```
**********************************************************
   77714 Mon Jul 28 07:45:22 2014
new/usr/src/uts/common/fs/zfs/dbuf.c
5047 don't use atomic_*_nv if you discard the return value
**********************************************************
______unchanged_portion_omitted_

1541 /*
1542  * "Clear" the contents of this dbuf.  This will mark the dbuf
1543  * EVICTING and clear *most* of its references.  Unfortunately,
1544  * when we are not holding the dn_dbufs_mtx, we can't clear the
1545  * entry in the dn_dbufs list.  We have to wait until dbuf_destroy()
1546  * in this case.  For callers from the DMU we will usually see:
1547  *      dbuf_clear()->arc_buf_evict()->dbuf_do_evict()->dbuf_destroy()
1548  * For the arc callback, we will usually see:
1549  *      dbuf_do_evict()->dbuf_clear();dbuf_destroy()
1550  * Sometimes, though, we will get a mix of these two:
1551  *      DMU: dbuf_clear()->arc_buf_evict()
1552  *      ARC: dbuf_do_evict()->dbuf_destroy()
1553  */
1554 void
1555 dbuf_clear(dmu_buf_impl_t *db)
1556 {
1557         dnode_t *dn;
1558         dmu_buf_impl_t *parent = db->db_parent;
1559         dmu_buf_impl_t *dndb;
1560         int dbuf_gone = FALSE;

1562         ASSERT(MUTEX_HELD(&db->db_mtx));
1563         ASSERT(refcount_is_zero(&db->db_holds));

1565         dbuf_evict_user(db);

1567         if (db->db_state == DB_CACHED) {
1568                 ASSERT(db->db.db_data != NULL);
1569                 if (db->db_blkid == DMU_BONUS_BLKID) {
1570                         zio_buf_free(db->db.db_data, DN_MAX_BONUSLEN);
1571                         arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
1572                 }
1573                 db->db.db_data = NULL;
1574                 db->db_state = DB_UNCACHED;
1575         }

1577         ASSERT(db->db_state == DB_UNCACHED || db->db_state == DB_NOFILL);
1578         ASSERT(db->db_data_pending == NULL);

1580         db->db_state = DB_EVICTING;
1581         db->db_blkptr = NULL;

1583         DB_DNODE_ENTER(db);
1584         dn = DB_DNODE(db);
1585         dndb = dn->dn_dbuf;
1586         if (db->db_blkid != DMU_BONUS_BLKID && MUTEX_HELD(&dn->dn_dbufs_mtx)) {
1587                 list_remove(&dn->dn_dbufs, db);
1588                 atomic_dec_32(&dn->dn_dbufs_count);
1588                 (void) atomic_dec_32_nv(&dn->dn_dbufs_count);
1589                 membar_producer();
1590                 DB_DNODE_EXIT(db);
1591                 /*
1592                  * Decrementing the dbuf count means that the hold corresponding
1593                  * to the removed dbuf is no longer discounted in dnode_move(),
1594                  * so the dnode cannot be moved until after we release the hold.
1595                  * The membar_producer() ensures visibility of the decremented
1596                  * value in dnode_move(), since DB_DNODE_EXIT doesn't actually
1597                  * release any lock.
1598                  */
```



```
1599                 dnode_rele(dn, db);
1600                 db->db_dnode_handle = NULL;
1601         } else {
1602                 DB_DNODE_EXIT(db);
1603         }

1605         if (db->db_buf)
1606                 dbuf_gone = arc_buf_evict(db->db_buf);

1608         if (!dbuf_gone)
1609                 mutex_exit(&db->db_mtx);

1611         /*
1612          * If this dbuf is referenced from an indirect dbuf,
1613          * decrement the ref count on the indirect dbuf.
1614          */
1615         if (parent && parent != dndb)
1616                 dbuf_rele(parent, db);
1617 }
______unchanged_portion_omitted_

1794 static void
1795 dbuf_destroy(dmu_buf_impl_t *db)
1796 {
1797         ASSERT(refcount_is_zero(&db->db_holds));

1799         if (db->db_blkid != DMU_BONUS_BLKID) {
1800                 /*
1801                  * If this dbuf is still on the dn_dbufs list,
1802                  * remove it from that list.
1803                  */
1804                 if (db->db_dnode_handle != NULL) {
1805                         dnode_t *dn;

1807                         DB_DNODE_ENTER(db);
1808                         dn = DB_DNODE(db);
1809                         mutex_enter(&dn->dn_dbufs_mtx);
1810                         list_remove(&dn->dn_dbufs, db);
1811                         atomic_dec_32(&dn->dn_dbufs_count);
1811                         (void) atomic_dec_32_nv(&dn->dn_dbufs_count);
1812                         mutex_exit(&dn->dn_dbufs_mtx);
1813                         DB_DNODE_EXIT(db);
1814                         /*
1815                          * Decrementing the dbuf count means that the hold
1816                          * corresponding to the removed dbuf is no longer
1817                          * discounted in dnode_move(), so the dnode cannot be
1818                          * moved until after we release the hold.
1819                          */
1820                         dnode_rele(dn, db);
1821                         db->db_dnode_handle = NULL;
1822                 }
1823                 dbuf_hash_remove(db);
1824         }
1825         db->db_parent = NULL;
1826         db->db_buf = NULL;

1828         ASSERT(!list_link_active(&db->db_link));
1829         ASSERT(db->db.db_data == NULL);
1830         ASSERT(db->db_hash_next == NULL);
1831         ASSERT(db->db_blkptr == NULL);
1832         ASSERT(db->db_data_pending == NULL);

1834         kmem_cache_free(dbuf_cache, db);
1835         arc_space_return(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1836 }
______unchanged_portion_omitted_
```

```
2056 /*
2057  * dbuf_rele() for an already-locked dbuf.  This is necessary to allow
2058  * db_dirtycnt and db_holds to be updated atomically.
2059  */
2060 void
2061 dbuf_rele_and_unlock(dmu_buf_impl_t *db, void *tag)
2062 {
2063         int64_t holds;

2065         ASSERT(MUTEX_HELD(&db->db_mtx));
2066         DBUF_VERIFY(db);

2068         /*
2069          * Remove the reference to the dbuf before removing its hold on the
2070          * dnode so we can guarantee in dnode_move() that a referenced bonus
2071          * buffer has a corresponding dnode hold.
2072          */
2073         holds = refcount_remove(&db->db_holds, tag);
2074         ASSERT(holds >= 0);

2076         /*
2077          * We can't freeze indirects if there is a possibility that they
2078          * may be modified in the current syncing context.
2079          */
2080         if (db->db_buf && holds == (db->db_level == 0 ? db->db_dirtycnt : 0))
2081                 arc_buf_freeze(db->db_buf);

2083         if (holds == db->db_dirtycnt &&
2084             db->db_level == 0 && db->db_immediate_evict)
2085                 dbuf_evict_user(db);

2087         if (holds == 0) {
2088                 if (db->db_blkid == DMU_BONUS_BLKID) {
2089                         mutex_exit(&db->db_mtx);

2091                         /*
2092                          * If the dnode moves here, we cannot cross this barrier
2093                          * until the move completes.
2094                          */
2095                         DB_DNODE_ENTER(db);
2096                         atomic_dec_32(&DB_DNODE(db)->dn_dbufs_count);
2096                         (void) atomic_dec_32_nv(&DB_DNODE(db)->dn_dbufs_count);
2097                         DB_DNODE_EXIT(db);
2098                         /*
2099                          * The bonus buffer's dnode hold is no longer discounted
2100                          * in dnode_move(). The dnode cannot move until after
2101                          * the dnode_rele().
2102                          */
2103                         dnode_rele(DB_DNODE(db), db);
2104                 } else if (db->db_buf == NULL) {
2105                         /*
2106                          * This is a special case: we never associated this
2107                          * dbuf with any data allocated from the ARC.
2108                          */
2109                         ASSERT(db->db_state == DB_UNCACHED ||
2110                             db->db_state == DB_NOFILL);
2111                         dbuf_evict(db);
2112                 } else if (arc_released(db->db_buf)) {
2113                         arc_buf_t *buf = db->db_buf;
2114                         /*
2115                          * This dbuf has anonymous data associated with it.
2116                          */
2117                         dbuf_set_data(db, NULL);
2118                         VERIFY(arc_buf_remove_ref(buf, db));
2119                         dbuf_evict(db);
```



```
2120                 } else {
2121                         VERIFY(!arc_buf_remove_ref(db->db_buf, db));

2123                         /*
2124                          * A dbuf will be eligible for eviction if either the
2125                          * 'primarycache' property is set or a duplicate
2126                          * copy of this buffer is already cached in the arc.
2127                          *
2128                          * In the case of the 'primarycache' a buffer
2129                          * is considered for eviction if it matches the
2130                          * criteria set in the property.
2131                          *
2132                          * To decide if our buffer is considered a
2133                          * duplicate, we must call into the arc to determine
2134                          * if multiple buffers are referencing the same
2135                          * block on-disk. If so, then we simply evict
2136                          * ourselves.
2137                          */
2138                         if (!DBUF_IS_CACHEABLE(db) ||
2139                             arc_buf_eviction_needed(db->db_buf))
2140                                 dbuf_clear(db);
2141                         else
2142                                 mutex_exit(&db->db_mtx);
2143                 }
2144         } else {
2145                 mutex_exit(&db->db_mtx);
2146         }
2147 }
```

_____***unchanged_portion_omitted_***

****
46854 Mon Jul 28 07:45:22 2014
new/usr/src/uts/common/fs/zfs/dmu.c
5047 don't use atomic_*_nv if you discard the return value
****
_____***unchanged_portion_omitted_***

```
261 /*
262  * returns ENOENT, EIO, or 0.
263  */
264 int
265 dmu_bonus_hold(objset_t *os, uint64_t object, void *tag, dmu_buf_t **dbp)
266 {
267         dnode_t *dn;
268         dmu_buf_impl_t *db;
269         int error;

271         error = dnode_hold(os, object, FTAG, &dn);
272         if (error)
273                 return (error);

275         rw_enter(&dn->dn_struct_rwlock, RW_READER);
276         if (dn->dn_bonus == NULL) {
277                 rw_exit(&dn->dn_struct_rwlock);
278                 rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
279                 if (dn->dn_bonus == NULL)
280                         dbuf_create_bonus(dn);
281         }
282         db = dn->dn_bonus;

284         /* as long as the bonus buf is held, the dnode will be held */
285         if (refcount_add(&db->db_holds, tag) == 1) {
286                 VERIFY(dnode_add_ref(dn, db));
287                 atomic_inc_32(&dn->dn_dbufs_count);
287                 (void) atomic_inc_32_nv(&dn->dn_dbufs_count);
288         }

290         /*
291          * Wait to drop dn_struct_rwlock until after adding the bonus dbuf's
292          * hold and incrementing the dbuf count to ensure that dnode_move() sees
293          * a dnode hold for every dbuf.
294          */
295         rw_exit(&dn->dn_struct_rwlock);

297         dnode_rele(dn, FTAG);

299         VERIFY(0 == dbuf_read(db, NULL, DB_RF_MUST_SUCCEED | DB_RF_NOPREFETCH));

301         *dbp = &db->db;
302         return (0);
303 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   62035 Mon Jul 28 07:45:22 2014
new/usr/src/uts/common/inet/sctp/sctp_addr.c
5047 don't use atomic_*_nv if you discard the return value
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/stream.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/kmem.h>
#include <sys/socket.h>
#include <sys/sysmacros.h>
#include <sys/list.h>

#include <netinet/in.h>
#include <netinet/ip6.h>
#include <netinet/sctp.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/ip_ire.h>
#include <inet/ip_if.h>
#include <inet/ipclassifier.h>
#include <inet/sctp_ip.h>
#include "sctp_impl.h"
#include "sctp_addr.h"

static void             sctp_ipif_inactive(sctp_ipif_t *);
static sctp_ipif_t      *sctp_lookup_ipif_addr(in6_addr_t *, boolean_t,
                            zoneid_t, boolean_t, uint_t, uint_t, boolean_t,
                            sctp_stack_t *);
static int              sctp_get_all_ipifs(sctp_t *, int);
static int              sctp_ipif_hash_insert(sctp_t *, sctp_ipif_t *, int,
                            boolean_t, boolean_t);
static void             sctp_ipif_hash_remove(sctp_t *, sctp_ipif_t *,
                            boolean_t);
static void             sctp_fix_saddr(sctp_t *, in6_addr_t *);
static int              sctp_compare_ipif_list(sctp_ipif_hash_t *,
                            sctp_ipif_hash_t *);
```



```
static int              sctp_copy_ipifs(sctp_ipif_hash_t *, sctp_t *, int);

#define SCTP_ADDR4_HASH(addr)   \
        (((addr) ^ ((addr) >> 8) ^ ((addr) >> 16) ^ ((addr) >> 24)) &   \
        (SCTP_IPIF_HASH - 1))

#define SCTP_ADDR6_HASH(addr)   \
        (((addr).s6_addr32[3] ^                                         \
        ((addr).s6_addr32[3] ^ (addr).s6_addr32[2]) >> 12)) &           \
        (SCTP_IPIF_HASH - 1))

#define SCTP_IPIF_ADDR_HASH(addr, isv6)                                 \
        ((isv6) ? SCTP_ADDR6_HASH((addr)) :                             \
        SCTP_ADDR4_HASH((addr)._S6_un._S6_u32[3]))

#define SCTP_IPIF_USABLE(sctp_ipif_state)       \
        ((sctp_ipif_state) == SCTP_IPIFS_UP ||  \
        (sctp_ipif_state) ==  SCTP_IPIFS_DOWN)

#define SCTP_IPIF_DISCARD(sctp_ipif_flags)      \
        ((sctp_ipif_flags) & (IPIF_PRIVATE | IPIF_DEPRECATED))

#define SCTP_IS_IPIF_LOOPBACK(ipif)             \
        ((ipif)->sctp_ipif_ill->sctp_ill_flags & PHYI_LOOPBACK)

#define SCTP_IS_IPIF_LINKLOCAL(ipif)            \
        ((ipif)->sctp_ipif_isv6 &&              \
        IN6_IS_ADDR_LINKLOCAL(&(ipif)->sctp_ipif_saddr))

#define SCTP_UNSUPP_AF(ipif, supp_af)   \
        ((!(ipif)->sctp_ipif_isv6 && !((supp_af) & PARM_SUPP_V4)) ||    \
        ((ipif)->sctp_ipif_isv6 && !((supp_af) & PARM_SUPP_V6)))

#define SCTP_IPIF_ZONE_MATCH(sctp, ipif)                                \
        IPCL_ZONE_MATCH((sctp)->sctp_connp, (ipif)->sctp_ipif_zoneid)

#define SCTP_ILL_HASH_FN(index)         ((index) % SCTP_ILL_HASH)
#define SCTP_ILL_TO_PHYINDEX(ill)       ((ill)->ill_phyint->phyint_ifindex)

/*
 * SCTP Interface list manipulation functions, locking used.
 */

/*
 * Delete an SCTP IPIF from the list if the refcount goes to 0 and it is
 * marked as condemned. Also, check if the ILL needs to go away.
 */
static void
sctp_ipif_inactive(sctp_ipif_t *sctp_ipif)
{
        sctp_ill_t      *sctp_ill;
        uint_t          hindex;
        uint_t          ill_index;
        sctp_stack_t    *sctps = sctp_ipif->sctp_ipif_ill->
            sctp_ill_netstack->netstack_sctp;

        rw_enter(&sctps->sctps_g_ills_lock, RW_READER);
        rw_enter(&sctps->sctps_g_ipifs_lock, RW_WRITER);

        hindex = SCTP_IPIF_ADDR_HASH(sctp_ipif->sctp_ipif_saddr,
            sctp_ipif->sctp_ipif_isv6);

        sctp_ill = sctp_ipif->sctp_ipif_ill;
        ASSERT(sctp_ill != NULL);
        ill_index = SCTP_ILL_HASH_FN(sctp_ill->sctp_ill_index);
        if (sctp_ipif->sctp_ipif_state != SCTP_IPIFS_CONDEMNED ||
```

```
 128             sctp_ipif->sctp_ipif_refcnt != 0) {
 129                 rw_exit(&sctps->sctps_g_ipifs_lock);
 130                 rw_exit(&sctps->sctps_g_ills_lock);
 131                 return;
 132         }
 133         list_remove(&sctps->sctps_g_ipifs[hindex].sctp_ipif_list,
 134             sctp_ipif);
 135         sctps->sctps_g_ipifs[hindex].ipif_count--;
 136         sctps->sctps_g_ipifs_count--;
 137         rw_destroy(&sctp_ipif->sctp_ipif_lock);
 138         kmem_free(sctp_ipif, sizeof (sctp_ipif_t));

 140         atomic_dec_32(&sctp_ill->sctp_ill_ipifcnt);
 140         (void) atomic_dec_32_nv(&sctp_ill->sctp_ill_ipifcnt);
 141         if (rw_tryupgrade(&sctps->sctps_g_ills_lock) != 0) {
 142                 rw_downgrade(&sctps->sctps_g_ipifs_lock);
 143                 if (sctp_ill->sctp_ill_ipifcnt == 0 &&
 144                     sctp_ill->sctp_ill_state == SCTP_ILLS_CONDEMNED) {
 145                         list_remove(&sctps->sctps_g_ills[ill_index].
 146                             sctp_ill_list, (void *)sctp_ill);
 147                         sctps->sctps_g_ills[ill_index].ill_count--;
 148                         sctps->sctps_ills_count--;
 149                         kmem_free(sctp_ill->sctp_ill_name,
 150                             sctp_ill->sctp_ill_name_length);
 151                         kmem_free(sctp_ill, sizeof (sctp_ill_t));
 152                 }
 153         }
 154         rw_exit(&sctps->sctps_g_ipifs_lock);
 155         rw_exit(&sctps->sctps_g_ills_lock);
 156 }
```

*unchanged_portion_omitted_*

```
 829 /* move ipif from f_ill to t_ill */
 830 void
 831 sctp_move_ipif(ipif_t *ipif, ill_t *f_ill, ill_t *t_ill)
 832 {
 833         sctp_ill_t      *fsctp_ill = NULL;
 834         sctp_ill_t      *tsctp_ill = NULL;
 835         sctp_ipif_t     *sctp_ipif;
 836         uint_t          hindex;
 837         int             i;
 838         netstack_t      *ns = ipif->ipif_ill->ill_ipst->ips_netstack;
 839         sctp_stack_t    *sctps = ns->netstack_sctp;

 841         rw_enter(&sctps->sctps_g_ills_lock, RW_READER);
 842         rw_enter(&sctps->sctps_g_ipifs_lock, RW_READER);

 844         hindex = SCTP_ILL_HASH_FN(SCTP_ILL_TO_PHYINDEX(f_ill));
 845         fsctp_ill = list_head(&sctps->sctps_g_ills[hindex].sctp_ill_list);
 846         for (i = 0; i < sctps->sctps_g_ills[hindex].ill_count; i++) {
 847                 if (fsctp_ill->sctp_ill_index == SCTP_ILL_TO_PHYINDEX(f_ill) &&
 848                     fsctp_ill->sctp_ill_isv6 == f_ill->ill_isv6) {
 849                         break;
 850                 }
 851                 fsctp_ill = list_next(
 852                     &sctps->sctps_g_ills[hindex].sctp_ill_list, fsctp_ill);
 853         }

 855         hindex = SCTP_ILL_HASH_FN(SCTP_ILL_TO_PHYINDEX(t_ill));
 856         tsctp_ill = list_head(&sctps->sctps_g_ills[hindex].sctp_ill_list);
 857         for (i = 0; i < sctps->sctps_g_ills[hindex].ill_count; i++) {
 858                 if (tsctp_ill->sctp_ill_index == SCTP_ILL_TO_PHYINDEX(t_ill) &&
 859                     tsctp_ill->sctp_ill_isv6 == t_ill->ill_isv6) {
 860                         break;
 861                 }
 862                 tsctp_ill = list_next(
```



```
 863                     &sctps->sctps_g_ills[hindex].sctp_ill_list, tsctp_ill);
 864         }

 866         hindex = SCTP_IPIF_ADDR_HASH(ipif->ipif_v6lcl_addr,
 867             ipif->ipif_ill->ill_isv6);
 868         sctp_ipif = list_head(&sctps->sctps_g_ipifs[hindex].sctp_ipif_list);
 869         for (i = 0; i < sctps->sctps_g_ipifs[hindex].ipif_count; i++) {
 870                 if (sctp_ipif->sctp_ipif_id == ipif->ipif_seqid)
 871                         break;
 872                 sctp_ipif = list_next(
 873                     &sctps->sctps_g_ipifs[hindex].sctp_ipif_list, sctp_ipif);
 874         }
 875         /* Should be an ASSERT? */
 876         if (fsctp_ill == NULL || tsctp_ill == NULL || sctp_ipif == NULL) {
 877                 ip1dbg(("sctp_move_ipif: error moving ipif %p from %p to %p\n",
 878                     (void *)ipif, (void *)f_ill, (void *)t_ill));
 879                 rw_exit(&sctps->sctps_g_ipifs_lock);
 880                 rw_exit(&sctps->sctps_g_ills_lock);
 881                 return;
 882         }
 883         rw_enter(&sctp_ipif->sctp_ipif_lock, RW_WRITER);
 884         ASSERT(sctp_ipif->sctp_ipif_ill == fsctp_ill);
 885         sctp_ipif->sctp_ipif_ill = tsctp_ill;
 886         rw_exit(&sctp_ipif->sctp_ipif_lock);
 887         atomic_dec_32(&fsctp_ill->sctp_ill_ipifcnt);
 887         (void) atomic_dec_32_nv(&fsctp_ill->sctp_ill_ipifcnt);
 888         atomic_inc_32(&tsctp_ill->sctp_ill_ipifcnt);
 889         rw_exit(&sctps->sctps_g_ipifs_lock);
 890         rw_exit(&sctps->sctps_g_ills_lock);
 891 }
```

*unchanged_portion_omitted_*

```
 994 /*
 995  * Insert a new SCTP ipif using 'ipif'. v6addr is the address that existed
 996  * prior to the current address in 'ipif'. Only when an existing address
 997  * is changed on an IPIF, will v6addr be specified. If the IPIF already
 998  * exists in the global SCTP ipif table, then we either removed it, if
 999  * it doesn't have any existing reference, or mark it condemned otherwise.
1000  * If an address is being brought up (IPIF_UP), then we need to scan
1001  * the SCTP list to check if there is any SCTP that points to the *same*
1002  * address on a different SCTP ipif and update in that case.
1003  */
1004 void
1005 sctp_update_ipif_addr(ipif_t *ipif, in6_addr_t v6addr)
1006 {
1007         ill_t           *ill = ipif->ipif_ill;
1008         int             i;
1009         sctp_ill_t      *sctp_ill;
1010         sctp_ill_t      *osctp_ill;
1011         sctp_ipif_t     *sctp_ipif = NULL;
1012         sctp_ipif_t     *osctp_ipif = NULL;
1013         uint_t          ill_index;
1014         int             hindex;
1015         sctp_stack_t    *sctps;

1017         sctps = ipif->ipif_ill->ill_ipst->ips_netstack->netstack_sctp;

1019         /* Index for new address */
1020         hindex = SCTP_IPIF_ADDR_HASH(ipif->ipif_v6lcl_addr, ill->ill_isv6);

1022         /*
1023          * The address on this IPIF is changing, we need to look for
1024          * this old address and mark it condemned, before creating
1025          * one for the new address.
1026          */
1027         osctp_ipif = sctp_lookup_ipif_addr(&v6addr, B_FALSE,
```

```
1028             ipif->ipif_zoneid, B_TRUE, SCTP_ILL_TO_PHYINDEX(ill),
1029             ipif->ipif_seqid, B_FALSE, sctps);

1031         rw_enter(&sctps->sctps_g_ills_lock, RW_READER);
1032         rw_enter(&sctps->sctps_g_ipifs_lock, RW_WRITER);

1034         ill_index = SCTP_ILL_HASH_FN(SCTP_ILL_TO_PHYINDEX(ill));
1035         sctp_ill = list_head(&sctps->sctps_g_ills[ill_index].sctp_ill_list);
1036         for (i = 0; i < sctps->sctps_g_ills[ill_index].ill_count; i++) {
1037                 if (sctp_ill->sctp_ill_index == SCTP_ILL_TO_PHYINDEX(ill) &&
1038                     sctp_ill->sctp_ill_isv6 == ill->ill_isv6) {
1039                         break;
1040                 }
1041                 sctp_ill = list_next(
1042                     &sctps->sctps_g_ills[ill_index].sctp_ill_list, sctp_ill);
1043         }

1045         if (sctp_ill == NULL) {
1046                 ip1dbg(("sctp_update_ipif_addr: ill not found ..\n"));
1047                 rw_exit(&sctps->sctps_g_ipifs_lock);
1048                 rw_exit(&sctps->sctps_g_ills_lock);
1049                 return;
1050         }

1052         if (osctp_ipif != NULL) {

1054                 /* The address is the same? */
1055                 if (IN6_ARE_ADDR_EQUAL(&ipif->ipif_v6lcl_addr, &v6addr)) {
1056                         boolean_t       chk_n_updt = B_FALSE;

1058                         rw_downgrade(&sctps->sctps_g_ipifs_lock);
1059                         rw_enter(&osctp_ipif->sctp_ipif_lock, RW_WRITER);
1060                         if (ipif->ipif_flags & IPIF_UP &&
1061                             osctp_ipif->sctp_ipif_state != SCTP_IPIFS_UP) {
1062                                 osctp_ipif->sctp_ipif_state = SCTP_IPIFS_UP;
1063                                 chk_n_updt = B_TRUE;
1064                         } else {
1065                                 osctp_ipif->sctp_ipif_state = SCTP_IPIFS_DOWN;
1066                         }
1067                         osctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1068                         rw_exit(&osctp_ipif->sctp_ipif_lock);
1069                         if (chk_n_updt) {
1070                                 sctp_chk_and_updt_saddr(hindex, osctp_ipif,
1071                                     sctps);
1072                         }
1073                         rw_exit(&sctps->sctps_g_ipifs_lock);
1074                         rw_exit(&sctps->sctps_g_ills_lock);
1075                         return;
1076                 }
1077                 /*
1078                  * We are effectively removing this address from the ILL.
1079                  */
1080                 if (osctp_ipif->sctp_ipif_refcnt != 0) {
1081                         osctp_ipif->sctp_ipif_state = SCTP_IPIFS_CONDEMNED;
1082                 } else {
1083                         list_t          *ipif_list;
1084                         int             ohindex;

1086                         osctp_ill = osctp_ipif->sctp_ipif_ill;
1087                         /* hash index for the old one */
1088                         ohindex = SCTP_IPIF_ADDR_HASH(
1089                             osctp_ipif->sctp_ipif_saddr,
1090                             osctp_ipif->sctp_ipif_isv6);

1092                         ipif_list =
1093                             &sctps->sctps_g_ipifs[ohindex].sctp_ipif_list;
```



```
1095                         list_remove(ipif_list, (void *)osctp_ipif);
1096                         sctps->sctps_g_ipifs[ohindex].ipif_count--;
1097                         sctps->sctps_g_ipifs_count--;
1098                         rw_destroy(&osctp_ipif->sctp_ipif_lock);
1099                         kmem_free(osctp_ipif, sizeof (sctp_ipif_t));
1100                         atomic_dec_32(&osctp_ill->sctp_ill_ipifcnt);
1100                         (void) atomic_dec_32_nv(&osctp_ill->sctp_ill_ipifcnt);
1101                 }
1102         }

1104         sctp_ipif = kmem_zalloc(sizeof (sctp_ipif_t), KM_NOSLEEP);
1105         /* Try again? */
1106         if (sctp_ipif == NULL) {
1107                 cmn_err(CE_WARN, "sctp_update_ipif_addr: error adding "
1108                     "IPIF %p to SCTP's IPIF list", (void *)ipif);
1109                 rw_exit(&sctps->sctps_g_ipifs_lock);
1110                 rw_exit(&sctps->sctps_g_ills_lock);
1111                 return;
1112         }
1113         sctps->sctps_g_ipifs_count++;
1114         rw_init(&sctp_ipif->sctp_ipif_lock, NULL, RW_DEFAULT, NULL);
1115         sctp_ipif->sctp_ipif_saddr = ipif->ipif_v6lcl_addr;
1116         sctp_ipif->sctp_ipif_ill = sctp_ill;
1117         sctp_ipif->sctp_ipif_isv6 = ill->ill_isv6;
1118         sctp_ipif->sctp_ipif_zoneid = ipif->ipif_zoneid;
1119         sctp_ipif->sctp_ipif_id = ipif->ipif_seqid;
1120         if (ipif->ipif_flags & IPIF_UP)
1121                 sctp_ipif->sctp_ipif_state = SCTP_IPIFS_UP;
1122         else
1123                 sctp_ipif->sctp_ipif_state = SCTP_IPIFS_DOWN;
1124         sctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1125         /*
1126          * We add it to the head so that it is quicker to find good/recent
1127          * additions.
1128          */
1129         list_insert_head(&sctps->sctps_g_ipifs[hindex].sctp_ipif_list,
1130             (void *)sctp_ipif);
1131         sctps->sctps_g_ipifs[hindex].ipif_count++;
1132         atomic_inc_32(&sctp_ill->sctp_ill_ipifcnt);
1133         if (sctp_ipif->sctp_ipif_state == SCTP_IPIFS_UP)
1134                 sctp_chk_and_updt_saddr(hindex, sctp_ipif, sctps);
1135         rw_exit(&sctps->sctps_g_ipifs_lock);
1136         rw_exit(&sctps->sctps_g_ills_lock);
1137 }

1139 /* Insert, Remove,  Mark up or Mark down the ipif */
1140 void
1141 sctp_update_ipif(ipif_t *ipif, int op)
1142 {
1143         ill_t           *ill = ipif->ipif_ill;
1144         int             i;
1145         sctp_ill_t      *sctp_ill;
1146         sctp_ipif_t     *sctp_ipif;
1147         uint_t          ill_index;
1148         uint_t          hindex;
1149         netstack_t      *ns = ipif->ipif_ill->ill_ipst->ips_netstack;
1150         sctp_stack_t    *sctps = ns->netstack_sctp;

1152         ip2dbg(("sctp_update_ipif: %s %d\n", ill->ill_name, ipif->ipif_seqid));

1154         rw_enter(&sctps->sctps_g_ills_lock, RW_READER);
1155         rw_enter(&sctps->sctps_g_ipifs_lock, RW_WRITER);

1157         ill_index = SCTP_ILL_HASH_FN(SCTP_ILL_TO_PHYINDEX(ill));
1158         sctp_ill = list_head(&sctps->sctps_g_ills[ill_index].sctp_ill_list);
```

```
1159         for (i = 0; i < sctps->sctps_g_ills[ill_index].ill_count; i++) {
1160                 if (sctp_ill->sctp_ill_index == SCTP_ILL_TO_PHYINDEX(ill) &&
1161                     sctp_ill->sctp_ill_isv6 == ill->ill_isv6) {
1162                         break;
1163                 }
1164                 sctp_ill = list_next(
1165                     &sctps->sctps_g_ills[ill_index].sctp_ill_list, sctp_ill);
1166         }
1167         if (sctp_ill == NULL) {
1168                 rw_exit(&sctps->sctps_g_ipifs_lock);
1169                 rw_exit(&sctps->sctps_g_ills_lock);
1170                 return;
1171         }

1173         hindex = SCTP_IPIF_ADDR_HASH(ipif->ipif_v6lcl_addr,
1174             ipif->ipif_ill->ill_isv6);
1175         sctp_ipif = list_head(&sctps->sctps_g_ipifs[hindex].sctp_ipif_list);
1176         for (i = 0; i < sctps->sctps_g_ipifs[hindex].ipif_count; i++) {
1177                 if (sctp_ipif->sctp_ipif_id == ipif->ipif_seqid) {
1178                         ASSERT(IN6_ARE_ADDR_EQUAL(&sctp_ipif->sctp_ipif_saddr,
1179                             &ipif->ipif_v6lcl_addr));
1180                         break;
1181                 }
1182                 sctp_ipif = list_next(
1183                     &sctps->sctps_g_ipifs[hindex].sctp_ipif_list,
1184                     sctp_ipif);
1185         }
1186         if (sctp_ipif == NULL) {
1187                 ip1dbg(("sctp_update_ipif: null sctp_ipif for %d\n", op));
1188                 rw_exit(&sctps->sctps_g_ipifs_lock);
1189                 rw_exit(&sctps->sctps_g_ills_lock);
1190                 return;
1191         }
1192         ASSERT(sctp_ill == sctp_ipif->sctp_ipif_ill);
1193         switch (op) {
1194         case SCTP_IPIF_REMOVE:
1195         {
1196                 list_t          *ipif_list;
1197                 list_t          *ill_list;

1199                 ill_list = &sctps->sctps_g_ills[ill_index].sctp_ill_list;
1200                 ipif_list = &sctps->sctps_g_ipifs[hindex].sctp_ipif_list;
1201                 if (sctp_ipif->sctp_ipif_refcnt != 0) {
1202                         sctp_ipif->sctp_ipif_state = SCTP_IPIFS_CONDEMNED;
1203                         rw_exit(&sctps->sctps_g_ipifs_lock);
1204                         rw_exit(&sctps->sctps_g_ills_lock);
1205                         return;
1206                 }
1207                 list_remove(ipif_list, (void *)sctp_ipif);
1208                 sctps->sctps_g_ipifs[hindex].ipif_count--;
1209                 sctps->sctps_g_ipifs_count--;
1210                 rw_destroy(&sctp_ipif->sctp_ipif_lock);
1211                 kmem_free(sctp_ipif, sizeof (sctp_ipif_t));
1212                 atomic_dec_32(&sctp_ill->sctp_ill_ipifcnt);
1212                 (void) atomic_dec_32_nv(&sctp_ill->sctp_ill_ipifcnt);
1213                 if (rw_tryupgrade(&sctps->sctps_g_ills_lock) != 0) {
1214                         rw_downgrade(&sctps->sctps_g_ipifs_lock);
1215                         if (sctp_ill->sctp_ill_ipifcnt == 0 &&
1216                             sctp_ill->sctp_ill_state == SCTP_ILLS_CONDEMNED) {
1217                                 list_remove(ill_list, (void *)sctp_ill);
1218                                 sctps->sctps_ills_count--;
1219                                 sctps->sctps_g_ills[ill_index].ill_count--;
1220                                 kmem_free(sctp_ill->sctp_ill_name,
1221                                     sctp_ill->sctp_ill_name_length);
1222                                 kmem_free(sctp_ill, sizeof (sctp_ill_t));
1223                         }
```



```
1224                 }
1225                 break;
1226         }

1228         case SCTP_IPIF_UP:

1230                 rw_downgrade(&sctps->sctps_g_ipifs_lock);
1231                 rw_enter(&sctp_ipif->sctp_ipif_lock, RW_WRITER);
1232                 sctp_ipif->sctp_ipif_state = SCTP_IPIFS_UP;
1233                 sctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1234                 rw_exit(&sctp_ipif->sctp_ipif_lock);
1235                 sctp_chk_and_updt_saddr(hindex, sctp_ipif,
1236                     ipif->ipif_ill->ill_ipst->ips_netstack->netstack_sctp);

1238                 break;

1240         case SCTP_IPIF_UPDATE:

1242                 rw_downgrade(&sctps->sctps_g_ipifs_lock);
1243                 rw_enter(&sctp_ipif->sctp_ipif_lock, RW_WRITER);
1244                 sctp_ipif->sctp_ipif_zoneid = ipif->ipif_zoneid;
1245                 sctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1246                 rw_exit(&sctp_ipif->sctp_ipif_lock);

1248                 break;

1250         case SCTP_IPIF_DOWN:

1252                 rw_downgrade(&sctps->sctps_g_ipifs_lock);
1253                 rw_enter(&sctp_ipif->sctp_ipif_lock, RW_WRITER);
1254                 sctp_ipif->sctp_ipif_state = SCTP_IPIFS_DOWN;
1255                 sctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1256                 rw_exit(&sctp_ipif->sctp_ipif_lock);

1258                 break;
1259         }
1260         rw_exit(&sctps->sctps_g_ipifs_lock);
1261         rw_exit(&sctps->sctps_g_ills_lock);
1262 }
______unchanged_portion_omitted_

2013 static void
2014 sctp_free_ipifs(sctp_stack_t *sctps)
2015 {
2016         int                     i;
2017         int                     l;
2018         sctp_ipif_t     *sctp_ipif;
2019         sctp_ill_t      *sctp_ill;

2021         if (sctps->sctps_g_ipifs_count == 0)
2022                 return;

2024         for (i = 0; i < SCTP_IPIF_HASH; i++) {
2025                 sctp_ipif = list_tail(&sctps->sctps_g_ipifs[i].sctp_ipif_list);
2026                 for (l = 0; l < sctps->sctps_g_ipifs[i].ipif_count; l++) {
2027                         sctp_ill = sctp_ipif->sctp_ipif_ill;

2029                         list_remove(&sctps->sctps_g_ipifs[i].sctp_ipif_list,
2030                             sctp_ipif);
2031                         sctps->sctps_g_ipifs_count--;
2032                         atomic_dec_32(&sctp_ill->sctp_ill_ipifcnt);
2032                         (void) atomic_dec_32_nv(&sctp_ill->sctp_ill_ipifcnt);
2033                         kmem_free(sctp_ipif, sizeof (sctp_ipif_t));
2034                         sctp_ipif =
2035                             list_tail(&sctps->sctps_g_ipifs[i].sctp_ipif_list);
2036                 }
```

```
2037                 sctps->sctps_g_ipifs[i].ipif_count = 0;
2038         }
2039         ASSERT(sctps->sctps_g_ipifs_count == 0);
2040 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   82534 Mon Jul 28 07:45:22 2014
new/usr/src/uts/common/io/comstar/port/fct/discovery.c
5047 don't use atomic_*_nv if you discard the return value
**********************************************************
______unchanged_portion_omitted_

1101 fct_status_t
1102 fct_register_remote_port(fct_local_port_t *port, fct_remote_port_t *rp,
1103                                 fct_cmd_t *cmd)
1104 {
1105         fct_status_t ret;
1106         fct_i_local_port_t      *iport;
1107         fct_i_remote_port_t     *irp;
1108         int                     i;
1109         char                    info[FCT_INFO_LEN];

1111         iport = (fct_i_local_port_t *)port->port_fct_private;
1112         irp = (fct_i_remote_port_t *)rp->rp_fct_private;

1114         if ((ret = port->port_register_remote_port(port, rp, cmd)) !=
1115             FCT_SUCCESS)
1116                 return (ret);

1118         rw_enter(&iport->iport_lock, RW_WRITER);
1119         rw_enter(&irp->irp_lock, RW_WRITER);
1120         if (rp->rp_handle != FCT_HANDLE_NONE) {
1121                 if (rp->rp_handle >= port->port_max_logins) {
1122                         (void) snprintf(info, sizeof (info),
1123                             "fct_register_remote_port: FCA "
1124                             "returned a handle (%d) for portid %x which is "
1125                             "out of range (max logins = %d)", rp->rp_handle,
1126                             rp->rp_id, port->port_max_logins);
1127                         goto hba_fatal_err;
1128                 }
1129                 if ((iport->iport_rp_slots[rp->rp_handle] != NULL) &&
1130                     (iport->iport_rp_slots[rp->rp_handle] != irp)) {
1131                         fct_i_remote_port_t *t_irp =
1132                             iport->iport_rp_slots[rp->rp_handle];
1133                         (void) snprintf(info, sizeof (info),
1134                             "fct_register_remote_port: "
1135                             "FCA returned a handle %d for portid %x "
1136                             "which was already in use for a different "
1137                             "portid (%x)", rp->rp_handle, rp->rp_id,
1138                             t_irp->irp_rp->rp_id);
1139                         goto hba_fatal_err;
1140                 }
1141         } else {
1142                 /* Pick a handle for this port */
1143                 for (i = 0; i < port->port_max_logins; i++) {
1144                         if (iport->iport_rp_slots[i] == NULL) {
1145                                 break;
1146                         }
1147                 }
1148                 if (i == port->port_max_logins) {
1149                         /* This is really pushing it. */
1150                         (void) snprintf(info, sizeof (info),
1151                             "fct_register_remote_port "
1152                             "Cannot register portid %x because all the "
1153                             "handles are used up", rp->rp_id);
1154                         goto hba_fatal_err;
1155                 }
1156                 rp->rp_handle = i;
1157         }
1158         /* By this time rport_handle is valid */
1159         if ((irp->irp_flags & IRP_HANDLE_OPENED) == 0) {
```



```
1160                 iport->iport_rp_slots[rp->rp_handle] = irp;
1161                 atomic_or_32(&irp->irp_flags, IRP_HANDLE_OPENED);
1162         }
1163         atomic_inc_64(&iport->iport_last_change);
1163         (void) atomic_inc_64_nv(&iport->iport_last_change);
1164         fct_log_remote_port_event(port, ESC_SUNFC_TARGET_ADD,
1165             rp->rp_pwwn, rp->rp_id);

1167 register_rp_done:;
1168         rw_exit(&irp->irp_lock);
1169         rw_exit(&iport->iport_lock);
1170         return (FCT_SUCCESS);

1172 hba_fatal_err:;
1173         rw_exit(&irp->irp_lock);
1174         rw_exit(&iport->iport_lock);
1175         /*
1176          * XXX Throw HBA fatal error event
1177          */
1178         (void) fct_port_shutdown(iport->iport_port,
1179             STMF_RFLAG_FATAL_ERROR | STMF_RFLAG_RESET, info);
1180         return (FCT_FAILURE);
1181 }

1183 fct_status_t
1184 fct_deregister_remote_port(fct_local_port_t *port, fct_remote_port_t *rp)
1185 {
1186         fct_status_t             ret   = FCT_SUCCESS;
1187         fct_i_local_port_t      *iport = PORT_TO_IPORT(port);
1188         fct_i_remote_port_t     *irp   = RP_TO_IRP(rp);

1190         if (irp->irp_snn) {
1191                 kmem_free(irp->irp_snn, strlen(irp->irp_snn) + 1);
1192                 irp->irp_snn = NULL;
1193         }
1194         if (irp->irp_spn) {
1195                 kmem_free(irp->irp_spn, strlen(irp->irp_spn) + 1);
1196                 irp->irp_spn = NULL;
1197         }

1199         if ((ret = port->port_deregister_remote_port(port, rp)) !=
1200             FCT_SUCCESS) {
1201                 return (ret);
1202         }

1204         if (irp->irp_flags & IRP_HANDLE_OPENED) {
1205                 atomic_and_32(&irp->irp_flags, ~IRP_HANDLE_OPENED);
1206                 iport->iport_rp_slots[rp->rp_handle] = NULL;
1207         }
1208         atomic_inc_64(&iport->iport_last_change);
1208         (void) atomic_inc_64_nv(&iport->iport_last_change);
1209         fct_log_remote_port_event(port, ESC_SUNFC_TARGET_REMOVE,
1210             rp->rp_pwwn, rp->rp_id);

1212         return (FCT_SUCCESS);
1213 }
______unchanged_portion_omitted_
```

****
44824 Mon Jul 28 07:45:23 2014
new/usr/src/uts/common/io/trill.c
5047 don't use atomic_*_nv if you discard the return value
****
_____***unchanged_portion_omitted_***

```
1109 static void
1110 trill_node_unref(trill_inst_t *tip, trill_node_t *tnp)
1111 {
1112         if (atomic_dec_uint_nv(&tnp->tn_refs) == 0) {
1113                 if (tnp->tn_tsp != NULL)
1114                         trill_sock_unref(tnp->tn_tsp);
1115                 trill_node_free(tnp);
1116                 atomic_dec_uint(&tip->ti_nodecount);
1116                 (void) atomic_dec_uint_nv(&tip->ti_nodecount);
1117         }
1118 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   48144 Mon Jul 28 07:45:23 2014
new/usr/src/uts/i86pc/os/cmi_hw.c
5047 don't use atomic_*_nv if you discard the return value
**********************************************************
______unchanged_portion_omitted_


1417 void
1418 cmi_hdl_rele(cmi_hdl_t ophdl)
1419 {
1420         cmi_hdl_impl_t *hdl = IMPLHDL(ophdl);

1422         ASSERT(*hdl->cmih_refcntp > 0);
1423         atomic_dec_32(hdl->cmih_refcntp);
1423         (void) atomic_dec_32_nv(hdl->cmih_refcntp);
1424 }

1426 void
1427 cmi_hdl_destroy(cmi_hdl_t ophdl)
1428 {
1429         cmi_hdl_impl_t *hdl = IMPLHDL(ophdl);
1430         cmi_hdl_ent_t *ent;

1432         /* Release the reference count held by cmi_hdl_create(). */
1433         ASSERT(*hdl->cmih_refcntp > 0);
1434         atomic_dec_32(hdl->cmih_refcntp);
1434         (void) atomic_dec_32_nv(hdl->cmih_refcntp);
1435         hdl->cmih_flags |= CMIH_F_DEAD;

1437         ent = cmi_hdl_ent_lookup(hdl->cmih_chipid, hdl->cmih_coreid,
1438             hdl->cmih_strandid);
1439         /*
1440          * Use busy polling instead of condition variable here because
1441          * cmi_hdl_rele() may be called from #MC handler.
1442          */
1443         while (cmi_hdl_canref(ent)) {
1444                 cmi_hdl_rele(ophdl);
1445                 delay(1);
1446         }
1447         ent->cmae_hdlp = NULL;

1449         kmem_free(hdl, sizeof (*hdl));
1450 }
______unchanged_portion_omitted_
```