prefilled in l1hdr.b_tmp_cdata and its
6128  * size in l2hdr->b_asize. This routine tries to compress the data and
6129  * depending on the compression result there are three possible outcomes:
6130  * *) The buffer was incompressible. The original l2hdr contents were left
6131  *    untouched and are ready for writing to an L2 device.
6132  * *) The buffer was all-zeros, so there is no need to write it to an L2
6133  *    device. To indicate this situation b_tmp_cdata is NULL'ed, b_asize is
6134  *    set to zero and b_compress is set to ZIO_COMPRESS_EMPTY.
6135  * *) Compression succeeded and b_tmp_cdata was replaced with a temporary
6136  *    data buffer which holds the compressed data to be written, and b_asize
6137  *    tells us how much data there is. b_compress is set to the appropriate
6138  *    compression algorithm. Once writing is done, invoke
6139  *    l2arc_release_cdata_buf on this l2hdr to free this temporary buffer.
6140  *
6141  * Returns B_TRUE if compression succeeded, or B_FALSE if it didn't (the
6142  * buffer was incompressible).
6143  */
6144 static boolean_t
6145 l2arc_compress_buf(arc_buf_hdr_t *hdr)
6146 {
6147         void *cdata;
6148         size_t csize, len, rounded;
6149         ASSERT(HDR_HAS_L2HDR(hdr));
6150         l2arc_buf_hdr_t *l2hdr = &hdr->b_l2hdr;
6151
6152         ASSERT(HDR_HAS_L1HDR(hdr));
6153         ASSERT(l2hdr->b_compress == ZIO_COMPRESS_OFF);
6154         ASSERT(hdr->b_l1hdr.b_tmp_cdata != NULL);
6155
6156         len = l2hdr->b_asize;
6157         cdata = zio_data_buf_alloc(len);
6158         ASSERT3P(cdata, !=, NULL);
6159         csize = zio_compress_data(ZIO_COMPRESS_LZ4, hdr->b_l1hdr.b_tmp_cdata,
6160             cdata, l2hdr->b_asize);
6161
6162         rounded = P2ROUNDUP(csize, (size_t)SPA_MINBLOCKSIZE);
6163         if (rounded > csize) {
6164                 bzero((char *)cdata + csize, rounded - csize);
6165                 csize = rounded;
6166         }
6167
6168         if (csize == 0) {
6169                 /* zero block, indicate that there's nothing to write */
6170                 zio_data_buf_free(cdata, len);
6171                 l2hdr->b_compress = ZIO_COMPRESS_EMPTY;
6172                 l2hdr->b_asize = 0;
6173                 hdr->b_l1hdr.b_tmp_cdata = NULL;
6174                 ARCSTAT_BUMP(arcstat_l2_compress_zeros);
6175                 return (B_TRUE);
6176         } else if (csize > 0 && csize < len) {
6177                 /*
6178                  * Compression succeeded, we'll keep the cdata around for
6179                  * writing and release it afterwards.
6180                  */
6181                 l2hdr->b_compress = ZIO_COMPRESS_LZ4;
6182                 l2hdr->b_asize = csize;
6183                 hdr->b_l1hdr.b_tmp_cdata = cdata;
6184                 ARCSTAT_BUMP(arcstat_l2_compress_successes);
6185                 return (B_TRUE);
6186         } else {
6187                 /*
6188                  * Compression failed, release the compressed buffer.
```



```
6189                  * l2hdr will be left unmodified.
6190                  */
6191                 zio_data_buf_free(cdata, len);
6192                 ARCSTAT_BUMP(arcstat_l2_compress_failures);
6193                 return (B_FALSE);
6194         }
6195 }
6196
6197 /*
6198  * Decompresses a zio read back from an l2arc device. On success, the
6199  * underlying zio's io_data buffer is overwritten by the uncompressed
6200  * version. On decompression error (corrupt compressed stream), the
6201  * zio->io_error value is set to signal an I/O error.
6202  *
6203  * Please note that the compressed data stream is not checksummed, so
6204  * if the underlying device is experiencing data corruption, we may feed
6205  * corrupt data to the decompressor, so the decompressor needs to be
6206  * able to handle this situation (LZ4 does).
6207  */
6208 static void
6209 l2arc_decompress_zio(zio_t *zio, arc_buf_hdr_t *hdr, enum zio_compress c)
6210 {
6211         ASSERT(L2ARC_IS_VALID_COMPRESS(c));
6212
6213         if (zio->io_error != 0) {
6214                 /*
6215                  * An io error has occured, just restore the original io
6216                  * size in preparation for a main pool read.
6217                  */
6218                 zio->io_orig_size = zio->io_size = hdr->b_size;
6219                 return;
6220         }
6221
6222         if (c == ZIO_COMPRESS_EMPTY) {
6223                 /*
6224                  * An empty buffer results in a null zio, which means we
6225                  * need to fill its io_data after we're done restoring the
6226                  * buffer's contents.
6227                  */
6228                 ASSERT(hdr->b_l1hdr.b_buf != NULL);
6229                 bzero(hdr->b_l1hdr.b_buf->b_data, hdr->b_size);
6230                 zio->io_data = zio->io_orig_data = hdr->b_l1hdr.b_buf->b_data;
6231         } else {
6232                 ASSERT(zio->io_data != NULL);
6233                 /*
6234                  * We copy the compressed data from the start of the arc buffer
6235                  * (the zio_read will have pulled in only what we need, the
6236                  * rest is garbage which we will overwrite at decompression)
6237                  * and then decompress back to the ARC data buffer. This way we
6238                  * can minimize copying by simply decompressing back over the
6239                  * original compressed data (rather than decompressing to an
6240                  * aux buffer and then copying back the uncompressed buffer,
6241                  * which is likely to be much larger).
6242                  */
6243                 uint64_t csize;
6244                 void *cdata;
6245
6246                 csize = zio->io_size;
6247                 cdata = zio_data_buf_alloc(csize);
6248                 bcopy(zio->io_data, cdata, csize);
6249                 if (zio_decompress_data(c, cdata, zio->io_data, csize,
6250                     hdr->b_size) != 0)
6251                         zio->io_error = EIO;
6252                 zio_data_buf_free(cdata, csize);
6253         }
```

```
6255         /* Restore the expected uncompressed IO size. */
6256         zio->io_orig_size = zio->io_size = hdr->b_size;
6257 }

6259 /*
6260  * Releases the temporary b_tmp_cdata buffer in an l2arc header structure.
6261  * This buffer serves as a temporary holder of compressed data while
6262  * the buffer entry is being written to an l2arc device. Once that is
6263  * done, we can dispose of it.
6264  */
6265 static void
6266 l2arc_release_cdata_buf(arc_buf_hdr_t *hdr)
6267 {
6268         ASSERT(HDR_HAS_L2HDR(hdr));
6269         enum zio_compress comp = hdr->b_l2hdr.b_compress;

6271         ASSERT(HDR_HAS_L1HDR(hdr));
6272         ASSERT(comp == ZIO_COMPRESS_OFF || L2ARC_IS_VALID_COMPRESS(comp));

6274         if (comp == ZIO_COMPRESS_OFF) {
6275                 /*
6276                  * In this case, b_tmp_cdata points to the same buffer
6277                  * as the arc_buf_t's b_data field. We don't want to
6278                  * free it, since the arc_buf_t will handle that.
6279                  */
6280                 hdr->b_l1hdr.b_tmp_cdata = NULL;
6281         } else if (comp == ZIO_COMPRESS_EMPTY) {
6282                 /*
6283                  * In this case, b_tmp_cdata was compressed to an empty
6284                  * buffer, thus there's nothing to free and b_tmp_cdata
6285                  * should have been set to NULL in l2arc_write_buffers().
6286                  */
6287                 ASSERT3P(hdr->b_l1hdr.b_tmp_cdata, ==, NULL);
6288         } else {
6289                 /*
6290                  * If the data was compressed, then we've allocated a
6291                  * temporary buffer for it, so now we need to release it.
6292                  */
6293                 ASSERT(hdr->b_l1hdr.b_tmp_cdata != NULL);
6294                 zio_data_buf_free(hdr->b_l1hdr.b_tmp_cdata,
6295                     hdr->b_size);
6296                 hdr->b_l1hdr.b_tmp_cdata = NULL;
6297         }

6299 }

6301 /*
6302  * This thread feeds the L2ARC at regular intervals.  This is the beating
6303  * heart of the L2ARC.
6304  */
6305 static void
6306 l2arc_feed_thread(void)
6307 {
6308         callb_cpr_t cpr;
6309         l2arc_dev_t *dev;
6310         spa_t *spa;
6311         uint64_t size, wrote;
6312         clock_t begin, next = ddi_get_lbolt();
6313         boolean_t headroom_boost = B_FALSE;

6315         CALLB_CPR_INIT(&cpr, &l2arc_feed_thr_lock, callb_generic_cpr, FTAG);

6317         mutex_enter(&l2arc_feed_thr_lock);

6319         while (l2arc_thread_exit == 0) {
6320                 CALLB_CPR_SAFE_BEGIN(&cpr);
```



```
6321                 (void) cv_timedwait(&l2arc_feed_thr_cv, &l2arc_feed_thr_lock,
6322                     next);
6323                 CALLB_CPR_SAFE_END(&cpr, &l2arc_feed_thr_lock);
6324                 next = ddi_get_lbolt() + hz;

6326                 /*
6327                  * Quick check for L2ARC devices.
6328                  */
6329                 mutex_enter(&l2arc_dev_mtx);
6330                 if (l2arc_ndev == 0) {
6331                         mutex_exit(&l2arc_dev_mtx);
6332                         continue;
6333                 }
6334                 mutex_exit(&l2arc_dev_mtx);
6335                 begin = ddi_get_lbolt();

6337                 /*
6338                  * This selects the next l2arc device to write to, and in
6339                  * doing so the next spa to feed from: dev->l2ad_spa.   This
6340                  * will return NULL if there are now no l2arc devices or if
6341                  * they are all faulted.
6342                  *
6343                  * If a device is returned, its spa's config lock is also
6344                  * held to prevent device removal.  l2arc_dev_get_next()
6345                  * will grab and release l2arc_dev_mtx.
6346                  */
6347                 if ((dev = l2arc_dev_get_next()) == NULL)
6348                         continue;

6350                 spa = dev->l2ad_spa;
6351                 ASSERT(spa != NULL);

6353                 /*
6354                  * If the pool is read-only then force the feed thread to
6355                  * sleep a little longer.
6356                  */
6357                 if (!spa_writeable(spa)) {
6358                         next = ddi_get_lbolt() + 5 * l2arc_feed_secs * hz;
6359                         spa_config_exit(spa, SCL_L2ARC, dev);
6360                         continue;
6361                 }

6363                 /*
6364                  * Avoid contributing to memory pressure.
6365                  */
6366                 if (arc_reclaim_needed()) {
6367                         ARCSTAT_BUMP(arcstat_l2_abort_lowmem);
6368                         spa_config_exit(spa, SCL_L2ARC, dev);
6369                         continue;
6370                 }

6372                 ARCSTAT_BUMP(arcstat_l2_feeds);

6374                 size = l2arc_write_size();

6376                 /*
6377                  * Evict L2ARC buffers that will be overwritten.
6378                  */
6379                 l2arc_evict(dev, size, B_FALSE);

6381                 /*
6382                  * Write ARC buffers.
6383                  */
6384                 wrote = l2arc_write_buffers(spa, dev, size, &headroom_boost);

6386                 /*
```

```
6387                 * Calculate interval between writes.
6388                 */
6389                next = l2arc_write_interval(begin, size, wrote);
6390                spa_config_exit(spa, SCL_L2ARC, dev);
6391        }
6392
6393        l2arc_thread_exit = 0;
6394        cv_broadcast(&l2arc_feed_thr_cv);
6395        CALLB_CPR_EXIT(&cpr);           /* drops l2arc_feed_thr_lock */
6396        thread_exit();
6397 }
6398
6399 boolean_t
6400 l2arc_vdev_present(vdev_t *vd)
6401 {
6402        l2arc_dev_t *dev;
6403
6404        mutex_enter(&l2arc_dev_mtx);
6405        for (dev = list_head(l2arc_dev_list); dev != NULL;
6406            dev = list_next(l2arc_dev_list, dev)) {
6407                if (dev->l2ad_vdev == vd)
6408                        break;
6409        }
6410        mutex_exit(&l2arc_dev_mtx);
6411
6412        return (dev != NULL);
6413 }
6414
6415 /*
6416  * Add a vdev for use by the L2ARC.  By this point the spa has already
6417  * validated the vdev and opened it.
6418  */
6419 void
6420 l2arc_add_vdev(spa_t *spa, vdev_t *vd)
6421 {
6422        l2arc_dev_t *adddev;
6423
6424        ASSERT(!l2arc_vdev_present(vd));
6425
6426        /*
6427         * Create a new l2arc device entry.
6428         */
6429        adddev = kmem_zalloc(sizeof (l2arc_dev_t), KM_SLEEP);
6430        adddev->l2ad_spa = spa;
6431        adddev->l2ad_vdev = vd;
6432        adddev->l2ad_start = VDEV_LABEL_START_SIZE;
6433        adddev->l2ad_end = VDEV_LABEL_START_SIZE + vdev_get_min_asize(vd);
6434        adddev->l2ad_hand = adddev->l2ad_start;
6435        adddev->l2ad_first = B_TRUE;
6436        adddev->l2ad_writing = B_FALSE;
6437
6438        mutex_init(&adddev->l2ad_mtx, NULL, MUTEX_DEFAULT, NULL);
6439        /*
6440         * This is a list of all ARC buffers that are still valid on the
6441         * device.
6442         */
6443        list_create(&adddev->l2ad_buflist, sizeof (arc_buf_hdr_t),
6444            offsetof(arc_buf_hdr_t, b_l2hdr.b_l2node));
6445
6446        vdev_space_update(vd, 0, 0, adddev->l2ad_end - adddev->l2ad_hand);
6447        refcount_create(&adddev->l2ad_alloc);
6448
6449        /*
6450         * Add device to global list
6451         */
6452        mutex_enter(&l2arc_dev_mtx);
```



```
6453        list_insert_head(l2arc_dev_list, adddev);
6454        atomic_inc_64(&l2arc_ndev);
6455        mutex_exit(&l2arc_dev_mtx);
6456 }
6457
6458 /*
6459  * Remove a vdev from the L2ARC.
6460  */
6461 void
6462 l2arc_remove_vdev(vdev_t *vd)
6463 {
6464        l2arc_dev_t *dev, *nextdev, *remdev = NULL;
6465
6466        /*
6467         * Find the device by vdev
6468         */
6469        mutex_enter(&l2arc_dev_mtx);
6470        for (dev = list_head(l2arc_dev_list); dev; dev = nextdev) {
6471                nextdev = list_next(l2arc_dev_list, dev);
6472                if (vd == dev->l2ad_vdev) {
6473                        remdev = dev;
6474                        break;
6475                }
6476        }
6477        ASSERT(remdev != NULL);
6478
6479        /*
6480         * Remove device from global list
6481         */
6482        list_remove(l2arc_dev_list, remdev);
6483        l2arc_dev_last = NULL;          /* may have been invalidated */
6484        atomic_dec_64(&l2arc_ndev);
6485        mutex_exit(&l2arc_dev_mtx);
6486
6487        /*
6488         * Clear all buflists and ARC references.  L2ARC device flush.
6489         */
6490        l2arc_evict(remdev, 0, B_TRUE);
6491        list_destroy(&remdev->l2ad_buflist);
6492        mutex_destroy(&remdev->l2ad_mtx);
6493        refcount_destroy(&remdev->l2ad_alloc);
6494        kmem_free(remdev, sizeof (l2arc_dev_t));
6495 }
6496
6497 void
6498 l2arc_init(void)
6499 {
6500        l2arc_thread_exit = 0;
6501        l2arc_ndev = 0;
6502        l2arc_writes_sent = 0;
6503        l2arc_writes_done = 0;
6504
6505        mutex_init(&l2arc_feed_thr_lock, NULL, MUTEX_DEFAULT, NULL);
6506        cv_init(&l2arc_feed_thr_cv, NULL, CV_DEFAULT, NULL);
6507        mutex_init(&l2arc_dev_mtx, NULL, MUTEX_DEFAULT, NULL);
6508        mutex_init(&l2arc_free_on_write_mtx, NULL, MUTEX_DEFAULT, NULL);
6509
6510        l2arc_dev_list = &L2ARC_dev_list;
6511        l2arc_free_on_write = &L2ARC_free_on_write;
6512        list_create(l2arc_dev_list, sizeof (l2arc_dev_t),
6513            offsetof(l2arc_dev_t, l2ad_node));
6514        list_create(l2arc_free_on_write, sizeof (l2arc_data_free_t),
6515            offsetof(l2arc_data_free_t, l2df_list_node));
6516 }
6517
6518 void
```

```
6519 l2arc_fini(void)
6520 {
6521         /*
6522          * This is called from dmu_fini(), which is called from spa_fini();
6523          * Because of this, we can assume that all l2arc devices have
6524          * already been removed when the pools themselves were removed.
6525          */

6527         l2arc_do_free_on_write();

6529         mutex_destroy(&l2arc_feed_thr_lock);
6530         cv_destroy(&l2arc_feed_thr_cv);
6531         mutex_destroy(&l2arc_dev_mtx);
6532         mutex_destroy(&l2arc_free_on_write_mtx);

6534         list_destroy(l2arc_dev_list);
6535         list_destroy(l2arc_free_on_write);
6536 }

6538 void
6539 l2arc_start(void)
6540 {
6541         if (!(spa_mode_global & FWRITE))
6542                 return;

6544         (void) thread_create(NULL, 0, l2arc_feed_thread, NULL, 0, &p0,
6545             TS_RUN, minclsyspri);
6546 }

6548 void
6549 l2arc_stop(void)
6550 {
6551         if (!(spa_mode_global & FWRITE))
6552                 return;

6554         mutex_enter(&l2arc_feed_thr_lock);
6555         cv_signal(&l2arc_feed_thr_cv);  /* kick thread out of startup */
6556         l2arc_thread_exit = 1;
6557         while (l2arc_thread_exit != 0)
6558                 cv_wait(&l2arc_feed_thr_cv, &l2arc_feed_thr_lock);
6559         mutex_exit(&l2arc_feed_thr_lock);
6560 }
```