```
**********************************************************
   81598 Wed Sep 17 12:15:31 2014
new/usr/src/cmd/mdb/common/modules/zfs/zfs.c
patch nuke-the-dbuf-hash
**********************************************************
______unchanged_portion_omitted_
```

```
503 /* ARGSUSED */
504 static int
505 dbuf_stats(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
506 {
507 #define HISTOSZ 32
508         uintptr_t dbp;
509         dmu_buf_impl_t db;
510         dbuf_hash_table_t ht;
511         uint64_t bucket, ndbufs;
512         uint64_t histo[HISTOSZ];
513         uint64_t histo2[HISTOSZ];
514         int i, maxidx;

516         if (mdb_readvar(&ht, "dbuf_hash_table") == -1) {
517                 mdb_warn("failed to read 'dbuf_hash_table'");
518                 return (DCMD_ERR);
519         }

521         for (i = 0; i < HISTOSZ; i++) {
522                 histo[i] = 0;
523                 histo2[i] = 0;
524         }

526         ndbufs = 0;
527         for (bucket = 0; bucket < ht.hash_table_mask+1; bucket++) {
528                 int len;

530                 if (mdb_vread(&dbp, sizeof (void *),
531                     (uintptr_t)(ht.hash_table+bucket)) == -1) {
532                         mdb_warn("failed to read hash bucket %u at %p",
533                             bucket, ht.hash_table+bucket);
534                         return (DCMD_ERR);
535                 }

537                 len = 0;
538                 while (dbp != 0) {
539                         if (mdb_vread(&db, sizeof (dmu_buf_impl_t),
540                             dbp) == -1) {
541                                 mdb_warn("failed to read dbuf at %p", dbp);
542                                 return (DCMD_ERR);
543                         }
544                         dbp = (uintptr_t)db.db_hash_next;
545                         for (i = MIN(len, HISTOSZ - 1); i >= 0; i--)
546                                 histo2[i]++;
547                         len++;
548                         ndbufs++;
549                 }

551                 if (len >= HISTOSZ)
552                         len = HISTOSZ-1;
553                 histo[len]++;
554         }

556         mdb_printf("hash table has %llu buckets, %llu dbufs "
557             "(avg %llu buckets/dbuf)\n",
558             ht.hash_table_mask+1, ndbufs,
559             (ht.hash_table_mask+1)/ndbufs);

561         mdb_printf("\n");
```



```
562         maxidx = 0;
563         for (i = 0; i < HISTOSZ; i++)
564                 if (histo[i] > 0)
565                         maxidx = i;
566         mdb_printf("hash chain length   number of buckets\n");
567         for (i = 0; i <= maxidx; i++)
568                 mdb_printf("%u                  %llu\n", i, histo[i]);

570         mdb_printf("\n");
571         maxidx = 0;
572         for (i = 0; i < HISTOSZ; i++)
573                 if (histo2[i] > 0)
574                         maxidx = i;
575         mdb_printf("hash chain depth    number of dbufs\n");
576         for (i = 0; i <= maxidx; i++)
577                 mdb_printf("%u or more          %llu    %llu%%\n",
578                     i, histo2[i], histo2[i]*100/ndbufs);


581         return (DCMD_OK);
582 }
```

```
503 #define CHAIN_END 0xffff
504 /*
505  * ::zap_leaf [-v]
506  *
507  * Print a zap_leaf_phys_t, assumed to be 16k
508  */
509 /* ARGSUSED */
510 static int
511 zap_leaf(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
512 {
513         char buf[16*1024];
514         int verbose = B_FALSE;
515         int four = B_FALSE;
516         zap_leaf_t l;
517         zap_leaf_phys_t *zlp = (void *)buf;
518         int i;

520         if (mdb_getopts(argc, argv,
521             'v', MDB_OPT_SETBITS, TRUE, &verbose,
522             '4', MDB_OPT_SETBITS, TRUE, &four,
523             NULL) != argc)
524                 return (DCMD_USAGE);

526         l.l_phys = zlp;
527         l.l_bs = 14; /* assume 16k blocks */
528         if (four)
529                 l.l_bs = 12;

531         if (!(flags & DCMD_ADDRSPEC)) {
532                 return (DCMD_USAGE);
533         }

535         if (mdb_vread(buf, sizeof (buf), addr) == -1) {
536                 mdb_warn("failed to read zap_leaf_phys_t at %p", addr);
537                 return (DCMD_ERR);
538         }

540         if (zlp->l_hdr.lh_block_type != ZBT_LEAF ||
541             zlp->l_hdr.lh_magic != ZAP_LEAF_MAGIC) {
542                 mdb_warn("This does not appear to be a zap_leaf_phys_t");
543                 return (DCMD_ERR);
544         }

546         mdb_printf("zap_leaf_phys_t at %p:\n", addr);
```

```
547         mdb_printf("    lh_prefix_len = %u\n", zlp->l_hdr.lh_prefix_len);
548         mdb_printf("    lh_prefix = %llx\n", zlp->l_hdr.lh_prefix);
549         mdb_printf("    lh_nentries = %u\n", zlp->l_hdr.lh_nentries);
550         mdb_printf("    lh_nfree = %u\n", zlp->l_hdr.lh_nfree,
551             zlp->l_hdr.lh_nfree * 100 / (ZAP_LEAF_NUMCHUNKS(&l)));
552         mdb_printf("    lh_freelist = %u\n", zlp->l_hdr.lh_freelist);
553         mdb_printf("    lh_flags = %x (%s)\n", zlp->l_hdr.lh_flags,
554             zlp->l_hdr.lh_flags & ZLF_ENTRIES_CDSORTED ?
555             "ENTRIES_CDSORTED" : "");

557         if (verbose) {
558                 mdb_printf(" hash table:\n");
559                 for (i = 0; i < ZAP_LEAF_HASH_NUMENTRIES(&l); i++) {
560                         if (zlp->l_hash[i] != CHAIN_END)
561                                 mdb_printf("    %u: %u\n", i, zlp->l_hash[i]);
562                 }
563         }

565         mdb_printf(" chunks:\n");
566         for (i = 0; i < ZAP_LEAF_NUMCHUNKS(&l); i++) {
567                 /* LINTED: alignment */
568                 zap_leaf_chunk_t *zlc = &ZAP_LEAF_CHUNK(&l, i);
569                 switch (zlc->l_entry.le_type) {
570                 case ZAP_CHUNK_FREE:
571                         if (verbose) {
572                                 mdb_printf("    %u: free; lf_next = %u\n",
573                                     i, zlc->l_free.lf_next);
574                         }
575                         break;
576                 case ZAP_CHUNK_ENTRY:
577                         mdb_printf("    %u: entry\n", i);
578                         if (verbose) {
579                                 mdb_printf("        le_next = %u\n",
580                                     zlc->l_entry.le_next);
581                         }
582                         mdb_printf("        le_name_chunk = %u\n",
583                             zlc->l_entry.le_name_chunk);
584                         mdb_printf("        le_name_numints = %u\n",
585                             zlc->l_entry.le_name_numints);
586                         mdb_printf("        le_value_chunk = %u\n",
587                             zlc->l_entry.le_value_chunk);
588                         mdb_printf("        le_value_intlen = %u\n",
589                             zlc->l_entry.le_value_intlen);
590                         mdb_printf("        le_value_numints = %u\n",
591                             zlc->l_entry.le_value_numints);
592                         mdb_printf("        le_cd = %u\n",
593                             zlc->l_entry.le_cd);
594                         mdb_printf("        le_hash = %llx\n",
595                             zlc->l_entry.le_hash);
596                         break;
597                 case ZAP_CHUNK_ARRAY:
598                         mdb_printf("    %u: array", i);
599                         if (strisprint((char *)zlc->l_array.la_array))
600                                 mdb_printf(" \"%s\"", zlc->l_array.la_array);
601                         mdb_printf("\n");
602                         if (verbose) {
603                                 int j;
604                                 mdb_printf("        ");
605                                 for (j = 0; j < ZAP_LEAF_ARRAY_BYTES; j++) {
606                                         mdb_printf("%02x ",
607                                             zlc->l_array.la_array[j]);
608                                 }
609                                 mdb_printf("\n");
610                         }
611                         if (zlc->l_array.la_next != CHAIN_END) {
612                                 mdb_printf("        lf_next = %u\n",
```



```
613                                     zlc->l_array.la_next);
614                         }
615                         break;
616                 default:
617                         mdb_printf("    %u: undefined type %u\n",
618                             zlc->l_entry.le_type);
619                 }
620         }

622         return (DCMD_OK);
623 }
```

*unchanged_portion_omitted_*

```
3181 /*
3182  * MDB module linkage information:
3183  *
3184  * We declare a list of structures describing our dcmds, and a function
3185  * named _mdb_init to return a pointer to our module information.
3186  */

3188 static const mdb_dcmd_t dcmds[] = {
3189         { "arc", "[-bkmg]", "print ARC variables", arc_print },
3190         { "blkptr", ":", "print blkptr_t", blkptr },
3191         { "dbuf", ":", "print dmu_buf_impl_t", dbuf },
3273         { "dbuf_stats", ":", "dbuf stats", dbuf_stats },
3192         { "dbufs",
3193             "\t[-O objset_t*] [-n objset_name | \"mos\"] "
3194             "[-o object | \"mdn\"] \n"
3195             "\t[-l level] [-b blkid | \"bonus\"]",
3196             "find dmu_buf_impl_t's that match specified criteria", dbufs },
3197         { "abuf_find", "dva_word[0] dva_word[1]",
3198             "find arc_buf_hdr_t of a specified DVA",
3199             abuf_find },
3200         { "spa", "?[-cevmMh]\n"
3201             "\t-c display spa config\n"
3202             "\t-e display vdev statistics\n"
3203             "\t-v display vdev information\n"
3204             "\t-m display metaslab statistics\n"
3205             "\t-M display metaslab group statistics\n"
3206             "\t-h display histogram (requires -m or -M)\n",
3207             "spa_t summary", spa_print },
3208         { "spa_config", ":", "print spa_t configuration", spa_print_config },
3209         { "spa_space", ":[-b]", "print spa_t on-disk space usage", spa_space },
3210         { "spa_vdevs", ":[-emMh]\n"
3211             "\t-e display vdev statistics\n"
3212             "\t-m dispaly metaslab statistics\n"
3213             "\t-M display metaslab group statistic\n"
3214             "\t-h display histogram (requires -m or -M)\n",
3215             "given a spa_t, print vdev summary", spa_vdevs },
3216         { "vdev", ":[-re]\n"
3217             "\t-r display recursively\n"
3218             "\t-e display statistics\n"
3219             "\t-m display metaslab statistics\n"
3220             "\t-M display metaslab group statistics\n"
3221             "\t-h display histogram (requires -m or -M)\n",
3222             "vdev_t summary", vdev_print },
3223         { "zio", ":[-cpr]\n"
3224             "\t-c display children\n"
3225             "\t-p display parents\n"
3226             "\t-r display recursively",
3227             "zio_t summary", zio_print },
3228         { "zio_state", "?", "print out all zio_t structures on system or "
3229             "for a particular pool", zio_state },
3230         { "zfs_blkstats", ":[-v]",
3231             "given a spa_t, print block type stats from last scrub",
3232             zfs_blkstats },
```

```
3233         { "zfs_params", "", "print zfs tunable parameters", zfs_params },
3234         { "refcount", ":[-r]\n"
3235             "\t-r display recently removed references",
3236             "print refcount_t holders", refcount },
3237         { "zap_leaf", "", "print zap_leaf_phys_t", zap_leaf },
3238         { "zfs_aces", ":[-v]", "print all ACEs from a zfs_acl_t",
3239             zfs_acl_dump },
3240         { "zfs_ace", ":[-v]", "print zfs_ace", zfs_ace_print },
3241         { "zfs_ace0", ":[-v]", "print zfs_ace0", zfs_ace0_print },
3242         { "sa_attr_table", ":", "print SA attribute table from sa_os_t",
3243             sa_attr_table},
3244         { "sa_attr", ": attr_id",
3245             "print SA attribute address when given sa_handle_t", sa_attr_print},
3246         { "zfs_dbgmsg", ":[-va]",
3247             "print zfs debug log", dbgmsg},
3248         { "rrwlock", ":",
3249             "print rrwlock_t, including readers", rrwlock},
3250         { NULL }
3251 };
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   75263 Wed Sep 17 12:15:31 2014
new/usr/src/uts/common/fs/zfs/dbuf.c
patch nuke-the-dbuf-hash
patch make-the-merge-easy
**********************************************************
_____unchanged_portion_omitted_

 87 /*
 88  * dbuf hash table routines
 89  */
 90 #pragma align 64(dbuf_hash_table)
 91 static dbuf_hash_table_t dbuf_hash_table;

 93 static uint64_t dbuf_hash_count;

 95 static uint64_t
 96 dbuf_hash(void *os, uint64_t obj, uint8_t lvl, uint64_t blkid)
 97 {
 98         uintptr_t osv = (uintptr_t)os;
 99         uint64_t crc = -1ULL;

101         ASSERT(zfs_crc64_table[128] == ZFS_CRC64_POLY);
102         crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (lvl)) & 0xFF];
103         crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (osv >> 6)) & 0xFF];
104         crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (obj >> 0)) & 0xFF];
105         crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (obj >> 8)) & 0xFF];
106         crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (blkid >> 0)) & 0xFF];
107         crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ (blkid >> 8)) & 0xFF];

109         crc ^= (osv>>14) ^ (obj>>16) ^ (blkid>>16);

111         return (crc);
112 }

114 #define DBUF_HASH(os, obj, level, blkid) dbuf_hash(os, obj, level, blkid);

116 #define DBUF_EQUAL(dbuf, os, obj, level, blkid)         \
117         ((dbuf)->db.db_object == (obj) &&               \
118         (dbuf)->db_objset == (os) &&                    \
119         (dbuf)->db_level == (level) &&                  \
120         (dbuf)->db_blkid == (blkid))

 87 dmu_buf_impl_t *
 88 dbuf_find(dnode_t *dn, uint8_t level, uint64_t blkid)
 89 {
125         dbuf_hash_table_t *h = &dbuf_hash_table;
 90         objset_t *os = dn->dn_objset;
 91         uint64_t obj = dn->dn_object;
128         uint64_t hv = DBUF_HASH(os, obj, level, blkid);
129         uint64_t idx = hv & h->hash_table_mask;
 92         dmu_buf_impl_t *db;
 93         dmu_buf_impl_t key;
 94         avl_index_t where;

 96         key.db_level = level;
 97         key.db_blkid = blkid;
 98         key.db_state = DB_SEARCH;

100         mutex_enter(&dn->dn_dbufs_mtx);
101         db = avl_find(&dn->dn_dbufs, &key, &where);
102         ASSERT3P(db, ==, NULL);
103         db = avl_nearest(&dn->dn_dbufs, where, AVL_AFTER);

105         for (; db; db = AVL_NEXT(&dn->dn_dbufs, db)) {
106                 if ((db->db_level != level) || (db->db_blkid != blkid))
```



```
107                         break;
108 #endif /* ! codereview */

131         mutex_enter(DBUF_HASH_MUTEX(h, idx));
132         for (db = h->hash_table[idx]; db != NULL; db = db->db_hash_next) {
133                 if (DBUF_EQUAL(db, os, obj, level, blkid)) {
110                 mutex_enter(&db->db_mtx);
111                 if (db->db_state != DB_EVICTING) {
112                         mutex_exit(&dn->dn_dbufs_mtx);
136                                 mutex_exit(DBUF_HASH_MUTEX(h, idx));
113                         return (db);
114                 }
115                 mutex_exit(&db->db_mtx);
116         }
141         }
142         mutex_exit(DBUF_HASH_MUTEX(h, idx));
143         return (NULL);
144 }

146 /*
147  * Insert an entry into the hash table.  If there is already an element
148  * equal to elem in the hash table, then the already existing element
149  * will be returned and the new element will not be inserted.
150  * Otherwise returns NULL.
151  */
152 static dmu_buf_impl_t *
153 dbuf_hash_insert(dmu_buf_impl_t *db)
154 {
155         dbuf_hash_table_t *h = &dbuf_hash_table;
156         objset_t *os = db->db_objset;
157         uint64_t obj = db->db.db_object;
158         int level = db->db_level;
159         uint64_t blkid = db->db_blkid;
160         uint64_t hv = DBUF_HASH(os, obj, level, blkid);
161         uint64_t idx = hv & h->hash_table_mask;
162         dmu_buf_impl_t *dbf;

164         mutex_enter(DBUF_HASH_MUTEX(h, idx));
165         for (dbf = h->hash_table[idx]; dbf != NULL; dbf = dbf->db_hash_next) {
166                 if (DBUF_EQUAL(dbf, os, obj, level, blkid)) {
167                         mutex_enter(&dbf->db_mtx);
168                         if (dbf->db_state != DB_EVICTING) {
169                                 mutex_exit(DBUF_HASH_MUTEX(h, idx));
170                                 return (dbf);
171                         }
172                         mutex_exit(&dbf->db_mtx);
173                 }
174         }

176         mutex_enter(&db->db_mtx);
177         db->db_hash_next = h->hash_table[idx];
178         h->hash_table[idx] = db;
179         mutex_exit(DBUF_HASH_MUTEX(h, idx));
180         atomic_inc_64(&dbuf_hash_count);

118         mutex_exit(&dn->dn_dbufs_mtx);
119 #endif /* ! codereview */
120         return (NULL);
121 }

182 /*
183  * Remove an entry from the hash table.  It must be in the EVICTING state.
184  */
185 static void
186 dbuf_hash_remove(dmu_buf_impl_t *db)
187 {
```

```
188         dbuf_hash_table_t *h = &dbuf_hash_table;
189         uint64_t hv = DBUF_HASH(db->db_objset, db->db.db_object,
190             db->db_level, db->db_blkid);
191         uint64_t idx = hv & h->hash_table_mask;
192         dmu_buf_impl_t *dbf, **dbp;

194         /*
195          * We musn't hold db_mtx to maintain lock ordering:
196          * DBUF_HASH_MUTEX > db_mtx.
197          */
198         ASSERT(refcount_is_zero(&db->db_holds));
199         ASSERT(db->db_state == DB_EVICTING);
200         ASSERT(!MUTEX_HELD(&db->db_mtx));

202         mutex_enter(DBUF_HASH_MUTEX(h, idx));
203         dbp = &h->hash_table[idx];
204         while ((dbf = *dbp) != db) {
205                 dbp = &dbf->db_hash_next;
206                 ASSERT(dbf != NULL);
207         }
208         *dbp = db->db_hash_next;
209         db->db_hash_next = NULL;
210         mutex_exit(DBUF_HASH_MUTEX(h, idx));
211         atomic_dec_64(&dbuf_hash_count);
212 }

123 static arc_evict_func_t dbuf_do_evict;

125 static void
126 dbuf_evict_user(dmu_buf_impl_t *db)
127 {
128         ASSERT(MUTEX_HELD(&db->db_mtx));

130         if (db->db_level != 0 || db->db_evict_func == NULL)
131                 return;

133         if (db->db_user_data_ptr_ptr)
134                 *db->db_user_data_ptr_ptr = db->db.db_data;
135         db->db_evict_func(&db->db, db->db_user_ptr);
136         db->db_user_ptr = NULL;
137         db->db_user_data_ptr_ptr = NULL;
138         db->db_evict_func = NULL;
139 }
```

unchanged_portion_omitted_

```
168 void
169 dbuf_init(void)
170 {
262         uint64_t hsize = 1ULL << 16;
263         dbuf_hash_table_t *h = &dbuf_hash_table;
264         int i;

266         /*
267          * The hash table is big enough to fill all of physical memory
268          * with an average 4K block size.  The table will take up
269          * totalmem*sizeof(void*)/4K (i.e. 2MB/GB with 8-byte pointers).
270          */
271         while (hsize * 4096 < physmem * PAGESIZE)
272                 hsize <<= 1;

274 retry:
275         h->hash_table_mask = hsize - 1;
276         h->hash_table = kmem_zalloc(hsize * sizeof (void *), KM_NOSLEEP);
277         if (h->hash_table == NULL) {
278                 /* XXX - we should really return an error instead of assert */
279                 ASSERT(hsize > (1ULL << 10));
```



```
280                 hsize >>= 1;
281                 goto retry;
282         }

171         dbuf_cache = kmem_cache_create("dmu_buf_impl_t",
172             sizeof (dmu_buf_impl_t),
173             0, dbuf_cons, dbuf_dest, NULL, NULL, NULL, 0);

288         for (i = 0; i < DBUF_MUTEXES; i++)
289                 mutex_init(DBUF_HASH_MUTEX(h, i), NULL, MUTEX_DEFAULT, NULL);
174 }

176 void
177 dbuf_fini(void)
178 {
295         dbuf_hash_table_t *h = &dbuf_hash_table;
296         int i;

298         for (i = 0; i < DBUF_MUTEXES; i++)
299                 mutex_destroy(DBUF_HASH_MUTEX(h, i));
300         kmem_free(h->hash_table, (h->hash_table_mask + 1) * sizeof (void *));
179         kmem_cache_destroy(dbuf_cache);
180 }
```

unchanged_portion_omitted_

```
1611 static dmu_buf_impl_t *
1612 dbuf_create(dnode_t *dn, uint8_t level, uint64_t blkid,
1613     dmu_buf_impl_t *parent, blkptr_t *blkptr)
1614 {
1615         objset_t *os = dn->dn_objset;
1616         dmu_buf_impl_t *db, *odb;
1617         avl_index_t where;
1618 #endif /* ! codereview */

1620         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1621         ASSERT(dn->dn_type != DMU_OT_NONE);

1623         db = kmem_cache_alloc(dbuf_cache, KM_SLEEP);

1625         db->db_objset = os;
1626         db->db.db_object = dn->dn_object;
1627         db->db_level = level;
1628         db->db_blkid = blkid;
1629         db->db_last_dirty = NULL;
1630         db->db_dirtycnt = 0;
1631         db->db_dnode_handle = dn->dn_handle;
1632         db->db_parent = parent;
1633         db->db_blkptr = blkptr;

1635         db->db_user_ptr = NULL;
1636         db->db_user_data_ptr_ptr = NULL;
1637         db->db_evict_func = NULL;
1638         db->db_immediate_evict = 0;
1639         db->db_freed_in_flight = 0;

1641         if (blkid == DMU_BONUS_BLKID) {
1642                 ASSERT3P(parent, ==, dn->dn_dbuf);
1643                 db->db.db_size = DN_MAX_BONUSLEN -
1644                     (dn->dn_nblkptr-1) * sizeof (blkptr_t);
1645                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
1646                 db->db.db_offset = DMU_BONUS_BLKID;
1647                 db->db_state = DB_UNCACHED;
1648                 /* the bonus dbuf is not placed into the dnode's dbuf tree */
1739                 /* the bonus dbuf is not placed in the hash table */
1649                 arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1650                 return (db);
```

```
1651         } else if (blkid == DMU_SPILL_BLKID) {
1652                 db->db.db_size = (blkptr != NULL) ?
1653                     BP_GET_LSIZE(blkptr) : SPA_MINBLOCKSIZE;
1654                 db->db.db_offset = 0;
1655         } else {
1656                 int blocksize =
1657                     db->db_level ? 1 << dn->dn_indblkshift : dn->dn_datablksz;
1658                 db->db.db_size = blocksize;
1659                 db->db.db_offset = db->db_blkid * blocksize;
1660         }

1753         /*
1754          * Hold the dn_dbufs_mtx while we get the new dbuf
1755          * in the hash table *and* added to the dbufs list.
1756          * This prevents a possible deadlock with someone
1757          * trying to look up this dbuf before its added to the
1758          * dn_dbufs list.
1759          */
1662         mutex_enter(&dn->dn_dbufs_mtx);
1663         mutex_enter(&db->db_mtx);
1664 #endif /* ! codereview */
1665         db->db_state = DB_EVICTING;
1666         if ((odb = avl_find(&dn->dn_dbufs, db, &where))) {
1761         if ((odb = dbuf_hash_insert(db)) != NULL) {
1667                 /* someone else inserted it first */
1668                 mutex_exit(&db->db_mtx);
1669 #endif /* ! codereview */
1670                 kmem_cache_free(dbuf_cache, db);
1671                 mutex_enter(&odb->db_mtx);
1672 #endif /* ! codereview */
1673                 mutex_exit(&dn->dn_dbufs_mtx);
1674                 return (odb);
1675         }
1676         avl_insert(&dn->dn_dbufs, db, where);
1763         avl_add(&dn->dn_dbufs, db);
1677         if (db->db_level == 0 && db->db_blkid >=
1678             dn->dn_unlisted_l0_blkid)
1679                 dn->dn_unlisted_l0_blkid = db->db_blkid + 1;
1680         db->db_state = DB_UNCACHED;
1681         mutex_exit(&dn->dn_dbufs_mtx);
1682         arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);

1684         if (parent && parent != dn->dn_dbuf)
1685                 dbuf_add_ref(parent, db);

1687         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1688             refcount_count(&dn->dn_holds) > 0);
1689         (void) refcount_add(&dn->dn_holds, db);
1690         atomic_inc_32(&dn->dn_dbufs_count);

1692         dprintf_dbuf(db, "db=%p\n", db);

1694         return (db);
1695 }
```

_____unchanged_portion_omitted_

```
1719 static void
1720 dbuf_destroy(dmu_buf_impl_t *db)
1721 {
1722         ASSERT(refcount_is_zero(&db->db_holds));

1724         if (db->db_blkid != DMU_BONUS_BLKID) {
1725                 /*
1726                  * If this dbuf is still on the dn_dbufs list,
1727                  * remove it from that list.
1728                  */
```



```
1729                 if (db->db_dnode_handle != NULL) {
1730                         dnode_t *dn;

1732                         DB_DNODE_ENTER(db);
1733                         dn = DB_DNODE(db);
1734                         mutex_enter(&dn->dn_dbufs_mtx);
1735                         avl_remove(&dn->dn_dbufs, db);
1736                         atomic_dec_32(&dn->dn_dbufs_count);
1737                         mutex_exit(&dn->dn_dbufs_mtx);
1738                         DB_DNODE_EXIT(db);
1739                         /*
1740                          * Decrementing the dbuf count means that the hold
1741                          * corresponding to the removed dbuf is no longer
1742                          * discounted in dnode_move(), so the dnode cannot be
1743                          * moved until after we release the hold.
1744                          */
1745                         dnode_rele(dn, db);
1746                         db->db_dnode_handle = NULL;
1747                 }
1835                 dbuf_hash_remove(db);
1748         }
1749         db->db_parent = NULL;
1750         db->db_buf = NULL;

1752         ASSERT(db->db.db_data == NULL);
1841         ASSERT(db->db_hash_next == NULL);
1753         ASSERT(db->db_blkptr == NULL);
1754         ASSERT(db->db_data_pending == NULL);

1756         kmem_cache_free(dbuf_cache, db);
1757         arc_space_return(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1758 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   11317 Wed Sep 17 12:15:31 2014
new/usr/src/uts/common/fs/zfs/sys/dbuf.h
patch nuke-the-dbuf-hash
**********************************************************
______unchanged_portion_omitted_

 154 typedef struct dmu_buf_impl {
 155         /*
 156          * The following members are immutable, with the exception of
 157          * db.db_data, which is protected by db_mtx.
 158          */

 160         /* the publicly visible structure */
 161         dmu_buf_t db;

 163         /* the objset we belong to */
 164         struct objset *db_objset;

 166         /*
 167          * handle to safely access the dnode we belong to (NULL when evicted)
 168          */
 169         struct dnode_handle *db_dnode_handle;

 171         /*
 172          * our parent buffer; if the dnode points to us directly,
 173          * db_parent == db_dnode_handle->dnh_dnode->dn_dbuf
 174          * only accessed by sync thread ???
 175          * (NULL when evicted)
 176          * May change from NULL to non-NULL under the protection of db_mtx
 177          * (see dbuf_check_blkptr())
 178          */
 179         struct dmu_buf_impl *db_parent;

 181         /*
 182          * link for hash table of all dmu_buf_impl_t's
 183          */
 184         struct dmu_buf_impl *db_hash_next;

 181         /* our block number */
 182         uint64_t db_blkid;

 184         /*
 185          * Pointer to the blkptr_t which points to us. May be NULL if we
 186          * don't have one yet. (NULL when evicted)
 187          */
 188         blkptr_t *db_blkptr;

 190         /*
 191          * Our indirection level.  Data buffers have db_level==0.
 192          * Indirect buffers which point to data buffers have
 193          * db_level==1. etc.  Buffers which contain dnodes have
 194          * db_level==0, since the dnodes are stored in a file.
 195          */
 196         uint8_t db_level;

 198         /* db_mtx protects the members below */
 199         kmutex_t db_mtx;

 201         /*
 202          * Current state of the buffer
 203          */
 204         dbuf_states_t db_state;

 206         /*
 207          * Refcount accessed by dmu_buf_{hold,rele}.
```



```
 208          * If nonzero, the buffer can't be destroyed.
 209          * Protected by db_mtx.
 210          */
 211         refcount_t db_holds;

 213         /* buffer holding our data */
 214         arc_buf_t *db_buf;

 216         kcondvar_t db_changed;
 217         dbuf_dirty_record_t *db_data_pending;

 219         /* pointer to most recent dirty record for this buffer */
 220         dbuf_dirty_record_t *db_last_dirty;

 222         /*
 223          * Our link on the owner dnodes's dn_dbufs list.
 224          * Protected by its dn_dbufs_mtx.
 225          */
 226         avl_node_t db_link;

 228         /* Data which is unique to data (leaf) blocks: */

 230         /* stuff we store for the user (see dmu_buf_set_user) */
 231         void *db_user_ptr;
 232         void **db_user_data_ptr_ptr;
 233         dmu_buf_evict_func_t *db_evict_func;

 235         uint8_t db_immediate_evict;
 236         uint8_t db_freed_in_flight;

 238         uint8_t db_dirtycnt;
 239 } dmu_buf_impl_t;
______unchanged_portion_omitted_
```