****
129268 Mon Jul 28 07:43:43 2014
new/usr/src/cmd/mdb/common/modules/genunix/genunix.c
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
3574 static int
3575 eqd_walk_init(mdb_walk_state_t *wsp)
3576 {
3577         eqd_walk_data_t *eqdp;
3578         errorq_elem_t eqe, *addr;
3579         errorq_t eq;
3580         ulong_t i;

3582         if (mdb_vread(&eq, sizeof (eq), wsp->walk_addr) == -1) {
3583                 mdb_warn("failed to read errorq at %p", wsp->walk_addr);
3584                 return (WALK_ERR);
3585         }

3587         if (eq.eq_ptail != NULL &&
3588             mdb_vread(&eqe, sizeof (eqe), (uintptr_t)eq.eq_ptail) == -1) {
3589                 mdb_warn("failed to read errorq element at %p", eq.eq_ptail);
3590                 return (WALK_ERR);
3591         }

3593         eqdp = mdb_alloc(sizeof (eqd_walk_data_t), UM_SLEEP);
3594         wsp->walk_data = eqdp;

3596         eqdp->eqd_stack = mdb_zalloc(sizeof (uintptr_t) * eq.eq_qlen, UM_SLEEP);
3597         eqdp->eqd_buf = mdb_alloc(eq.eq_size, UM_SLEEP);
3598         eqdp->eqd_qlen = eq.eq_qlen;
3599         eqdp->eqd_qpos = 0;
3600         eqdp->eqd_size = eq.eq_size;

3602         /*
3603          * The newest elements in the queue are on the pending list, so we
3604          * push those on to our stack first.
3605          */
3606         eqd_push_list(eqdp, (uintptr_t)eq.eq_pend);

3608         /*
3609          * If eq_ptail is set, it may point to a subset of the errors on the
3610          * pending list in the event a atomic_cas_ptr() failed; if ptail's
3611          * data is already in our stack, NULL out eq_ptail and ignore it.
3610          * pending list in the event a casptr() failed; if ptail's data is
3611          * already in our stack, NULL out eq_ptail and ignore it.
3612          */
3613         if (eq.eq_ptail != NULL) {
3614                 for (i = 0; i < eqdp->eqd_qpos; i++) {
3615                         if (eqdp->eqd_stack[i] == (uintptr_t)eqe.eqe_data) {
3616                                 eq.eq_ptail = NULL;
3617                                 break;
3618                         }
3619                 }
3620         }

3622         /*
3623          * If eq_phead is set, it has the processing list in order from oldest
3624          * to newest.  Use this to recompute eq_ptail as best we can and then
3625          * we nicely fall into eqd_push_list() of eq_ptail below.
3626          */
3627         for (addr = eq.eq_phead; addr != NULL && mdb_vread(&eqe, sizeof (eqe),
3628             (uintptr_t)addr) == sizeof (eqe); addr = eqe.eqe_next)
3629                 eq.eq_ptail = addr;
```



```
3631         /*
3632          * The oldest elements in the queue are on the processing list, subject
3633          * to machinations in the if-clauses above.  Push any such elements.
3634          */
3635         eqd_push_list(eqdp, (uintptr_t)eq.eq_ptail);
3636         return (WALK_NEXT);
3637 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   70398 Mon Jul 28 07:43:43 2014
new/usr/src/uts/common/disp/fss.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

1306 /*
1307  * Thread functions
1308  */
1309 static int
1310 fss_enterclass(kthread_t *t, id_t cid, void *parmsp, cred_t *reqpcredp,
1311     void *bufp)
1312 {
1313         fssparms_t      *fssparmsp = (fssparms_t *)parmsp;
1314         fssproc_t       *fssproc;
1315         pri_t           reqfssuprilim;
1316         pri_t           reqfssupri;
1317         static uint32_t fssexists = 0;
1318         fsspset_t       *fsspset;
1319         fssproj_t       *fssproj;
1320         fsszone_t       *fsszone;
1321         kproject_t      *kpj;
1322         zone_t          *zone;
1323         int             fsszone_allocated = 0;

1325         fssproc = (fssproc_t *)bufp;
1326         ASSERT(fssproc != NULL);

1328         ASSERT(MUTEX_HELD(&ttoproc(t)->p_lock));

1330         /*
1331          * Only root can move threads to FSS class.
1332          */
1333         if (reqpcredp != NULL && secpolicy_setpriority(reqpcredp) != 0)
1334                 return (EPERM);
1335         /*
1336          * Initialize the fssproc structure.
1337          */
1338         fssproc->fss_umdpri = fss_maxumdpri / 2;

1340         if (fssparmsp == NULL) {
1341                 /*
1342                  * Use default values.
1343                  */
1344                 fssproc->fss_nice = NZERO;
1345                 fssproc->fss_uprilim = fssproc->fss_upri = 0;
1346         } else {
1347                 /*
1348                  * Use supplied values.
1349                  */
1350                 if (fssparmsp->fss_uprilim == FSS_NOCHANGE) {
1351                         reqfssuprilim = 0;
1352                 } else {
1353                         if (fssparmsp->fss_uprilim > 0 &&
1354                             secpolicy_setpriority(reqpcredp) != 0)
1355                                 return (EPERM);
1356                         reqfssuprilim = fssparmsp->fss_uprilim;
1357                 }
1358                 if (fssparmsp->fss_upri == FSS_NOCHANGE) {
1359                         reqfssupri = reqfssuprilim;
1360                 } else {
1361                         if (fssparmsp->fss_upri > 0 &&
1362                             secpolicy_setpriority(reqpcredp) != 0)
1363                                 return (EPERM);
1364                         /*
```



```
1365                          * Set the user priority to the requested value or
1366                          * the upri limit, whichever is lower.
1367                          */
1368                         reqfssupri = fssparmsp->fss_upri;
1369                         if (reqfssupri > reqfssuprilim)
1370                                 reqfssupri = reqfssuprilim;
1371                 }
1372                 fssproc->fss_uprilim = reqfssuprilim;
1373                 fssproc->fss_upri = reqfssupri;
1374                 fssproc->fss_nice = NZERO - (NZERO * reqfssupri) / fss_maxupri;
1375                 if (fssproc->fss_nice > FSS_NICE_MAX)
1376                         fssproc->fss_nice = FSS_NICE_MAX;
1377         }

1379         fssproc->fss_timeleft = fss_quantum;
1380         fssproc->fss_tp = t;
1381         cpucaps_sc_init(&fssproc->fss_caps);

1383         /*
1384          * Put a lock on our fsspset structure.
1385          */
1386         mutex_enter(&fsspsets_lock);
1387         fsspset = fss_find_fsspset(t->t_cpupart);
1388         mutex_enter(&fsspset->fssps_lock);
1389         mutex_exit(&fsspsets_lock);

1391         zone = ttoproc(t)->p_zone;
1392         if ((fsszone = fss_find_fsszone(fsspset, zone)) == NULL) {
1393                 if ((fsszone = kmem_zalloc(sizeof (fsszone_t), KM_NOSLEEP))
1394                     == NULL) {
1395                         mutex_exit(&fsspset->fssps_lock);
1396                         return (ENOMEM);
1397                 } else {
1398                         fsszone_allocated = 1;
1399                         fss_insert_fsszone(fsspset, zone, fsszone);
1400                 }
1401         }
1402         kpj = ttoproj(t);
1403         if ((fssproj = fss_find_fssproj(fsspset, kpj)) == NULL) {
1404                 if ((fssproj = kmem_zalloc(sizeof (fssproj_t), KM_NOSLEEP))
1405                     == NULL) {
1406                         if (fsszone_allocated) {
1407                                 fss_remove_fsszone(fsspset, fsszone);
1408                                 kmem_free(fsszone, sizeof (fsszone_t));
1409                         }
1410                         mutex_exit(&fsspset->fssps_lock);
1411                         return (ENOMEM);
1412                 } else {
1413                         fss_insert_fssproj(fsspset, kpj, fsszone, fssproj);
1414                 }
1415         }
1416         fssproj->fssp_threads++;
1417         fssproc->fss_proj = fssproj;

1419         /*
1420          * Reset priority. Process goes to a "user mode" priority here
1421          * regardless of whether or not it has slept since entering the kernel.
1422          */
1423         thread_lock(t);
1424         t->t_clfuncs = &(sclass[cid].cl_funcs->thread);
1425         t->t_cid = cid;
1426         t->t_cldata = (void *)fssproc;
1427         t->t_schedflag |= TS_RUNQMATCH;
1428         fss_change_priority(t, fssproc);
1429         if (t->t_state == TS_RUN || t->t_state == TS_ONPROC ||
1430             t->t_state == TS_WAIT)
```

```
1431                 fss_active(t);
1432         thread_unlock(t);

1434         mutex_exit(&fsspset->fssps_lock);

1436         /*
1437          * Link new structure into fssproc list.
1438          */
1439         FSS_LIST_INSERT(fssproc);

1441         /*
1442          * If this is the first fair-sharing thread to occur since boot,
1443          * we set up the initial call to fss_update() here. Use an atomic
1444          * compare-and-swap since that's easier and faster than a mutex
1445          * (but check with an ordinary load first since most of the time
1446          * this will already be done).
1447          */
1448         if (fssexists == 0 && atomic_cas_32(&fssexists, 0, 1) == 0)
1448         if (fssexists == 0 && cas32(&fssexists, 0, 1) == 0)
1449                 (void) timeout(fss_update, NULL, hz);

1451         return (0);
1452 }
```

*unchanged_portion_omitted_*

****
61978 Mon Jul 28 07:43:44 2014
new/usr/src/uts/common/disp/ts.c
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
538 /*
539  * Allocate a time-sharing class specific thread structure and
540  * initialize it with the parameters supplied. Also move the thread
541  * to specified time-sharing priority.
542  */
543 static int
544 ts_enterclass(kthread_t *t, id_t cid, void *parmsp,
545         cred_t *reqpcredp, void *bufp)
546 {
547         tsparms_t       *tsparmsp = (tsparms_t *)parmsp;
548         tsproc_t        *tspp;
549         pri_t           reqtsuprilim;
550         pri_t           reqtsupri;
551         static uint32_t tspexists = 0;  /* set on first occurrence of */
552                                         /*   a time-sharing process */

554         tspp = (tsproc_t *)bufp;
555         ASSERT(tspp != NULL);

557         /*
558          * Initialize the tsproc structure.
559          */
560         tspp->ts_cpupri = tsmedumdpri;
561         if (cid == ia_cid) {
562                 /*
563                  * Check to make sure caller is either privileged or the
564                  * window system.  When the window system is converted
565                  * to using privileges, the second check can go away.
566                  */
567                 if (reqpcredp != NULL && !groupmember(IA_gid, reqpcredp) &&
568                     secpolicy_setpriority(reqpcredp) != 0)
569                         return (EPERM);
570                 /*
571                  * Belongs to IA "class", so set appropriate flags.
572                  * Mark as 'on' so it will not be a swap victim
573                  * while forking.
574                  */
575                 tspp->ts_flags = TSIA | TSIASET;
576                 tspp->ts_boost = ia_boost;
577         } else {
578                 tspp->ts_flags = 0;
579                 tspp->ts_boost = 0;
580         }

582         if (tsparmsp == NULL) {
583                 /*
584                  * Use default values.
585                  */
586                 tspp->ts_uprilim = tspp->ts_upri = 0;
587                 tspp->ts_nice = NZERO;
588         } else {
589                 /*
590                  * Use supplied values.
591                  */
592                 if (tsparmsp->ts_uprilim == TS_NOCHANGE)
593                         reqtsuprilim = 0;
594                 else {
595                         if (tsparmsp->ts_uprilim > 0 &&
```



```
596                             secpolicy_setpriority(reqpcredp) != 0)
597                                 return (EPERM);
598                         reqtsuprilim = tsparmsp->ts_uprilim;
599                 }

601                 if (tsparmsp->ts_upri == TS_NOCHANGE) {
602                         reqtsupri = reqtsuprilim;
603                 } else {
604                         if (tsparmsp->ts_upri > 0 &&
605                             secpolicy_setpriority(reqpcredp) != 0)
606                                 return (EPERM);
607                         /*
608                          * Set the user priority to the requested value
609                          * or the upri limit, whichever is lower.
610                          */
611                         reqtsupri = tsparmsp->ts_upri;
612                         if (reqtsupri > reqtsuprilim)
613                                 reqtsupri = reqtsuprilim;
614                 }


617                 tspp->ts_uprilim = reqtsuprilim;
618                 tspp->ts_upri = reqtsupri;
619                 tspp->ts_nice = NZERO - (NZERO * reqtsupri) / ts_maxupri;
620         }
621         TS_NEWUMDPRI(tspp);

623         tspp->ts_dispwait = 0;
624         tspp->ts_timeleft = ts_dptbl[tspp->ts_cpupri].ts_quantum;
625         tspp->ts_tp = t;
626         cpucaps_sc_init(&tspp->ts_caps);

628         /*
629          * Reset priority. Process goes to a "user mode" priority
630          * here regardless of whether or not it has slept since
631          * entering the kernel.
632          */
633         thread_lock(t);                 /* get dispatcher lock on thread */
634         t->t_clfuncs = &(sclass[cid].cl_funcs->thread);
635         t->t_cid = cid;
636         t->t_cldata = (void *)tspp;
637         t->t_schedflag &= ~TS_RUNQMATCH;
638         ts_change_priority(t, tspp);
639         thread_unlock(t);

641         /*
642          * Link new structure into tsproc list.
643          */
644         TS_LIST_INSERT(tspp);

646         /*
647          * If this is the first time-sharing thread to occur since
648          * boot we set up the initial call to ts_update() here.
649          * Use an atomic compare-and-swap since that's easier and
650          * faster than a mutex (but check with an ordinary load first
651          * since most of the time this will already be done).
652          */
653         if (tspexists == 0 && atomic_cas_32(&tspexists, 0, 1) == 0)
653         if (tspexists == 0 && cas32(&tspexists, 0, 1) == 0)
654                 (void) timeout(ts_update, NULL, hz);

656         return (0);
657 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    9130 Mon Jul 28 07:43:44 2014
new/usr/src/uts/common/dtrace/systrace.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 142 /*ARGSUSED*/
 143 static int
 144 systrace_enable(void *arg, dtrace_id_t id, void *parg)
 145 {
 146         int sysnum = SYSTRACE_SYSNUM((uintptr_t)parg);
 147         int enabled = (systrace_sysent[sysnum].stsy_entry != DTRACE_IDNONE ||
 148             systrace_sysent[sysnum].stsy_return != DTRACE_IDNONE);

 150         if (SYSTRACE_ISENTRY((uintptr_t)parg)) {
 151                 systrace_sysent[sysnum].stsy_entry = id;
 152 #ifdef _SYSCALL32_IMPL
 153                 systrace_sysent32[sysnum].stsy_entry = id;
 154 #endif
 155         } else {
 156                 systrace_sysent[sysnum].stsy_return = id;
 157 #ifdef _SYSCALL32_IMPL
 158                 systrace_sysent32[sysnum].stsy_return = id;
 159 #endif
 160         }

 162         if (enabled) {
 163                 ASSERT(sysent[sysnum].sy_callc == dtrace_systrace_syscall);
 164                 return (0);
 165         }

 167         (void) atomic_cas_ptr(&sysent[sysnum].sy_callc,
 167         (void) casptr(&sysent[sysnum].sy_callc,
 168             (void *)systrace_sysent[sysnum].stsy_underlying,
 169             (void *)dtrace_systrace_syscall);
 170 #ifdef _SYSCALL32_IMPL
 171         (void) atomic_cas_ptr(&sysent32[sysnum].sy_callc,
 171         (void) casptr(&sysent32[sysnum].sy_callc,
 172             (void *)systrace_sysent32[sysnum].stsy_underlying,
 173             (void *)dtrace_systrace_syscall32);
 174 #endif
 175         return (0);
 176 }

 178 /*ARGSUSED*/
 179 static void
 180 systrace_disable(void *arg, dtrace_id_t id, void *parg)
 181 {
 182         int sysnum = SYSTRACE_SYSNUM((uintptr_t)parg);
 183         int disable = (systrace_sysent[sysnum].stsy_entry == DTRACE_IDNONE ||
 184             systrace_sysent[sysnum].stsy_return == DTRACE_IDNONE);

 186         if (disable) {
 187                 (void) atomic_cas_ptr(&sysent[sysnum].sy_callc,
 187                 (void) casptr(&sysent[sysnum].sy_callc,
 188                     (void *)dtrace_systrace_syscall,
 189                     (void *)systrace_sysent[sysnum].stsy_underlying);

 191 #ifdef _SYSCALL32_IMPL
 192                 (void) atomic_cas_ptr(&sysent32[sysnum].sy_callc,
 192                 (void) casptr(&sysent32[sysnum].sy_callc,
 193                     (void *)dtrace_systrace_syscall32,
 194                     (void *)systrace_sysent32[sysnum].stsy_underlying);
 195 #endif
 196         }
```



```
 198         if (SYSTRACE_ISENTRY((uintptr_t)parg)) {
 199                 systrace_sysent[sysnum].stsy_entry = DTRACE_IDNONE;
 200 #ifdef _SYSCALL32_IMPL
 201                 systrace_sysent32[sysnum].stsy_entry = DTRACE_IDNONE;
 202 #endif
 203         } else {
 204                 systrace_sysent[sysnum].stsy_return = DTRACE_IDNONE;
 205 #ifdef _SYSCALL32_IMPL
 206                 systrace_sysent32[sysnum].stsy_return = DTRACE_IDNONE;
 207 #endif
 208         }
 209 }
______unchanged_portion_omitted_
```

```
***********************************************************
   87005 Mon Jul 28 07:43:44 2014
new/usr/src/uts/common/fs/fem.c
5042 stop using deprecated atomic functions
***********************************************************
______unchanged_portion_omitted_


2860 /*
2861  * Create a new fem_head and associate with the vnode.
2862  * To keep the unaugmented vnode access path lock free, we spin
2863  * update this - create a new one, then try and install it. If
2864  * we fail to install, release the old one and pretend we succeeded.
2865  */

2867 static struct fem_head *
2868 new_femhead(struct fem_head **hp)
2869 {
2870         struct fem_head *head;

2872         head = kmem_alloc(sizeof (*head), KM_SLEEP);
2873         mutex_init(&head->femh_lock, NULL, MUTEX_DEFAULT, NULL);
2874         head->femh_list = NULL;
2875         if (atomic_cas_ptr(hp, NULL, head) != NULL) {
2875         (casptr(hp, NULL, head) != NULL) {
2876                 kmem_free(head, sizeof (*head));
2877                 head = *hp;
2878         }
2879         return (head);
2880 }
______unchanged_portion_omitted_

3348 void
3349 fem_setvnops(vnode_t *v, vnodeops_t *newops)
3350 {
3351         vnodeops_t      *r;

3353         ASSERT(v != NULL);
3354         ASSERT(newops != NULL);

3356         do {
3357                 r = v->v_op;
3358                 membar_consumer();
3359                 if (v->v_femhead != NULL) {
3360                         struct fem_list *fl;
3361                         if ((fl = fem_lock(v->v_femhead)) != NULL) {
3362                                 fl->feml_nodes[1].fn_op.vnode = newops;
3363                                 fem_unlock(v->v_femhead);
3364                                 return;
3365                         }
3366                         fem_unlock(v->v_femhead);
3367                 }
3368         } while (atomic_cas_ptr(&v->v_op, r, newops) != r);
3368         } while (casptr(&v->v_op, r, newops) != r);
3369 }
______unchanged_portion_omitted_

3490 void
3491 fsem_setvfsops(vfs_t *v, vfsops_t *newops)
3492 {
3493         vfsops_t        *r;

3495         ASSERT(v != NULL);
3496         ASSERT(newops != NULL);
3497         ASSERT(v->vfs_implp);
```



```
3499         do {
3500                 r = v->vfs_op;
3501                 membar_consumer();
3502                 if (v->vfs_femhead != NULL) {
3503                         struct fem_list *fl;
3504                         if ((fl = fem_lock(v->vfs_femhead)) != NULL) {
3505                                 fl->feml_nodes[1].fn_op.vfs = newops;
3506                                 fem_unlock(v->vfs_femhead);
3507                                 return;
3508                         }
3509                         fem_unlock(v->vfs_femhead);
3510                 }
3511         } while (atomic_cas_ptr(&v->vfs_op, r, newops) != r);
3511         } while (casptr(&v->vfs_op, r, newops) != r);
3512 }
______unchanged_portion_omitted_
```

```
**********************************************************
   63925 Mon Jul 28 07:43:44 2014
new/usr/src/uts/common/fs/portfs/port_fop.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 282 fem_t *fop_femop;
 283 fsem_t *fop_fsemop;

 285 static fem_t *
 286 port_fop_femop()
 287 {
 288         fem_t *femp;
 289         if (fop_femop != NULL)
 290                 return (fop_femop);
 291         if (fem_create("portfop_fem",
 292             (const struct fs_operation_def *)port_vnodesrc_template,
 293             (fem_t **)&femp)) {
 294                 return (NULL);
 295         }
 296         if (atomic_cas_ptr(&fop_femop, NULL, femp) != NULL) {
 296         if (casptr(&fop_femop, NULL, femp) != NULL) {
 297                 /*
 298                  * some other thread beat us to it.
 299                  */
 300                 fem_free(femp);
 301         }
 302         return (fop_femop);
 303 }

 305 static fsem_t *
 306 port_fop_fsemop()
 307 {
 308         fsem_t *fsemp;
 309         if (fop_fsemop != NULL)
 310                 return (fop_fsemop);
 311         if (fsem_create("portfop_fsem", port_vfssrc_template, &fsemp)) {
 312                 return (NULL);
 313         }
 314         if (atomic_cas_ptr(&fop_fsemop, NULL, fsemp) != NULL) {
 314         if (casptr(&fop_fsemop, NULL, fsemp) != NULL) {
 315                 /*
 316                  * some other thread beat us to it.
 317                  */
 318                 fsem_free(fsemp);
 319         }
 320         return (fop_fsemop);
 321 }
______unchanged_portion_omitted_

1071 /*
1072  * Installs the portfop_vp_t data structure on the
1073  * vnode. The 'pvp_femp == NULL' indicates it is not
1074  * active. The fem hooks have to be installed.
1075  * The portfop_vp_t is only freed when the vnode gets freed.
1076  */
1077 void
1078 port_install_fopdata(vnode_t *vp)
1079 {
1080         portfop_vp_t *npvp;

1082         npvp = kmem_zalloc(sizeof (*npvp), KM_SLEEP);
1083         mutex_init(&npvp->pvp_mutex, NULL, MUTEX_DEFAULT, NULL);
1084         list_create(&npvp->pvp_pfoplist, sizeof (portfop_t),
1085             offsetof(portfop_t, pfop_node));
```



```
1086         npvp->pvp_vp = vp;
1087         /*
1088          * If v_fopdata is not null, some other thread beat us to it.
1089          */
1090         if (atomic_cas_ptr(&vp->v_fopdata, NULL, npvp) != NULL) {
1090         if (casptr(&vp->v_fopdata, NULL, npvp) != NULL) {
1091                 mutex_destroy(&npvp->pvp_mutex);
1092                 list_destroy(&npvp->pvp_pfoplist);
1093                 kmem_free(npvp, sizeof (*npvp));
1094         }
1095 }
______unchanged_portion_omitted_
```

```
***********************************************************
  118246 Mon Jul 28 07:43:45 2014
new/usr/src/uts/common/fs/vfs.c
5042 stop using deprecated atomic functions
***********************************************************
______unchanged_portion_omitted_

 456 /* Support routines used to reference vfs_op */

 458 /* Set the operations vector for a vfs */
 459 void
 460 vfs_setops(vfs_t *vfsp, vfsops_t *vfsops)
 461 {
 462         vfsops_t        *op;

 464         ASSERT(vfsp != NULL);
 465         ASSERT(vfsops != NULL);

 467         op = vfsp->vfs_op;
 468         membar_consumer();
 469         if (vfsp->vfs_femhead == NULL &&
 470             atomic_cas_ptr(&vfsp->vfs_op, op, vfsops) == op) {
 470             casptr(&vfsp->vfs_op, op, vfsops) == op) {
 471                 return;
 472         }
 473         fsem_setvfsops(vfsp, vfsops);
 474 }
______unchanged_portion_omitted_

2954 /* Provide a unique and monotonically-increasing timestamp. */
2955 void
2956 vfs_mono_time(timespec_t *ts)
2957 {
2958         static volatile hrtime_t hrt;           /* The saved time. */
2959         hrtime_t        newhrt, oldhrt;         /* For effecting the CAS. */
2960         timespec_t      newts;

2962         /*
2963          * Try gethrestime() first, but be prepared to fabricate a sensible
2964          * answer at the first sign of any trouble.
2965          */
2966         gethrestime(&newts);
2967         newhrt = ts2hrt(&newts);
2968         for (;;) {
2969                 oldhrt = hrt;
2970                 if (newhrt <= hrt)
2971                         newhrt = hrt + 1;
2972                 if (atomic_cas_64((uint64_t *)&hrt, oldhrt, newhrt) == oldhrt)
2972                 if (cas64((uint64_t *)&hrt, oldhrt, newhrt) == oldhrt)
2973                         break;
2974         }
2975         hrt2ts(newhrt, ts);
2976 }
______unchanged_portion_omitted_
```

```
**********************************************************
  105377 Mon Jul 28 07:43:45 2014
new/usr/src/uts/common/fs/vnode.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

2820 /*
2821  * Set the operations vector for a vnode.
2822  *
2823  * FEM ensures that the v_femhead pointer is filled in before the
2824  * v_op pointer is changed.  This means that if the v_femhead pointer
2825  * is NULL, and the v_op field hasn't changed since before which checked
2826  * the v_femhead pointer; then our update is ok - we are not racing with
2827  * FEM.
2828  */
2829 void
2830 vn_setops(vnode_t *vp, vnodeops_t *vnodeops)
2831 {
2832         vnodeops_t      *op;

2834         ASSERT(vp != NULL);
2835         ASSERT(vnodeops != NULL);

2837         op = vp->v_op;
2838         membar_consumer();
2839         /*
2840          * If vp->v_femhead == NULL, then we'll call atomic_cas_ptr() to do
2841          * the compare-and-swap on vp->v_op.  If either fails, then FEM is
2840          * If vp->v_femhead == NULL, then we'll call casptr() to do the
2841          * compare-and-swap on vp->v_op.  If either fails, then FEM is
2842          * in effect on the vnode and we need to have FEM deal with it.
2843          */
2844         if (vp->v_femhead != NULL || atomic_cas_ptr(&vp->v_op, op, vnodeops) !=
2845             op) {
2844         if (vp->v_femhead != NULL || casptr(&vp->v_op, op, vnodeops) != op) {
2846                 fem_setvnops(vp, vnodeops);
2847         }
2848 }
______unchanged_portion_omitted_
```

****
*** 219335 Mon Jul 28 07:43:45 2014***
new/usr/src/uts/common/inet/ip/sadb.c
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
1713 /*
1714  * Set up a global pfkey_q instance for AH, ESP, or some other consumer.
1715  */
1716 void
1717 sadb_keysock_hello(queue_t **pfkey_qp, queue_t *q, mblk_t *mp,
1718     void (*ager)(void *), void *agerarg, timeout_id_t *top, int satype)
1719 {
1720         keysock_hello_ack_t *kha;
1721         queue_t *oldq;

1723         ASSERT(OTHERQ(q) != NULL);

1725         /*
1726          * First, check atomically that I'm the first and only keysock
1727          * instance.
1728          *
1729          * Use OTHERQ(q), because qreply(q, mp) == putnext(OTHERQ(q), mp),
1730          * and I want this module to say putnext(*_pfkey_q, mp) for PF_KEY
1731          * messages.
1732          */

1734         oldq = atomic_cas_ptr((void **)pfkey_qp, NULL, OTHERQ(q));
1734         oldq = casptr((void **)pfkey_qp, NULL, OTHERQ(q));
1735         if (oldq != NULL) {
1736                 ASSERT(oldq != q);
1737                 cmn_err(CE_WARN, "Danger!  Multiple keysocks on top of %s.\n",
1738                     (satype == SADB_SATYPE_ESP)? "ESP" : "AH or other");
1739                 freemsg(mp);
1740                 return;
1741         }

1743         kha = (keysock_hello_ack_t *)mp->b_rptr;
1744         kha->ks_hello_len = sizeof (keysock_hello_ack_t);
1745         kha->ks_hello_type = KEYSOCK_HELLO_ACK;
1746         kha->ks_hello_satype = (uint8_t)satype;

1748         /*
1749          * If we made it past the atomic_cas_ptr, then we have "exclusive"
1750          * access to the timeout handle.  Fire it off after the default ager
1749          * If we made it past the casptr, then we have "exclusive" access
1750          * to the timeout handle.  Fire it off after the default ager
1751          * interval.
1752          */
1753         *top = qtimeout(*pfkey_qp, ager, agerarg,
1754             drv_usectohz(SADB_AGE_INTERVAL_DEFAULT * 1000));

1756         putnext(*pfkey_qp, mp);
1757 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    4054 Mon Jul 28 07:43:46 2014
new/usr/src/uts/common/inet/ipsecah.h
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

  88 /*
  89  * IPSECAH stack instances
  90  */
  91 struct ipsecah_stack {
  92         netstack_t              *ipsecah_netstack;      /* Common netstack */

  94         caddr_t                 ipsecah_g_nd;
  95         ipsecahparam_t          *ipsecah_params;
  96         kmutex_t                ipsecah_param_lock;     /* Protects params */

  98         sadbp_t                 ah_sadb;

 100         /* Packet dropper for AH drops. */
 101         ipdropper_t             ah_dropper;

 103         kstat_t                 *ah_ksp;
 104         ah_kstats_t             *ah_kstats;

 106         /*
 107          * Keysock instance of AH.  There can be only one per stack instance.
 108          * Use atomic_cas_ptr() on this because I don't set it until
 109          * KEYSOCK_HELLO comes down.
 108          * Use casptr() on this because I don't set it until KEYSOCK_HELLO
 109          * comes down.
 110          * Paired up with the ah_pfkey_q is the ah_event, which will age SAs.
 111          */
 112         queue_t                 *ah_pfkey_q;
 113         timeout_id_t            ah_event;
 114 };
______unchanged_portion_omitted_
```

```
**********************************************************
    2541 Mon Jul 28 07:43:46 2014
new/usr/src/uts/common/inet/ipsecesp.h
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 46 /*
 47  * IPSECESP stack instances
 48  */
 49 struct ipsecesp_stack {
 50         netstack_t              *ipsecesp_netstack;     /* Common netstack */

 52         caddr_t                 ipsecesp_g_nd;
 53         struct ipsecespparam_s  *ipsecesp_params;
 54         kmutex_t                ipsecesp_param_lock;    /* Protects params */

 56         /* Packet dropper for ESP drops. */
 57         ipdropper_t             esp_dropper;

 59         kstat_t                 *esp_ksp;
 60         struct esp_kstats_s     *esp_kstats;

 62         /*
 63          * Keysock instance of ESP.  There can be only one per stack instance.
 64          * Use atomic_cas_ptr() on this because I don't set it until
 65          * KEYSOCK_HELLO comes down.
 64          * Use casptr() on this because I don't set it until KEYSOCK_HELLO
 65          * comes down.
 66          * Paired up with the esp_pfkey_q is the esp_event, which will age SAs.
 67          */
 68         queue_t                 *esp_pfkey_q;
 69         timeout_id_t            esp_event;

 71         sadbp_t                 esp_sadb;
 72 };
______unchanged_portion_omitted_
```

```
**********************************************************
    4947 Mon Jul 28 07:43:46 2014
new/usr/src/uts/common/inet/keysock.h
5042 stop using deprecated atomic functions
**********************************************************
```

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

#ifndef _INET_KEYSOCK_H
#define _INET_KEYSOCK_H

#ifdef  __cplusplus
extern "C" {
#endif

extern int keysock_opt_get(queue_t *, int, int, uchar_t *);
extern int keysock_opt_set(queue_t *, uint_t, int, int, uint_t,
    uchar_t *, uint_t *, uchar_t *, void *, cred_t *cr);

/*
 * Object to represent database of options to search passed to
 * {sock,tpi}optcom_req() interface routine to take care of option
 * management and associated methods.
 */

extern optdb_obj_t      keysock_opt_obj;
extern uint_t           keysock_max_optsize;

/*
 * KEYSOCK stack instances
 */
struct keysock_stack {
        netstack_t              *keystack_netstack;     /* Common netstack */
        /*
         * keysock_plumbed: zero if plumb not attempted, positive if it
         * succeeded,  negative if it failed.
         */
        int                     keystack_plumbed;
        caddr_t                 keystack_g_nd;
        struct keysockparam_s   *keystack_params;

        kmutex_t                keystack_param_lock;
                                /* Protects the NDD variables. */
```



```
        /* List of open PF_KEY sockets, protected by keysock_list_lock. */
        kmutex_t                keystack_list_lock;
        struct keysock_s        *keystack_list;

        /*
         * Consumers table. If an entry is NULL, keysock maintains
         * the table.
         */
        kmutex_t                keystack_consumers_lock;

#define KEYSOCK_MAX_CONSUMERS 256
        struct keysock_consumer_s *keystack_consumers[KEYSOCK_MAX_CONSUMERS];

        /*
         * State for flush/dump.  This would normally be a boolean_t, but
         * atomic_cas_32() works best for a known 32-bit quantity.
         * cas32() works best for a known 32-bit quantity.
         */
        uint32_t                keystack_flushdump;
        int                     keystack_flushdump_errno;

        /*
         * This integer counts the number of extended REGISTERed sockets.  This
         * determines if we should send extended REGISTERs.
         */
        uint32_t                keystack_num_extended;

        /*
         * Global sequence space for SADB_ACQUIRE messages of any sort.
         */
        uint32_t                keystack_acquire_seq;
};
```

*unchanged_portion_omitted_*

```
**********************************************************
   70052 Mon Jul 28 07:43:46 2014
new/usr/src/uts/common/inet/nca/nca.h
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_
 682 #endif

 684 #define NODE_TV_SZ 8192

 686 extern struct node_ts node_tv[NODE_TV_SZ];
 687 extern struct node_ts *node_tp;

 689 #define NODE_T_TRACE(p, a) {                                    \
 690         struct node_ts *_p;                                     \
 691         struct node_ts *_np;                                    \
 692         int    _ix;                                             \
 693                                                                 \
 694         do {                                                    \
 695                 _p = node_tp;                                   \
 696                 if ((_np = _p + 1) == &node_tv[NODE_TV_SZ])     \
 697                         _np = node_tv;                          \
 698         } while (atomic_cas_ptr(&node_tp, _p, _np) != _p);      \
 698         } while (casptr(&node_tp, _p, _np) != _p);              \
 699         _p->node = (p);                                         \
 700         _p->action = (a);                                       \
 701         _p->ref = (p) ? (p)->ref : 0;                           \
 702         _p->cnt = (p) ? (p)->cnt : 0;                           \
 703         _p->cpu = CPU->cpu_seqid;                               \
 704         NODE_T_TRACE_STK();                                     \
 705 }
______unchanged_portion_omitted_
 770 #endif

 772 #define DOOR_TV_SZ 8192

 774 extern struct door_ts door_tv[DOOR_TV_SZ];
 775 extern struct door_ts *door_tp;

 777 #define DOOR_TRACE(io, d, d_sz, a) {                            \
 778         nca_conn_t *_cp = (io) ? (nca_conn_t *)(io)->cid : (nca_conn_t *)NULL; \
 779         node_t *_req_np = _cp ? _cp->req_np : (node_t *)NULL;   \
 780         struct door_ts *_p;                                     \
 781         struct door_ts *_np;                                    \
 782         int    _ix;                                             \
 783                                                                 \
 784         do {                                                    \
 785                 _p = door_tp;                                   \
 786                 if ((_np = _p + 1) == &door_tv[DOOR_TV_SZ])     \
 787                         _np = door_tv;                          \
 788         } while (atomic_cas_ptr(&door_tp, _p, _np) != _p);      \
 788         } while (casptr(&door_tp, _p, _np) != _p);              \
 789         _p->cp = _cp;                                           \
 790         _p->np = _req_np;                                       \
 791         _p->action = (a);                                       \
 792         _p->ref = _req_np ? _req_np->ref : 0;                   \
 793         if ((io)) {                                             \
 794                 _p->state = ((io)->op == http_op ? 0x80000000 : 0) |    \
 795                             ((io)->more ? 0x40000000 : 0) |     \
 796                             ((io)->first ? 0x20000000 : 0) |    \
 797                             ((io)->advisory ? 0x10000000 : 0) | \
 798                             ((io)->nocache ? 0x08000000 : 0) |  \
 799                             ((io)->preempt ? 0x04000000 : 0) |  \
 800                             ((io)->peer_len ? 0x02000000 : 0) | \
 801                             ((io)->local_len ? 0x01000000 : 0) |    \
 802                             ((io)->data_len ? 0x00800000 : 0) |     \
```



```
 803                             (((io)->direct_type << 20) & 0x00700000) |  \
 804                             ((io)->direct_len ? 0x00080000 : 0) |       \
 805                             ((io)->trailer_len ? 0x00040000 : 0) |      \
 806                             (((io)->peer_len + (io)->local_len +        \
 807                             (io)->data_len + (io)->direct_len +         \
 808                             (io)->trailer_len) & 0x3FFFF);              \
 809         } else {                                                        \
 810                 _p->state = 0;                                          \
 811         }                                                               \
 812         if ((d_sz)) {                                                   \
 813                 int _n = MIN((d_sz), 63);                               \
 814                                                                         \
 815                 bcopy((d), _p->data, _n);                               \
 816                 bzero(&_p->data[_n], 64 - _n);                          \
 817         } else {                                                        \
 818                 bzero(_p->data, 64);                                    \
 819         }                                                               \
 820         DOOR_TRACE_STK();                                               \
 821 }
______unchanged_portion_omitted_

 972 #define DCB_WR_ENTER() {                                        \
 973         int cpu;                                                \
 974         int readers;                                            \
 975                                                                 \
 976         mutex_enter(&nca_dcb_readers);                          \
 977         mutex_enter(&nca_dcb_lock);                             \
 978         for (;;) {                                              \
 979                 readers = 0;                                    \
 980                 for (cpu = 0; cpu < max_ncpus; cpu++) {         \
 981                         int new;                                \
 982                         uint32_t *rp = &nca_gv[cpu].dcb_readers;        \
 983                         int old = *rp;                          \
 984                                                                 \
 985                         if (old & DCB_COUNT_USELOCK) {          \
 986                                 readers += old & DCB_COUNT_MASK;        \
 987                                 continue;                       \
 988                         }                                       \
 989                         new = old | DCB_COUNT_USELOCK;          \
 990                         while (atomic_cas_32(rp, old, new) != old) {    \
 990                         while (cas32(rp, old, new) != old) {    \
 991                                 old = *rp;                      \
 992                                 new = old | DCB_COUNT_USELOCK;  \
 993                         }                                       \
 994                         readers += (new & DCB_COUNT_MASK);      \
 995                 }                                               \
 996                 if (readers == 0)                               \
 997                         break;                                  \
 998                 cv_wait(&nca_dcb_wait, &nca_dcb_lock);          \
 999         }                                                       \
1000         mutex_exit(&nca_dcb_lock);                              \
1001 }

1003 #define DCB_WR_EXIT() {                                         \
1004         int cpu;                                                \
1005                                                                 \
1006         mutex_enter(&nca_dcb_lock);                             \
1007         for (cpu = 0; cpu < max_ncpus; cpu++) {                 \
1008                 int new;                                        \
1009                 uint32_t *rp = &nca_gv[cpu].dcb_readers;        \
1010                 int old = *rp;                                  \
1011                                                                 \
1012                 new = old & ~DCB_COUNT_USELOCK;                 \
1013                 while (atomic_cas_32(rp, old, new) != old) {    \
1013                 while (cas32(rp, old, new) != old) {            \
1014                         old = *rp;                              \
```

```
1015                        new = old & ~DCB_COUNT_USELOCK;                \
1016                }                                                      \
1017        }                                                              \
1018        mutex_exit(&nca_dcb_lock);                                     \
1019        mutex_exit(&nca_dcb_readers);                                  \
1020 }
______unchanged_portion_omitted_
1530 #endif

1532 #define CON_TV_SZ 4096

1534 extern struct conn_ts con_tv[CON_TV_SZ];
1535 extern struct conn_ts *conn_tp;

1537 #define NCA_CONN_T_TRACE(p, a) {                                       \
1538        struct conn_ts *_p;                                            \
1539        struct conn_ts *_np;                                           \
1540        int    _ix;                                                    \
1541                                                                       \
1542        do {                                                           \
1543                _p = conn_tp;                                  \
1544                if ((_np = _p + 1) == &con_tv[CON_TV_SZ])      \
1545                        _np = con_tv;                          \
1546        } while (atomic_cas_ptr(&conn_tp, _p, _np) != _p);             \
1546        } while (casptr(&conn_tp, _p, _np) != _p);                     \
1547        _p->conn = (p);                                                \
1548        _p->action = (a);                                              \
1549        _p->ref = (p)->ref;                                            \
1550        _p->cpu = CPU->cpu_seqid;                                      \
1551        NCA_CONN_T_TRACE_STK();                                        \
1552 }
______unchanged_portion_omitted_

1752 extern nca_counter_t nca_counter_tv[];
1753 extern nca_counter_t *nca_counter_tp;

1755 #define NCA_COUNTER(_p, _v) {                                          \
1756        unsigned long   *p = _p;                                       \
1757        long            v = _v;                                        \
1758        unsigned long   _nv;                                           \
1759        nca_counter_t   *_otp;                                         \
1760        nca_counter_t   *_ntp;                                         \
1761                                                                       \
1762        _nv = atomic_add_long_nv(p, v);                                \
1763        do {                                                           \
1764                _otp = nca_counter_tp;                                 \
1765                _ntp = _otp + 1;                                       \
1766                if (_ntp == &nca_counter_tv[NCA_COUNTER_TRACE_SZ])     \
1767                        _ntp = nca_counter_tv;                         \
1768        } while (atomic_cas_ptr((void *)&nca_counter_tp, (void *)_otp, \
1768        } while (casptr((void *)&nca_counter_tp, (void *)_otp,         \
1769            (void *)_ntp) != (void *)_otp);                            \
1770        _ntp->t = gethrtime();                                         \
1771        _ntp->p = p;                                                   \
1772        _ntp->v = v;                                                   \
1773        _ntp->nv = _nv;                                                \
1774 }
______unchanged_portion_omitted_
```

```
**********************************************************
  172578 Mon Jul 28 07:43:46 2014
new/usr/src/uts/common/inet/tcp/tcp_input.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

1877 /*
1878  * In an ideal case of vertical partition in NUMA architecture, its
1879  * beneficial to have the listener and all the incoming connections
1880  * tied to the same squeue. The other constraint is that incoming
1881  * connections should be tied to the squeue attached to interrupted
1882  * CPU for obvious locality reason so this leaves the listener to
1883  * be tied to the same squeue. Our only problem is that when listener
1884  * is binding, the CPU that will get interrupted by the NIC whose
1885  * IP address the listener is binding to is not even known. So
1886  * the code below allows us to change that binding at the time the
1887  * CPU is interrupted by virtue of incoming connection's squeue.
1888  *
1889  * This is usefull only in case of a listener bound to a specific IP
1890  * address. For other kind of listeners, they get bound the
1891  * very first time and there is no attempt to rebind them.
1892  */
1893 void
1894 tcp_input_listener_unbound(void *arg, mblk_t *mp, void *arg2,
1895     ip_recv_attr_t *ira)
1896 {
1897         conn_t          *connp = (conn_t *)arg;
1898         squeue_t        *sqp = (squeue_t *)arg2;
1899         squeue_t        *new_sqp;
1900         uint32_t        conn_flags;

1902         /*
1903          * IP sets ira_sqp to either the senders conn_sqp (for loopback)
1904          * or based on the ring (for packets from GLD). Otherwise it is
1905          * set based on lbolt i.e., a somewhat random number.
1906          */
1907         ASSERT(ira->ira_sqp != NULL);
1908         new_sqp = ira->ira_sqp;

1910         if (connp->conn_fanout == NULL)
1911                 goto done;

1913         if (!(connp->conn_flags & IPCL_FULLY_BOUND)) {
1914                 mutex_enter(&connp->conn_fanout->connf_lock);
1915                 mutex_enter(&connp->conn_lock);
1916                 /*
1917                  * No one from read or write side can access us now
1918                  * except for already queued packets on this squeue.
1919                  * But since we haven't changed the squeue yet, they
1920                  * can't execute. If they are processed after we have
1921                  * changed the squeue, they are sent back to the
1922                  * correct squeue down below.
1923                  * But a listner close can race with processing of
1924                  * incoming SYN. If incoming SYN processing changes
1925                  * the squeue then the listener close which is waiting
1926                  * to enter the squeue would operate on the wrong
1927                  * squeue. Hence we don't change the squeue here unless
1928                  * the refcount is exactly the minimum refcount. The
1929                  * minimum refcount of 4 is counted as - 1 each for
1930                  * TCP and IP, 1 for being in the classifier hash, and
1931                  * 1 for the mblk being processed.
1932                  */

1934                 if (connp->conn_ref != 4 ||
1935                     connp->conn_tcp->tcp_state != TCPS_LISTEN) {
```



```
1936                         mutex_exit(&connp->conn_lock);
1937                         mutex_exit(&connp->conn_fanout->connf_lock);
1938                         goto done;
1939                 }
1940                 if (connp->conn_sqp != new_sqp) {
1941                         while (connp->conn_sqp != new_sqp)
1942                                 (void) atomic_cas_ptr(&connp->conn_sqp, sqp,
1943                                     new_sqp);
1942                                 (void) casptr(&connp->conn_sqp, sqp, new_sqp);
1944                         /* No special MT issues for outbound ixa_sqp hint */
1945                         connp->conn_ixa->ixa_sqp = new_sqp;
1946                 }

1948                 do {
1949                         conn_flags = connp->conn_flags;
1950                         conn_flags |= IPCL_FULLY_BOUND;
1951                         (void) atomic_cas_32(&connp->conn_flags,
1952                             connp->conn_flags, conn_flags);
1950                         (void) cas32(&connp->conn_flags, connp->conn_flags,
1951                             conn_flags);
1953                 } while (!(connp->conn_flags & IPCL_FULLY_BOUND));

1955                 mutex_exit(&connp->conn_fanout->connf_lock);
1956                 mutex_exit(&connp->conn_lock);

1958                 /*
1959                  * Assume we have picked a good squeue for the listener. Make
1960                  * subsequent SYNs not try to change the squeue.
1961                  */
1962                 connp->conn_recv = tcp_input_listener;
1963         }

1965 done:
1966         if (connp->conn_sqp != sqp) {
1967                 CONN_INC_REF(connp);
1968                 SQUEUE_ENTER_ONE(connp->conn_sqp, mp, connp->conn_recv, connp,
1969                     ira, SQ_FILL, SQTAG_TCP_CONN_REQ_UNBOUND);
1970         } else {
1971                 tcp_input_listener(connp, mp, sqp, ira);
1972         }
1973 }
______unchanged_portion_omitted_
```

```
**********************************************************
    3472 Mon Jul 28 07:43:47 2014
new/usr/src/uts/common/io/bge/bge_atomic.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * 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 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include "bge_impl.h"

/*
 * Atomically decrement a counter, but only if it will remain
 * strictly positive (greater than zero) afterwards.  We return
 * the decremented value if so, otherwise zero (in which case
 * the counter is unchanged).
 *
 * This is used for keeping track of available resources such
 * as transmit ring slots ...
 */
uint64_t
bge_atomic_reserve(uint64_t *count_p, uint64_t n)
{
        uint64_t oldval;
        uint64_t newval;

        /* ATOMICALLY */
        do {
                oldval = *count_p;
                newval = oldval - n;
                if (oldval <= n)
                        return (0);             /* no resources left    */
        } while (atomic_cas_64(count_p, oldval, newval) != oldval);
        } while (cas64(count_p, oldval, newval) != oldval);

        return (newval);
}

/*
 * Atomically increment a counter
 */
void
```



```
bge_atomic_renounce(uint64_t *count_p, uint64_t n)
{
        uint64_t oldval;
        uint64_t newval;

        /* ATOMICALLY */
        do {
                oldval = *count_p;
                newval = oldval + n;
        } while (atomic_cas_64(count_p, oldval, newval) != oldval);
        } while (cas64(count_p, oldval, newval) != oldval);
}

/*
 * Atomically claim a slot in a descriptor ring
 */
uint64_t
bge_atomic_claim(uint64_t *count_p, uint64_t limit)
{
        uint64_t oldval;
        uint64_t newval;

        /* ATOMICALLY */
        do {
                oldval = *count_p;
                newval = NEXT(oldval, limit);
        } while (atomic_cas_64(count_p, oldval, newval) != oldval);
        } while (cas64(count_p, oldval, newval) != oldval);

        return (oldval);
}

/*
 * Atomically NEXT a 64-bit integer, returning the
 * value it had *before* the NEXT was applied
 */
uint64_t
bge_atomic_next(uint64_t *sp, uint64_t limit)
{
        uint64_t oldval;
        uint64_t newval;

        /* ATOMICALLY */
        do {
                oldval = *sp;
                newval = NEXT(oldval, limit);
        } while (atomic_cas_64(sp, oldval, newval) != oldval);
        } while (cas64(sp, oldval, newval) != oldval);

        return (oldval);
}

/*
 * Atomically decrement a counter
 */
void
bge_atomic_sub64(uint64_t *count_p, uint64_t n)
{
        uint64_t oldval;
        uint64_t newval;

        /* ATOMICALLY */
        do {
                oldval = *count_p;
                newval = oldval - n;
        } while (atomic_cas_64(count_p, oldval, newval) != oldval);
```

```
123         } while (cas64(count_p, oldval, newval) != oldval);
122 }

124 /*
125  * Atomically clear bits in a 64-bit word, returning
126  * the value it had *before* the bits were cleared.
127  */
128 uint64_t
129 bge_atomic_clr64(uint64_t *sp, uint64_t bits)
130 {
131         uint64_t oldval;
132         uint64_t newval;

134         /* ATOMICALLY */
135         do {
136                 oldval = *sp;
137                 newval = oldval & ~bits;
138         } while (atomic_cas_64(sp, oldval, newval) != oldval);
140         } while (cas64(sp, oldval, newval) != oldval);

140         return (oldval);
141 }

143 /*
144  * Atomically shift a 32-bit word left, returning
145  * the value it had *before* the shift was applied
146  */
147 uint32_t
148 bge_atomic_shl32(uint32_t *sp, uint_t count)
149 {
150         uint32_t oldval;
151         uint32_t newval;

153         /* ATOMICALLY */
154         do {
155                 oldval = *sp;
156                 newval = oldval << count;
157         } while (atomic_cas_32(sp, oldval, newval) != oldval);
159         } while (cas32(sp, oldval, newval) != oldval);

159         return (oldval);
160 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   25886 Mon Jul 28 07:43:47 2014
new/usr/src/uts/common/io/hxge/hxge_send.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

  87 static int
  88 hxge_start(p_hxge_t hxgep, p_tx_ring_t tx_ring_p, p_mblk_t mp)
  89 {
  90         int                     dma_status, status = 0;
  91         p_tx_desc_t             tx_desc_ring_vp;
  92         hpi_handle_t            hpi_desc_handle;
  93         hxge_os_dma_handle_t    tx_desc_dma_handle;
  94         p_tx_desc_t             tx_desc_p;
  95         p_tx_msg_t              tx_msg_ring;
  96         p_tx_msg_t              tx_msg_p;
  97         tx_desc_t               tx_desc, *tmp_desc_p;
  98         tx_desc_t               sop_tx_desc, *sop_tx_desc_p;
  99         p_tx_pkt_header_t       hdrp;
 100         p_tx_pkt_hdr_all_t      pkthdrp;
 101         uint8_t                 npads = 0;
 102         uint64_t                dma_ioaddr;
 103         uint32_t                dma_flags;
 104         int                     last_bidx;
 105         uint8_t                 *b_rptr;
 106         caddr_t                 kaddr;
 107         uint32_t                nmblks;
 108         uint32_t                ngathers;
 109         uint32_t                clen;
 110         int                     len;
 111         uint32_t                pkt_len, pack_len, min_len;
 112         uint32_t                bcopy_thresh;
 113         int                     i, cur_index, sop_index;
 114         uint16_t                tail_index;
 115         boolean_t               tail_wrap = B_FALSE;
 116         hxge_dma_common_t       desc_area;
 117         hxge_os_dma_handle_t    dma_handle;
 118         ddi_dma_cookie_t        dma_cookie;
 119         hpi_handle_t            hpi_handle;
 120         p_mblk_t                nmp;
 121         p_mblk_t                t_mp;
 122         uint32_t                ncookies;
 123         boolean_t               good_packet;
 124         boolean_t               mark_mode = B_FALSE;
 125         p_hxge_stats_t          statsp;
 126         p_hxge_tx_ring_stats_t  tdc_stats;
 127         t_uscalar_t             start_offset = 0;
 128         t_uscalar_t             stuff_offset = 0;
 129         t_uscalar_t             end_offset = 0;
 130         t_uscalar_t             value = 0;
 131         t_uscalar_t             cksum_flags = 0;
 132         boolean_t               cksum_on = B_FALSE;
 133         uint32_t                boff = 0;
 134         uint64_t                tot_xfer_len = 0, tmp_len = 0;
 135         boolean_t               header_set = B_FALSE;
 136         tdc_tdr_kick_t          kick;
 137         uint32_t                offset;
 138 #ifdef HXGE_DEBUG
 139         p_tx_desc_t             tx_desc_ring_pp;
 140         p_tx_desc_t             tx_desc_pp;
 141         tx_desc_t               *save_desc_p;
 142         int                     dump_len;
 143         int                     sad_len;
 144         uint64_t                sad;
 145         int                     xfer_len;
```



```
 146         uint32_t                msgsize;
 147 #endif

 149         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 150             "==> hxge_start: tx dma channel %d", tx_ring_p->tdc));
 151         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 152             "==> hxge_start: Starting tdc %d desc pending %d",
 153             tx_ring_p->tdc, tx_ring_p->descs_pending));

 155         statsp = hxgep->statsp;

 157         if (hxgep->statsp->port_stats.lb_mode == hxge_lb_normal) {
 158                 if (!statsp->mac_stats.link_up) {
 159                         freemsg(mp);
 160                         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start: "
 161                             "link not up or LB mode"));
 162                         goto hxge_start_fail1;
 163                 }
 164         }

 166         mac_hcksum_get(mp, &start_offset, &stuff_offset, &end_offset, &value,
 167             &cksum_flags);
 168         if (!HXGE_IS_VLAN_PACKET(mp->b_rptr)) {
 169                 start_offset += sizeof (ether_header_t);
 170                 stuff_offset += sizeof (ether_header_t);
 171         } else {
 172                 start_offset += sizeof (struct ether_vlan_header);
 173                 stuff_offset += sizeof (struct ether_vlan_header);
 174         }

 176         if (cksum_flags & HCK_PARTIALCKSUM) {
 177                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
 178                     "==> hxge_start: mp $%p len %d "
 179                     "cksum_flags 0x%x (partial checksum) ",
 180                     mp, MBLKL(mp), cksum_flags));
 181                 cksum_on = B_TRUE;
 182         }

 184         MUTEX_ENTER(&tx_ring_p->lock);
 185 start_again:
 186         ngathers = 0;
 187         sop_index = tx_ring_p->wr_index;
 188 #ifdef  HXGE_DEBUG
 189         if (tx_ring_p->descs_pending) {
 190                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
 191                     "==> hxge_start: desc pending %d ",
 192                     tx_ring_p->descs_pending));
 193         }

 195         dump_len = (int)(MBLKL(mp));
 196         dump_len = (dump_len > 128) ? 128: dump_len;

 198         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 199             "==> hxge_start: tdc %d: dumping ...: b_rptr $%p "
 200             "(Before header reserve: ORIGINAL LEN %d)",
 201             tx_ring_p->tdc, mp->b_rptr, dump_len));

 203         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 204             "==> hxge_start: dump packets (IP ORIGINAL b_rptr $%p): %s",
 205             mp->b_rptr, hxge_dump_packet((char *)mp->b_rptr, dump_len)));
 206 #endif

 208         tdc_stats = tx_ring_p->tdc_stats;
 209         mark_mode = (tx_ring_p->descs_pending &&
 210             ((tx_ring_p->tx_ring_size - tx_ring_p->descs_pending) <
 211             hxge_tx_minfree));
```

```
213         HXGE_DEBUG_MSG((hxgep, TX_CTL,
214             "TX Descriptor ring is channel %d mark mode %d",
215             tx_ring_p->tdc, mark_mode));
216
217         if (!hxge_txdma_reclaim(hxgep, tx_ring_p, hxge_tx_minfree)) {
218                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
219                     "TX Descriptor ring is full: channel %d", tx_ring_p->tdc));
220                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
221                     "TX Descriptor ring is full: channel %d", tx_ring_p->tdc));
222                 (void) atomic_cas_32((uint32_t *)&tx_ring_p->queueing, 0, 1);
222                 (void) cas32((uint32_t *)&tx_ring_p->queueing, 0, 1);
223                 tdc_stats->tx_no_desc++;
224                 MUTEX_EXIT(&tx_ring_p->lock);
225                 status = 1;
226                 goto hxge_start_fail1;
227         }
228
229         nmp = mp;
230         i = sop_index = tx_ring_p->wr_index;
231         nmblks = 0;
232         ngathers = 0;
233         pkt_len = 0;
234         pack_len = 0;
235         clen = 0;
236         last_bidx = -1;
237         good_packet = B_TRUE;
238
239         desc_area = tx_ring_p->tdc_desc;
240         hpi_handle = desc_area.hpi_handle;
241         hpi_desc_handle.regh = (hxge_os_acc_handle_t)
242             DMA_COMMON_ACC_HANDLE(desc_area);
243         hpi_desc_handle.hxgep = hxgep;
244         tx_desc_ring_vp = (p_tx_desc_t)DMA_COMMON_VPTR(desc_area);
245 #ifdef  HXGE_DEBUG
246 #if defined(__i386)
247         tx_desc_ring_pp = (p_tx_desc_t)(uint32_t)DMA_COMMON_IOADDR(desc_area);
248 #else
249         tx_desc_ring_pp = (p_tx_desc_t)DMA_COMMON_IOADDR(desc_area);
250 #endif
251 #endif
252         tx_desc_dma_handle = (hxge_os_dma_handle_t)DMA_COMMON_HANDLE(desc_area);
253         tx_msg_ring = tx_ring_p->tx_msg_ring;
254
255         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start: wr_index %d i %d",
256             sop_index, i));
257
258 #ifdef  HXGE_DEBUG
259         msgsize = msgdsize(nmp);
260         HXGE_DEBUG_MSG((hxgep, TX_CTL,
261             "==> hxge_start(1): wr_index %d i %d msgdsize %d",
262             sop_index, i, msgsize));
263 #endif
264         /*
265          * The first 16 bytes of the premapped buffer are reserved
266          * for header. No padding will be used.
267          */
268         pkt_len = pack_len = boff = TX_PKT_HEADER_SIZE;
269         if (hxge_tx_use_bcopy) {
270                 bcopy_thresh = (hxge_bcopy_thresh - TX_PKT_HEADER_SIZE);
271         } else {
272                 bcopy_thresh = (TX_BCOPY_SIZE - TX_PKT_HEADER_SIZE);
273         }
274         while (nmp) {
275                 good_packet = B_TRUE;
276                 b_rptr = nmp->b_rptr;
```



```
277                 len = MBLKL(nmp);
278                 if (len <= 0) {
279                         nmp = nmp->b_cont;
280                         continue;
281                 }
282                 nmblks++;
283
284                 HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start(1): nmblks %d "
285                     "len %d pkt_len %d pack_len %d",
286                     nmblks, len, pkt_len, pack_len));
287                 /*
288                  * Hardware limits the transfer length to 4K.
289                  * If len is more than 4K, we need to break
290                  * nmp into two chunks: Make first chunk smaller
291                  * than 4K. The second chunk will be broken into
292                  * less than 4K (if needed) during the next pass.
293                  */
294                 if (len > (TX_MAX_TRANSFER_LENGTH - TX_PKT_HEADER_SIZE)) {
295                         if ((t_mp = dupb(nmp)) != NULL) {
296                                 nmp->b_wptr = nmp->b_rptr +
297                                     (TX_MAX_TRANSFER_LENGTH -
298                                     TX_PKT_HEADER_SIZE);
299                                 t_mp->b_rptr = nmp->b_wptr;
300                                 t_mp->b_cont = nmp->b_cont;
301                                 nmp->b_cont = t_mp;
302                                 len = MBLKL(nmp);
303                         } else {
304                                 good_packet = B_FALSE;
305                                 goto hxge_start_fail2;
306                         }
307                 }
308                 tx_desc.value = 0;
309                 tx_desc_p = &tx_desc_ring_vp[i];
310 #ifdef  HXGE_DEBUG
311                 tx_desc_pp = &tx_desc_ring_pp[i];
312 #endif
313                 tx_msg_p = &tx_msg_ring[i];
314 #if defined(__i386)
315                 hpi_desc_handle.regp = (uint32_t)tx_desc_p;
316 #else
317                 hpi_desc_handle.regp = (uint64_t)tx_desc_p;
318 #endif
319                 if (!header_set &&
320                     ((!hxge_tx_use_bcopy && (len > TX_BCOPY_SIZE)) ||
321                     (len >= bcopy_thresh))) {
322                         header_set = B_TRUE;
323                         bcopy_thresh += TX_PKT_HEADER_SIZE;
324                         boff = 0;
325                         pack_len = 0;
326                         kaddr = (caddr_t)DMA_COMMON_VPTR(tx_msg_p->buf_dma);
327                         hdrp = (p_tx_pkt_header_t)kaddr;
328                         clen = pkt_len;
329                         dma_handle = tx_msg_p->buf_dma_handle;
330                         dma_ioaddr = DMA_COMMON_IOADDR(tx_msg_p->buf_dma);
331                         offset = tx_msg_p->offset_index * hxge_bcopy_thresh;
332                         (void) ddi_dma_sync(dma_handle,
333                             offset, hxge_bcopy_thresh, DDI_DMA_SYNC_FORDEV);
334
335                         tx_msg_p->flags.dma_type = USE_BCOPY;
336                         goto hxge_start_control_header_only;
337                 }
338
339                 pkt_len += len;
340                 pack_len += len;
341
342                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
```

```
343                     "==> hxge_start(3): desc entry %d DESC IOADDR $%p "
344                     "desc_vp $%p tx_desc_p $%p desc_pp $%p tx_desc_pp $%p "
345                     "len %d pkt_len %d pack_len %d",
346                     i,
347                     DMA_COMMON_IOADDR(desc_area),
348                     tx_desc_ring_vp, tx_desc_p,
349                     tx_desc_ring_pp, tx_desc_pp,
350                     len, pkt_len, pack_len));

352                 if (len < bcopy_thresh) {
353                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
354                             "==> hxge_start(4): USE BCOPY: "));
355                         if (hxge_tx_tiny_pack) {
356                                 uint32_t blst = TXDMA_DESC_NEXT_INDEX(i, -1,
357                                     tx_ring_p->tx_wrap_mask);
358                                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
359                                     "==> hxge_start(5): pack"));
360                                 if ((pack_len <= bcopy_thresh) &&
361                                     (last_bidx == blst)) {
362                                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
363                                             "==> hxge_start: pack(6) "
364                                             "(pkt_len %d pack_len %d)",
365                                             pkt_len, pack_len));
366                                         i = blst;
367                                         tx_desc_p = &tx_desc_ring_vp[i];
368 #ifdef  HXGE_DEBUG
369                                         tx_desc_pp = &tx_desc_ring_pp[i];
370 #endif
371                                         tx_msg_p = &tx_msg_ring[i];
372                                         boff = pack_len - len;
373                                         ngathers--;
374                                 } else if (pack_len > bcopy_thresh &&
375                                     header_set) {
376                                         pack_len = len;
377                                         boff = 0;
378                                         bcopy_thresh = hxge_bcopy_thresh;
379                                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
380                                             "==> hxge_start(7): > max NEW "
381                                             "bcopy thresh %d "
382                                             "pkt_len %d pack_len %d(next)",
383                                             bcopy_thresh, pkt_len, pack_len));
384                                 }
385                                 last_bidx = i;
386                         }
387                         kaddr = (caddr_t)DMA_COMMON_VPTR(tx_msg_p->buf_dma);
388                         if ((boff == TX_PKT_HEADER_SIZE) && (nmblks == 1)) {
389                                 hdrp = (p_tx_pkt_header_t)kaddr;
390                                 header_set = B_TRUE;
391                                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
392                                     "==> hxge_start(7_x2): "
393                                     "pkt_len %d pack_len %d (new hdrp $%p)",
394                                     pkt_len, pack_len, hdrp));
395                         }
396                         tx_msg_p->flags.dma_type = USE_BCOPY;
397                         kaddr += boff;
398                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
399                             "==> hxge_start(8): USE BCOPY: before bcopy "
400                             "DESC IOADDR $%p entry %d bcopy packets %d "
401                             "bcopy kaddr $%p bcopy ioaddr (SAD) $%p "
402                             "bcopy clen %d bcopy boff %d",
403                             DMA_COMMON_IOADDR(desc_area), i,
404                             tdc_stats->tx_hdr_pkts, kaddr, dma_ioaddr,
405                             clen, boff));
406                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
407                             "==> hxge_start: 1USE BCOPY: "));
408                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
```



```
409                             "==> hxge_start: 2USE BCOPY: "));
410                         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start: "
411                             "last USE BCOPY: copy from b_rptr $%p "
412                             "to KADDR $%p (len %d offset %d",
413                             b_rptr, kaddr, len, boff));
414                         bcopy(b_rptr, kaddr, len);
415 #ifdef  HXGE_DEBUG
416                         dump_len = (len > 128) ? 128: len;
417                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
418                             "==> hxge_start: dump packets "
419                             "(After BCOPY len %d)"
420                             "(b_rptr $%p): %s", len, nmp->b_rptr,
421                             hxge_dump_packet((char *)nmp->b_rptr,
422                             dump_len)));
423 #endif
424                         dma_handle = tx_msg_p->buf_dma_handle;
425                         dma_ioaddr = DMA_COMMON_IOADDR(tx_msg_p->buf_dma);
426                         offset = tx_msg_p->offset_index * hxge_bcopy_thresh;
427                         (void) ddi_dma_sync(dma_handle,
428                             offset, hxge_bcopy_thresh, DDI_DMA_SYNC_FORDEV);
429                         clen = len + boff;
430                         tdc_stats->tx_hdr_pkts++;
431                         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start(9): "
432                             "USE BCOPY: DESC IOADDR $%p entry %d "
433                             "bcopy packets %d bcopy kaddr $%p "
434                             "bcopy ioaddr (SAD) $%p bcopy clen %d "
435                             "bcopy boff %d",
436                             DMA_COMMON_IOADDR(desc_area), i,
437                             tdc_stats->tx_hdr_pkts, kaddr, dma_ioaddr,
438                             clen, boff));
439                 } else {
440                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
441                             "==> hxge_start(12): USE DVMA: len %d", len));
442                         tx_msg_p->flags.dma_type = USE_DMA;
443                         dma_flags = DDI_DMA_WRITE;
444                         if (len < hxge_dma_stream_thresh) {
445                                 dma_flags |= DDI_DMA_CONSISTENT;
446                         } else {
447                                 dma_flags |= DDI_DMA_STREAMING;
448                         }

450                         dma_handle = tx_msg_p->dma_handle;
451                         dma_status = ddi_dma_addr_bind_handle(dma_handle, NULL,
452                             (caddr_t)b_rptr, len, dma_flags,
453                             DDI_DMA_DONTWAIT, NULL,
454                             &dma_cookie, &ncookies);
455                         if (dma_status == DDI_DMA_MAPPED) {
456                                 dma_ioaddr = dma_cookie.dmac_laddress;
457                                 len = (int)dma_cookie.dmac_size;
458                                 clen = (uint32_t)dma_cookie.dmac_size;
459                                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
460                                     "==> hxge_start(12_1): "
461                                     "USE DVMA: len %d clen %d ngathers %d",
462                                     len, clen, ngathers));
463 #if defined(__i386)
464                                 hpi_desc_handle.regp = (uint32_t)tx_desc_p;
465 #else
466                                 hpi_desc_handle.regp = (uint64_t)tx_desc_p;
467 #endif
468                                 while (ncookies > 1) {
469                                         ngathers++;
470                                         /*
471                                          * this is the fix for multiple
472                                          * cookies, which are basically
473                                          * a descriptor entry, we don't set
474                                          * SOP bit as well as related fields
```

```
475                                      */
477                                     (void) hpi_txdma_desc_gather_set(
478                                         hpi_desc_handle, &tx_desc,
479                                         (ngathers -1), mark_mode,
480                                         ngathers, dma_ioaddr, clen);
481                                     tx_msg_p->tx_msg_size = clen;
482                                     HXGE_DEBUG_MSG((hxgep, TX_CTL,
483                                         "==> hxge_start:  DMA "
484                                         "ncookie %d ngathers %d "
485                                         "dma_ioaddr $%p len %d"
486                                         "desc $%p descp $%p (%d)",
487                                         ncookies, ngathers,
488                                         dma_ioaddr, clen,
489                                         *tx_desc_p, tx_desc_p, i));
491                                     ddi_dma_nextcookie(dma_handle,
492                                         &dma_cookie);
493                                     dma_ioaddr = dma_cookie.dmac_laddress;
495                                     len = (int)dma_cookie.dmac_size;
496                                     clen = (uint32_t)dma_cookie.dmac_size;
497                                     HXGE_DEBUG_MSG((hxgep, TX_CTL,
498                                         "==> hxge_start(12_2): "
499                                         "USE DVMA: len %d clen %d ",
500                                         len, clen));
502                                     i = TXDMA_DESC_NEXT_INDEX(i, 1,
503                                         tx_ring_p->tx_wrap_mask);
504                                     tx_desc_p = &tx_desc_ring_vp[i];
506                                     hpi_desc_handle.regp =
507 #if defined(__i386)
508                                         (uint32_t)tx_desc_p;
509 #else
510                                             (uint64_t)tx_desc_p;
511 #endif
512                                     tx_msg_p = &tx_msg_ring[i];
513                                     tx_msg_p->flags.dma_type = USE_NONE;
514                                     tx_desc.value = 0;
515                                     ncookies--;
516                                 }
517                                 tdc_stats->tx_ddi_pkts++;
518                                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
519                                     "==> hxge_start: DMA: ddi packets %d",
520                                     tdc_stats->tx_ddi_pkts));
521                         } else {
522                                 HXGE_ERROR_MSG((hxgep, HXGE_ERR_CTL,
523                                     "dma mapping failed for %d "
524                                     "bytes addr $%p flags %x (%d)",
525                                     len, b_rptr, status, status));
526                                 good_packet = B_FALSE;
527                                 tdc_stats->tx_dma_bind_fail++;
528                                 tx_msg_p->flags.dma_type = USE_NONE;
529                                 status = 1;
530                                 goto hxge_start_fail2;
531                         }
532                 } /* ddi dvma */
534                 nmp = nmp->b_cont;
535 hxge_start_control_header_only:
536 #if defined(__i386)
537                 hpi_desc_handle.regp = (uint32_t)tx_desc_p;
538 #else
539                 hpi_desc_handle.regp = (uint64_t)tx_desc_p;
540 #endif
```



```
541                 ngathers++;
543                 if (ngathers == 1) {
544 #ifdef  HXGE_DEBUG
545                         save_desc_p = &sop_tx_desc;
546 #endif
547                         sop_tx_desc_p = &sop_tx_desc;
548                         sop_tx_desc_p->value = 0;
549                         sop_tx_desc_p->bits.tr_len = clen;
550                         sop_tx_desc_p->bits.sad = dma_ioaddr >> 32;
551                         sop_tx_desc_p->bits.sad_l = dma_ioaddr & 0xffffffff;
552                 } else {
553 #ifdef  HXGE_DEBUG
554                         save_desc_p = &tx_desc;
555 #endif
556                         tmp_desc_p = &tx_desc;
557                         tmp_desc_p->value = 0;
558                         tmp_desc_p->bits.tr_len = clen;
559                         tmp_desc_p->bits.sad = dma_ioaddr >> 32;
560                         tmp_desc_p->bits.sad_l = dma_ioaddr & 0xffffffff;
562                         tx_desc_p->value = tmp_desc_p->value;
563                 }
565                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
566                     "==> hxge_start(13): Desc_entry %d ngathers %d "
567                     "desc_vp $%p tx_desc_p $%p "
568                     "len %d clen %d pkt_len %d pack_len %d nmblks %d "
569                     "dma_ioaddr (SAD) $%p mark %d",
570                     i, ngathers, tx_desc_ring_vp, tx_desc_p,
571                     len, clen, pkt_len, pack_len, nmblks,
572                     dma_ioaddr, mark_mode));
574 #ifdef HXGE_DEBUG
575                 hpi_desc_handle.hxgep = hxgep;
576                 hpi_desc_handle.function.function = 0;
577                 hpi_desc_handle.function.instance = hxgep->instance;
578                 sad = save_desc_p->bits.sad;
579                 sad = (sad << 32) | save_desc_p->bits.sad_l;
580                 xfer_len = save_desc_p->bits.tr_len;
582                 HXGE_DEBUG_MSG((hxgep, TX_CTL, "\n\t: value 0x%llx\n"
583                     "\t\tsad $%p\ttr_len %d len %d\tnptrs %d\t"
584                     "mark %d sop %d\n",
585                     save_desc_p->value, sad, save_desc_p->bits.tr_len,
586                     xfer_len, save_desc_p->bits.num_ptr,
587                     save_desc_p->bits.mark, save_desc_p->bits.sop));
589                 hpi_txdma_dump_desc_one(hpi_desc_handle, NULL, i);
590 #endif
592                 tx_msg_p->tx_msg_size = clen;
593                 i = TXDMA_DESC_NEXT_INDEX(i, 1, tx_ring_p->tx_wrap_mask);
594                 if (ngathers > hxge_tx_max_gathers) {
595                         good_packet = B_FALSE;
596                         mac_hcksum_get(mp, &start_offset, &stuff_offset,
597                             &end_offset, &value, &cksum_flags);
599                         HXGE_DEBUG_MSG((NULL, TX_CTL,
600                             "==> hxge_start(14): pull msg - "
601                             "len %d pkt_len %d ngathers %d",
602                             len, pkt_len, ngathers));
603                         goto hxge_start_fail2;
604                 }
605         } /* while (nmp) */
```

```
607         tx_msg_p->tx_message = mp;
608         tx_desc_p = &tx_desc_ring_vp[sop_index];
609 #if defined(__i386)
610         hpi_desc_handle.regp = (uint32_t)tx_desc_p;
611 #else
612         hpi_desc_handle.regp = (uint64_t)tx_desc_p;
613 #endif

615         pkthdrp = (p_tx_pkt_hdr_all_t)hdrp;
616         pkthdrp->reserved = 0;
617         hdrp->value = 0;
618         (void) hxge_fill_tx_hdr(mp, B_FALSE, cksum_on,
619             (pkt_len - TX_PKT_HEADER_SIZE), npads, pkthdrp);

621         /*
622          * Hardware header should not be counted as part of the frame
623          * when determining the frame size
624          */
625         if ((pkt_len - TX_PKT_HEADER_SIZE) > (STD_FRAME_SIZE - ETHERFCSL)) {
626                 tdc_stats->tx_jumbo_pkts++;
627         }

629         min_len = (hxgep->msg_min + TX_PKT_HEADER_SIZE + (npads * 2));
630         if (pkt_len < min_len) {
631                 /* Assume we use bcopy to premapped buffers */
632                 kaddr = (caddr_t)DMA_COMMON_VPTR(tx_msg_p->buf_dma);
633                 HXGE_DEBUG_MSG((NULL, TX_CTL,
634                     "==> hxge_start(14-1): < (msg_min + 16)"
635                     "len %d pkt_len %d min_len %d bzero %d ngathers %d",
636                     len, pkt_len, min_len, (min_len - pkt_len), ngathers));
637                 bzero((kaddr + pkt_len), (min_len - pkt_len));
638                 pkt_len = tx_msg_p->tx_msg_size = min_len;

640                 sop_tx_desc_p->bits.tr_len = min_len;

642                 HXGE_MEM_PIO_WRITE64(hpi_desc_handle, sop_tx_desc_p->value);
643                 tx_desc_p->value = sop_tx_desc_p->value;

645                 HXGE_DEBUG_MSG((NULL, TX_CTL,
646                     "==> hxge_start(14-2): < msg_min - "
647                     "len %d pkt_len %d min_len %d ngathers %d",
648                     len, pkt_len, min_len, ngathers));
649         }

651         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start: cksum_flags 0x%x ",
652             cksum_flags));
653         if (cksum_flags & HCK_PARTIALCKSUM) {
654                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
655                     "==> hxge_start: cksum_flags 0x%x (partial checksum) ",
656                     cksum_flags));
657                 cksum_on = B_TRUE;
658                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
659                     "==> hxge_start: from IP cksum_flags 0x%x "
660                     "(partial checksum) "
661                     "start_offset %d stuff_offset %d",
662                     cksum_flags, start_offset, stuff_offset));
663                 tmp_len = (uint64_t)(start_offset >> 1);
664                 hdrp->value |= (tmp_len << TX_PKT_HEADER_L4START_SHIFT);
665                 tmp_len = (uint64_t)(stuff_offset >> 1);
666                 hdrp->value |= (tmp_len << TX_PKT_HEADER_L4STUFF_SHIFT);

668                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
669                     "==> hxge_start: from IP cksum_flags 0x%x "
670                     "(partial checksum) "
671                     "after SHIFT start_offset %d stuff_offset %d",
672                     cksum_flags, start_offset, stuff_offset));
```



```
673         }

675         /*
676          * pkt_len already includes 16 + paddings!!
677          * Update the control header length
678          */

680         /*
681          * Note that Hydra is different from Neptune where
682          * tot_xfer_len = (pkt_len - TX_PKT_HEADER_SIZE);
683          */
684         tot_xfer_len = pkt_len;
685         tmp_len = hdrp->value |
686             (tot_xfer_len << TX_PKT_HEADER_TOT_XFER_LEN_SHIFT);

688         HXGE_DEBUG_MSG((hxgep, TX_CTL,
689             "==> hxge_start(15_x1): setting SOP "
690             "tot_xfer_len 0x%llx (%d) pkt_len %d tmp_len "
691             "0x%llx hdrp->value 0x%llx",
692             tot_xfer_len, tot_xfer_len, pkt_len, tmp_len, hdrp->value));
693 #if defined(_BIG_ENDIAN)
694         hdrp->value = ddi_swap64(tmp_len);
695 #else
696         hdrp->value = tmp_len;
697 #endif
698         HXGE_DEBUG_MSG((hxgep,
699             TX_CTL, "==> hxge_start(15_x2): setting SOP "
700             "after SWAP: tot_xfer_len 0x%llx pkt_len %d "
701             "tmp_len 0x%llx hdrp->value 0x%llx",
702             tot_xfer_len, pkt_len, tmp_len, hdrp->value));

704         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start(15): setting SOP "
705             "wr_index %d tot_xfer_len (%d) pkt_len %d npads %d",
706             sop_index, tot_xfer_len, pkt_len, npads));

708         sop_tx_desc_p->bits.sop = 1;
709         sop_tx_desc_p->bits.mark = mark_mode;
710         sop_tx_desc_p->bits.num_ptr = ngathers;

712         if (mark_mode)
713                 tdc_stats->tx_marks++;

715         HXGE_MEM_PIO_WRITE64(hpi_desc_handle, sop_tx_desc_p->value);
716         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start(16): set SOP done"));

718 #ifdef HXGE_DEBUG
719         hpi_desc_handle.hxgep = hxgep;
720         hpi_desc_handle.function.function = 0;
721         hpi_desc_handle.function.instance = hxgep->instance;

723         HXGE_DEBUG_MSG((hxgep, TX_CTL, "\n\t: value 0x%llx\n"
724             "\t\tsad $%p\ttr_len %d len %d\tnptrs %d\tmark %d sop %d\n",
725             save_desc_p->value, sad, save_desc_p->bits.tr_len,
726             xfer_len, save_desc_p->bits.num_ptr, save_desc_p->bits.mark,
727             save_desc_p->bits.sop));
728         (void) hpi_txdma_dump_desc_one(hpi_desc_handle, NULL, sop_index);

730         dump_len = (pkt_len > 128) ? 128: pkt_len;
731         HXGE_DEBUG_MSG((hxgep, TX_CTL,
732             "==> hxge_start: dump packets(17) (after sop set, len "
733             " (len/dump_len/pkt_len/tot_xfer_len) %d/%d/%d/%d):\n"
734             "ptr $%p: %s", len, dump_len, pkt_len, tot_xfer_len,
735             (char *)hdrp, hxge_dump_packet((char *)hdrp, dump_len)));
736         HXGE_DEBUG_MSG((hxgep, TX_CTL,
737             "==> hxge_start(18): TX desc sync: sop_index %d", sop_index));
738 #endif
```

```
740         if ((ngathers == 1) || tx_ring_p->wr_index < i) {
741                 (void) ddi_dma_sync(tx_desc_dma_handle,
742                     sop_index * sizeof (tx_desc_t),
743                     ngathers * sizeof (tx_desc_t), DDI_DMA_SYNC_FORDEV);

745                 HXGE_DEBUG_MSG((hxgep, TX_CTL, "hxge_start(19): sync 1 "
746                     "cs_off = 0x%02X cs_s_off = 0x%02X "
747                     "pkt_len %d ngathers %d sop_index %d\n",
748                     stuff_offset, start_offset,
749                     pkt_len, ngathers, sop_index));
750         } else { /* more than one descriptor and wrap around */
751                 uint32_t nsdescs = tx_ring_p->tx_ring_size - sop_index;
752                 (void) ddi_dma_sync(tx_desc_dma_handle,
753                     sop_index * sizeof (tx_desc_t),
754                     nsdescs * sizeof (tx_desc_t), DDI_DMA_SYNC_FORDEV);
755                 HXGE_DEBUG_MSG((hxgep, TX_CTL, "hxge_start(20): sync 1 "
756                     "cs_off = 0x%02X cs_s_off = 0x%02X "
757                     "pkt_len %d ngathers %d sop_index %d\n",
758                     stuff_offset, start_offset, pkt_len, ngathers, sop_index));

760                 (void) ddi_dma_sync(tx_desc_dma_handle, 0,
761                     (ngathers - nsdescs) * sizeof (tx_desc_t),
762                     DDI_DMA_SYNC_FORDEV);
763                 HXGE_DEBUG_MSG((hxgep, TX_CTL, "hxge_start(21): sync 2 "
764                     "cs_off = 0x%02X cs_s_off = 0x%02X "
765                     "pkt_len %d ngathers %d sop_index %d\n",
766                     stuff_offset, start_offset,
767                     pkt_len, ngathers, sop_index));
768         }

770         tail_index = tx_ring_p->wr_index;
771         tail_wrap = tx_ring_p->wr_index_wrap;

773         tx_ring_p->wr_index = i;
774         if (tx_ring_p->wr_index <= tail_index) {
775                 tx_ring_p->wr_index_wrap = ((tail_wrap == B_TRUE) ?
776                     B_FALSE : B_TRUE);
777         }

779         tx_ring_p->descs_pending += ngathers;
780         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start: TX kick: "
781             "channel %d wr_index %d wrap %d ngathers %d desc_pend %d",
782             tx_ring_p->tdc, tx_ring_p->wr_index, tx_ring_p->wr_index_wrap,
783             ngathers, tx_ring_p->descs_pending));
784         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start: TX KICKING: "));

786         kick.value = 0;
787         kick.bits.wrap = tx_ring_p->wr_index_wrap;
788         kick.bits.tail = (uint16_t)tx_ring_p->wr_index;

790         /* Kick start the Transmit kick register */
791         TXDMA_REG_WRITE64(HXGE_DEV_HPI_HANDLE(hxgep),
792             TDC_TDR_KICK, (uint8_t)tx_ring_p->tdc, kick.value);
793         tdc_stats->tx_starts++;
794         MUTEX_EXIT(&tx_ring_p->lock);
795         HXGE_DEBUG_MSG((hxgep, TX_CTL, "<== hxge_start"));
796         return (status);

798 hxge_start_fail2:
799         if (good_packet == B_FALSE) {
800                 cur_index = sop_index;
801                 HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_start: clean up"));
802                 for (i = 0; i < ngathers; i++) {
803                         tx_desc_p = &tx_desc_ring_vp[cur_index];
804 #if defined(__i386)
```



```
805                         hpi_handle.regp = (uint32_t)tx_desc_p;
806 #else
807                         hpi_handle.regp = (uint64_t)tx_desc_p;
808 #endif
809                         tx_msg_p = &tx_msg_ring[cur_index];
810                         (void) hpi_txdma_desc_set_zero(hpi_handle, 1);
811                         if (tx_msg_p->flags.dma_type == USE_DVMA) {
812                                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
813                                     "tx_desc_p = %X index = %d",
814                                     tx_desc_p, tx_ring_p->rd_index));
815                                 (void) dvma_unload(tx_msg_p->dvma_handle,
816                                     0, -1);
817                                 tx_msg_p->dvma_handle = NULL;
818                                 if (tx_ring_p->dvma_wr_index ==
819                                     tx_ring_p->dvma_wrap_mask)
820                                         tx_ring_p->dvma_wr_index = 0;
821                                 else
822                                         tx_ring_p->dvma_wr_index++;
823                                 tx_ring_p->dvma_pending--;
824                         } else if (tx_msg_p->flags.dma_type == USE_DMA) {
825                                 if (ddi_dma_unbind_handle(
826                                     tx_msg_p->dma_handle)) {
827                                         cmn_err(CE_WARN, "hxge_start: "
828                                             "ddi_dma_unbind_handle failed");
829                                 }
830                         }
831                         tx_msg_p->flags.dma_type = USE_NONE;
832                         cur_index = TXDMA_DESC_NEXT_INDEX(cur_index, 1,
833                             tx_ring_p->tx_wrap_mask);

835                 }
836         }

838         MUTEX_EXIT(&tx_ring_p->lock);

840 hxge_start_fail1:
841         /* Add FMA to check the access handle hxge_hregh */
842         HXGE_DEBUG_MSG((hxgep, TX_CTL, "<== hxge_start"));
843         return (status);
844 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   82932 Mon Jul 28 07:43:47 2014
new/usr/src/uts/common/io/hxge/hxge_txdma.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 601 boolean_t
 602 hxge_txdma_reclaim(p_hxge_t hxgep, p_tx_ring_t tx_ring_p, int nmblks)
 603 {
 604         boolean_t               status = B_TRUE;
 605         p_hxge_dma_common_t     tx_desc_dma_p;
 606         hxge_dma_common_t       desc_area;
 607         p_tx_desc_t             tx_desc_ring_vp;
 608         p_tx_desc_t             tx_desc_p;
 609         p_tx_desc_t             tx_desc_pp;
 610         tx_desc_t               r_tx_desc;
 611         p_tx_msg_t              tx_msg_ring;
 612         p_tx_msg_t              tx_msg_p;
 613         hpi_handle_t            handle;
 614         tdc_tdr_head_t          tx_head;
 615         uint32_t                pkt_len;
 616         uint_t                  tx_rd_index;
 617         uint16_t                head_index, tail_index;
 618         uint8_t                 tdc;
 619         boolean_t               head_wrap, tail_wrap;
 620         p_hxge_tx_ring_stats_t  tdc_stats;
 621         tdc_byte_cnt_t          byte_cnt;
 622         tdc_tdr_qlen_t          qlen;
 623         int                     rc;

 625         HXGE_DEBUG_MSG((hxgep, TX_CTL, "==> hxge_txdma_reclaim"));

 627         status = ((tx_ring_p->descs_pending < hxge_reclaim_pending) &&
 628             (nmblks != 0));
 629         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 630             "==> hxge_txdma_reclaim: pending %d  reclaim %d nmblks %d",
 631             tx_ring_p->descs_pending, hxge_reclaim_pending, nmblks));

 633         if (!status) {
 634                 tx_desc_dma_p = &tx_ring_p->tdc_desc;
 635                 desc_area = tx_ring_p->tdc_desc;
 636                 tx_desc_ring_vp = tx_desc_dma_p->kaddrp;
 637                 tx_desc_ring_vp = (p_tx_desc_t)DMA_COMMON_VPTR(desc_area);
 638                 tx_rd_index = tx_ring_p->rd_index;
 639                 tx_desc_p = &tx_desc_ring_vp[tx_rd_index];
 640                 tx_msg_ring = tx_ring_p->tx_msg_ring;
 641                 tx_msg_p = &tx_msg_ring[tx_rd_index];
 642                 tdc = tx_ring_p->tdc;
 643                 tdc_stats = tx_ring_p->tdc_stats;
 644                 if (tx_ring_p->descs_pending > tdc_stats->tx_max_pend) {
 645                         tdc_stats->tx_max_pend = tx_ring_p->descs_pending;
 646                 }
 647                 tail_index = tx_ring_p->wr_index;
 648                 tail_wrap = tx_ring_p->wr_index_wrap;

 650                 /*
 651                  * tdc_byte_cnt reg can be used to get bytes transmitted. It
 652                  * includes padding too in case of runt packets.
 653                  */
 654                 handle = HXGE_DEV_HPI_HANDLE(hxgep);
 655                 TXDMA_REG_READ64(handle, TDC_BYTE_CNT, tdc, &byte_cnt.value);
 656                 tdc_stats->obytes_with_pad += byte_cnt.bits.byte_count;

 658                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
 659                     "==> hxge_txdma_reclaim: tdc %d tx_rd_index %d "
```



```
 660                     "tail_index %d tail_wrap %d tx_desc_p $%p ($%p) ",
 661                     tdc, tx_rd_index, tail_index, tail_wrap,
 662                     tx_desc_p, (*(uint64_t *)tx_desc_p)));

 664                 /*
 665                  * Read the hardware maintained transmit head and wrap around
 666                  * bit.
 667                  */
 668                 TXDMA_REG_READ64(handle, TDC_TDR_HEAD, tdc, &tx_head.value);
 669                 head_index = tx_head.bits.head;
 670                 head_wrap = tx_head.bits.wrap;
 671                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
 672                     "==> hxge_txdma_reclaim: "
 673                     "tx_rd_index %d tail %d tail_wrap %d head %d wrap %d",
 674                     tx_rd_index, tail_index, tail_wrap, head_index, head_wrap));

 676                 /*
 677                  * For debug only. This can be used to verify the qlen and make
 678                  * sure the hardware is wrapping the Tdr correctly.
 679                  */
 680                 TXDMA_REG_READ64(handle, TDC_TDR_QLEN, tdc, &qlen.value);
 681                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
 682                     "==> hxge_txdma_reclaim: tdr_qlen %d tdr_pref_qlen %d",
 683                     qlen.bits.tdr_qlen, qlen.bits.tdr_pref_qlen));

 685                 if (head_index == tail_index) {
 686                         if (TXDMA_RING_EMPTY(head_index, head_wrap, tail_index,
 687                             tail_wrap) && (head_index == tx_rd_index)) {
 688                                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
 689                                     "==> hxge_txdma_reclaim: EMPTY"));
 690                                 return (B_TRUE);
 691                         }
 692                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 693                             "==> hxge_txdma_reclaim: Checking if ring full"));
 694                         if (TXDMA_RING_FULL(head_index, head_wrap, tail_index,
 695                             tail_wrap)) {
 696                                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
 697                                     "==> hxge_txdma_reclaim: full"));
 698                                 return (B_FALSE);
 699                         }
 700                 }
 701                 HXGE_DEBUG_MSG((hxgep, TX_CTL,
 702                     "==> hxge_txdma_reclaim: tx_rd_index and head_index"));

 704                 /* XXXX: limit the # of reclaims */
 705                 tx_desc_pp = &r_tx_desc;
 706                 while ((tx_rd_index != head_index) &&
 707                     (tx_ring_p->descs_pending != 0)) {
 708                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 709                             "==> hxge_txdma_reclaim: Checking if pending"));
 710                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 711                             "==> hxge_txdma_reclaim: descs_pending %d ",
 712                             tx_ring_p->descs_pending));
 713                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 714                             "==> hxge_txdma_reclaim: "
 715                             "(tx_rd_index %d head_index %d (tx_desc_p $%p)",
 716                             tx_rd_index, head_index, tx_desc_p));

 718                         tx_desc_pp->value = tx_desc_p->value;
 719                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
 720                             "==> hxge_txdma_reclaim: "
 721                             "(tx_rd_index %d head_index %d "
 722                             "tx_desc_p $%p (desc value 0x%llx) ",
 723                             tx_rd_index, head_index,
 724                             tx_desc_pp, (*(uint64_t *)tx_desc_pp)));
 725                         HXGE_DEBUG_MSG((hxgep, TX_CTL,
```

```
726                             "==> hxge_txdma_reclaim: dump desc:"));

728                     /*
729                      * tdc_byte_cnt reg can be used to get bytes
730                      * transmitted
731                      */
732                     pkt_len = tx_desc_pp->bits.tr_len;
733                     tdc_stats->obytes += pkt_len;
734                     tdc_stats->opackets += tx_desc_pp->bits.sop;
735                     HXGE_DEBUG_MSG((hxgep, TX_CTL,
736                         "==> hxge_txdma_reclaim: pkt_len %d "
737                         "tdc channel %d opackets %d",
738                         pkt_len, tdc, tdc_stats->opackets));

740                     if (tx_msg_p->flags.dma_type == USE_DVMA) {
741                             HXGE_DEBUG_MSG((hxgep, TX_CTL,
742                                 "tx_desc_p = $%p tx_desc_pp = $%p "
743                                 "index = %d",
744                                 tx_desc_p, tx_desc_pp,
745                                 tx_ring_p->rd_index));
746                             (void) dvma_unload(tx_msg_p->dvma_handle,
747                                 0, -1);
748                             tx_msg_p->dvma_handle = NULL;
749                             if (tx_ring_p->dvma_wr_index ==
750                                 tx_ring_p->dvma_wrap_mask) {
751                                     tx_ring_p->dvma_wr_index = 0;
752                             } else {
753                                     tx_ring_p->dvma_wr_index++;
754                             }
755                             tx_ring_p->dvma_pending--;
756                     } else if (tx_msg_p->flags.dma_type == USE_DMA) {
757                             HXGE_DEBUG_MSG((hxgep, TX_CTL,
758                                 "==> hxge_txdma_reclaim: USE DMA"));
759                             if (rc = ddi_dma_unbind_handle
760                                 (tx_msg_p->dma_handle)) {
761                                     cmn_err(CE_WARN, "hxge_reclaim: "
762                                         "ddi_dma_unbind_handle "
763                                         "failed. status %d", rc);
764                             }
765                     }

767                     HXGE_DEBUG_MSG((hxgep, TX_CTL,
768                         "==> hxge_txdma_reclaim: count packets"));

770                     /*
771                      * count a chained packet only once.
772                      */
773                     if (tx_msg_p->tx_message != NULL) {
774                             freemsg(tx_msg_p->tx_message);
775                             tx_msg_p->tx_message = NULL;
776                     }
777                     tx_msg_p->flags.dma_type = USE_NONE;
778                     tx_rd_index = tx_ring_p->rd_index;
779                     tx_rd_index = (tx_rd_index + 1) &
780                         tx_ring_p->tx_wrap_mask;
781                     tx_ring_p->rd_index = tx_rd_index;
782                     tx_ring_p->descs_pending--;
783                     tx_desc_p = &tx_desc_ring_vp[tx_rd_index];
784                     tx_msg_p = &tx_msg_ring[tx_rd_index];
785             }

787             status = (nmblks <= ((int)tx_ring_p->tx_ring_size -
788                 (int)tx_ring_p->descs_pending - TX_FULL_MARK));
789             if (status) {
790                     (void) atomic_cas_32((uint32_t *)&tx_ring_p->queueing,
791                         1, 0);
```



```
790                     (void) cas32((uint32_t *)&tx_ring_p->queueing, 1, 0);
792                 }
793             } else {
794                     status = (nmblks <= ((int)tx_ring_p->tx_ring_size -
795                         (int)tx_ring_p->descs_pending - TX_FULL_MARK));
796             }

798             HXGE_DEBUG_MSG((hxgep, TX_CTL,
799                 "<== hxge_txdma_reclaim status = 0x%08x", status));
800             return (status);
801 }
```

*unchanged_portion_omitted_*

****
10945 Mon Jul 28 07:43:47 2014
new/usr/src/uts/common/io/ksyms.c
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
206 /*
207  * Copy a snapshot of the kernel symbol table into the user's address space.
208  * The symbol table is copied in fragments so that we do not have to
209  * do a large kmem_alloc() which could fail/block if the kernel memory is
210  * fragmented.
211  */
212 /* ARGSUSED */
213 static int
214 ksyms_open(dev_t *devp, int flag, int otyp, struct cred *cred)
215 {
216         minor_t clone;
217         size_t size = 0;
218         size_t realsize;
219         char *addr;
220         void *hptr = NULL;
221         ksyms_buflist_hdr_t hdr;
222         bzero(&hdr, sizeof (struct ksyms_buflist_hdr));
223         list_create(&hdr.blist, PAGESIZE,
224             offsetof(ksyms_buflist_t, buflist_node));

226         if (getminor(*devp) != 0)
227                 return (ENXIO);

229         for (;;) {
230                 realsize = ksyms_snapshot(ksyms_bcopy, hptr, size);
231                 if (realsize <= size)
232                         break;
233                 size = realsize;
234                 size = ksyms_buflist_alloc(&hdr, size);
235                 if (size == 0)
236                         return (ENOMEM);
237                 hptr = (void *)&hdr;
238         }

240         addr = ksyms_mapin(&hdr, realsize);
241         ksyms_buflist_free(&hdr);
242         if (addr == NULL)
243                 return (EOVERFLOW);

245         /*
246          * Reserve a clone entry.  Note that we don't use clone 0
247          * since that's the "real" minor number.
248          */
249         for (clone = 1; clone < nksyms_clones; clone++) {
250                 if (atomic_cas_ptr(&ksyms_clones[clone].ksyms_base, 0, addr) ==
251                     0) {
250                 if (casptr(&ksyms_clones[clone].ksyms_base, 0, addr) == 0) {
252                         ksyms_clones[clone].ksyms_size = realsize;
253                         *devp = makedevice(getemajor(*devp), clone);
254                         (void) ddi_prop_update_int(*devp, ksyms_devi,
255                             "size", realsize);
256                         modunload_disable();
257                         return (0);
258                 }
259         }
260         cmn_err(CE_NOTE, "ksyms: too many open references");
261         (void) as_unmap(curproc->p_as, addr, roundup(realsize, PAGESIZE));
262         return (ENXIO);
263 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   40190 Mon Jul 28 07:43:47 2014
new/usr/src/uts/common/io/multidata.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License, Version 1.0 only
   6  * (the "License").  You may not use this file except in compliance
   7  * with the License.
   8  *
   9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10  * or http://www.opensolaris.org/os/licensing.
  11  * See the License for the specific language governing permissions
  12  * and limitations under the License.
  13  *
  14  * When distributing Covered Code, include this CDDL HEADER in each
  15  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16  * If applicable, add the following below this CDDL HEADER, with the
  17  * fields enclosed by brackets "[]" replaced with your own identifying
  18  * information: Portions Copyright [yyyy] [name of copyright owner]
  19  *
  20  * CDDL HEADER END
  21  */
  22 /*
  23  * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
  24  * Use is subject to license terms.
  25  */

  27 #pragma ident   "%Z%%M% %I%     %E% SMI"

  27 /*
  28  * Multidata, as described in the following papers:
  29  *
  30  * Adi Masputra,
  31  * Multidata V.2: VA-Disjoint Packet Extents Framework Interface
  32  * Design Specification.  August 2004.
  33  * Available as http://sac.sfbay/PSARC/2004/594/materials/mmd2.pdf.
  34  *
  35  * Adi Masputra,
  36  * Multidata Interface Design Specification.  Sep 2002.
  37  * Available as http://sac.sfbay/PSARC/2002/276/materials/mmd.pdf.
  38  *
  39  * Adi Masputra, Frank DiMambro, Kacheong Poon,
  40  * An Efficient Networking Transmit Mechanism for Solaris:
  41  * Multidata Transmit (MDT).  May 2002.
  42  * Available as http://sac.sfbay/PSARC/2002/276/materials/mdt.pdf.
  43  */

  45 #include <sys/types.h>
  46 #include <sys/stream.h>
  47 #include <sys/dlpi.h>
  48 #include <sys/stropts.h>
  49 #include <sys/strsun.h>
  50 #include <sys/strlog.h>
  51 #include <sys/strsubr.h>
  52 #include <sys/sysmacros.h>
  53 #include <sys/cmn_err.h>
  54 #include <sys/debug.h>
  55 #include <sys/kmem.h>
  56 #include <sys/atomic.h>

  58 #include <sys/multidata.h>
  59 #include <sys/multidata_impl.h>
```



```
  61 static int mmd_constructor(void *, void *, int);
  62 static void mmd_destructor(void *, void *);
  63 static int pdslab_constructor(void *, void *, int);
  64 static void pdslab_destructor(void *, void *);
  65 static int pattbl_constructor(void *, void *, int);
  66 static void pattbl_destructor(void *, void *);
  67 static void mmd_esballoc_free(caddr_t);
  68 static int mmd_copy_pattbl(patbkt_t *, multidata_t *, pdesc_t *, int);

  70 static boolean_t pbuf_ref_valid(multidata_t *, pdescinfo_t *);
  71 #pragma inline(pbuf_ref_valid)

  73 static boolean_t pdi_in_range(pdescinfo_t *, pdescinfo_t *);
  74 #pragma inline(pdi_in_range)

  76 static pdesc_t *mmd_addpdesc_int(multidata_t *, pdescinfo_t *, int *, int);
  77 #pragma inline(mmd_addpdesc_int)

  79 static void mmd_destroy_pattbl(patbkt_t **);
  80 #pragma inline(mmd_destroy_pattbl)

  82 static pattr_t *mmd_find_pattr(patbkt_t *, uint_t);
  83 #pragma inline(mmd_find_pattr)

  85 static pdesc_t *mmd_destroy_pdesc(multidata_t *, pdesc_t *);
  86 #pragma inline(mmd_destroy_pdesc)

  88 static pdesc_t *mmd_getpdesc(multidata_t *, pdesc_t *, pdescinfo_t *, uint_t,
  89     boolean_t);
  90 #pragma inline(mmd_getpdesc)

  92 static struct kmem_cache *mmd_cache;
  93 static struct kmem_cache *pd_slab_cache;
  94 static struct kmem_cache *pattbl_cache;

  96 int mmd_debug = 1;
  97 #define MMD_DEBUG(s)    if (mmd_debug > 0) cmn_err s

  99 /*
 100  * Set to this to true to bypass pdesc bounds checking.
 101  */
 102 boolean_t mmd_speed_over_safety = B_FALSE;

 104 /*
 105  * Patchable kmem_cache flags.
 106  */
 107 int mmd_kmem_flags = 0;
 108 int pdslab_kmem_flags = 0;
 109 int pattbl_kmem_flags = 0;

 111 /*
 112  * Alignment (in bytes) of our kmem caches.
 113  */
 114 #define MULTIDATA_CACHE_ALIGN   64

 116 /*
 117  * Default number of packet descriptors per descriptor slab.  Making
 118  * this too small will trigger more descriptor slab allocation; making
 119  * it too large will create too many unclaimed descriptors.
 120  */
 121 #define PDSLAB_SZ       15
 122 uint_t pdslab_sz = PDSLAB_SZ;

 124 /*
 125  * Default attribute hash table size.  It's okay to set this to a small
```

```
 126  * value (even to 1) because there aren't that many attributes currently
 127  * defined, and because we assume there won't be many attributes associated
 128  * with a Multidata at a given time.  Increasing the size will reduce
 129  * attribute search time (given a large number of attributes in a Multidata),
 130  * and decreasing it will reduce the memory footprints and the overhead
 131  * associated with managing the table.
 132  */
 133 #define PATTBL_SZ       1
 134 uint_t pattbl_sz = PATTBL_SZ;

 136 /*
 137  * Attribute hash key.
 138  */
 139 #define PATTBL_HASH(x, sz)      ((x) % (sz))

 141 /*
 142  * Structure that precedes each Multidata metadata.
 143  */
 144 struct mmd_buf_info {
 145         frtn_t  frp;            /* free routine */
 146         uint_t  buf_len;        /* length of kmem buffer */
 147 };
______unchanged_portion_omitted_

1263 /*
1264  * Add a global or local attribute to a Multidata.  Global attribute
1265  * association is specified by a NULL packet descriptor.
1266  */
1267 pattr_t *
1268 mmd_addpattr(multidata_t *mmd, pdesc_t *pd, pattrinfo_t *pai,
1269     boolean_t persistent, int kmflags)
1270 {
1271         patbkt_t **tbl_p;
1272         patbkt_t *tbl, *o_tbl;
1273         patbkt_t *bkt;
1274         pattr_t *pa;
1275         uint_t size;

1277         ASSERT(mmd != NULL);
1278         ASSERT(mmd->mmd_magic == MULTIDATA_MAGIC);
1279         ASSERT(pd == NULL || pd->pd_magic == PDESC_MAGIC);
1280         ASSERT(pai != NULL);

1282         /* pointer to the attribute hash table (local or global) */
1283         tbl_p = pd != NULL ? &(pd->pd_pattbl) : &(mmd->mmd_pattbl);

1285         /*
1286          * See if the hash table has not yet been created; if so,
1287          * we create the table and store its address atomically.
1288          */
1289         if ((tbl = *tbl_p) == NULL) {
1290                 tbl = kmem_cache_alloc(pattbl_cache, kmflags);
1291                 if (tbl == NULL)
1292                         return (NULL);

1294                 /* if someone got there first, use his table instead */
1295                 if ((o_tbl = atomic_cas_ptr(tbl_p, NULL, tbl)) != NULL) {
1297                 if ((o_tbl = casptr(tbl_p, NULL, tbl)) != NULL) {
1296                         kmem_cache_free(pattbl_cache, tbl);
1297                         tbl = o_tbl;
1298                 }
1299         }

1301         ASSERT(tbl->pbkt_tbl_sz > 0);
1302         bkt = &(tbl[PATTBL_HASH(pai->type, tbl->pbkt_tbl_sz)]);
```



```
1304         /* attribute of the same type already exists? */
1305         if ((pa = mmd_find_pattr(bkt, pai->type)) != NULL)
1306                 return (NULL);

1308         size = sizeof (*pa) + pai->len;
1309         if ((pa = kmem_zalloc(size, kmflags)) == NULL)
1310                 return (NULL);

1312         pa->pat_magic = PATTR_MAGIC;
1313         pa->pat_lock = &(bkt->pbkt_lock);
1314         pa->pat_mmd = mmd;
1315         pa->pat_buflen = size;
1316         pa->pat_type = pai->type;
1317         pai->buf = pai->len > 0 ? ((uchar_t *)(pa + 1)) : NULL;

1319         if (persistent)
1320                 pa->pat_flags = PATTR_PERSIST;

1322         /* insert attribute at end of hash chain */
1323         mutex_enter(&(bkt->pbkt_lock));
1324         insque(&(pa->pat_next), bkt->pbkt_pattr_q.ql_prev);
1325         mutex_exit(&(bkt->pbkt_lock));

1327         return (pa);
1328 }
______unchanged_portion_omitted_
```

```
**********************************************************
    1950 Mon Jul 28 07:43:48 2014
new/usr/src/uts/common/io/nge/nge_atomic.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * 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 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include "nge.h"

/*
 * Atomically decrement a counter, but only if it will remain
 * positive (>=0) afterwards.
 */
boolean_t
nge_atomic_decrease(uint64_t *count_p, uint64_t n)
{
        uint64_t oldval;
        uint64_t newval;

        /* ATOMICALLY */
        do {
                oldval = *count_p;
                newval = oldval - n;
                if (oldval < n)
                        return (B_FALSE);
        } while (atomic_cas_64(count_p, oldval, newval) != oldval);
        } while (cas64(count_p, oldval, newval) != oldval);

        return (B_TRUE);
}

/*
 * Atomically increment a counter
 */
void
nge_atomic_increase(uint64_t *count_p, uint64_t n)
{
        uint64_t oldval;
        uint64_t newval;
```



```
        /* ATOMICALLY */
        do {
                oldval = *count_p;
                newval = oldval + n;
        } while (atomic_cas_64(count_p, oldval, newval) != oldval);
        } while (cas64(count_p, oldval, newval) != oldval);
}


/*
 * Atomically shift a 32-bit word left, returning
 * the value it had *before* the shift was applied
 */
uint32_t
nge_atomic_shl32(uint32_t *sp, uint_t count)
{
        uint32_t oldval;
        uint32_t newval;

        /* ATOMICALLY */
        do {
                oldval = *sp;
                newval = oldval << count;
        } while (atomic_cas_32(sp, oldval, newval) != oldval);
        } while (cas32(sp, oldval, newval) != oldval);

        return (oldval);
}
```

_____unchanged_portion_omitted_

```
**********************************************************
   50092 Mon Jul 28 07:43:48 2014
new/usr/src/uts/common/io/nxge/nxge_send.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 135 int
 136 nxge_start(p_nxge_t nxgep, p_tx_ring_t tx_ring_p, p_mblk_t mp)
 137 {
 138         int                     dma_status, status = 0;
 139         p_tx_desc_t             tx_desc_ring_vp;
 140         npi_handle_t            npi_desc_handle;
 141         nxge_os_dma_handle_t    tx_desc_dma_handle;
 142         p_tx_desc_t             tx_desc_p;
 143         p_tx_msg_t              tx_msg_ring;
 144         p_tx_msg_t              tx_msg_p;
 145         tx_desc_t               tx_desc, *tmp_desc_p;
 146         tx_desc_t               sop_tx_desc, *sop_tx_desc_p;
 147         p_tx_pkt_header_t       hdrp;
 148         tx_pkt_hdr_all_t        tmp_hdrp;
 149         p_tx_pkt_hdr_all_t      pkthdrp;
 150         uint8_t                 npads = 0;
 151         uint64_t                dma_ioaddr;
 152         uint32_t                dma_flags;
 153         int                     last_bidx;
 154         uint8_t                 *b_rptr;
 155         caddr_t                 kaddr;
 156         uint32_t                nmblks;
 157         uint32_t                ngathers;
 158         uint32_t                clen;
 159         int                     len;
 160         uint32_t                pkt_len, pack_len, min_len;
 161         uint32_t                bcopy_thresh;
 162         int                     i, cur_index, sop_index;
 163         uint16_t                tail_index;
 164         boolean_t               tail_wrap = B_FALSE;
 165         nxge_dma_common_t       desc_area;
 166         nxge_os_dma_handle_t    dma_handle;
 167         ddi_dma_cookie_t        dma_cookie;
 168         npi_handle_t            npi_handle;
 169         p_mblk_t                nmp;
 170         p_mblk_t                t_mp;
 171         uint32_t                ncookies;
 172         boolean_t               good_packet;
 173         boolean_t               mark_mode = B_FALSE;
 174         p_nxge_stats_t          statsp;
 175         p_nxge_tx_ring_stats_t tdc_stats;
 176         t_uscalar_t             start_offset = 0;
 177         t_uscalar_t             stuff_offset = 0;
 178         t_uscalar_t             end_offset = 0;
 179         t_uscalar_t             value = 0;
 180         t_uscalar_t             cksum_flags = 0;
 181         boolean_t               cksum_on = B_FALSE;
 182         uint32_t                boff = 0;
 183         uint64_t                tot_xfer_len = 0;
 184         boolean_t               header_set = B_FALSE;
 185 #ifdef NXGE_DEBUG
 186         p_tx_desc_t             tx_desc_ring_pp;
 187         p_tx_desc_t             tx_desc_pp;
 188         tx_desc_t               *save_desc_p;
 189         int                     dump_len;
 190         int                     sad_len;
 191         uint64_t                sad;
 192         int                     xfer_len;
 193         uint32_t                msgsize;
```



```
 194 #endif
 195         p_mblk_t                mp_chain = NULL;
 196         boolean_t               is_lso = B_FALSE;
 197         boolean_t               lso_again;
 198         int                     cur_index_lso;
 199         p_mblk_t                nmp_lso_save;
 200         uint32_t                lso_ngathers;
 201         boolean_t               lso_tail_wrap = B_FALSE;

 203         NXGE_DEBUG_MSG((nxgep, TX_CTL,
 204             "==> nxge_start: tx dma channel %d", tx_ring_p->tdc));
 205         NXGE_DEBUG_MSG((nxgep, TX_CTL,
 206             "==> nxge_start: Starting tdc %d desc pending %d",
 207             tx_ring_p->tdc, tx_ring_p->descs_pending));

 209         statsp = nxgep->statsp;

 211         if (!isLDOMguest(nxgep)) {
 212                 switch (nxgep->mac.portmode) {
 213                 default:
 214                         if (nxgep->statsp->port_stats.lb_mode ==
 215                             nxge_lb_normal) {
 216                                 if (!statsp->mac_stats.link_up) {
 217                                         freemsg(mp);
 218                                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
 219                                             "==> nxge_start: "
 220                                             "link not up"));
 221                                         goto nxge_start_fail1;
 222                                 }
 223                         }
 224                         break;
 225                 case PORT_10G_FIBER:
 226                         /*
 227                          * For the following modes, check the link status
 228                          * before sending the packet out:
 229                          * nxge_lb_normal,
 230                          * nxge_lb_ext10g,
 231                          * nxge_lb_ext1000,
 232                          * nxge_lb_ext100,
 233                          * nxge_lb_ext10.
 234                          */
 235                         if (nxgep->statsp->port_stats.lb_mode <
 236                             nxge_lb_phy10g) {
 237                                 if (!statsp->mac_stats.link_up) {
 238                                         freemsg(mp);
 239                                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
 240                                             "==> nxge_start: "
 241                                             "link not up"));
 242                                         goto nxge_start_fail1;
 243                                 }
 244                         }
 245                         break;
 246                 }
 247         }

 249         if ((!(nxgep->drv_state & STATE_HW_INITIALIZED)) ||
 250             (nxgep->nxge_mac_state != NXGE_MAC_STARTED)) {
 251                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
 252                     "==> nxge_start: hardware not initialized or stopped"));
 253                 freemsg(mp);
 254                 goto nxge_start_fail1;
 255         }

 257         if (nxgep->soft_lso_enable) {
 258                 mp_chain = nxge_lso_eliminate(mp);
 259                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
```

```
260                     "==> nxge_start(0): LSO mp $%p mp_chain $%p",
261                     mp, mp_chain));
262                 if (mp_chain == NULL) {
263                         NXGE_ERROR_MSG((nxgep, TX_CTL,
264                             "==> nxge_send(0): NULL mp_chain $%p != mp $%p",
265                             mp_chain, mp));
266                         goto nxge_start_fail1;
267                 }
268                 if (mp_chain != mp) {
269                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
270                             "==> nxge_send(1): IS LSO mp_chain $%p != mp $%p",
271                             mp_chain, mp));
272                         is_lso = B_TRUE;
273                         mp = mp_chain;
274                         mp_chain = mp_chain->b_next;
275                         mp->b_next = NULL;
276                 }
277         }

279         mac_hcksum_get(mp, &start_offset, &stuff_offset, &end_offset,
280             &value, &cksum_flags);
281         if (!NXGE_IS_VLAN_PACKET(mp->b_rptr)) {
282                 start_offset += sizeof (ether_header_t);
283                 stuff_offset += sizeof (ether_header_t);
284         } else {
285                 start_offset += sizeof (struct ether_vlan_header);
286                 stuff_offset += sizeof (struct ether_vlan_header);
287         }

289         if (cksum_flags & HCK_PARTIALCKSUM) {
290                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
291                     "==> nxge_start: mp $%p len %d "
292                     "cksum_flags 0x%x (partial checksum) ",
293                     mp, MBLKL(mp), cksum_flags));
294                 cksum_on = B_TRUE;
295         }

297         pkthdrp = (p_tx_pkt_hdr_all_t)&tmp_hdrp;
298         pkthdrp->reserved = 0;
299         tmp_hdrp.pkthdr.value = 0;
300         nxge_fill_tx_hdr(mp, B_FALSE, cksum_on,
301             0, 0, pkthdrp,
302             start_offset, stuff_offset);

304         lso_again = B_FALSE;
305         lso_ngathers = 0;

307         MUTEX_ENTER(&tx_ring_p->lock);

309         if (isLDOMservice(nxgep)) {
310                 tx_ring_p->tx_ring_busy = B_TRUE;
311                 if (tx_ring_p->tx_ring_offline) {
312                         freemsg(mp);
313                         tx_ring_p->tx_ring_busy = B_FALSE;
314                         (void) atomic_swap_32(&tx_ring_p->tx_ring_offline,
315                             NXGE_TX_RING_OFFLINED);
316                         MUTEX_EXIT(&tx_ring_p->lock);
317                         return (status);
318                 }
319         }

321         cur_index_lso = tx_ring_p->wr_index;
322         lso_tail_wrap = tx_ring_p->wr_index_wrap;
323 start_again:
324         ngathers = 0;
325         sop_index = tx_ring_p->wr_index;
```



```
326 #ifdef  NXGE_DEBUG
327         if (tx_ring_p->descs_pending) {
328                 NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: "
329                     "desc pending %d ", tx_ring_p->descs_pending));
330         }

332         dump_len = (int)(MBLKL(mp));
333         dump_len = (dump_len > 128) ? 128: dump_len;

335         NXGE_DEBUG_MSG((nxgep, TX_CTL,
336             "==> nxge_start: tdc %d: dumping ...: b_rptr $%p "
337             "(Before header reserve: ORIGINAL LEN %d)",
338             tx_ring_p->tdc,
339             mp->b_rptr,
340             dump_len));

342         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: dump packets "
343             "(IP ORIGINAL b_rptr $%p): %s", mp->b_rptr,
344             nxge_dump_packet((char *)mp->b_rptr, dump_len)));
345 #endif

347         tdc_stats = tx_ring_p->tdc_stats;
348         mark_mode = (tx_ring_p->descs_pending &&
349             (((int)tx_ring_p->tx_ring_size - (int)tx_ring_p->descs_pending) <
350             (int)nxge_tx_minfree));

352         NXGE_DEBUG_MSG((nxgep, TX_CTL,
353             "TX Descriptor ring is channel %d mark mode %d",
354             tx_ring_p->tdc, mark_mode));

356         if ((tx_ring_p->descs_pending + lso_ngathers) >= nxge_reclaim_pending) {
357                 if (!nxge_txdma_reclaim(nxgep, tx_ring_p,
358                     (nxge_tx_minfree + lso_ngathers))) {
359                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
360                             "TX Descriptor ring is full: channel %d",
361                             tx_ring_p->tdc));
362                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
363                             "TX Descriptor ring is full: channel %d",
364                             tx_ring_p->tdc));
365                         if (is_lso) {
366                                 /*
367                                  * free the current mp and mp_chain if not FULL.
368                                  */
369                                 tdc_stats->tx_no_desc++;
370                                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
371                                     "LSO packet: TX Descriptor ring is full: "
372                                     "channel %d",
373                                     tx_ring_p->tdc));
374                                 goto nxge_start_fail_lso;
375                         } else {
376                                 (void) atomic_cas_32(
377                                     (uint32_t *)&tx_ring_p->queueing, 0, 1);
376                                 (void) cas32((uint32_t *)&tx_ring_p->queueing,
377                                     0, 1);
378                                 tdc_stats->tx_no_desc++;

380                                 if (isLDOMservice(nxgep)) {
381                                         tx_ring_p->tx_ring_busy = B_FALSE;
382                                         if (tx_ring_p->tx_ring_offline) {
383                                                 (void) atomic_swap_32(
384                                                     &tx_ring_p->tx_ring_offline,
385                                                     NXGE_TX_RING_OFFLINED);
386                                         }
387                                 }

389                                 MUTEX_EXIT(&tx_ring_p->lock);
```

```
390                                 status = 1;
391                                 goto nxge_start_fail1;
392                         }
393                 }
394         }

396         nmp = mp;
397         i = sop_index = tx_ring_p->wr_index;
398         nmblks = 0;
399         ngathers = 0;
400         pkt_len = 0;
401         pack_len = 0;
402         clen = 0;
403         last_bidx = -1;
404         good_packet = B_TRUE;

406         desc_area = tx_ring_p->tdc_desc;
407         npi_handle = desc_area.npi_handle;
408         npi_desc_handle.regh = (nxge_os_acc_handle_t)
409             DMA_COMMON_ACC_HANDLE(desc_area);
410         tx_desc_ring_vp = (p_tx_desc_t)DMA_COMMON_VPTR(desc_area);
411         tx_desc_dma_handle = (nxge_os_dma_handle_t)
412             DMA_COMMON_HANDLE(desc_area);
413         tx_msg_ring = tx_ring_p->tx_msg_ring;

415         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: wr_index %d i %d",
416             sop_index, i));

418 #ifdef  NXGE_DEBUG
419         msgsize = msgdsize(nmp);
420         NXGE_DEBUG_MSG((nxgep, TX_CTL,
421             "==> nxge_start(1): wr_index %d i %d msgdsize %d",
422             sop_index, i, msgsize));
423 #endif
424         /*
425          * The first 16 bytes of the premapped buffer are reserved
426          * for header. No padding will be used.
427          */
428         pkt_len = pack_len = boff = TX_PKT_HEADER_SIZE;
429         if (nxge_tx_use_bcopy && (nxgep->niu_type != N2_NIU)) {
430                 bcopy_thresh = (nxge_bcopy_thresh - TX_PKT_HEADER_SIZE);
431         } else {
432                 bcopy_thresh = (TX_BCOPY_SIZE - TX_PKT_HEADER_SIZE);
433         }
434         while (nmp) {
435                 good_packet = B_TRUE;
436                 b_rptr = nmp->b_rptr;
437                 len = MBLKL(nmp);
438                 if (len <= 0) {
439                         nmp = nmp->b_cont;
440                         continue;
441                 }
442                 nmblks++;

444                 NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start(1): nmblks %d "
445                     "len %d pkt_len %d pack_len %d",
446                     nmblks, len, pkt_len, pack_len));
447                 /*
448                  * Hardware limits the transfer length to 4K for NIU and
449                  * 4076 (TX_MAX_TRANSFER_LENGTH) for Neptune. But we just
450                  * use TX_MAX_TRANSFER_LENGTH as the limit for both.
451                  * If len is longer than the limit, then we break nmp into
452                  * two chunks: Make the first chunk equal to the limit and
453                  * the second chunk for the remaining data. If the second
454                  * chunk is still larger than the limit, then it will be
455                  * broken into two in the next pass.
```



```
456                  */
457                 if (len > TX_MAX_TRANSFER_LENGTH - TX_PKT_HEADER_SIZE) {
458                         if ((t_mp = dupb(nmp)) != NULL) {
459                                 nmp->b_wptr = nmp->b_rptr +
460                                     (TX_MAX_TRANSFER_LENGTH
461                                     - TX_PKT_HEADER_SIZE);
462                                 t_mp->b_rptr = nmp->b_wptr;
463                                 t_mp->b_cont = nmp->b_cont;
464                                 nmp->b_cont = t_mp;
465                                 len = MBLKL(nmp);
466                         } else {
467                                 if (is_lso) {
468                                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
469                                             "LSO packet: dupb failed: "
470                                             "channel %d",
471                                             tx_ring_p->tdc));
472                                         mp = nmp;
473                                         goto nxge_start_fail_lso;
474                                 } else {
475                                         good_packet = B_FALSE;
476                                         goto nxge_start_fail2;
477                                 }
478                         }
479                 }
480                 tx_desc.value = 0;
481                 tx_desc_p = &tx_desc_ring_vp[i];
482 #ifdef  NXGE_DEBUG
483                 tx_desc_pp = &tx_desc_ring_pp[i];
484 #endif
485                 tx_msg_p = &tx_msg_ring[i];
486 #if defined(__i386)
487                 npi_desc_handle.regp = (uint32_t)tx_desc_p;
488 #else
489                 npi_desc_handle.regp = (uint64_t)tx_desc_p;
490 #endif
491                 if (!header_set &&
492                     ((!nxge_tx_use_bcopy && (len > TX_BCOPY_SIZE)) ||
493                     (len >= bcopy_thresh))) {
494                         header_set = B_TRUE;
495                         bcopy_thresh += TX_PKT_HEADER_SIZE;
496                         boff = 0;
497                         pack_len = 0;
498                         kaddr = (caddr_t)DMA_COMMON_VPTR(tx_msg_p->buf_dma);
499                         hdrp = (p_tx_pkt_header_t)kaddr;
500                         clen = pkt_len;
501                         dma_handle = tx_msg_p->buf_dma_handle;
502                         dma_ioaddr = DMA_COMMON_IOADDR(tx_msg_p->buf_dma);
503                         (void) ddi_dma_sync(dma_handle,
504                             i * nxge_bcopy_thresh, nxge_bcopy_thresh,
505                             DDI_DMA_SYNC_FORDEV);

507                         tx_msg_p->flags.dma_type = USE_BCOPY;
508                         goto nxge_start_control_header_only;
509                 }

511                 pkt_len += len;
512                 pack_len += len;

514                 NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start(3): "
515                     "desc entry %d "
516                     "DESC IOADDR $%p "
517                     "desc_vp $%p tx_desc_p $%p "
518                     "desc_pp $%p tx_desc_pp $%p "
519                     "len %d pkt_len %d pack_len %d",
520                     i,
521                     DMA_COMMON_IOADDR(desc_area),
```

```
522                     tx_desc_ring_vp, tx_desc_p,
523                     tx_desc_ring_pp, tx_desc_pp,
524                     len, pkt_len, pack_len));
525
526                 if (len < bcopy_thresh) {
527                         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start(4): "
528                             "USE BCOPY: "));
529                         if (nxge_tx_tiny_pack) {
530                                 uint32_t blst =
531                                     TXDMA_DESC_NEXT_INDEX(i, -1,
532                                     tx_ring_p->tx_wrap_mask);
533                                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
534                                     "==> nxge_start(5): pack"));
535                                 if ((pack_len <= bcopy_thresh) &&
536                                     (last_bidx == blst)) {
537                                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
538                                             "==> nxge_start: pack(6) "
539                                             "(pkt_len %d pack_len %d)",
540                                             pkt_len, pack_len));
541                                         i = blst;
542                                         tx_desc_p = &tx_desc_ring_vp[i];
543 #ifdef  NXGE_DEBUG
544                                         tx_desc_pp = &tx_desc_ring_pp[i];
545 #endif
546                                         tx_msg_p = &tx_msg_ring[i];
547                                         boff = pack_len - len;
548                                         ngathers--;
549                                 } else if (pack_len > bcopy_thresh &&
550                                     header_set) {
551                                         pack_len = len;
552                                         boff = 0;
553                                         bcopy_thresh = nxge_bcopy_thresh;
554                                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
555                                             "==> nxge_start(7): > max NEW "
556                                             "bcopy thresh %d "
557                                             "pkt_len %d pack_len %d(next)",
558                                             bcopy_thresh,
559                                             pkt_len, pack_len));
560                                 }
561                                 last_bidx = i;
562                         }
563                         kaddr = (caddr_t)DMA_COMMON_VPTR(tx_msg_p->buf_dma);
564                         if ((boff == TX_PKT_HEADER_SIZE) && (nmblks == 1)) {
565                                 hdrp = (p_tx_pkt_header_t)kaddr;
566                                 header_set = B_TRUE;
567                                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
568                                     "==> nxge_start(7_x2): "
569                                     "pkt_len %d pack_len %d (new hdrp $%p)",
570                                     pkt_len, pack_len, hdrp));
571                         }
572                         tx_msg_p->flags.dma_type = USE_BCOPY;
573                         kaddr += boff;
574                         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start(8): "
575                             "USE BCOPY: before bcopy "
576                             "DESC IOADDR $%p entry %d "
577                             "bcopy packets %d "
578                             "bcopy kaddr $%p "
579                             "bcopy ioaddr (SAD) $%p "
580                             "bcopy clen %d "
581                             "bcopy boff %d",
582                             DMA_COMMON_IOADDR(desc_area), i,
583                             tdc_stats->tx_hdr_pkts,
584                             kaddr,
585                             dma_ioaddr,
586                             clen,
587                             boff));
```



```
588                         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: "
589                             "1USE BCOPY: "));
590                         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: "
591                             "2USE BCOPY: "));
592                         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: "
593                             "last USE BCOPY: copy from b_rptr $%p "
594                             "to KADDR $%p (len %d offset %d",
595                             b_rptr, kaddr, len, boff));
596
597                         bcopy(b_rptr, kaddr, len);
598
599 #ifdef  NXGE_DEBUG
600                         dump_len = (len > 128) ? 128: len;
601                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
602                             "==> nxge_start: dump packets "
603                             "(After BCOPY len %d)"
604                             "(b_rptr $%p): %s", len, nmp->b_rptr,
605                             nxge_dump_packet((char *)nmp->b_rptr,
606                             dump_len)));
607 #endif
608
609                         dma_handle = tx_msg_p->buf_dma_handle;
610                         dma_ioaddr = DMA_COMMON_IOADDR(tx_msg_p->buf_dma);
611                         (void) ddi_dma_sync(dma_handle,
612                             i * nxge_bcopy_thresh, nxge_bcopy_thresh,
613                             DDI_DMA_SYNC_FORDEV);
614                         clen = len + boff;
615                         tdc_stats->tx_hdr_pkts++;
616                         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start(9): "
617                             "USE BCOPY: "
618                             "DESC IOADDR $%p entry %d "
619                             "bcopy packets %d "
620                             "bcopy kaddr $%p "
621                             "bcopy ioaddr (SAD) $%p "
622                             "bcopy clen %d "
623                             "bcopy boff %d",
624                             DMA_COMMON_IOADDR(desc_area),
625                             i,
626                             tdc_stats->tx_hdr_pkts,
627                             kaddr,
628                             dma_ioaddr,
629                             clen,
630                             boff));
631                 } else {
632                         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start(12): "
633                             "USE DVMA: len %d", len));
634                         tx_msg_p->flags.dma_type = USE_DMA;
635                         dma_flags = DDI_DMA_WRITE;
636                         if (len < nxge_dma_stream_thresh) {
637                                 dma_flags |= DDI_DMA_CONSISTENT;
638                         } else {
639                                 dma_flags |= DDI_DMA_STREAMING;
640                         }
641
642                         dma_handle = tx_msg_p->dma_handle;
643                         dma_status = ddi_dma_addr_bind_handle(dma_handle, NULL,
644                             (caddr_t)b_rptr, len, dma_flags,
645                             DDI_DMA_DONTWAIT, NULL,
646                             &dma_cookie, &ncookies);
647                         if (dma_status == DDI_DMA_MAPPED) {
648                                 dma_ioaddr = dma_cookie.dmac_laddress;
649                                 len = (int)dma_cookie.dmac_size;
650                                 clen = (uint32_t)dma_cookie.dmac_size;
651                                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
652                                     "==> nxge_start(12_1): "
653                                     "USE DVMA: len %d clen %d "
```

```
654                                     "ngathers %d",
655                                     len, clen,
656                                     ngathers));
657 #if defined(__i386)
658                                 npi_desc_handle.regp = (uint32_t)tx_desc_p;
659 #else
660                                 npi_desc_handle.regp = (uint64_t)tx_desc_p;
661 #endif
662                                 while (ncookies > 1) {
663                                         ngathers++;
664                                         /*
665                                          * this is the fix for multiple
666                                          * cookies, which are basically
667                                          * a descriptor entry, we don't set
668                                          * SOP bit as well as related fields
669                                          */
670
671                                         (void) npi_txdma_desc_gather_set(
672                                             npi_desc_handle,
673                                             &tx_desc,
674                                             (ngathers -1),
675                                             mark_mode,
676                                             ngathers,
677                                             dma_ioaddr,
678                                             clen);
679
680                                         tx_msg_p->tx_msg_size = clen;
681                                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
682                                             "==> nxge_start:  DMA "
683                                             "ncookie %d "
684                                             "ngathers %d "
685                                             "dma_ioaddr $%p len %d"
686                                             "desc $%p descp $%p (%d)",
687                                             ncookies,
688                                             ngathers,
689                                             dma_ioaddr, clen,
690                                             *tx_desc_p, tx_desc_p, i));
691
692                                         ddi_dma_nextcookie(dma_handle,
693                                             &dma_cookie);
694                                         dma_ioaddr =
695                                             dma_cookie.dmac_laddress;
696
697                                         len = (int)dma_cookie.dmac_size;
698                                         clen = (uint32_t)dma_cookie.dmac_size;
699                                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
700                                             "==> nxge_start(12_2): "
701                                             "USE DVMA: len %d clen %d ",
702                                             len, clen));
703
704                                         i = TXDMA_DESC_NEXT_INDEX(i, 1,
705                                             tx_ring_p->tx_wrap_mask);
706                                         tx_desc_p = &tx_desc_ring_vp[i];
707
708 #if defined(__i386)
709                                         npi_desc_handle.regp =
710                                             (uint32_t)tx_desc_p;
711 #else
712                                         npi_desc_handle.regp =
713                                             (uint64_t)tx_desc_p;
714 #endif
715                                         tx_msg_p = &tx_msg_ring[i];
716                                         tx_msg_p->flags.dma_type = USE_NONE;
717                                         tx_desc.value = 0;
718
719                                         ncookies--;
```



```
720                                 }
721                                 tdc_stats->tx_ddi_pkts++;
722                                 NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start:"
723                                     "DMA: ddi packets %d",
724                                     tdc_stats->tx_ddi_pkts));
725                         } else {
726                                 NXGE_ERROR_MSG((nxgep, NXGE_ERR_CTL,
727                                     "dma mapping failed for %d "
728                                     "bytes addr $%p flags %x (%d)",
729                                     len, b_rptr, status, status));
730                                 good_packet = B_FALSE;
731                                 tdc_stats->tx_dma_bind_fail++;
732                                 tx_msg_p->flags.dma_type = USE_NONE;
733                                 if (is_lso) {
734                                         mp = nmp;
735                                         goto nxge_start_fail_lso;
736                                 } else {
737                                         status = 1;
738                                         goto nxge_start_fail2;
739                                 }
740                         }
741                 } /* ddi dvma */
742
743                 if (is_lso) {
744                         nmp_lso_save = nmp;
745                 }
746                 nmp = nmp->b_cont;
747 nxge_start_control_header_only:
748 #if defined(__i386)
749                 npi_desc_handle.regp = (uint32_t)tx_desc_p;
750 #else
751                 npi_desc_handle.regp = (uint64_t)tx_desc_p;
752 #endif
753                 ngathers++;
754
755                 if (ngathers == 1) {
756 #ifdef  NXGE_DEBUG
757                         save_desc_p = &sop_tx_desc;
758 #endif
759                         sop_tx_desc_p = &sop_tx_desc;
760                         sop_tx_desc_p->value = 0;
761                         sop_tx_desc_p->bits.hdw.tr_len = clen;
762                         sop_tx_desc_p->bits.hdw.sad = dma_ioaddr >> 32;
763                         sop_tx_desc_p->bits.ldw.sad = dma_ioaddr & 0xffffffff;
764                 } else {
765 #ifdef  NXGE_DEBUG
766                         save_desc_p = &tx_desc;
767 #endif
768                         tmp_desc_p = &tx_desc;
769                         tmp_desc_p->value = 0;
770                         tmp_desc_p->bits.hdw.tr_len = clen;
771                         tmp_desc_p->bits.hdw.sad = dma_ioaddr >> 32;
772                         tmp_desc_p->bits.ldw.sad = dma_ioaddr & 0xffffffff;
773
774                         tx_desc_p->value = tmp_desc_p->value;
775                 }
776
777                 NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start(13): "
778                     "Desc_entry %d ngathers %d "
779                     "desc_vp $%p tx_desc_p $%p "
780                     "len %d clen %d pkt_len %d pack_len %d nmblks %d "
781                     "dma_ioaddr (SAD) $%p mark %d",
782                     i, ngathers,
783                     tx_desc_ring_vp, tx_desc_p,
784                     len, clen, pkt_len, pack_len, nmblks,
785                     dma_ioaddr, mark_mode));
```

```
787 #ifdef NXGE_DEBUG
788                 npi_desc_handle.nxgep = nxgep;
789                 npi_desc_handle.function.function = nxgep->function_num;
790                 npi_desc_handle.function.instance = nxgep->instance;
791                 sad = (save_desc_p->value & TX_PKT_DESC_SAD_MASK);
792                 xfer_len = ((save_desc_p->value & TX_PKT_DESC_TR_LEN_MASK) >>
793                     TX_PKT_DESC_TR_LEN_SHIFT);


796                 NXGE_DEBUG_MSG((nxgep, TX_CTL, "\n\t: value 0x%llx\n"
797                     "\t\tsad $%p\ttr_len %d len %d\tnptrs %d\t"
798                     "mark %d sop %d\n",
799                     save_desc_p->value,
800                     sad,
801                     save_desc_p->bits.hdw.tr_len,
802                     xfer_len,
803                     save_desc_p->bits.hdw.num_ptr,
804                     save_desc_p->bits.hdw.mark,
805                     save_desc_p->bits.hdw.sop));

807                 npi_txdma_dump_desc_one(npi_desc_handle, NULL, i);
808 #endif

810                 tx_msg_p->tx_msg_size = clen;
811                 i = TXDMA_DESC_NEXT_INDEX(i, 1, tx_ring_p->tx_wrap_mask);
812                 if (ngathers > nxge_tx_max_gathers) {
813                         good_packet = B_FALSE;
814                         mac_hcksum_get(mp, &start_offset,
815                             &stuff_offset, &end_offset, &value,
816                             &cksum_flags);

818                         NXGE_DEBUG_MSG((NULL, TX_CTL,
819                             "==> nxge_start(14): pull msg - "
820                             "len %d pkt_len %d ngathers %d",
821                             len, pkt_len, ngathers));

823                         /*
824                          * Just give up on this packet.
825                          */
826                         if (is_lso) {
827                                 mp = nmp_lso_save;
828                                 goto nxge_start_fail_lso;
829                         }
830                         status = 0;
831                         goto nxge_start_fail2;
832                 }
833         } /* while (nmp) */

835         tx_msg_p->tx_message = mp;
836         tx_desc_p = &tx_desc_ring_vp[sop_index];
837 #if defined(__i386)
838         npi_desc_handle.regp = (uint32_t)tx_desc_p;
839 #else
840         npi_desc_handle.regp = (uint64_t)tx_desc_p;
841 #endif

843         pkthdrp = (p_tx_pkt_hdr_all_t)hdrp;
844         pkthdrp->reserved = 0;
845         hdrp->value = 0;
846         bcopy(&tmp_hdrp, hdrp, sizeof (tx_pkt_header_t));

848         if (pkt_len > NXGE_MTU_DEFAULT_MAX) {
849                 tdc_stats->tx_jumbo_pkts++;
850         }
```



```
852         min_len = (ETHERMIN + TX_PKT_HEADER_SIZE + (npads * 2));
853         if (pkt_len < min_len) {
854                 /* Assume we use bcopy to premapped buffers */
855                 kaddr = (caddr_t)DMA_COMMON_VPTR(tx_msg_p->buf_dma);
856                 NXGE_DEBUG_MSG((NULL, TX_CTL,
857                     "==> nxge_start(14-1): < (msg_min + 16)"
858                     "len %d pkt_len %d min_len %d bzero %d ngathers %d",
859                     len, pkt_len, min_len, (min_len - pkt_len), ngathers));
860                 bzero((kaddr + pkt_len), (min_len - pkt_len));
861                 pkt_len = tx_msg_p->tx_msg_size = min_len;

863                 sop_tx_desc_p->bits.hdw.tr_len = min_len;

865                 NXGE_MEM_PIO_WRITE64(npi_desc_handle, sop_tx_desc_p->value);
866                 tx_desc_p->value = sop_tx_desc_p->value;

868                 NXGE_DEBUG_MSG((NULL, TX_CTL,
869                     "==> nxge_start(14-2): < msg_min - "
870                     "len %d pkt_len %d min_len %d ngathers %d",
871                     len, pkt_len, min_len, ngathers));
872         }

874         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: cksum_flags 0x%x ",
875             cksum_flags));
876         {
877                 uint64_t        tmp_len;

879                 /* pkt_len already includes 16 + paddings!! */
880                 /* Update the control header length */
881                 tot_xfer_len = (pkt_len - TX_PKT_HEADER_SIZE);
882                 tmp_len = hdrp->value |
883                     (tot_xfer_len << TX_PKT_HEADER_TOT_XFER_LEN_SHIFT);

885                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
886                     "==> nxge_start(15_x1): setting SOP "
887                     "tot_xfer_len 0x%llx (%d) pkt_len %d tmp_len "
888                     "0x%llx hdrp->value 0x%llx",
889                     tot_xfer_len, tot_xfer_len, pkt_len,
890                     tmp_len, hdrp->value));
891 #if defined(_BIG_ENDIAN)
892                 hdrp->value = ddi_swap64(tmp_len);
893 #else
894                 hdrp->value = tmp_len;
895 #endif
896                 NXGE_DEBUG_MSG((nxgep,
897                     TX_CTL, "==> nxge_start(15_x2): setting SOP "
898                     "after SWAP: tot_xfer_len 0x%llx pkt_len %d "
899                     "tmp_len 0x%llx hdrp->value 0x%llx",
900                     tot_xfer_len, pkt_len,
901                     tmp_len, hdrp->value));
902         }

904         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start(15): setting SOP "
905             "wr_index %d "
906             "tot_xfer_len (%d) pkt_len %d npads %d",
907             sop_index,
908             tot_xfer_len, pkt_len,
909             npads));

911         sop_tx_desc_p->bits.hdw.sop = 1;
912         sop_tx_desc_p->bits.hdw.mark = mark_mode;
913         sop_tx_desc_p->bits.hdw.num_ptr = ngathers;

915         NXGE_MEM_PIO_WRITE64(npi_desc_handle, sop_tx_desc_p->value);

917         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start(16): set SOP done"));
```

```
#ifdef NXGE_DEBUG
        npi_desc_handle.nxgep = nxgep;
        npi_desc_handle.function.function = nxgep->function_num;
        npi_desc_handle.function.instance = nxgep->instance;

        NXGE_DEBUG_MSG((nxgep, TX_CTL, "\n\t: value 0x%llx\n"
            "\t\tsad $%p\ttr_len %d len %d\tnptrs %d\tmark %d sop %d\n",
            save_desc_p->value,
            sad,
            save_desc_p->bits.hdw.tr_len,
            xfer_len,
            save_desc_p->bits.hdw.num_ptr,
            save_desc_p->bits.hdw.mark,
            save_desc_p->bits.hdw.sop));
        (void) npi_txdma_dump_desc_one(npi_desc_handle, NULL, sop_index);

        dump_len = (pkt_len > 128) ? 128: pkt_len;
        NXGE_DEBUG_MSG((nxgep, TX_CTL,
            "==> nxge_start: dump packets(17) (after sop set, len "
            " (len/dump_len/pkt_len/tot_xfer_len) %d/%d/%d/%d):\n"
            "ptr $%p: %s", len, dump_len, pkt_len, tot_xfer_len,
            (char *)hdrp,
            nxge_dump_packet((char *)hdrp, dump_len)));
        NXGE_DEBUG_MSG((nxgep, TX_CTL,
            "==> nxge_start(18): TX desc sync: sop_index %d",
            sop_index));
#endif

        if ((ngathers == 1) || tx_ring_p->wr_index < i) {
                (void) ddi_dma_sync(tx_desc_dma_handle,
                    sop_index * sizeof (tx_desc_t),
                    ngathers * sizeof (tx_desc_t),
                    DDI_DMA_SYNC_FORDEV);

                NXGE_DEBUG_MSG((nxgep, TX_CTL, "nxge_start(19): sync 1 "
                    "cs_off = 0x%02X cs_s_off = 0x%02X "
                    "pkt_len %d ngathers %d sop_index %d\n",
                    stuff_offset, start_offset,
                    pkt_len, ngathers, sop_index));
        } else { /* more than one descriptor and wrap around */
                uint32_t nsdescs = tx_ring_p->tx_ring_size - sop_index;
                (void) ddi_dma_sync(tx_desc_dma_handle,
                    sop_index * sizeof (tx_desc_t),
                    nsdescs * sizeof (tx_desc_t),
                    DDI_DMA_SYNC_FORDEV);
                NXGE_DEBUG_MSG((nxgep, TX_CTL, "nxge_start(20): sync 1 "
                    "cs_off = 0x%02X cs_s_off = 0x%02X "
                    "pkt_len %d ngathers %d sop_index %d\n",
                    stuff_offset, start_offset,
                    pkt_len, ngathers, sop_index));

                (void) ddi_dma_sync(tx_desc_dma_handle,
                    0,
                    (ngathers - nsdescs) * sizeof (tx_desc_t),
                    DDI_DMA_SYNC_FORDEV);
                NXGE_DEBUG_MSG((nxgep, TX_CTL, "nxge_start(21): sync 2 "
                    "cs_off = 0x%02X cs_s_off = 0x%02X "
                    "pkt_len %d ngathers %d sop_index %d\n",
                    stuff_offset, start_offset,
                    pkt_len, ngathers, sop_index));
        }

        tail_index = tx_ring_p->wr_index;
        tail_wrap = tx_ring_p->wr_index_wrap;
```



```
        tx_ring_p->wr_index = i;
        if (tx_ring_p->wr_index <= tail_index) {
                tx_ring_p->wr_index_wrap = ((tail_wrap == B_TRUE) ?
                    B_FALSE : B_TRUE);
        }

        NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: TX kick: "
            "channel %d wr_index %d wrap %d ngathers %d desc_pend %d",
            tx_ring_p->tdc,
            tx_ring_p->wr_index,
            tx_ring_p->wr_index_wrap,
            ngathers,
            tx_ring_p->descs_pending));

        if (is_lso) {
                lso_ngathers += ngathers;
                if (mp_chain != NULL) {
                        mp = mp_chain;
                        mp_chain = mp_chain->b_next;
                        mp->b_next = NULL;
                        if (nxge_lso_kick_cnt == lso_ngathers) {
                                tx_ring_p->descs_pending += lso_ngathers;
                                {
                                        tx_ring_kick_t          kick;

                                        kick.value = 0;
                                        kick.bits.ldw.wrap =
                                            tx_ring_p->wr_index_wrap;
                                        kick.bits.ldw.tail =
                                            (uint16_t)tx_ring_p->wr_index;

                                        /* Kick the Transmit kick register */
                                        TXDMA_REG_WRITE64(
                                            NXGE_DEV_NPI_HANDLE(nxgep),
                                            TX_RING_KICK_REG,
                                            (uint8_t)tx_ring_p->tdc,
                                            kick.value);
                                        tdc_stats->tx_starts++;

                                        NXGE_DEBUG_MSG((nxgep, TX_CTL,
                                            "==> nxge_start: more LSO: "
                                            "LSO_CNT %d",
                                            lso_ngathers));
                                }
                                lso_ngathers = 0;
                                ngathers = 0;
                                cur_index_lso = sop_index = tx_ring_p->wr_index;
                                lso_tail_wrap = tx_ring_p->wr_index_wrap;
                        }
                        NXGE_DEBUG_MSG((nxgep, TX_CTL,
                            "==> nxge_start: lso again: "
                            "lso_gathers %d ngathers %d cur_index_lso %d "
                            "wr_index %d sop_index %d",
                            lso_ngathers, ngathers, cur_index_lso,
                            tx_ring_p->wr_index, sop_index));

                        NXGE_DEBUG_MSG((nxgep, TX_CTL,
                            "==> nxge_start: next : count %d",
                            lso_ngathers));
                        lso_again = B_TRUE;
                        goto start_again;
                }
                ngathers = lso_ngathers;
        }

        NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: TX KICKING: "));
```

```
1051        {
1052                tx_ring_kick_t          kick;

1054                kick.value = 0;
1055                kick.bits.ldw.wrap = tx_ring_p->wr_index_wrap;
1056                kick.bits.ldw.tail = (uint16_t)tx_ring_p->wr_index;

1058                /* Kick start the Transmit kick register */
1059                TXDMA_REG_WRITE64(NXGE_DEV_NPI_HANDLE(nxgep),
1060                    TX_RING_KICK_REG,
1061                    (uint8_t)tx_ring_p->tdc,
1062                    kick.value);
1063        }

1065        tx_ring_p->descs_pending += ngathers;
1066        tdc_stats->tx_starts++;

1068        if (isLDOMservice(nxgep)) {
1069                tx_ring_p->tx_ring_busy = B_FALSE;
1070                if (tx_ring_p->tx_ring_offline) {
1071                        (void) atomic_swap_32(&tx_ring_p->tx_ring_offline,
1072                            NXGE_TX_RING_OFFLINED);
1073                }
1074        }

1076        MUTEX_EXIT(&tx_ring_p->lock);

1078        NXGE_DEBUG_MSG((nxgep, TX_CTL, "<== nxge_start"));
1079        return (status);

1081 nxge_start_fail_lso:
1082        status = 0;
1083        good_packet = B_FALSE;
1084        if (mp != NULL)
1085                freemsg(mp);
1086        if (mp_chain != NULL)
1087                freemsgchain(mp_chain);

1089        if (!lso_again && !ngathers) {
1090                if (isLDOMservice(nxgep)) {
1091                        tx_ring_p->tx_ring_busy = B_FALSE;
1092                        if (tx_ring_p->tx_ring_offline) {
1093                                (void) atomic_swap_32(
1094                                    &tx_ring_p->tx_ring_offline,
1095                                    NXGE_TX_RING_OFFLINED);
1096                        }
1097                }

1099                MUTEX_EXIT(&tx_ring_p->lock);
1100                NXGE_DEBUG_MSG((nxgep, TX_CTL,
1101                    "==> nxge_start: lso exit (nothing changed)"));
1102                goto nxge_start_fail1;
1103        }

1105        NXGE_DEBUG_MSG((nxgep, TX_CTL,
1106            "==> nxge_start (channel %d): before lso "
1107            "lso_gathers %d ngathers %d cur_index_lso %d "
1108            "wr_index %d sop_index %d lso_again %d",
1109            tx_ring_p->tdc,
1110            lso_ngathers, ngathers, cur_index_lso,
1111            tx_ring_p->wr_index, sop_index, lso_again));

1113        if (lso_again) {
1114                lso_ngathers += ngathers;
1115                ngathers = lso_ngathers;
```



```
1116                sop_index = cur_index_lso;
1117                tx_ring_p->wr_index = sop_index;
1118                tx_ring_p->wr_index_wrap = lso_tail_wrap;
1119        }

1121        NXGE_DEBUG_MSG((nxgep, TX_CTL,
1122            "==> nxge_start (channel %d): after lso "
1123            "lso_gathers %d ngathers %d cur_index_lso %d "
1124            "wr_index %d sop_index %d lso_again %d",
1125            tx_ring_p->tdc,
1126            lso_ngathers, ngathers, cur_index_lso,
1127            tx_ring_p->wr_index, sop_index, lso_again));

1129 nxge_start_fail2:
1130        if (good_packet == B_FALSE) {
1131                cur_index = sop_index;
1132                NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_start: clean up"));
1133                for (i = 0; i < ngathers; i++) {
1134                        tx_desc_p = &tx_desc_ring_vp[cur_index];
1135 #if defined(__i386)
1136                        npi_handle.regp = (uint32_t)tx_desc_p;
1137 #else
1138                        npi_handle.regp = (uint64_t)tx_desc_p;
1139 #endif
1140                        tx_msg_p = &tx_msg_ring[cur_index];
1141                        (void) npi_txdma_desc_set_zero(npi_handle, 1);
1142                        if (tx_msg_p->flags.dma_type == USE_DVMA) {
1143                                NXGE_DEBUG_MSG((nxgep, TX_CTL,
1144                                    "tx_desc_p = %X index = %d",
1145                                    tx_desc_p, tx_ring_p->rd_index));
1146                                (void) dvma_unload(tx_msg_p->dvma_handle,
1147                                    0, -1);
1148                                tx_msg_p->dvma_handle = NULL;
1149                                if (tx_ring_p->dvma_wr_index ==
1150                                    tx_ring_p->dvma_wrap_mask)
1151                                        tx_ring_p->dvma_wr_index = 0;
1152                                else
1153                                        tx_ring_p->dvma_wr_index++;
1154                                tx_ring_p->dvma_pending--;
1155                        } else if (tx_msg_p->flags.dma_type == USE_DMA) {
1156                                if (ddi_dma_unbind_handle(
1157                                    tx_msg_p->dma_handle)) {
1158                                        cmn_err(CE_WARN, "!nxge_start: "
1159                                            "ddi_dma_unbind_handle failed");
1160                                }
1161                        }
1162                        tx_msg_p->flags.dma_type = USE_NONE;
1163                        cur_index = TXDMA_DESC_NEXT_INDEX(cur_index, 1,
1164                            tx_ring_p->tx_wrap_mask);

1166                }
1167        }

1169        if (isLDOMservice(nxgep)) {
1170                tx_ring_p->tx_ring_busy = B_FALSE;
1171                if (tx_ring_p->tx_ring_offline) {
1172                        (void) atomic_swap_32(&tx_ring_p->tx_ring_offline,
1173                            NXGE_TX_RING_OFFLINED);
1174                }
1175        }

1177        MUTEX_EXIT(&tx_ring_p->lock);

1179 nxge_start_fail1:
1180        /* Add FMA to check the access handle nxge_hregh */
```

```
1182         NXGE_DEBUG_MSG((nxgep, TX_CTL, "<== nxge_start"));
1183         return (status);
1184 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   95472 Mon Jul 28 07:43:48 2014
new/usr/src/uts/common/io/nxge/nxge_txdma.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 900 boolean_t
 901 nxge_txdma_reclaim(p_nxge_t nxgep, p_tx_ring_t tx_ring_p, int nmblks)
 902 {
 903         boolean_t               status = B_TRUE;
 904         p_nxge_dma_common_t     tx_desc_dma_p;
 905         nxge_dma_common_t       desc_area;
 906         p_tx_desc_t             tx_desc_ring_vp;
 907         p_tx_desc_t             tx_desc_p;
 908         p_tx_desc_t             tx_desc_pp;
 909         tx_desc_t               r_tx_desc;
 910         p_tx_msg_t              tx_msg_ring;
 911         p_tx_msg_t              tx_msg_p;
 912         npi_handle_t            handle;
 913         tx_ring_hdl_t           tx_head;
 914         uint32_t                pkt_len;
 915         uint_t                  tx_rd_index;
 916         uint16_t                head_index, tail_index;
 917         uint8_t                 tdc;
 918         boolean_t               head_wrap, tail_wrap;
 919         p_nxge_tx_ring_stats_t tdc_stats;
 920         int                     rc;

 922         NXGE_DEBUG_MSG((nxgep, TX_CTL, "==> nxge_txdma_reclaim"));

 924         status = ((tx_ring_p->descs_pending < nxge_reclaim_pending) &&
 925             (nmblks != 0));
 926         NXGE_DEBUG_MSG((nxgep, TX_CTL,
 927             "==> nxge_txdma_reclaim: pending %d  reclaim %d nmblks %d",
 928             tx_ring_p->descs_pending, nxge_reclaim_pending,
 929             nmblks));
 930         if (!status) {
 931                 tx_desc_dma_p = &tx_ring_p->tdc_desc;
 932                 desc_area = tx_ring_p->tdc_desc;
 933                 handle = NXGE_DEV_NPI_HANDLE(nxgep);
 934                 tx_desc_ring_vp = tx_desc_dma_p->kaddrp;
 935                 tx_desc_ring_vp =
 936                     (p_tx_desc_t)DMA_COMMON_VPTR(desc_area);
 937                 tx_rd_index = tx_ring_p->rd_index;
 938                 tx_desc_p = &tx_desc_ring_vp[tx_rd_index];
 939                 tx_msg_ring = tx_ring_p->tx_msg_ring;
 940                 tx_msg_p = &tx_msg_ring[tx_rd_index];
 941                 tdc = tx_ring_p->tdc;
 942                 tdc_stats = tx_ring_p->tdc_stats;
 943                 if (tx_ring_p->descs_pending > tdc_stats->tx_max_pend) {
 944                         tdc_stats->tx_max_pend = tx_ring_p->descs_pending;
 945                 }

 947                 tail_index = tx_ring_p->wr_index;
 948                 tail_wrap = tx_ring_p->wr_index_wrap;

 950                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
 951                     "==> nxge_txdma_reclaim: tdc %d tx_rd_index %d "
 952                     "tail_index %d tail_wrap %d "
 953                     "tx_desc_p $%p ($%p) ",
 954                     tdc, tx_rd_index, tail_index, tail_wrap,
 955                     tx_desc_p, (*(uint64_t *)tx_desc_p)));
 956                 /*
 957                  * Read the hardware maintained transmit head
 958                  * and wrap around bit.
```



```
 959                  */
 960                 TXDMA_REG_READ64(handle, TX_RING_HDL_REG, tdc, &tx_head.value);
 961                 head_index =  tx_head.bits.ldw.head;
 962                 head_wrap = tx_head.bits.ldw.wrap;
 963                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
 964                     "==> nxge_txdma_reclaim: "
 965                     "tx_rd_index %d tail %d tail_wrap %d "
 966                     "head %d wrap %d",
 967                     tx_rd_index, tail_index, tail_wrap,
 968                     head_index, head_wrap));

 970                 if (head_index == tail_index) {
 971                         if (TXDMA_RING_EMPTY(head_index, head_wrap,
 972                             tail_index, tail_wrap) &&
 973                             (head_index == tx_rd_index)) {
 974                                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
 975                                     "==> nxge_txdma_reclaim: EMPTY"));
 976                                 return (B_TRUE);
 977                         }

 979                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
 980                             "==> nxge_txdma_reclaim: Checking "
 981                             "if ring full"));
 982                         if (TXDMA_RING_FULL(head_index, head_wrap, tail_index,
 983                             tail_wrap)) {
 984                                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
 985                                     "==> nxge_txdma_reclaim: full"));
 986                                 return (B_FALSE);
 987                         }
 988                 }

 990                 NXGE_DEBUG_MSG((nxgep, TX_CTL,
 991                     "==> nxge_txdma_reclaim: tx_rd_index and head_index"));

 993                 tx_desc_pp = &r_tx_desc;
 994                 while ((tx_rd_index != head_index) &&
 995                     (tx_ring_p->descs_pending != 0)) {

 997                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
 998                             "==> nxge_txdma_reclaim: Checking if pending"));

1000                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
1001                             "==> nxge_txdma_reclaim: "
1002                             "descs_pending %d ",
1003                             tx_ring_p->descs_pending));

1005                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
1006                             "==> nxge_txdma_reclaim: "
1007                             "(tx_rd_index %d head_index %d "
1008                             "(tx_desc_p $%p)",
1009                             tx_rd_index, head_index,
1010                             tx_desc_p));

1012                         tx_desc_pp->value = tx_desc_p->value;
1013                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
1014                             "==> nxge_txdma_reclaim: "
1015                             "(tx_rd_index %d head_index %d "
1016                             "tx_desc_p $%p (desc value 0x%llx) ",
1017                             tx_rd_index, head_index,
1018                             tx_desc_pp, (*(uint64_t *)tx_desc_pp)));

1020                         NXGE_DEBUG_MSG((nxgep, TX_CTL,
1021                             "==> nxge_txdma_reclaim: dump desc:"));

1023                         pkt_len = tx_desc_pp->bits.hdw.tr_len;
1024                         tdc_stats->obytes += (pkt_len - TX_PKT_HEADER_SIZE);
```

```
1025                        tdc_stats->opackets += tx_desc_pp->bits.hdw.sop;
1026                        NXGE_DEBUG_MSG((nxgep, TX_CTL,
1027                            "==> nxge_txdma_reclaim: pkt_len %d "
1028                            "tdc channel %d opackets %d",
1029                            pkt_len,
1030                            tdc,
1031                            tdc_stats->opackets));

1033                        if (tx_msg_p->flags.dma_type == USE_DVMA) {
1034                                NXGE_DEBUG_MSG((nxgep, TX_CTL,
1035                                    "tx_desc_p = $%p "
1036                                    "tx_desc_pp = $%p "
1037                                    "index = %d",
1038                                    tx_desc_p,
1039                                    tx_desc_pp,
1040                                    tx_ring_p->rd_index));
1041                                (void) dvma_unload(tx_msg_p->dvma_handle,
1042                                    0, -1);
1043                                tx_msg_p->dvma_handle = NULL;
1044                                if (tx_ring_p->dvma_wr_index ==
1045                                    tx_ring_p->dvma_wrap_mask) {
1046                                        tx_ring_p->dvma_wr_index = 0;
1047                                } else {
1048                                        tx_ring_p->dvma_wr_index++;
1049                                }
1050                                tx_ring_p->dvma_pending--;
1051                        } else if (tx_msg_p->flags.dma_type ==
1052                            USE_DMA) {
1053                                NXGE_DEBUG_MSG((nxgep, TX_CTL,
1054                                    "==> nxge_txdma_reclaim: "
1055                                    "USE DMA"));
1056                                if (rc = ddi_dma_unbind_handle
1057                                    (tx_msg_p->dma_handle)) {
1058                                        cmn_err(CE_WARN, "!nxge_reclaim: "
1059                                            "ddi_dma_unbind_handle "
1060                                            "failed. status %d", rc);
1061                                }
1062                        }
1063                        NXGE_DEBUG_MSG((nxgep, TX_CTL,
1064                            "==> nxge_txdma_reclaim: count packets"));
1065                        /*
1066                         * count a chained packet only once.
1067                         */
1068                        if (tx_msg_p->tx_message != NULL) {
1069                                freemsg(tx_msg_p->tx_message);
1070                                tx_msg_p->tx_message = NULL;
1071                        }

1073                        tx_msg_p->flags.dma_type = USE_NONE;
1074                        tx_rd_index = tx_ring_p->rd_index;
1075                        tx_rd_index = (tx_rd_index + 1) &
1076                            tx_ring_p->tx_wrap_mask;
1077                        tx_ring_p->rd_index = tx_rd_index;
1078                        tx_ring_p->descs_pending--;
1079                        tx_desc_p = &tx_desc_ring_vp[tx_rd_index];
1080                        tx_msg_p = &tx_msg_ring[tx_rd_index];
1081                }

1083                status = (nmblks <= ((int)tx_ring_p->tx_ring_size -
1084                    (int)tx_ring_p->descs_pending - TX_FULL_MARK));
1085                if (status) {
1086                        (void) atomic_cas_32((uint32_t *)&tx_ring_p->queueing,
1087                            1, 0);
1086                        (void) cas32((uint32_t *)&tx_ring_p->queueing, 1, 0);
1088                }
1089        } else {
```



```
1090                status = (nmblks <= ((int)tx_ring_p->tx_ring_size -
1091                    (int)tx_ring_p->descs_pending - TX_FULL_MARK));
1092        }

1094        NXGE_DEBUG_MSG((nxgep, TX_CTL,
1095            "<== nxge_txdma_reclaim status = 0x%08x", status));

1097        return (status);
1098 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   55103 Mon Jul 28 07:43:48 2014
new/usr/src/uts/common/io/rge/rge_chip.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 271 /*
 272  * Atomically shift a 32-bit word left, returning
 273  * the value it had *before* the shift was applied
 274  */
 275 static uint32_t rge_atomic_shl32(uint32_t *sp, uint_t count);
 276 #pragma inline(rge_mii_put16)

 278 static uint32_t
 279 rge_atomic_shl32(uint32_t *sp, uint_t count)
 280 {
 281         uint32_t oldval;
 282         uint32_t newval;

 284         /* ATOMICALLY */
 285         do {
 286                 oldval = *sp;
 287                 newval = oldval << count;
 288         } while (atomic_cas_32(sp, oldval, newval) != oldval);
 288         } while (cas32(sp, oldval, newval) != oldval);

 290         return (oldval);
 291 }
______unchanged_portion_omitted_
```

```
**********************************************************
   17781 Mon Jul 28 07:43:49 2014
new/usr/src/uts/common/io/rge/rge_rxtx.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #include "rge.h"

  28 #define U32TOPTR(x)     ((void *)(uintptr_t)(uint32_t)(x))
  29 #define PTRTOU32(x)     ((uint32_t)(uintptr_t)(void *)(x))

  31 /*
  32  * ========== RX side routines ==========
  33  */

  35 #define RGE_DBG         RGE_DBG_RECV    /* debug flag for this code     */

  37 static uint32_t rge_atomic_reserve(uint32_t *count_p, uint32_t n);
  38 #pragma inline(rge_atomic_reserve)

  40 static uint32_t
  41 rge_atomic_reserve(uint32_t *count_p, uint32_t n)
  42 {
  43         uint32_t oldval;
  44         uint32_t newval;

  46         /* ATOMICALLY */
  47         do {
  48                 oldval = *count_p;
  49                 newval = oldval - n;
  50                 if (oldval <= n)
  51                         return (0);             /* no resources left    */
  52         } while (atomic_cas_32(count_p, oldval, newval) != oldval);
  52         } while (cas32(count_p, oldval, newval) != oldval);

  54         return (newval);
  55 }

  57 /*
  58  * Atomically increment a counter
  59  */
  60 static void rge_atomic_renounce(uint32_t *count_p, uint32_t n);
```



```
  61 #pragma inline(rge_atomic_renounce)

  63 static void
  64 rge_atomic_renounce(uint32_t *count_p, uint32_t n)
  65 {
  66         uint32_t oldval;
  67         uint32_t newval;

  69         /* ATOMICALLY */
  70         do {
  71                 oldval = *count_p;
  72                 newval = oldval + n;
  73         } while (atomic_cas_32(count_p, oldval, newval) != oldval);
  73         } while (cas32(count_p, oldval, newval) != oldval);
  74 }
______unchanged_portion_omitted_
```

****
7954 Mon Jul 28 07:43:49 2014
new/usr/src/uts/common/io/smbios.c
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
 63 static dev_info_t *smb_devi;
 64 static smb_clone_t *smb_clones;
 65 static int smb_nclones;

 67 /*ARGSUSED*/
 68 static int
 69 smb_open(dev_t *dp, int flag, int otyp, cred_t *cred)
 70 {
 71         minor_t c;

 73         if (ksmbios == NULL)
 74                 return (ENXIO);

 76         /*
 77          * Locate and reserve a clone structure.  We skip clone 0 as that is
 78          * the real minor number, and we assign a new minor to each clone.
 79          */
 80         for (c = 1; c < smb_nclones; c++) {
 81                 if (atomic_cas_ptr(&smb_clones[c].c_hdl, NULL, ksmbios) == NULL)
 81                 if (casptr(&smb_clones[c].c_hdl, NULL, ksmbios) == NULL)
 82                         break;
 83         }

 85         if (c >= smb_nclones)
 86                 return (EAGAIN);

 88         smb_clones[c].c_eplen = P2ROUNDUP(sizeof (smbios_entry_t), 16);
 89         smb_clones[c].c_stlen = smbios_buflen(smb_clones[c].c_hdl);

 91         *dp = makedevice(getemajor(*dp), c);

 93         (void) ddi_prop_update_int(*dp, smb_devi, "size",
 94             smb_clones[c].c_eplen + smb_clones[c].c_stlen);

 96         return (0);
 97 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  106139 Mon Jul 28 07:43:49 2014
new/usr/src/uts/common/io/stream.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 769 mblk_t *
 770 dupb(mblk_t *mp)
 771 {
 772         dblk_t *dbp = mp->b_datap;
 773         mblk_t *new_mp;
 774         uint32_t oldrtfu, newrtfu;

 776         if ((new_mp = kmem_cache_alloc(mblk_cache, KM_NOSLEEP)) == NULL)
 777                 goto out;

 779         new_mp->b_next = new_mp->b_prev = new_mp->b_cont = NULL;
 780         new_mp->b_rptr = mp->b_rptr;
 781         new_mp->b_wptr = mp->b_wptr;
 782         new_mp->b_datap = dbp;
 783         new_mp->b_queue = NULL;
 784         MBLK_BAND_FLAG_WORD(new_mp) = MBLK_BAND_FLAG_WORD(mp);

 786         STR_FTEVENT_MBLK(mp, caller(), FTEV_DUPB, dbp->db_ref);

 788         dbp->db_free = dblk_decref;
 789         do {
 790                 ASSERT(dbp->db_ref > 0);
 791                 oldrtfu = DBLK_RTFU_WORD(dbp);
 792                 newrtfu = oldrtfu + (1 << DBLK_RTFU_SHIFT(db_ref));
 793                 /*
 794                  * If db_ref is maxed out we can't dup this message anymore.
 795                  */
 796                 if ((oldrtfu & DBLK_RTFU_REF_MASK) == DBLK_RTFU_REF_MASK) {
 797                         kmem_cache_free(mblk_cache, new_mp);
 798                         new_mp = NULL;
 799                         goto out;
 800                 }
 801         } while (atomic_cas_32(&DBLK_RTFU_WORD(dbp), oldrtfu, newrtfu) !=
 802             oldrtfu);
 801         } while (cas32(&DBLK_RTFU_WORD(dbp), oldrtfu, newrtfu) != oldrtfu);

 804 out:
 805         FTRACE_1("dupb(): new_mp=0x%lx", (uintptr_t)new_mp);
 806         return (new_mp);
 807 }
______unchanged_portion_omitted_

4190 /*
4191  * STREAMS Flow Trace - record STREAMS Flow Trace events as an mblk flows
4192  * through a stream.
4193  *
4194  * Data currently record per-event is a timestamp, module/driver name,
4195  * downstream module/driver name, optional callstack, event type and a per
4196  * type datum.  Much of the STREAMS framework is instrumented for automatic
4197  * flow tracing (when enabled).  Events can be defined and used by STREAMS
4198  * modules and drivers.
4199  *
4200  * Global objects:
4201  *
4202  *      str_ftevent() - Add a flow-trace event to a dblk.
4203  *      str_ftfree() - Free flow-trace data
4204  *
4205  * Local objects:
4206  *
```



```
4207  *      fthdr_cache - pointer to the kmem cache for trace header.
4208  *      ftblk_cache - pointer to the kmem cache for trace data blocks.
4209  */

4211 int str_ftnever = 1;    /* Don't do STREAMS flow tracing */
4212 int str_ftstack = 0;    /* Don't record event call stacks */

4214 void
4215 str_ftevent(fthdr_t *hp, void *p, ushort_t evnt, ushort_t data)
4216 {
4217         ftblk_t *bp = hp->tail;
4218         ftblk_t *nbp;
4219         ftevnt_t *ep;
4220         int ix, nix;

4222         ASSERT(hp != NULL);

4224         for (;;) {
4225                 if ((ix = bp->ix) == FTBLK_EVNTS) {
4226                         /*
4227                          * Tail doesn't have room, so need a new tail.
4228                          *
4229                          * To make this MT safe, first, allocate a new
4230                          * ftblk, and initialize it.  To make life a
4231                          * little easier, reserve the first slot (mostly
4232                          * by making ix = 1).  When we are finished with
4233                          * the initialization, CAS this pointer to the
4234                          * tail.  If this succeeds, this is the new
4235                          * "next" block.  Otherwise, another thread
4236                          * got here first, so free the block and start
4237                          * again.
4238                          */
4239                         nbp = kmem_cache_alloc(ftblk_cache, KM_NOSLEEP);
4240                         if (nbp == NULL) {
4241                                 /* no mem, so punt */
4242                                 str_ftnever++;
4243                                 /* free up all flow data? */
4244                                 return;
4245                         }
4246                         nbp->nxt = NULL;
4247                         nbp->ix = 1;
4248                         /*
4249                          * Just in case there is another thread about
4250                          * to get the next index, we need to make sure
4251                          * the value is there for it.
4252                          */
4253                         membar_producer();
4254                         if (atomic_cas_ptr(&hp->tail, bp, nbp) == bp) {
4253                         if (casptr(&hp->tail, bp, nbp) == bp) {
4255                                 /* CAS was successful */
4256                                 bp->nxt = nbp;
4257                                 membar_producer();
4258                                 bp = nbp;
4259                                 ix = 0;
4260                                 goto cas_good;
4261                         } else {
4262                                 kmem_cache_free(ftblk_cache, nbp);
4263                                 bp = hp->tail;
4264                                 continue;
4265                         }
4266                 }
4267                 nix = ix + 1;
4268                 if (atomic_cas_32((uint32_t *)&bp->ix, ix, nix) == ix) {
4267                 if (cas32((uint32_t *)&bp->ix, ix, nix) == ix) {
4269                 cas_good:
4270                         if (curthread != hp->thread) {
```

```
4271                                    hp->thread = curthread;
4272                                    evnt |= FTEV_CS;
4273                            }
4274                            if (CPU->cpu_seqid != hp->cpu_seqid) {
4275                                    hp->cpu_seqid = CPU->cpu_seqid;
4276                                    evnt |= FTEV_PS;
4277                            }
4278                            ep = &bp->ev[ix];
4279                            break;
4280                    }
4281            }

4283            if (evnt & FTEV_QMASK) {
4284                    queue_t *qp = p;

4286                    if (!(qp->q_flag & QREADR))
4287                            evnt |= FTEV_ISWR;

4289                    ep->mid = Q2NAME(qp);

4291                    /*
4292                     * We only record the next queue name for FTEV_PUTNEXT since
4293                     * that's the only time we *really* need it, and the putnext()
4294                     * code ensures that qp->q_next won't vanish.  (We could use
4295                     * claimstr()/releasestr() but at a performance cost.)
4296                     */
4297                    if ((evnt & FTEV_MASK) == FTEV_PUTNEXT && qp->q_next != NULL)
4298                            ep->midnext = Q2NAME(qp->q_next);
4299                    else
4300                            ep->midnext = NULL;
4301            } else {
4302                    ep->mid = p;
4303                    ep->midnext = NULL;
4304            }

4306            if (ep->stk != NULL)
4307                    ep->stk->fs_depth = getpcstack(ep->stk->fs_stk, FTSTK_DEPTH);

4309            ep->ts = gethrtime();
4310            ep->evnt = evnt;
4311            ep->data = data;
4312            hp->hash = (hp->hash << 9) + hp->hash;
4313            hp->hash += (evnt << 16) | data;
4314            hp->hash += (uintptr_t)ep->mid;
4315 }
```

_____***unchanged_portion_omitted_***

****
11155 Mon Jul 28 07:43:49 2014
new/usr/src/uts/common/io/xge/drv/xge_osdep.h
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
 166 #define xge_os_println                  xge_os_printf

 168 /* -------------------- synchronization primitives ------------------------- */

 170 #define xge_os_spin_lock_init(lockp, ctxh) \
 171         mutex_init(lockp, NULL, MUTEX_DRIVER, NULL)
 172 #define xge_os_spin_lock_init_irq(lockp, irqh) \
 173         mutex_init(lockp, NULL, MUTEX_DRIVER, DDI_INTR_PRI(irqh))
 174 #define xge_os_spin_lock_destroy(lockp, cthx) \
 175         (cthx = cthx, mutex_destroy(lockp))
 176 #define xge_os_spin_lock_destroy_irq(lockp, cthx) \
 177         (cthx = cthx, mutex_destroy(lockp))
 178 #define xge_os_spin_lock(lockp)                 mutex_enter(lockp)
 179 #define xge_os_spin_unlock(lockp)               mutex_exit(lockp)
 180 #define xge_os_spin_lock_irq(lockp, flags) (flags = flags, mutex_enter(lockp))
 181 #define xge_os_spin_unlock_irq(lockp, flags)    mutex_exit(lockp)

 183 /* x86 arch will never re-order writes, Sparc can */
 184 #define xge_os_wmb()                            membar_producer()

 186 #define xge_os_udelay(us)                       drv_usecwait(us)
 187 #define xge_os_mdelay(ms)                       drv_usecwait(ms * 1000)

 189 #define xge_os_cmpxchg(targetp, cmp, newval)            \
 190         sizeof (*(targetp)) == 4 ?                      \
 191         atomic_cas_32((uint32_t *)targetp, cmp, newval) :       \
 192         atomic_cas_64((uint64_t *)targetp, cmp, newval)
 191         cas32((uint32_t *)targetp, cmp, newval) :       \
 192         cas64((uint64_t *)targetp, cmp, newval)

 194 /* ------------------------- misc primitives ------------------------------ */

 196 #define xge_os_unlikely(x)              (x)
 197 #define xge_os_prefetch(a)              (a = a)
 198 #define xge_os_prefetchw
 199 #ifdef __GNUC__
 200 #define xge_os_bug(fmt...)              cmn_err(CE_PANIC, fmt)
 201 #else
 202 static inline void xge_os_bug(char *fmt, ...) {
 203         va_list ap;

 205         va_start(ap, fmt);
 206         vcmn_err(CE_PANIC, fmt, ap);
 207         va_end(ap);
 208 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   74613 Mon Jul 28 07:43:49 2014
new/usr/src/uts/common/os/clock.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

1638 void
1639 profil_tick(uintptr_t upc)
1640 {
1641         int ticks;
1642         proc_t *p = ttoproc(curthread);
1643         klwp_t *lwp = ttolwp(curthread);
1644         struct prof *pr = &p->p_prof;

1646         do {
1647                 ticks = lwp->lwp_oweupc;
1648         } while (atomic_cas_32(&lwp->lwp_oweupc, ticks, 0) != ticks);
1648         } while (cas32(&lwp->lwp_oweupc, ticks, 0) != ticks);

1650         mutex_enter(&p->p_pflock);
1651         if (pr->pr_scale >= 2 && upc >= pr->pr_off) {
1652                 /*
1653                  * Old-style profiling
1654                  */
1655                 uint16_t *slot = pr->pr_base;
1656                 uint16_t old, new;
1657                 if (pr->pr_scale != 2) {
1658                         uintptr_t delta = upc - pr->pr_off;
1659                         uintptr_t byteoff = ((delta >> 16) * pr->pr_scale) +
1660                             (((delta & 0xffff) * pr->pr_scale) >> 16);
1661                         if (byteoff >= (uintptr_t)pr->pr_size) {
1662                                 mutex_exit(&p->p_pflock);
1663                                 return;
1664                         }
1665                         slot += byteoff / sizeof (uint16_t);
1666                 }
1667                 if (fuword16(slot, &old) < 0 ||
1668                     (new = old + ticks) > SHRT_MAX ||
1669                     suword16(slot, new) < 0) {
1670                         pr->pr_scale = 0;
1671                 }
1672         } else if (pr->pr_scale == 1) {
1673                 /*
1674                  * PC Sampling
1675                  */
1676                 model_t model = lwp_getdatamodel(lwp);
1677                 int result;
1678 #ifdef __lint
1679                 model = model;
1680 #endif
1681                 while (ticks-- > 0) {
1682                         if (pr->pr_samples == pr->pr_size) {
1683                                 /* buffer full, turn off sampling */
1684                                 pr->pr_scale = 0;
1685                                 break;
1686                         }
1687                         switch (SIZEOF_PTR(model)) {
1688                         case sizeof (uint32_t):
1689                                 result = suword32(pr->pr_base, (uint32_t)upc);
1690                                 break;
1691 #ifdef _LP64
1692                         case sizeof (uint64_t):
1693                                 result = suword64(pr->pr_base, (uint64_t)upc);
1694                                 break;
1695 #endif
```



```
1696                         default:
1697                                 cmn_err(CE_WARN, "profil_tick: unexpected "
1698                                     "data model");
1699                                 result = -1;
1700                                 break;
1701                         }
1702                         if (result != 0) {
1703                                 pr->pr_scale = 0;
1704                                 break;
1705                         }
1706                         pr->pr_base = (caddr_t)pr->pr_base + SIZEOF_PTR(model);
1707                         pr->pr_samples++;
1708                 }
1709         }
1710         mutex_exit(&p->p_pflock);
1711 }
______unchanged_portion_omitted_
```

```
**********************************************************
    9174 Mon Jul 28 07:43:50 2014
new/usr/src/uts/common/os/clock_highres.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

  87 static void
  88 clock_highres_fire(void *arg)
  89 {
  90         itimer_t *it = (itimer_t *)arg;
  91         hrtime_t *addr = &it->it_hrtime;
  92         hrtime_t old = *addr, new = gethrtime();

  94         do {
  95                 old = *addr;
  96         } while (atomic_cas_64((uint64_t *)addr, old, new) != old);
  96         } while (cas64((uint64_t *)addr, old, new) != old);

  98         timer_fire(it);
  99 }
______unchanged_portion_omitted_

 225 static int
 226 clock_highres_timer_gettime(itimer_t *it, struct itimerspec *when)
 227 {
 228         /*
 229          * CLOCK_HIGHRES doesn't update it_itime.
 230          */
 231         hrtime_t start = ts2hrt(&it->it_itime.it_value);
 232         hrtime_t interval = ts2hrt(&it->it_itime.it_interval);
 233         hrtime_t diff, now = gethrtime();
 234         hrtime_t *addr = &it->it_hrtime;
 235         hrtime_t last;

 237         /*
 238          * We're using atomic_cas_64() here only to assure that we slurp the
 239          * entire timestamp atomically.
 238          * We're using cas64() here only to assure that we slurp the entire
 239          * timestamp atomically.
 240          */
 241         last = atomic_cas_64((uint64_t *)addr, 0, 0);
 241         last = cas64((uint64_t *)addr, 0, 0);

 243         *when = it->it_itime;

 245         if (!timerspecisset(&when->it_value))
 246                 return (0);

 248         if (start > now) {
 249                 /*
 250                  * We haven't gone off yet...
 251                  */
 252                 diff = start - now;
 253         } else {
 254                 if (interval == 0) {
 255                         /*
 256                          * This is a one-shot which should have already
 257                          * fired; set it_value to 0.
 258                          */
 259                         timerspecclear(&when->it_value);
 260                         return (0);
 261                 }

 263                 /*
 264                  * Calculate how far we are into this interval.
```



```
 265                  */
 266                 diff = (now - start) % interval;

 268                 /*
 269                  * Now check to see if we've dealt with the last interval
 270                  * yet.
 271                  */
 272                 if (now - diff > last) {
 273                         /*
 274                          * The last interval hasn't fired; set it_value to 0.
 275                          */
 276                         timerspecclear(&when->it_value);
 277                         return (0);
 278                 }

 280                 /*
 281                  * The last interval _has_ fired; we can return the amount
 282                  * of time left in this interval.
 283                  */
 284                 diff = interval - diff;
 285         }

 287         hrt2ts(diff, &when->it_value);

 289         return (0);
 290 }
______unchanged_portion_omitted_
```

```
**********************************************************
  117534 Mon Jul 28 07:43:50 2014
new/usr/src/uts/common/os/cyclic.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 651 static void
 652 cyclic_coverage(char *why, int level, uint64_t arg0, uint64_t arg1)
 653 {
 654         uint_t ndx, orig;

 656         for (ndx = orig = cyclic_coverage_hash(why) % CY_NCOVERAGE; ; ) {
 657                 if (cyc_coverage[ndx].cyv_why == why)
 658                         break;

 660                 if (cyc_coverage[ndx].cyv_why != NULL ||
 661                     atomic_cas_ptr(&cyc_coverage[ndx].cyv_why, NULL, why) !=
 662                     NULL) {
 661                     casptr(&cyc_coverage[ndx].cyv_why, NULL, why) != NULL) {

 664                         if (++ndx == CY_NCOVERAGE)
 665                                 ndx = 0;

 667                         if (ndx == orig)
 668                                 panic("too many cyclic coverage points");
 669                         continue;
 670                 }

 672                 /*
 673                  * If we're here, we have successfully swung our guy into
 674                  * the position at "ndx".
 675                  */
 676                 break;
 677         }

 679         if (level == CY_PASSIVE_LEVEL)
 680                 cyc_coverage[ndx].cyv_passive_count++;
 681         else
 682                 cyc_coverage[ndx].cyv_count[level]++;

 684         cyc_coverage[ndx].cyv_arg0 = arg0;
 685         cyc_coverage[ndx].cyv_arg1 = arg1;
 686 }
______unchanged_portion_omitted_

1072 /*
1073  *  cyclic_softint(cpu_t *cpu, cyc_level_t level)
1074  *
1075  *  Overview
1076  *
1077  *    cyclic_softint() is the cyclic subsystem's CY_LOCK_LEVEL and CY_LOW_LEVEL
1078  *    soft interrupt handler.  Called by the cyclic backend.
1079  *
1080  *  Arguments and notes
1081  *
1082  *    The first argument to cyclic_softint() is the CPU on which the interrupt
1083  *    is executing; backends must call into cyclic_softint() on the specified
1084  *    CPU.  The second argument is the level of the soft interrupt; it must
1085  *    be one of CY_LOCK_LEVEL or CY_LOW_LEVEL.
1086  *
1087  *    cyclic_softint() will call the handlers for cyclics pending at the
1088  *    specified level.  cyclic_softint() will not return until all pending
1089  *    cyclics at the specified level have been dealt with; intervening
1090  *    CY_HIGH_LEVEL interrupts which enqueue cyclics at the specified level
1091  *    may therefore prolong cyclic_softint().
```



```
1092  *
1093  *    cyclic_softint() never disables interrupts, and, if neither a
1094  *    cyclic_add() nor a cyclic_remove() is pending on the specified CPU, is
1095  *    lock-free.  This assures that in the common case, cyclic_softint()
1096  *    completes without blocking, and never starves cyclic_fire().  If either
1097  *    cyclic_add() or cyclic_remove() is pending, cyclic_softint() may grab
1098  *    a dispatcher lock.
1099  *
1100  *    While cyclic_softint() is designed for bounded latency, it is obviously
1101  *    at the mercy of its cyclic handlers.  Because cyclic handlers may block
1102  *    arbitrarily, callers of cyclic_softint() should not rely upon
1103  *    deterministic completion.
1104  *
1105  *    cyclic_softint() may be called spuriously without ill effect.
1106  *
1107  *  Return value
1108  *
1109  *    None.
1110  *
1111  *  Caller's context
1112  *
1113  *    The caller must be executing in soft interrupt context at either
1114  *    CY_LOCK_LEVEL or CY_LOW_LEVEL.  The level passed to cyclic_softint()
1115  *    must match the level at which it is executing.  On optimal backends,
1116  *    the caller will hold no locks.  In any case, the caller may not hold
1117  *    cpu_lock or any lock acquired by any cyclic handler or held across
1118  *    any of cyclic_add(), cyclic_remove(), cyclic_bind() or cyclic_juggle().
1119  */
1120 void
1121 cyclic_softint(cpu_t *c, cyc_level_t level)
1122 {
1123         cyc_cpu_t *cpu = c->cpu_cyclic;
1124         cyc_softbuf_t *softbuf;
1125         int soft, *buf, consndx, resized = 0, intr_resized = 0;
1126         cyc_pcbuffer_t *pc;
1127         cyclic_t *cyclics = cpu->cyp_cyclics;
1128         int sizemask;

1130         CYC_TRACE(cpu, level, "softint", cyclics, 0);

1132         ASSERT(level < CY_LOW_LEVEL + CY_SOFT_LEVELS);

1134         softbuf = &cpu->cyp_softbuf[level];
1135 top:
1136         soft = softbuf->cys_soft;
1137         ASSERT(soft == 0 || soft == 1);

1139         pc = &softbuf->cys_buf[soft];
1140         buf = pc->cypc_buf;
1141         consndx = pc->cypc_consndx;
1142         sizemask = pc->cypc_sizemask;

1144         CYC_TRACE(cpu, level, "softint-top", cyclics, pc);

1146         while (consndx != pc->cypc_prodndx) {
1147                 uint32_t pend, npend, opend;
1148                 int consmasked = consndx & sizemask;
1149                 cyclic_t *cyclic = &cyclics[buf[consmasked]];
1150                 cyc_func_t handler = cyclic->cy_handler;
1151                 void *arg = cyclic->cy_arg;

1153                 ASSERT(buf[consmasked] < cpu->cyp_size);
1154                 CYC_TRACE(cpu, level, "consuming", consndx, cyclic);

1156                 /*
1157                  * We have found this cyclic in the pcbuffer.  We know that
```

```
1158                  * one of the following is true:
1159                  *
1160                  *  (a) The pend is non-zero.  We need to execute the handler
1161                  *      at least once.
1162                  *
1163                  *  (b) The pend _was_ non-zero, but it's now zero due to a
1164                  *      resize.  We will call the handler once, see that we
1165                  *      are in this case, and read the new cyclics buffer
1166                  *      (and hence the old non-zero pend).
1167                  *
1168                  *  (c) The pend _was_ non-zero, but it's now zero due to a
1169                  *      removal.  We will call the handler once, see that we
1170                  *      are in this case, and call into cyclic_remove_pend()
1171                  *      to call the cyclic rpend times.  We will take into
1172                  *      account that we have already called the handler once.
1173                  *
1174                  * Point is:  it's safe to call the handler without first
1175                  * checking the pend.
1176                  */
1177                 do {
1178                         CYC_TRACE(cpu, level, "handler-in", handler, arg);
1179                         DTRACE_PROBE1(cyclic__start, cyclic_t *, cyclic);

1181                         (*handler)(arg);

1183                         DTRACE_PROBE1(cyclic__end, cyclic_t *, cyclic);
1184                         CYC_TRACE(cpu, level, "handler-out", handler, arg);
1185 reread:
1186                         pend = cyclic->cy_pend;
1187                         npend = pend - 1;

1189                         if (pend == 0) {
1190                                 if (cpu->cyp_state == CYS_REMOVING) {
1191                                         /*
1192                                          * This cyclic has been removed while
1193                                          * it had a non-zero pend count (we
1194                                          * know it was non-zero because we
1195                                          * found this cyclic in the pcbuffer).
1196                                          * There must be a non-zero rpend for
1197                                          * this CPU, and there must be a remove
1198                                          * operation blocking; we'll call into
1199                                          * cyclic_remove_pend() to clean this
1200                                          * up, and break out of the pend loop.
1201                                          */
1202                                         cyclic_remove_pend(cpu, level, cyclic);
1203                                         break;
1204                                 }

1206                                 /*
1207                                  * We must have had a resize interrupt us.
1208                                  */
1209                                 CYC_TRACE(cpu, level, "resize-int", cyclics, 0);
1210                                 ASSERT(cpu->cyp_state == CYS_EXPANDING);
1211                                 ASSERT(cyclics != cpu->cyp_cyclics);
1212                                 ASSERT(resized == 0);
1213                                 ASSERT(intr_resized == 0);
1214                                 intr_resized = 1;
1215                                 cyclics = cpu->cyp_cyclics;
1216                                 cyclic = &cyclics[buf[consmasked]];
1217                                 ASSERT(cyclic->cy_handler == handler);
1218                                 ASSERT(cyclic->cy_arg == arg);
1219                                 goto reread;
1220                         }

1222                         if ((opend =
1223                             atomic_cas_32(&cyclic->cy_pend, pend, npend)) !=
```



```
1224                             pend) {
1222                             cas32(&cyclic->cy_pend, pend, npend)) != pend) {
1225                                 /*
1226                                  * Our atomic_cas_32 can fail for one of several
1224                                  * Our cas32 can fail for one of several
1227                                  * reasons:
1228                                  *
1229                                  *  (a) An intervening high level bumped up the
1230                                  *      pend count on this cyclic.  In this
1231                                  *      case, we will see a higher pend.
1232                                  *
1233                                  *  (b) The cyclics array has been yanked out
1234                                  *      from underneath us by a resize
1235                                  *      operation.  In this case, pend is 0 and
1236                                  *      cyp_state is CYS_EXPANDING.
1237                                  *
1238                                  *  (c) The cyclic has been removed by an
1239                                  *      intervening remove-xcall.  In this case,
1240                                  *      pend will be 0, the cyp_state will be
1241                                  *      CYS_REMOVING, and the cyclic will be
1242                                  *      marked CYF_FREE.
1243                                  *
1244                                  * The assertion below checks that we are
1245                                  * in one of the above situations.  The
1246                                  * action under all three is to return to
1247                                  * the top of the loop.
1248                                  */
1249                                 CYC_TRACE(cpu, level, "cas-fail", opend, pend);
1250                                 ASSERT(opend > pend || (opend == 0 &&
1251                                     ((cyclics != cpu->cyp_cyclics &&
1252                                     cpu->cyp_state == CYS_EXPANDING) ||
1253                                     (cpu->cyp_state == CYS_REMOVING &&
1254                                     (cyclic->cy_flags & CYF_FREE)))));
1255                                 goto reread;
1256                         }

1258                         /*
1259                          * Okay, so we've managed to successfully decrement
1260                          * pend.  If we just decremented the pend to 0, we're
1261                          * done.
1262                          */
1263                 } while (npend > 0);

1265                 pc->cypc_consndx = ++consndx;
1266         }

1268         /*
1269          * If the high level handler is no longer writing to the same
1270          * buffer, then we've had a resize.  We need to switch our soft
1271          * index, and goto top.
1272          */
1273         if (soft != softbuf->cys_hard) {
1274                 /*
1275                  * We can assert that the other buffer has grown by exactly
1276                  * one factor of two.
1277                  */
1278                 CYC_TRACE(cpu, level, "buffer-grow", 0, 0);
1279                 ASSERT(cpu->cyp_state == CYS_EXPANDING);
1280                 ASSERT(softbuf->cys_buf[softbuf->cys_hard].cypc_sizemask ==
1281                     (softbuf->cys_buf[soft].cypc_sizemask << 1) + 1 ||
1282                     softbuf->cys_buf[soft].cypc_sizemask == 0);
1283                 ASSERT(softbuf->cys_hard == (softbuf->cys_soft ^ 1));

1285                 /*
1286                  * If our cached cyclics pointer doesn't match cyp_cyclics,
1287                  * then we took a resize between our last iteration of the
```

```
1288                  * pend loop and the check against softbuf->cys_hard.
1289                  */
1290                 if (cpu->cyp_cyclics != cyclics) {
1291                         CYC_TRACE1(cpu, level, "resize-int-int", consndx);
1292                         cyclics = cpu->cyp_cyclics;
1293                 }

1295                 softbuf->cys_soft = softbuf->cys_hard;

1297                 ASSERT(resized == 0);
1298                 resized = 1;
1299                 goto top;
1300         }

1302         /*
1303          * If we were interrupted by a resize operation, then we must have
1304          * seen the hard index change.
1305          */
1306         ASSERT(!(intr_resized == 1 && resized == 0));

1308         if (resized) {
1309                 uint32_t lev, nlev;

1311                 ASSERT(cpu->cyp_state == CYS_EXPANDING);

1313                 do {
1314                         lev = cpu->cyp_modify_levels;
1315                         nlev = lev + 1;
1316                 } while (atomic_cas_32(&cpu->cyp_modify_levels, lev, nlev) !=
1317                     lev);
1314                 } while (cas32(&cpu->cyp_modify_levels, lev, nlev) != lev);

1319                 /*
1320                  * If we are the last soft level to see the modification,
1321                  * post on cyp_modify_wait.  Otherwise, (if we're not
1322                  * already at low level), post down to the next soft level.
1323                  */
1324                 if (nlev == CY_SOFT_LEVELS) {
1325                         CYC_TRACE0(cpu, level, "resize-kick");
1326                         sema_v(&cpu->cyp_modify_wait);
1327                 } else {
1328                         ASSERT(nlev < CY_SOFT_LEVELS);
1329                         if (level != CY_LOW_LEVEL) {
1330                                 cyc_backend_t *be = cpu->cyp_backend;

1332                                 CYC_TRACE0(cpu, level, "resize-post");
1333                                 be->cyb_softint(be->cyb_arg, level - 1);
1334                         }
1335                 }
1336         }
1337 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
    9590 Mon Jul 28 07:43:50 2014
new/usr/src/uts/common/os/dtrace_subr.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 183 void
 184 dtrace_vtime_enable(void)
 185 {
 186         dtrace_vtime_state_t state, nstate;

 188         do {
 189                 state = dtrace_vtime_active;

 191                 switch (state) {
 192                 case DTRACE_VTIME_INACTIVE:
 193                         nstate = DTRACE_VTIME_ACTIVE;
 194                         break;

 196                 case DTRACE_VTIME_INACTIVE_TNF:
 197                         nstate = DTRACE_VTIME_ACTIVE_TNF;
 198                         break;

 200                 case DTRACE_VTIME_ACTIVE:
 201                 case DTRACE_VTIME_ACTIVE_TNF:
 202                         panic("DTrace virtual time already enabled");
 203                         /*NOTREACHED*/
 204                 }

 206         } while (atomic_cas_32((uint32_t *)&dtrace_vtime_active,
 206         } while (cas32((uint32_t *)&dtrace_vtime_active,
 207             state, nstate) != state);
 208 }

 210 void
 211 dtrace_vtime_disable(void)
 212 {
 213         dtrace_vtime_state_t state, nstate;

 215         do {
 216                 state = dtrace_vtime_active;

 218                 switch (state) {
 219                 case DTRACE_VTIME_ACTIVE:
 220                         nstate = DTRACE_VTIME_INACTIVE;
 221                         break;

 223                 case DTRACE_VTIME_ACTIVE_TNF:
 224                         nstate = DTRACE_VTIME_INACTIVE_TNF;
 225                         break;

 227                 case DTRACE_VTIME_INACTIVE:
 228                 case DTRACE_VTIME_INACTIVE_TNF:
 229                         panic("DTrace virtual time already disabled");
 230                         /*NOTREACHED*/
 231                 }

 233         } while (atomic_cas_32((uint32_t *)&dtrace_vtime_active,
 233         } while (cas32((uint32_t *)&dtrace_vtime_active,
 234             state, nstate) != state);
 235 }

 237 void
 238 dtrace_vtime_enable_tnf(void)
 239 {
```



```
 240         dtrace_vtime_state_t state, nstate;

 242         do {
 243                 state = dtrace_vtime_active;

 245                 switch (state) {
 246                 case DTRACE_VTIME_ACTIVE:
 247                         nstate = DTRACE_VTIME_ACTIVE_TNF;
 248                         break;

 250                 case DTRACE_VTIME_INACTIVE:
 251                         nstate = DTRACE_VTIME_INACTIVE_TNF;
 252                         break;

 254                 case DTRACE_VTIME_ACTIVE_TNF:
 255                 case DTRACE_VTIME_INACTIVE_TNF:
 256                         panic("TNF already active");
 257                         /*NOTREACHED*/
 258                 }

 260         } while (atomic_cas_32((uint32_t *)&dtrace_vtime_active,
 260         } while (cas32((uint32_t *)&dtrace_vtime_active,
 261             state, nstate) != state);
 262 }

 264 void
 265 dtrace_vtime_disable_tnf(void)
 266 {
 267         dtrace_vtime_state_t state, nstate;

 269         do {
 270                 state = dtrace_vtime_active;

 272                 switch (state) {
 273                 case DTRACE_VTIME_ACTIVE_TNF:
 274                         nstate = DTRACE_VTIME_ACTIVE;
 275                         break;

 277                 case DTRACE_VTIME_INACTIVE_TNF:
 278                         nstate = DTRACE_VTIME_INACTIVE;
 279                         break;

 281                 case DTRACE_VTIME_ACTIVE:
 282                 case DTRACE_VTIME_INACTIVE:
 283                         panic("TNF already inactive");
 284                         /*NOTREACHED*/
 285                 }

 287         } while (atomic_cas_32((uint32_t *)&dtrace_vtime_active,
 287         } while (cas32((uint32_t *)&dtrace_vtime_active,
 288             state, nstate) != state);
 289 }
______unchanged_portion_omitted_
```

```
**********************************************************
   36418 Mon Jul 28 07:43:50 2014
new/usr/src/uts/common/os/errorq.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 514 /*
 515  * Dispatch a new error into the queue for later processing.  The specified
 516  * data buffer is copied into a preallocated queue element.  If 'len' is
 517  * smaller than the queue element size, the remainder of the queue element is
 518  * filled with zeroes.  This function may be called from any context subject
 519  * to the Platform Considerations described above.
 520  */
 521 void
 522 errorq_dispatch(errorq_t *eqp, const void *data, size_t len, uint_t flag)
 523 {
 524         errorq_elem_t *eep, *old;

 526         if (eqp == NULL || !(eqp->eq_flags & ERRORQ_ACTIVE)) {
 527                 atomic_add_64(&errorq_lost, 1);
 528                 return; /* drop error if queue is uninitialized or disabled */
 529         }

 531         for (;;) {
 532                 int i, rval;

 534                 if ((i = errorq_availbit(eqp->eq_bitmap, eqp->eq_qlen,
 535                     eqp->eq_rotor)) == -1) {
 536                         atomic_add_64(&eqp->eq_kstat.eqk_dropped.value.ui64, 1);
 537                         return;
 538                 }
 539                 BT_ATOMIC_SET_EXCL(eqp->eq_bitmap, i, rval);
 540                 if (rval == 0) {
 541                         eqp->eq_rotor = i;
 542                         eep = &eqp->eq_elems[i];
 543                         break;
 544                 }
 545         }

 547         ASSERT(len <= eqp->eq_size);
 548         bcopy(data, eep->eqe_data, MIN(eqp->eq_size, len));

 550         if (len < eqp->eq_size)
 551                 bzero((caddr_t)eep->eqe_data + len, eqp->eq_size - len);

 553         for (;;) {
 554                 old = eqp->eq_pend;
 555                 eep->eqe_prev = old;
 556                 membar_producer();

 558                 if (atomic_cas_ptr(&eqp->eq_pend, old, eep) == old)
 558                 if (casptr(&eqp->eq_pend, old, eep) == old)
 559                         break;
 560         }

 562         atomic_add_64(&eqp->eq_kstat.eqk_dispatched.value.ui64, 1);

 564         if (flag == ERRORQ_ASYNC && eqp->eq_id != NULL)
 565                 ddi_trigger_softintr(eqp->eq_id);
 566 }

 568 /*
 569  * Drain the specified error queue by calling eq_func() for each pending error.
 570  * This function must be called at or below LOCK_LEVEL or from panic context.
 571  * In order to synchronize with other attempts to drain the queue, we acquire
```



```
 572  * the adaptive eq_lock, blocking other consumers.  Once this lock is held,
 573  * we must use compare-and-swap to move the pending list to the processing
 574  * list and to return elements to the free pool in order to synchronize
 575  * with producers, who do not acquire any locks and only use atomic set/clear.
 576  *
 577  * An additional constraint on this function is that if the system panics
 578  * while this function is running, the panic code must be able to detect and
 579  * handle all intermediate states and correctly dequeue all errors.  The
 580  * errorq_panic() function below will be used for detecting and handling
 581  * these intermediate states.  The comments in errorq_drain() below explain
 582  * how we make sure each intermediate state is distinct and consistent.
 583  */
 584 void
 585 errorq_drain(errorq_t *eqp)
 586 {
 587         errorq_elem_t *eep, *dep;

 589         ASSERT(eqp != NULL);
 590         mutex_enter(&eqp->eq_lock);

 592         /*
 593          * If there are one or more pending errors, set eq_ptail to point to
 594          * the first element on the pending list and then attempt to compare-
 595          * and-swap NULL to the pending list.  We use membar_producer() to
 596          * make sure that eq_ptail will be visible to errorq_panic() below
 597          * before the pending list is NULLed out.  This section is labeled
 598          * case (1) for errorq_panic, below.  If eq_ptail is not yet set (1A)
 599          * eq_pend has all the pending errors.  If atomic_cas_ptr fails or
 600          * has not been called yet (1B), eq_pend still has all the pending
 601          * errors.  If atomic_cas_ptr succeeds (1C), eq_ptail has all the
 602          * pending errors.
 599          * eq_pend has all the pending errors.  If casptr fails or has not
 600          * been called yet (1B), eq_pend still has all the pending errors.
 601          * If casptr succeeds (1C), eq_ptail has all the pending errors.
 603          */
 604         while ((eep = eqp->eq_pend) != NULL) {
 605                 eqp->eq_ptail = eep;
 606                 membar_producer();

 608                 if (atomic_cas_ptr(&eqp->eq_pend, eep, NULL) == eep)
 607                 if (casptr(&eqp->eq_pend, eep, NULL) == eep)
 609                         break;
 610         }

 612         /*
 613          * If no errors were pending, assert that eq_ptail is set to NULL,
 614          * drop the consumer lock, and return without doing anything.
 615          */
 616         if (eep == NULL) {
 617                 ASSERT(eqp->eq_ptail == NULL);
 618                 mutex_exit(&eqp->eq_lock);
 619                 return;
 620         }

 622         /*
 623          * Now iterate from eq_ptail (a.k.a. eep, the newest error) to the
 624          * oldest error, setting the eqe_next pointer so that we can iterate
 625          * over the errors from oldest to newest.  We use membar_producer()
 626          * to make sure that these stores are visible before we set eq_phead.
 627          * If we panic before, during, or just after this loop (case 2),
 628          * errorq_panic() will simply redo this work, as described below.
 629          */
 630         for (eep->eqe_next = NULL; eep->eqe_prev != NULL; eep = eep->eqe_prev)
 631                 eep->eqe_prev->eqe_next = eep;
 632         membar_producer();
```

```
634         /*
635          * Now set eq_phead to the head of the processing list (the oldest
636          * error) and issue another membar_producer() to make sure that
637          * eq_phead is seen as non-NULL before we clear eq_ptail.  If we panic
638          * after eq_phead is set (case 3), we will detect and log these errors
639          * in errorq_panic(), as described below.
640          */
641         eqp->eq_phead = eep;
642         membar_producer();

644         eqp->eq_ptail = NULL;
645         membar_producer();

647         /*
648          * If we enter from errorq_panic_drain(), we may already have
649          * errorq elements on the dump list.  Find the tail of
650          * the list ready for append.
651          */
652         if (panicstr && (dep = eqp->eq_dump) != NULL) {
653                 while (dep->eqe_dump != NULL)
654                         dep = dep->eqe_dump;
655         }

657         /*
658          * Now iterate over the processing list from oldest (eq_phead) to
659          * newest and log each error.  Once an error is logged, we use
660          * atomic clear to return it to the free pool.  If we panic before,
661          * during, or after calling eq_func() (case 4), the error will still be
662          * found on eq_phead and will be logged in errorq_panic below.
663          */

665         while ((eep = eqp->eq_phead) != NULL) {
666                 eqp->eq_func(eqp->eq_private, eep->eqe_data, eep);
667                 eqp->eq_kstat.eqk_logged.value.ui64++;

669                 eqp->eq_phead = eep->eqe_next;
670                 membar_producer();

672                 eep->eqe_next = NULL;

674                 /*
675                  * On panic, we add the element to the dump list for each
676                  * nvlist errorq.  Elements are stored oldest to newest.
677                  * Then continue, so we don't free and subsequently overwrite
678                  * any elements which we've put on the dump queue.
679                  */
680                 if (panicstr && (eqp->eq_flags & ERRORQ_NVLIST)) {
681                         if (eqp->eq_dump == NULL)
682                                 dep = eqp->eq_dump = eep;
683                         else
684                                 dep = dep->eqe_dump = eep;
685                         membar_producer();
686                         continue;
687                 }

689                 eep->eqe_prev = NULL;
690                 BT_ATOMIC_CLEAR(eqp->eq_bitmap, eep - eqp->eq_elems);
691         }

693         mutex_exit(&eqp->eq_lock);
694 }
```

*unchanged_portion_omitted_*

```
733 /*
734  * This function is designed to be called from panic context only, and
735  * therefore does not need to acquire errorq_lock when iterating over
```



```
736  * errorq_list.  This function must be called no more than once for each
737  * 'what' value (if you change this then review the manipulation of 'dep'.
738  */
739 static uint64_t
740 errorq_panic_drain(uint_t what)
741 {
742         errorq_elem_t *eep, *nep, *dep;
743         errorq_t *eqp;
744         uint64_t loggedtmp;
745         uint64_t logged = 0;

747         for (eqp = errorq_list; eqp != NULL; eqp = eqp->eq_next) {
748                 if ((eqp->eq_flags & (ERRORQ_VITAL | ERRORQ_NVLIST)) != what)
749                         continue; /* do not drain this queue on this pass */

751                 loggedtmp = eqp->eq_kstat.eqk_logged.value.ui64;

753                 /*
754                  * In case (1B) above, eq_ptail may be set but the
755                  * atomic_cas_ptr may not have been executed yet or may have
756                  * failed.  Either way, we must log errors in chronological
757                  * order.  So we search the pending list for the error
758                  * pointed to by eq_ptail.  If it is found, we know that all
759                  * subsequent errors are also still on the pending list, so
760                  * just NULL out eq_ptail and let errorq_drain(), below,
761                  * take care of the logging.
753                  * In case (1B) above, eq_ptail may be set but the casptr may
754                  * not have been executed yet or may have failed.  Either way,
755                  * we must log errors in chronological order.  So we search
756                  * the pending list for the error pointed to by eq_ptail.  If
757                  * it is found, we know that all subsequent errors are also
758                  * still on the pending list, so just NULL out eq_ptail and let
759                  * errorq_drain(), below, take care of the logging.
762                  */
763                 for (eep = eqp->eq_pend; eep != NULL; eep = eep->eqe_prev) {
764                         if (eep == eqp->eq_ptail) {
765                                 ASSERT(eqp->eq_phead == NULL);
766                                 eqp->eq_ptail = NULL;
767                                 break;
768                         }
769                 }

771                 /*
772                  * In cases (1C) and (2) above, eq_ptail will be set to the
773                  * newest error on the processing list but eq_phead will still
774                  * be NULL.  We set the eqe_next pointers so we can iterate
775                  * over the processing list in order from oldest error to the
776                  * newest error.  We then set eq_phead to point to the oldest
777                  * error and fall into the for-loop below.
778                  */
779                 if (eqp->eq_phead == NULL && (eep = eqp->eq_ptail) != NULL) {
780                         for (eep->eqe_next = NULL; eep->eqe_prev != NULL;
781                             eep = eep->eqe_prev)
782                                 eep->eqe_prev->eqe_next = eep;

784                         eqp->eq_phead = eep;
785                         eqp->eq_ptail = NULL;
786                 }

788                 /*
789                  * In cases (3) and (4) above (or after case (1C/2) handling),
790                  * eq_phead will be set to the oldest error on the processing
791                  * list.  We log each error and return it to the free pool.
792                  *
793                  * Unlike errorq_drain(), we don't need to worry about updating
794                  * eq_phead because errorq_panic() will be called at most once.
```

```
                 * However, we must use atomic_cas_ptr to update the
                 * freelist in case errors are still being enqueued during
                 * panic.
                 * However, we must use casptr to update the freelist in case
                 * errors are still being enqueued during panic.
                 */
                for (eep = eqp->eq_phead; eep != NULL; eep = nep) {
                        eqp->eq_func(eqp->eq_private, eep->eqe_data, eep);
                        eqp->eq_kstat.eqk_logged.value.ui64++;

                        nep = eep->eqe_next;
                        eep->eqe_next = NULL;

                        /*
                         * On panic, we add the element to the dump list for
                         * each nvlist errorq, stored oldest to newest. Then
                         * continue, so we don't free and subsequently overwrite
                         * any elements which we've put on the dump queue.
                         */
                        if (eqp->eq_flags & ERRORQ_NVLIST) {
                                if (eqp->eq_dump == NULL)
                                        dep = eqp->eq_dump = eep;
                                else
                                        dep = dep->eqe_dump = eep;
                                membar_producer();
                                continue;
                        }

                        eep->eqe_prev = NULL;
                        BT_ATOMIC_CLEAR(eqp->eq_bitmap, eep - eqp->eq_elems);
                }

                /*
                 * Now go ahead and drain any other errors on the pending list.
                 * This call transparently handles case (1A) above, as well as
                 * any other errors that were dispatched after errorq_drain()
                 * completed its first compare-and-swap.
                 */
                errorq_drain(eqp);

                logged += eqp->eq_kstat.eqk_logged.value.ui64 - loggedtmp;
        }
        return (logged);
}
```

unchanged_portion_omitted_

```

/*
 * Commit an errorq element (eqep) for dispatching.
 * This function may be called from any context subject
 * to the Platform Considerations described above.
 */
void
errorq_commit(errorq_t *eqp, errorq_elem_t *eqep, uint_t flag)
{
        errorq_elem_t *old;

        if (eqep == NULL || !(eqp->eq_flags & ERRORQ_ACTIVE)) {
                atomic_add_64(&eqp->eq_kstat.eqk_commit_fail.value.ui64, 1);
                return;
        }

        for (;;) {
                old = eqp->eq_pend;
                eqep->eqe_prev = old;
                membar_producer();
```



```
                if (atomic_cas_ptr(&eqp->eq_pend, old, eqep) == old)
                if (casptr(&eqp->eq_pend, old, eqep) == old)
                        break;
        }

        atomic_add_64(&eqp->eq_kstat.eqk_committed.value.ui64, 1);

        if (flag == ERRORQ_ASYNC && eqp->eq_id != NULL)
                ddi_trigger_softintr(eqp->eq_id);
}
```

unchanged_portion_omitted_

```
**********************************************************
   35950 Mon Jul 28 07:43:50 2014
new/usr/src/uts/common/os/fm.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 364 /*
 365  * Wrapper for panic() that first produces an FMA-style message for admins.
 366  * Normally such messages are generated by fmd(1M)'s syslog-msgs agent: this
 367  * is the one exception to that rule and the only error that gets messaged.
 368  * This function is intended for use by subsystems that have detected a fatal
 369  * error and enqueued appropriate ereports and wish to then force a panic.
 370  */
 371 /*PRINTFLIKE1*/
 372 void
 373 fm_panic(const char *format, ...)
 374 {
 375         va_list ap;

 377         (void) atomic_cas_ptr((void *)&fm_panicstr, NULL, (void *)format);
 377         (void) casptr((void *)&fm_panicstr, NULL, (void *)format);
 378 #if defined(__i386) || defined(__amd64)
 379         fastreboot_disable_highpil();
 380 #endif /* __i386 || __amd64 */
 381         va_start(ap, format);
 382         vpanic(format, ap);
 383         va_end(ap);
 384 }
______unchanged_portion_omitted_
```

```
**********************************************************
    3304 Mon Jul 28 07:43:51 2014
new/usr/src/uts/common/os/kdi.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * 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 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/cpuvar.h>
#include <sys/kdi_impl.h>
#include <sys/reboot.h>
#include <sys/errno.h>
#include <sys/atomic.h>
#include <sys/kmem.h>

kdi_debugvec_t  *kdi_dvec;
struct modctl   *kdi_dmods;

static kdi_dtrace_state_t kdi_dtrace_state = KDI_DTSTATE_IDLE;

void
kdi_dvec_vmready(void)
{
        kdi_dvec->dv_kctl_vmready();
        kdi_dvec->dv_vmready();
}
```

_____unchanged_portion_omitted_

```
int
kdi_dtrace_set(kdi_dtrace_set_t transition)
{
        kdi_dtrace_state_t new, cur;

        do {
                cur = kdi_dtrace_state;

                switch (transition) {
                case KDI_DTSET_DTRACE_ACTIVATE:
                        if (cur == KDI_DTSTATE_KMDB_BPT_ACTIVE)
                                return (EBUSY);
                        if (cur == KDI_DTSTATE_DTRACE_ACTIVE)
                                return (0);
```



```
                        new = KDI_DTSTATE_DTRACE_ACTIVE;
                        break;
                case KDI_DTSET_DTRACE_DEACTIVATE:
                        if (cur == KDI_DTSTATE_KMDB_BPT_ACTIVE)
                                return (EBUSY);
                        if (cur == KDI_DTSTATE_IDLE)
                                return (0);
                        new = KDI_DTSTATE_IDLE;
                        break;
                case KDI_DTSET_KMDB_BPT_ACTIVATE:
                        if (cur == KDI_DTSTATE_DTRACE_ACTIVE)
                                return (EBUSY);
                        if (cur == KDI_DTSTATE_KMDB_BPT_ACTIVE)
                                return (0);
                        new = KDI_DTSTATE_KMDB_BPT_ACTIVE;
                        break;
                case KDI_DTSET_KMDB_BPT_DEACTIVATE:
                        if (cur == KDI_DTSTATE_DTRACE_ACTIVE)
                                return (EBUSY);
                        if (cur == KDI_DTSTATE_IDLE)
                                return (0);
                        new = KDI_DTSTATE_IDLE;
                        break;
                default:
                        return (EINVAL);
                }
        } while (atomic_cas_32((uint_t *)&kdi_dtrace_state, cur, new) != cur);
        } while (cas32((uint_t *)&kdi_dtrace_state, cur, new) != cur);

        return (0);
}
```

_____unchanged_portion_omitted_

****
181712 Mon Jul 28 07:43:51 2014
new/usr/src/uts/common/os/kmem.c
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
3163 static void
3164 kmem_reap_common(void *flag_arg)
3165 {
3166         uint32_t *flag = (uint32_t *)flag_arg;

3168         if (MUTEX_HELD(&kmem_cache_lock) || kmem_taskq == NULL ||
3169             atomic_cas_32(flag, 0, 1) != 0)
3169             cas32(flag, 0, 1) != 0)
3170                 return;

3172         /*
3173          * It may not be kosher to do memory allocation when a reap is called
3174          * is called (for example, if vmem_populate() is in the call chain).
3175          * So we start the reap going with a TQ_NOALLOC dispatch.  If the
3176          * dispatch fails, we reset the flag, and the next reap will try again.
3177          */
3178         if (!taskq_dispatch(kmem_taskq, kmem_reap_start, flag, TQ_NOALLOC))
3179                 *flag = 0;
3180 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  119454 Mon Jul 28 07:43:51 2014
new/usr/src/uts/common/os/lgrp.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

2705 /*
2706  * Recompute load average for the specified partition/lgrp fragment.
2707  *
2708  * We rely on the fact that this routine is called from the clock thread
2709  * at a point before the clock thread can block (i.e. before its first
2710  * lock request).  Since the clock thread can not be preempted (since it
2711  * runs at highest priority), we know that cpu partitions can not change
2712  * (since doing so would require either the repartition requester or the
2713  * cpu_pause thread to run on this cpu), so we can update the cpu's load
2714  * without grabbing cpu_lock.
2715  */
2716 void
2717 lgrp_loadavg(lpl_t *lpl, uint_t nrcpus, int ageflag)
2718 {
2719         uint_t          ncpu;
2720         int64_t         old, new, f;

2722         /*
2723          * 1 - exp(-1/(20 * ncpu)) << 13 = 400 for 1 cpu...
2724          */
2725         static short expval[] = {
2726             0, 3196, 1618, 1083,
2727             814, 652, 543, 466,
2728             408, 363, 326, 297,
2729             272, 251, 233, 218,
2730             204, 192, 181, 172,
2731             163, 155, 148, 142,
2732             136, 130, 125, 121,
2733             116, 112, 109, 105
2734         };

2736         /* ASSERT (called from clock level) */

2738         if ((lpl == NULL) ||    /* we're booting - this is easiest for now */
2739             ((ncpu = lpl->lpl_ncpu) == 0)) {
2740                 return;
2741         }

2743         for (;;) {

2745                 if (ncpu >= sizeof (expval) / sizeof (expval[0]))
2746                         f = expval[1]/ncpu; /* good approx. for large ncpu */
2747                 else
2748                         f = expval[ncpu];

2750                 /*
2751                  * Modify the load average atomically to avoid losing
2752                  * anticipatory load updates (see lgrp_move_thread()).
2753                  */
2754                 if (ageflag) {
2755                         /*
2756                          * We're supposed to both update and age the load.
2757                          * This happens 10 times/sec. per cpu.  We do a
2758                          * little hoop-jumping to avoid integer overflow.
2759                          */
2760                         int64_t         q, r;

2762                         do {
2763                                 old = new = lpl->lpl_loadavg;
```



```
2764                                 q = (old  >> 16) << 7;
2765                                 r = (old  & 0xffff) << 7;
2766                                 new += ((long long)(nrcpus - q) * f -
2767                                     ((r * f) >> 16)) >> 7;

2769                                 /*
2770                                  * Check for overflow
2771                                  */
2772                                 if (new > LGRP_LOADAVG_MAX)
2773                                         new = LGRP_LOADAVG_MAX;
2774                                 else if (new < 0)
2775                                         new = 0;
2776                         } while (atomic_cas_32((lgrp_load_t *)&lpl->lpl_loadavg,
2777                             old, new) != old);
2776                         } while (cas32((lgrp_load_t *)&lpl->lpl_loadavg, old,
2777                             new) != old);
2778                 } else {
2779                         /*
2780                          * We're supposed to update the load, but not age it.
2781                          * This option is used to update the load (which either
2782                          * has already been aged in this 1/10 sec. interval or
2783                          * soon will be) to account for a remotely executing
2784                          * thread.
2785                          */
2786                         do {
2787                                 old = new = lpl->lpl_loadavg;
2788                                 new += f;
2789                                 /*
2790                                  * Check for overflow
2791                                  * Underflow not possible here
2792                                  */
2793                                 if (new < old)
2794                                         new = LGRP_LOADAVG_MAX;
2795                         } while (atomic_cas_32((lgrp_load_t *)&lpl->lpl_loadavg,
2796                             old, new) != old);
2795                         } while (cas32((lgrp_load_t *)&lpl->lpl_loadavg, old,
2796                             new) != old);
2797                 }

2799                 /*
2800                  * Do the same for this lpl's parent
2801                  */
2802                 if ((lpl = lpl->lpl_parent) == NULL)
2803                         break;
2804                 ncpu = lpl->lpl_ncpu;
2805         }
2806 }
______unchanged_portion_omitted_

3230 /*
3231  * An LWP is expected to be assigned to an lgroup for at least this long
3232  * for its anticipatory load to be justified.  NOTE that this value should
3233  * not be set extremely huge (say, larger than 100 years), to avoid problems
3234  * with overflow in the calculation that uses it.
3235  */
3236 #define LGRP_MIN_NSEC   (NANOSEC / 10)          /* 1/10 of a second */
3237 hrtime_t lgrp_min_nsec = LGRP_MIN_NSEC;

3239 /*
3240  * Routine to change a thread's lgroup affiliation.  This routine updates
3241  * the thread's kthread_t struct and its process' proc_t struct to note the
3242  * thread's new lgroup affiliation, and its lgroup affinities.
3243  *
3244  * Note that this is the only routine that modifies a thread's t_lpl field,
3245  * and that adds in or removes anticipatory load.
3246  *
```

```
3247  * If the thread is exiting, newlpl is NULL.
3248  *
3249  * Locking:
3250  * The following lock must be held on entry:
3251  *      cpu_lock, kpreempt_disable(), or thread_lock -- to assure t's new lgrp
3252  *              doesn't get removed from t's partition
3253  *
3254  * This routine is not allowed to grab any locks, since it may be called
3255  * with cpus paused (such as from cpu_offline).
3256  */
3257 void
3258 lgrp_move_thread(kthread_t *t, lpl_t *newlpl, int do_lgrpset_delete)
3259 {
3260         proc_t          *p;
3261         lpl_t           *lpl, *oldlpl;
3262         lgrp_id_t       oldid;
3263         kthread_t       *tp;
3264         uint_t          ncpu;
3265         lgrp_load_t     old, new;

3267         ASSERT(t);
3268         ASSERT(MUTEX_HELD(&cpu_lock) || curthread->t_preempt > 0 ||
3269             THREAD_LOCK_HELD(t));

3271         /*
3272          * If not changing lpls, just return
3273          */
3274         if ((oldlpl = t->t_lpl) == newlpl)
3275                 return;

3277         /*
3278          * Make sure the thread's lwp hasn't exited (if so, this thread is now
3279          * associated with process 0 rather than with its original process).
3280          */
3281         if (t->t_proc_flag & TP_LWPEXIT) {
3282                 if (newlpl != NULL) {
3283                         t->t_lpl = newlpl;
3284                 }
3285                 return;
3286         }

3288         p = ttoproc(t);

3290         /*
3291          * If the thread had a previous lgroup, update its process' p_lgrpset
3292          * to account for it being moved from its old lgroup.
3293          */
3294         if ((oldlpl != NULL) && /* thread had a previous lgroup */
3295             (p->p_tlist != NULL)) {
3296                 oldid = oldlpl->lpl_lgrpid;

3298                 if (newlpl != NULL)
3299                         lgrp_stat_add(oldid, LGRP_NUM_MIGR, 1);

3301                 if ((do_lgrpset_delete) &&
3302                     (klgrpset_ismember(p->p_lgrpset, oldid))) {
3303                         for (tp = p->p_tlist->t_forw; ; tp = tp->t_forw) {
3304                                 /*
3305                                  * Check if a thread other than the thread
3306                                  * that's moving is assigned to the same
3307                                  * lgroup as the thread that's moving.  Note
3308                                  * that we have to compare lgroup IDs, rather
3309                                  * than simply comparing t_lpl's, since the
3310                                  * threads may belong to different partitions
3311                                  * but be assigned to the same lgroup.
3312                                  */
```



```
3313                                 ASSERT(tp->t_lpl != NULL);

3315                                 if ((tp != t) &&
3316                                     (tp->t_lpl->lpl_lgrpid == oldid)) {
3317                                         /*
3318                                          * Another thread is assigned to the
3319                                          * same lgroup as the thread that's
3320                                          * moving, p_lgrpset doesn't change.
3321                                          */
3322                                         break;
3323                                 } else if (tp == p->p_tlist) {
3324                                         /*
3325                                          * No other thread is assigned to the
3326                                          * same lgroup as the exiting thread,
3327                                          * clear the lgroup's bit in p_lgrpset.
3328                                          */
3329                                         klgrpset_del(p->p_lgrpset, oldid);
3330                                         break;
3331                                 }
3332                         }
3333                 }

3335                 /*
3336                  * If this thread was assigned to its old lgroup for such a
3337                  * short amount of time that the anticipatory load that was
3338                  * added on its behalf has aged very little, remove that
3339                  * anticipatory load.
3340                  */
3341                 if ((t->t_anttime + lgrp_min_nsec > gethrtime()) &&
3342                     ((ncpu = oldlpl->lpl_ncpu) > 0)) {
3343                         lpl = oldlpl;
3344                         for (;;) {
3345                                 do {
3346                                         old = new = lpl->lpl_loadavg;
3347                                         new -= LGRP_LOADAVG_MAX_EFFECT(ncpu);
3348                                         if (new > old) {
3349                                                 /*
3350                                                  * this can happen if the load
3351                                                  * average was aged since we
3352                                                  * added in the anticipatory
3353                                                  * load
3354                                                  */
3355                                                 new = 0;
3356                                         }
3357                                 } while (atomic_cas_32(
3357                                 } while (cas32(
3358                                     (lgrp_load_t *)&lpl->lpl_loadavg, old,
3359                                     new) != old);

3361                                 lpl = lpl->lpl_parent;
3362                                 if (lpl == NULL)
3363                                         break;

3365                                 ncpu = lpl->lpl_ncpu;
3366                                 ASSERT(ncpu > 0);
3367                         }
3368                 }
3369         }
3370         /*
3371          * If the thread has a new lgroup (i.e. it's not exiting), update its
3372          * t_lpl and its process' p_lgrpset, and apply an anticipatory load
3373          * to its new lgroup to account for its move to its new lgroup.
3374          */
3375         if (newlpl != NULL) {
3376                 /*
3377                  * This thread is moving to a new lgroup
```

```
3378                  */
3379                 t->t_lpl = newlpl;
3380                 if (t->t_tid == 1 && p->p_t1_lgrpid != newlpl->lpl_lgrpid) {
3381                         p->p_t1_lgrpid = newlpl->lpl_lgrpid;
3382                         membar_producer();
3383                         if (p->p_tr_lgrpid != LGRP_NONE &&
3384                             p->p_tr_lgrpid != p->p_t1_lgrpid) {
3385                                 lgrp_update_trthr_migrations(1);
3386                         }
3387                 }

3389                 /*
3390                  * Reflect move in load average of new lgroup
3391                  * unless it is root lgroup
3392                  */
3393                 if (lgrp_table[newlpl->lpl_lgrpid] == lgrp_root)
3394                         return;

3396                 if (!klgrpset_ismember(p->p_lgrpset, newlpl->lpl_lgrpid)) {
3397                         klgrpset_add(p->p_lgrpset, newlpl->lpl_lgrpid);
3398                 }

3400                 /*
3401                  * It'll take some time for the load on the new lgroup
3402                  * to reflect this thread's placement on it.  We'd
3403                  * like not, however, to have all threads between now
3404                  * and then also piling on to this lgroup.  To avoid
3405                  * this pileup, we anticipate the load this thread
3406                  * will generate on its new lgroup.  The goal is to
3407                  * make the lgroup's load appear as though the thread
3408                  * had been there all along.  We're very conservative
3409                  * in calculating this anticipatory load, we assume
3410                  * the worst case case (100% CPU-bound thread).  This
3411                  * may be modified in the future to be more accurate.
3412                  */
3413                 lpl = newlpl;
3414                 for (;;) {
3415                         ncpu = lpl->lpl_ncpu;
3416                         ASSERT(ncpu > 0);
3417                         do {
3418                                 old = new = lpl->lpl_loadavg;
3419                                 new += LGRP_LOADAVG_MAX_EFFECT(ncpu);
3420                                 /*
3421                                  * Check for overflow
3422                                  * Underflow not possible here
3423                                  */
3424                                 if (new < old)
3425                                         new = UINT32_MAX;
3426                         } while (atomic_cas_32((lgrp_load_t *)&lpl->lpl_loadavg,
3427                             old, new) != old);
3426                         } while (cas32((lgrp_load_t *)&lpl->lpl_loadavg, old,
3427                             new) != old);

3429                         lpl = lpl->lpl_parent;
3430                         if (lpl == NULL)
3431                                 break;
3432                 }
3433                 t->t_anttime = gethrtime();
3434         }
3435 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   22844 Mon Jul 28 07:43:52 2014
new/usr/src/uts/common/os/msacct.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_


 545 /*
 546  * Defined to determine whether a lwp is still on a processor.
 547  */

 549 #define T_ONPROC(kt)    \
 550         ((kt)->t_mstate < LMS_SLEEP)
 551 #define T_OFFPROC(kt)   \
 552         ((kt)->t_mstate >= LMS_SLEEP)

 554 uint_t
 555 cpu_update_pct(kthread_t *t, hrtime_t newtime)
 556 {
 557         hrtime_t delta;
 558         hrtime_t hrlb;
 559         uint_t pctcpu;
 560         uint_t npctcpu;

 562         /*
 563          * This routine can get called at PIL > 0, this *has* to be
 564          * done atomically. Holding locks here causes bad things to happen.
 565          * (read: deadlock).
 566          */

 568         do {
 569                 if (T_ONPROC(t) && t->t_waitrq == 0) {
 570                         hrlb = t->t_hrtime;
 571                         delta = newtime - hrlb;
 572                         if (delta < 0) {
 573                                 newtime = gethrtime_unscaled();
 574                                 delta = newtime - hrlb;
 575                         }
 576                         t->t_hrtime = newtime;
 577                         scalehrtime(&delta);
 578                         pctcpu = t->t_pctcpu;
 579                         npctcpu = cpu_grow(pctcpu, delta);
 580                 } else {
 581                         hrlb = t->t_hrtime;
 582                         delta = newtime - hrlb;
 583                         if (delta < 0) {
 584                                 newtime = gethrtime_unscaled();
 585                                 delta = newtime - hrlb;
 586                         }
 587                         t->t_hrtime = newtime;
 588                         scalehrtime(&delta);
 589                         pctcpu = t->t_pctcpu;
 590                         npctcpu = cpu_decay(pctcpu, delta);
 591                 }
 592         } while (atomic_cas_32(&t->t_pctcpu, pctcpu, npctcpu) != pctcpu);
 592         } while (cas32(&t->t_pctcpu, pctcpu, npctcpu) != pctcpu);

 594         return (npctcpu);
 595 }

 597 /*
 598  * Change the microstate level for the LWP and update the
 599  * associated accounting information.  Return the previous
 600  * LWP state.
 601  */
```



```
 602 int
 603 new_mstate(kthread_t *t, int new_state)
 604 {
 605         struct mstate *ms;
 606         unsigned state;
 607         hrtime_t *mstimep;
 608         hrtime_t curtime;
 609         hrtime_t newtime;
 610         hrtime_t oldtime;
 611         hrtime_t ztime;
 612         hrtime_t origstart;
 613         klwp_t *lwp;
 614         zone_t *z;

 616         ASSERT(new_state != LMS_WAIT_CPU);
 617         ASSERT((unsigned)new_state < NMSTATES);
 618         ASSERT(t == curthread || THREAD_LOCK_HELD(t));

 620         /*
 621          * Don't do microstate processing for threads without a lwp (kernel
 622          * threads).  Also, if we're an interrupt thread that is pinning another
 623          * thread, our t_mstate hasn't been initialized.  We'd be modifying the
 624          * microstate of the underlying lwp which doesn't realize that it's
 625          * pinned.  In this case, also don't change the microstate.
 626          */
 627         if (((lwp = ttolwp(t)) == NULL) || t->t_intr)
 628                 return (LMS_SYSTEM);

 630         curtime = gethrtime_unscaled();

 632         /* adjust cpu percentages before we go any further */
 633         (void) cpu_update_pct(t, curtime);

 635         ms = &lwp->lwp_mstate;
 636         state = t->t_mstate;
 637         origstart = ms->ms_state_start;
 638         do {
 639                 switch (state) {
 640                 case LMS_TFAULT:
 641                 case LMS_DFAULT:
 642                 case LMS_KFAULT:
 643                 case LMS_USER_LOCK:
 644                         mstimep = &ms->ms_acct[LMS_SYSTEM];
 645                         break;
 646                 default:
 647                         mstimep = &ms->ms_acct[state];
 648                         break;
 649                 }
 650                 ztime = newtime = curtime - ms->ms_state_start;
 651                 if (newtime < 0) {
 652                         curtime = gethrtime_unscaled();
 653                         oldtime = *mstimep - 1; /* force CAS to fail */
 654                         continue;
 655                 }
 656                 oldtime = *mstimep;
 657                 newtime += oldtime;
 658                 t->t_mstate = new_state;
 659                 ms->ms_state_start = curtime;
 660         } while (atomic_cas_64((uint64_t *)mstimep, oldtime, newtime) !=
 661             oldtime);
 660         } while (cas64((uint64_t *)mstimep, oldtime, newtime) != oldtime);

 663         /*
 664          * When the system boots the initial startup thread will have a
 665          * ms_state_start of 0 which would add a huge system time to the global
 666          * zone.  We want to skip aggregating that initial bit of work.
```

```
667          */
668         if (origstart != 0) {
669                 z = ttozone(t);
670                 if (state == LMS_USER)
671                         atomic_add_64(&z->zone_utime, ztime);
672                 else if (state == LMS_SYSTEM)
673                         atomic_add_64(&z->zone_stime, ztime);
674         }

676         /*
677          * Remember the previous running microstate.
678          */
679         if (state != LMS_SLEEP && state != LMS_STOPPED)
680                 ms->ms_prev = state;

682         /*
683          * Switch CPU microstate if appropriate
684          */

686         kpreempt_disable(); /* MUST disable kpreempt before touching t->cpu */
687         ASSERT(t->t_cpu == CPU);
688         if (!CPU_ON_INTR(t->t_cpu) && curthread->t_intr == NULL) {
689                 if (new_state == LMS_USER && t->t_cpu->cpu_mstate != CMS_USER)
690                         new_cpu_mstate(CMS_USER, curtime);
691                 else if (new_state != LMS_USER &&
692                     t->t_cpu->cpu_mstate != CMS_SYSTEM)
693                         new_cpu_mstate(CMS_SYSTEM, curtime);
694         }
695         kpreempt_enable();

697         return (ms->ms_prev);
698 }

700 /*
701  * Restore the LWP microstate to the previous runnable state.
702  * Called from disp() with the newly selected lwp.
703  */
704 void
705 restore_mstate(kthread_t *t)
706 {
707         struct mstate *ms;
708         hrtime_t *mstimep;
709         klwp_t *lwp;
710         hrtime_t curtime;
711         hrtime_t waitrq;
712         hrtime_t newtime;
713         hrtime_t oldtime;
714         hrtime_t waittime;
715         zone_t *z;

717         /*
718          * Don't call restore mstate of threads without lwps.  (Kernel threads)
719          *
720          * threads with t_intr set shouldn't be in the dispatcher, so assert
721          * that nobody here has t_intr.
722          */
723         ASSERT(t->t_intr == NULL);

725         if ((lwp = ttolwp(t)) == NULL)
726                 return;

728         curtime = gethrtime_unscaled();
729         (void) cpu_update_pct(t, curtime);
730         ms = &lwp->lwp_mstate;
731         ASSERT((unsigned)t->t_mstate < NMSTATES);
732         do {
```



```
733                 switch (t->t_mstate) {
734                 case LMS_SLEEP:
735                         /*
736                          * Update the timer for the current sleep state.
737                          */
738                         ASSERT((unsigned)ms->ms_prev < NMSTATES);
739                         switch (ms->ms_prev) {
740                         case LMS_TFAULT:
741                         case LMS_DFAULT:
742                         case LMS_KFAULT:
743                         case LMS_USER_LOCK:
744                                 mstimep = &ms->ms_acct[ms->ms_prev];
745                                 break;
746                         default:
747                                 mstimep = &ms->ms_acct[LMS_SLEEP];
748                                 break;
749                         }
750                         /*
751                          * Return to the previous run state.
752                          */
753                         t->t_mstate = ms->ms_prev;
754                         break;
755                 case LMS_STOPPED:
756                         mstimep = &ms->ms_acct[LMS_STOPPED];
757                         /*
758                          * Return to the previous run state.
759                          */
760                         t->t_mstate = ms->ms_prev;
761                         break;
762                 case LMS_TFAULT:
763                 case LMS_DFAULT:
764                 case LMS_KFAULT:
765                 case LMS_USER_LOCK:
766                         mstimep = &ms->ms_acct[LMS_SYSTEM];
767                         break;
768                 default:
769                         mstimep = &ms->ms_acct[t->t_mstate];
770                         break;
771                 }
772                 waitrq = t->t_waitrq;   /* hopefully atomic */
773                 if (waitrq == 0) {
774                         waitrq = curtime;
775                 }
776                 t->t_waitrq = 0;
777                 newtime = waitrq - ms->ms_state_start;
778                 if (newtime < 0) {
779                         curtime = gethrtime_unscaled();
780                         oldtime = *mstimep - 1; /* force CAS to fail */
781                         continue;
782                 }
783                 oldtime = *mstimep;
784                 newtime += oldtime;
785         } while (atomic_cas_64((uint64_t *)mstimep, oldtime, newtime) !=
786             oldtime);
784         } while (cas64((uint64_t *)mstimep, oldtime, newtime) != oldtime);

788         /*
789          * Update the WAIT_CPU timer and per-cpu waitrq total.
790          */
791         z = ttozone(t);
792         waittime = curtime - waitrq;
793         ms->ms_acct[LMS_WAIT_CPU] += waittime;
794         atomic_add_64(&z->zone_wtime, waittime);
795         CPU->cpu_waitrq += waittime;
796         ms->ms_state_start = curtime;
797 }
```

*unchanged_portion_omitted_*

****
22695 Mon Jul 28 07:43:52 2014
new/usr/src/uts/common/os/mutex.c
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
 233 /*
 234  * If the system panics on a mutex, save the address of the offending
 235  * mutex in panic_mutex_addr, and save the contents in panic_mutex.
 236  */
 237 static mutex_impl_t panic_mutex;
 238 static mutex_impl_t *panic_mutex_addr;

 240 static void
 241 mutex_panic(char *msg, mutex_impl_t *lp)
 242 {
 243         if (panicstr)
 244                 return;

 246         if (atomic_cas_ptr(&panic_mutex_addr, NULL, lp) == NULL)
 246         if (casptr(&panic_mutex_addr, NULL, lp) == NULL)
 247                 panic_mutex = *lp;

 249         panic("%s, lp=%p owner=%p thread=%p",
 250             msg, (void *)lp, (void *)MUTEX_OWNER(&panic_mutex),
 251             (void *)curthread);
 252 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    8586 Mon Jul 28 07:43:52 2014
new/usr/src/uts/common/os/printf.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 *
 * Copyright 2012 Nexenta Systems, Inc.  All rights reserved.
 */

#include <sys/param.h>
#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/inline.h>
#include <sys/varargs.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/cmn_err.h>
#include <sys/syslog.h>
#include <sys/log.h>
#include <sys/proc.h>
#include <sys/vnode.h>
#include <sys/session.h>
#include <sys/stream.h>
#include <sys/kmem.h>
#include <sys/kobj.h>
#include <sys/atomic.h>
#include <sys/console.h>
#include <sys/cpuvar.h>
#include <sys/modctl.h>
#include <sys/reboot.h>
#include <sys/debug.h>
#include <sys/panic.h>
#include <sys/spl.h>
#include <sys/zone.h>
#include <sys/sunddi.h>

/*
 * In some debugging situations it's useful to log all messages to panicbuf,
 * which is persistent across reboots (on most platforms).  The range
 * panicbuf[panicbuf_log..PANICBUFSIZE-1] may be used for this purpose.
 * By default, panicbuf_log == PANICBUFSIZE and no messages are logged.
 * To enable panicbuf logging, set panicbuf_log to a small value, say 1K;
 * this will reserve 1K for panic information and 7K for message logging.
```



```
 */
uint32_t panicbuf_log = PANICBUFSIZE;
uint32_t panicbuf_index = PANICBUFSIZE;

static int aask, aok;
static int ce_to_sl[CE_IGNORE] = { SL_NOTE, SL_NOTE, SL_WARN, SL_FATAL };
static char ce_prefix[CE_IGNORE][10] = { "", "NOTICE: ", "WARNING: ", "" };
static char ce_suffix[CE_IGNORE][2] = { "", "\n", "\n", "" };

static void
cprintf(const char *fmt, va_list adx, int sl, const char *prefix,
        const char *suffix, void *site, int mid, int sid, int level,
        zoneid_t zoneid)
{
        uint32_t msgid;
        size_t bufsize = LOG_MSGSIZE;
        char buf[LOG_MSGSIZE];
        char *bufp = buf;
        char *body, *msgp, *bufend;
        mblk_t *mp;
        int s, on_intr;
        size_t len;

        s = splhi();
        on_intr = CPU_ON_INTR(CPU) ||
            (interrupts_unleashed && (spltoipl(s) > LOCK_LEVEL));
        splx(s);

        ASSERT(zoneid == GLOBAL_ZONEID || !on_intr);

        STRLOG_MAKE_MSGID(fmt, msgid);

        if (strchr("^!?", fmt[0]) != NULL) {
                if (fmt[0] == '^')
                        sl |= SL_CONSONLY;
                else if (fmt[0] == '!' ||
                    (prefix[0] == '\0' && !(boothowto & RB_VERBOSE)))
                        sl = (sl & ~(SL_USER | SL_NOTE)) | SL_LOGONLY;
                fmt++;
        }

        if ((sl & SL_USER) && (MUTEX_HELD(&pidlock) || on_intr)) {
                zoneid = getzoneid();
                sl = sl & ~(SL_USER | SL_LOGONLY) | SL_CONSOLE;
        }

retry:
        bufend = bufp + bufsize;
        msgp = bufp;
        body = msgp += snprintf(msgp, bufend - msgp,
            "%s: [ID %u FACILITY_AND_PRIORITY] ",
            mod_containing_pc(site), msgid);
        msgp += snprintf(msgp, bufend - msgp, prefix);
        msgp += vsnprintf(msgp, bufend - msgp, fmt, adx);
        msgp += snprintf(msgp, bufend - msgp, suffix);
        len = strlen(body);

        if (((sl & SL_CONSONLY) && panicstr) ||
            (zoneid == GLOBAL_ZONEID && log_global.lz_active == 0)) {
                console_printf("%s", body);
                goto out;
        }

        if (msgp - bufp >= bufsize && !on_intr) {
                ASSERT(bufp == buf);
                bufsize = msgp - bufp + 1;
```

```
128                 bufp = kmem_alloc(bufsize, KM_NOSLEEP);
129                 if (bufp != NULL)
130                         goto retry;
131                 bufsize = LOG_MSGSIZE;
132                 bufp = buf;
133         }

135         mp = log_makemsg(mid, sid, level, sl, LOG_KERN, bufp,
136             MIN(bufsize, msgp - bufp + 1), on_intr);
137         if (mp == NULL) {
138                 if ((sl & (SL_CONSOLE | SL_LOGONLY)) == SL_CONSOLE && !on_intr)
139                         console_printf("%s", body);
140                 goto out;
141         }

143         if (sl & SL_USER) {
144                 ssize_t resid;
145                 sess_t *sp;

147                 if ((sp = tty_hold()) != NULL) {
148                         if (sp->s_vp != NULL)
149                                 (void) vn_rdwr(UIO_WRITE, sp->s_vp, body,
150                                     len, 0LL, UIO_SYSSPACE, FAPPEND,
151                                     (rlim64_t)LOG_HIWAT, kcred, &resid);
152                         tty_rele(sp);
153                 }
154         }

156         if (on_intr && !panicstr) {
157                 (void) putq(log_intrq, mp);
158                 softcall((void (*)(void *))log_flushq, log_intrq);
159         } else {
160                 log_sendmsg(mp, zoneid);
161         }
162 out:
163         if (panicbuf_log + len < PANICBUFSIZE) {
164                 uint32_t old, new;
165                 do {
166                         old = panicbuf_index;
167                         new = old + len;
168                         if (new >= PANICBUFSIZE)
169                                 new = panicbuf_log + len;
170                 } while (atomic_cas_32(&panicbuf_index, old, new) != old);
170                 } while (cas32(&panicbuf_index, old, new) != old);
171                 bcopy(body, &panicbuf[new - len], len);
172         }
173         if (bufp != buf)
174                 kmem_free(bufp, bufsize);
175 }
```

unchanged_portion_omitted_

****
22032 Mon Jul 28 07:43:52 2014
new/usr/src/uts/common/os/rwlock.c
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
 207 /*
 208  * If the system panics on an rwlock, save the address of the offending
 209  * rwlock in panic_rwlock_addr, and save the contents in panic_rwlock.
 210  */
 211 static rwlock_impl_t panic_rwlock;
 212 static rwlock_impl_t *panic_rwlock_addr;

 214 static void
 215 rw_panic(char *msg, rwlock_impl_t *lp)
 216 {
 217         if (panicstr)
 218                 return;

 220         if (atomic_cas_ptr(&panic_rwlock_addr, NULL, lp) == NULL)
 220         if (casptr(&panic_rwlock_addr, NULL, lp) == NULL)
 221                 panic_rwlock = *lp;

 223         panic("%s, lp=%p wwwh=%lx thread=%p",
 224             msg, (void *)lp, panic_rwlock.rw_wwwh, (void *)curthread);
 225 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    5732 Mon Jul 28 07:43:52 2014
new/usr/src/uts/common/sys/bitmap.h
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * 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 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2014 by Delphix. All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


#ifndef _SYS_BITMAP_H
#define _SYS_BITMAP_H

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/feature_tests.h>
#if defined(__GNUC__) && defined(_ASM_INLINES) && \
        (defined(__i386) || defined(__amd64))
#include <asm/bitmap.h>
#endif

/*
 * Operations on bitmaps of arbitrary size
 * A bitmap is a vector of 1 or more ulong_t's.
 * The user of the package is responsible for range checks and keeping
 * track of sizes.
 */

#ifdef _LP64
#define BT_ULSHIFT      6 /* log base 2 of BT_NBIPUL, to extract word index */
#define BT_ULSHIFT32    5 /* log base 2 of BT_NBIPUL, to extract word index */
#else
#define BT_ULSHIFT      5 /* log base 2 of BT_NBIPUL, to extract word index */
#endif
```



```
#define BT_NBIPUL       (1 << BT_ULSHIFT)       /* n bits per ulong_t */
#define BT_ULMASK       (BT_NBIPUL - 1)         /* to extract bit index */

#ifdef _LP64
#define BT_NBIPUL32     (1 << BT_ULSHIFT32)     /* n bits per ulong_t */
#define BT_ULMASK32     (BT_NBIPUL32 - 1)       /* to extract bit index */
#define BT_ULMAXMASK    0xffffffffffffffff      /* used by bt_getlowbit */
#else
#define BT_ULMAXMASK    0xffffffff
#endif

/*
 * bitmap is a ulong_t *, bitindex an index_t
 *
 * The macros BT_WIM and BT_BIW internal; there is no need
 * for users of this package to use them.
 */

/*
 * word in map
 */
#define BT_WIM(bitmap, bitindex) \
        ((bitmap)[(bitindex) >> BT_ULSHIFT])
/*
 * bit in word
 */
#define BT_BIW(bitindex) \
        (1UL << ((bitindex) & BT_ULMASK))

#ifdef _LP64
#define BT_WIM32(bitmap, bitindex) \
        ((bitmap)[(bitindex) >> BT_ULSHIFT32])

#define BT_BIW32(bitindex) \
        (1UL << ((bitindex) & BT_ULMASK32))
#endif

/*
 * These are public macros
 *
 * BT_BITOUL == n bits to n ulong_t's
 */
#define BT_BITOUL(nbits) \
        (((nbits) + BT_NBIPUL - 1l) / BT_NBIPUL)
#define BT_SIZEOFMAP(nbits) \
        (BT_BITOUL(nbits) * sizeof (ulong_t))
#define BT_TEST(bitmap, bitindex) \
        ((BT_WIM((bitmap), (bitindex)) & BT_BIW(bitindex)) ? 1 : 0)
#define BT_SET(bitmap, bitindex) \
        { BT_WIM((bitmap), (bitindex)) |= BT_BIW(bitindex); }
#define BT_CLEAR(bitmap, bitindex) \
        { BT_WIM((bitmap), (bitindex)) &= ~BT_BIW(bitindex); }

#ifdef _LP64
#define BT_BITOUL32(nbits) \
        (((nbits) + BT_NBIPUL32 - 1l) / BT_NBIPUL32)
#define BT_SIZEOFMAP32(nbits) \
        (BT_BITOUL32(nbits) * sizeof (uint_t))
#define BT_TEST32(bitmap, bitindex) \
        ((BT_WIM32((bitmap), (bitindex)) & BT_BIW32(bitindex)) ? 1 : 0)
#define BT_SET32(bitmap, bitindex) \
        { BT_WIM32((bitmap), (bitindex)) |= BT_BIW32(bitindex); }
#define BT_CLEAR32(bitmap, bitindex) \
        { BT_WIM32((bitmap), (bitindex)) &= ~BT_BIW32(bitindex); }
#endif /* _LP64 */
```

```
 129 /*
 130  * BIT_ONLYONESET is a private macro not designed for bitmaps of
 131  * arbitrary size.  u must be an unsigned integer/long.  It returns
 132  * true if one and only one bit is set in u.
 133  */
 134 #define BIT_ONLYONESET(u) \
 135         ((((u) == 0) ? 0 : ((u) & ((u) - 1)) == 0))

 137 #if defined(_KERNEL) && !defined(_ASM)
 138 #include <sys/atomic.h>

 140 /*
 141  * return next available bit index from map with specified number of bits
 142  */
 143 extern index_t  bt_availbit(ulong_t *bitmap, size_t nbits);
 144 /*
 145  * find the highest order bit that is on, and is within or below
 146  * the word specified by wx
 147  */
 148 extern int      bt_gethighbit(ulong_t *mapp, int wx);
 149 extern int      bt_range(ulong_t *bitmap, size_t *pos1, size_t *pos2,
 150                         size_t end_pos);
 151 /*
 152  * Find highest and lowest one bit set.
 153  *      Returns bit number + 1 of bit that is set, otherwise returns 0.
 154  * Low order bit is 0, high order bit is 31.
 155  */
 156 extern int      highbit(ulong_t);
 157 extern int      highbit64(uint64_t);
 158 extern int      lowbit(ulong_t);
 159 extern int      bt_getlowbit(ulong_t *bitmap, size_t start, size_t stop);
 160 extern void     bt_copy(ulong_t *, ulong_t *, ulong_t);

 162 /*
 163  * find the parity
 164  */
 165 extern int      odd_parity(ulong_t);

 167 /*
 168  * Atomically set/clear bits
 169  * Atomic exclusive operations will set "result" to "-1"
 170  * if the bit is already set/cleared. "result" will be set
 171  * to 0 otherwise.
 172  */
 173 #define BT_ATOMIC_SET(bitmap, bitindex) \
 174         { atomic_or_ulong(&(BT_WIM(bitmap, bitindex)), BT_BIW(bitindex)); }
 174         { atomic_or_long(&(BT_WIM(bitmap, bitindex)), BT_BIW(bitindex)); }
 175 #define BT_ATOMIC_CLEAR(bitmap, bitindex) \
 176         { atomic_and_ulong(&(BT_WIM(bitmap, bitindex)), ~BT_BIW(bitindex)); }
 176         { atomic_and_long(&(BT_WIM(bitmap, bitindex)), ~BT_BIW(bitindex)); }

 178 #define BT_ATOMIC_SET_EXCL(bitmap, bitindex, result) \
 179         { result = atomic_set_long_excl(&(BT_WIM(bitmap, bitindex)),    \
 180             (bitindex) % BT_NBIPUL); }
 181 #define BT_ATOMIC_CLEAR_EXCL(bitmap, bitindex, result) \
 182         { result = atomic_clear_long_excl(&(BT_WIM(bitmap, bitindex)),  \
 183             (bitindex) % BT_NBIPUL); }

 185 /*
 186  * Extracts bits between index h (high, inclusive) and l (low, exclusive) from
 187  * u, which must be an unsigned integer.
 188  */
 189 #define BITX(u, h, l)   (((u) >> (l)) & ((1LU << ((h) - (l) + 1LU)) - 1LU))

 191 #endif  /* _KERNEL && !_ASM */
```



```
 193 #ifdef  __cplusplus
 194 }
______unchanged_portion_omitted_
```

```
**********************************************************
   30223 Mon Jul 28 07:43:52 2014
new/usr/src/uts/common/sys/cpuvar.h
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

531 #define CPUSET_ATOMIC_DEL(set, cpu)     atomic_and_ulong(&(set), ~CPUSET(cpu))
532 #define CPUSET_ATOMIC_ADD(set, cpu)     atomic_or_ulong(&(set), CPUSET(cpu))
531 #define CPUSET_ATOMIC_DEL(set, cpu)     atomic_and_long(&(set), ~CPUSET(cpu))
532 #define CPUSET_ATOMIC_ADD(set, cpu)     atomic_or_long(&(set), CPUSET(cpu))

534 #define CPUSET_ATOMIC_XADD(set, cpu, result) \
535         { result = atomic_set_long_excl(&(set), (cpu)); }

537 #define CPUSET_ATOMIC_XDEL(set, cpu, result) \
538         { result = atomic_clear_long_excl(&(set), (cpu)); }

540 #else   /* CPUSET_WORDS <= 0 */

542 #error NCPU is undefined or invalid

544 #endif  /* CPUSET_WORDS */

546 extern cpuset_t cpu_seqid_inuse;

548 #endif  /* (_KERNEL || _KMEMUSER) && _MACHDEP */

550 #define CPU_CPR_OFFLINE         0x0
551 #define CPU_CPR_ONLINE          0x1
552 #define CPU_CPR_IS_OFFLINE(cpu) (((cpu)->cpu_cpr_flags & CPU_CPR_ONLINE) == 0)
553 #define CPU_CPR_IS_ONLINE(cpu)  ((cpu)->cpu_cpr_flags & CPU_CPR_ONLINE)
554 #define CPU_SET_CPR_FLAGS(cpu, flag)    ((cpu)->cpu_cpr_flags |= flag)

556 #if defined(_KERNEL) || defined(_KMEMUSER)

558 extern struct cpu       *cpu[];         /* indexed by CPU number */
559 extern struct cpu       **cpu_seq;      /* indexed by sequential CPU id */
560 extern cpu_t            *cpu_list;      /* list of CPUs */
561 extern cpu_t            *cpu_active;    /* list of active CPUs */
562 extern int              ncpus;          /* number of CPUs present */
563 extern int              ncpus_online;   /* number of CPUs not quiesced */
564 extern int              max_ncpus;      /* max present before ncpus is known */
565 extern int              boot_max_ncpus; /* like max_ncpus but for real */
566 extern int              boot_ncpus;     /* # cpus present @ boot */
567 extern processorid_t    max_cpuid;      /* maximum CPU number */
568 extern struct cpu       *cpu_inmotion;  /* offline or partition move target */
569 extern cpu_t            *clock_cpu_list;
570 extern processorid_t    max_cpu_seqid_ever;     /* maximum seqid ever given */

572 #if defined(__i386) || defined(__amd64)
573 extern struct cpu *curcpup(void);
574 #define CPU             (curcpup())     /* Pointer to current CPU */
575 #else
576 #define CPU             (curthread->t_cpu)      /* Pointer to current CPU */
577 #endif

579 /*
580  * CPU_CURRENT indicates to thread_affinity_set to use CPU->cpu_id
581  * as the target and to grab cpu_lock instead of requiring the caller
582  * to grab it.
583  */
584 #define CPU_CURRENT     -3

586 /*
587  * Per-CPU statistics
```



```
588  *
589  * cpu_stats_t contains numerous system and VM-related statistics, in the form
590  * of gauges or monotonically-increasing event occurrence counts.
591  */

593 #define CPU_STATS_ENTER_K()     kpreempt_disable()
594 #define CPU_STATS_EXIT_K()      kpreempt_enable()

596 #define CPU_STATS_ADD_K(class, stat, amount) \
597         {       kpreempt_disable(); /* keep from switching CPUs */\
598                 CPU_STATS_ADDQ(CPU, class, stat, amount); \
599                 kpreempt_enable(); \
600         }

602 #define CPU_STATS_ADDQ(cp, class, stat, amount) {                       \
603         extern void __dtrace_probe___cpu_##class##info_##stat(uint_t,   \
604             uint64_t *, cpu_t *);                                       \
605         uint64_t *stataddr = &((cp)->cpu_stats.class.stat);             \
606         __dtrace_probe___cpu_##class##info_##stat((amount),             \
607             stataddr, cp);                                              \
608         *(stataddr) += (amount);                                        \
609 }
______unchanged_portion_omitted_
```

```
**********************************************************
   37194 Mon Jul 28 07:43:53 2014
new/usr/src/uts/common/vm/seg_kp.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 758 /*
 759  * segkp_map_red() will check the current frame pointer against the
 760  * stack base.  If the amount of stack remaining is questionable
 761  * (less than red_minavail), then segkp_map_red() will map in the redzone
 762  * and return 1.  Otherwise, it will return 0.  segkp_map_red() can
 763  * _only_ be called when:
 764  *
 765  *   - it is safe to sleep on page_create_va().
 766  *   - the caller is non-swappable.
 767  *
 768  * It is up to the caller to remember whether segkp_map_red() successfully
 769  * mapped the redzone, and, if so, to call segkp_unmap_red() at a later
 770  * time.  Note that the caller must _remain_ non-swappable until after
 771  * calling segkp_unmap_red().
 772  *
 773  * Currently, this routine is only called from pagefault() (which necessarily
 774  * satisfies the above conditions).
 775  */
 776 #if defined(STACK_GROWTH_DOWN)
 777 int
 778 segkp_map_red(void)
 779 {
 780         uintptr_t fp = STACK_BIAS + (uintptr_t)getfp();
 781 #ifndef _LP64
 782         caddr_t stkbase;
 783 #endif

 785         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

 787         /*
 788          * Optimize for the common case where we simply return.
 789          */
 790         if ((curthread->t_red_pp == NULL) &&
 791             (fp - (uintptr_t)curthread->t_stkbase >= red_minavail))
 792                 return (0);

 794 #if defined(_LP64)
 795         /*
 796          * XXX  We probably need something better than this.
 797          */
 798         panic("kernel stack overflow");
 799         /*NOTREACHED*/
 800 #else /* _LP64 */
 801         if (curthread->t_red_pp == NULL) {
 802                 page_t *red_pp;
 803                 struct seg kseg;

 805                 caddr_t red_va = (caddr_t)
 806                     (((uintptr_t)curthread->t_stkbase & (uintptr_t)PAGEMASK) -
 807                     PAGESIZE);

 809                 ASSERT(page_exists(&kvp, (u_offset_t)(uintptr_t)red_va) ==
 810                     NULL);

 812                 /*
 813                  * Allocate the physical for the red page.
 814                  */
 815                 /*
 816                  * No PG_NORELOC here to avoid waits. Unlikely to get
```



```
 817                  * a relocate happening in the short time the page exists
 818                  * and it will be OK anyway.
 819                  */

 821                 kseg.s_as = &kas;
 822                 red_pp = page_create_va(&kvp, (u_offset_t)(uintptr_t)red_va,
 823                     PAGESIZE, PG_WAIT | PG_EXCL, &kseg, red_va);
 824                 ASSERT(red_pp != NULL);

 826                 /*
 827                  * So we now have a page to jam into the redzone...
 828                  */
 829                 page_io_unlock(red_pp);

 831                 hat_memload(kas.a_hat, red_va, red_pp,
 832                     (PROT_READ|PROT_WRITE), HAT_LOAD_LOCK);
 833                 page_downgrade(red_pp);

 835                 /*
 836                  * The page is left SE_SHARED locked so we can hold on to
 837                  * the page_t pointer.
 838                  */
 839                 curthread->t_red_pp = red_pp;

 841                 atomic_add_32(&red_nmapped, 1);
 842                 while (fp - (uintptr_t)curthread->t_stkbase < red_closest) {
 843                         (void) atomic_cas_32(&red_closest, red_closest,
 843                         (void) cas32(&red_closest, red_closest,
 844                             (uint32_t)(fp - (uintptr_t)curthread->t_stkbase));
 845                 }
 846                 return (1);
 847         }

 849         stkbase = (caddr_t)(((uintptr_t)curthread->t_stkbase &
 850             (uintptr_t)PAGEMASK) - PAGESIZE);

 852         atomic_add_32(&red_ndoubles, 1);

 854         if (fp - (uintptr_t)stkbase < RED_DEEP_THRESHOLD) {
 855                 /*
 856                  * Oh boy.  We're already deep within the mapped-in
 857                  * redzone page, and the caller is trying to prepare
 858                  * for a deep stack run.  We're running without a
 859                  * redzone right now:  if the caller plows off the
 860                  * end of the stack, it'll plow another thread or
 861                  * LWP structure.  That situation could result in
 862                  * a very hard-to-debug panic, so, in the spirit of
 863                  * recording the name of one's killer in one's own
 864                  * blood, we're going to record hrestime and the calling
 865                  * thread.
 866                  */
 867                 red_deep_hires = hrestime.tv_nsec;
 868                 red_deep_thread = curthread;
 869         }

 871         /*
 872          * If this is a DEBUG kernel, and we've run too deep for comfort, toss.
 873          */
 874         ASSERT(fp - (uintptr_t)stkbase >= RED_DEEP_THRESHOLD);
 875         return (0);
 876 #endif /* _LP64 */
 877 }
______unchanged_portion_omitted_
```

```
**********************************************************
  280601 Mon Jul 28 07:43:53 2014
new/usr/src/uts/common/vm/seg_vn.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

9519 /*
9520  * Bind text vnode segment to an amp. If we bind successfully mappings will be
9521  * established to per vnode mapping per lgroup amp pages instead of to vnode
9522  * pages. There's one amp per vnode text mapping per lgroup. Many processes
9523  * may share the same text replication amp. If a suitable amp doesn't already
9524  * exist in svntr hash table create a new one.  We may fail to bind to amp if
9525  * segment is not eligible for text replication.  Code below first checks for
9526  * these conditions. If binding is successful segment tr_state is set to on
9527  * and svd->amp points to the amp to use. Otherwise tr_state is set to off and
9528  * svd->amp remains as NULL.
9529  */
9530 static void
9531 segvn_textrepl(struct seg *seg)
9532 {
9533         struct segvn_data       *svd = (struct segvn_data *)seg->s_data;
9534         vnode_t                 *vp = svd->vp;
9535         u_offset_t              off = svd->offset;
9536         size_t                  size = seg->s_size;
9537         u_offset_t              eoff = off + size;
9538         uint_t                  szc = seg->s_szc;
9539         ulong_t                 hash = SVNTR_HASH_FUNC(vp);
9540         svntr_t                 *svntrp;
9541         struct vattr            va;
9542         proc_t                  *p = seg->s_as->a_proc;
9543         lgrp_id_t               lgrp_id;
9544         lgrp_id_t               olid;
9545         int                     first;
9546         struct anon_map         *amp;

9548         ASSERT(AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
9549         ASSERT(SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
9550         ASSERT(p != NULL);
9551         ASSERT(svd->tr_state == SEGVN_TR_INIT);
9552         ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
9553         ASSERT(svd->flags & MAP_TEXT);
9554         ASSERT(svd->type == MAP_PRIVATE);
9555         ASSERT(vp != NULL && svd->amp == NULL);
9556         ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
9557         ASSERT(!(svd->flags & MAP_NORESERVE) && svd->swresv == 0);
9558         ASSERT(seg->s_as != &kas);
9559         ASSERT(off < eoff);
9560         ASSERT(svntr_hashtab != NULL);

9562         /*
9563          * If numa optimizations are no longer desired bail out.
9564          */
9565         if (!lgrp_optimizations()) {
9566                 svd->tr_state = SEGVN_TR_OFF;
9567                 return;
9568         }

9570         /*
9571          * Avoid creating anon maps with size bigger than the file size.
9572          * If VOP_GETATTR() call fails bail out.
9573          */
9574         va.va_mask = AT_SIZE | AT_MTIME | AT_CTIME;
9575         if (VOP_GETATTR(vp, &va, 0, svd->cred, NULL) != 0) {
9576                 svd->tr_state = SEGVN_TR_OFF;
9577                 SEGVN_TR_ADDSTAT(gaerr);
```



```
9578                 return;
9579         }
9580         if (btopr(va.va_size) < btopr(eoff)) {
9581                 svd->tr_state = SEGVN_TR_OFF;
9582                 SEGVN_TR_ADDSTAT(overmap);
9583                 return;
9584         }

9586         /*
9587          * VVMEXEC may not be set yet if exec() prefaults text segment. Set
9588          * this flag now before vn_is_mapped(V_WRITE) so that MAP_SHARED
9589          * mapping that checks if trcache for this vnode needs to be
9590          * invalidated can't miss us.
9591          */
9592         if (!(vp->v_flag & VVMEXEC)) {
9593                 mutex_enter(&vp->v_lock);
9594                 vp->v_flag |= VVMEXEC;
9595                 mutex_exit(&vp->v_lock);
9596         }
9597         mutex_enter(&svntr_hashtab[hash].tr_lock);
9598         /*
9599          * Bail out if potentially MAP_SHARED writable mappings exist to this
9600          * vnode.  We don't want to use old file contents from existing
9601          * replicas if this mapping was established after the original file
9602          * was changed.
9603          */
9604         if (vn_is_mapped(vp, V_WRITE)) {
9605                 mutex_exit(&svntr_hashtab[hash].tr_lock);
9606                 svd->tr_state = SEGVN_TR_OFF;
9607                 SEGVN_TR_ADDSTAT(wrcnt);
9608                 return;
9609         }
9610         svntrp = svntr_hashtab[hash].tr_head;
9611         for (; svntrp != NULL; svntrp = svntrp->tr_next) {
9612                 ASSERT(svntrp->tr_refcnt != 0);
9613                 if (svntrp->tr_vp != vp) {
9614                         continue;
9615                 }

9617                 /*
9618                  * Bail out if the file or its attributes were changed after
9619                  * this replication entry was created since we need to use the
9620                  * latest file contents. Note that mtime test alone is not
9621                  * sufficient because a user can explicitly change mtime via
9622                  * utimes(2) interfaces back to the old value after modifiying
9623                  * the file contents. To detect this case we also have to test
9624                  * ctime which among other things records the time of the last
9625                  * mtime change by utimes(2). ctime is not changed when the file
9626                  * is only read or executed so we expect that typically existing
9627                  * replication amp's can be used most of the time.
9628                  */
9629                 if (!svntrp->tr_valid ||
9630                     svntrp->tr_mtime.tv_sec != va.va_mtime.tv_sec ||
9631                     svntrp->tr_mtime.tv_nsec != va.va_mtime.tv_nsec ||
9632                     svntrp->tr_ctime.tv_sec != va.va_ctime.tv_sec ||
9633                     svntrp->tr_ctime.tv_nsec != va.va_ctime.tv_nsec) {
9634                         mutex_exit(&svntr_hashtab[hash].tr_lock);
9635                         svd->tr_state = SEGVN_TR_OFF;
9636                         SEGVN_TR_ADDSTAT(stale);
9637                         return;
9638                 }
9639                 /*
9640                  * if off, eoff and szc match current segment we found the
9641                  * existing entry we can use.
9642                  */
9643                 if (svntrp->tr_off == off && svntrp->tr_eoff == eoff &&
```

```
9644                    svntrp->tr_szc == szc) {
9645                        break;
9646                }
9647                /*
9648                 * Don't create different but overlapping in file offsets
9649                 * entries to avoid replication of the same file pages more
9650                 * than once per lgroup.
9651                 */
9652                if ((off >= svntrp->tr_off && off < svntrp->tr_eoff) ||
9653                    (eoff > svntrp->tr_off && eoff <= svntrp->tr_eoff)) {
9654                        mutex_exit(&svntr_hashtab[hash].tr_lock);
9655                        svd->tr_state = SEGVN_TR_OFF;
9656                        SEGVN_TR_ADDSTAT(overlap);
9657                        return;
9658                }
9659        }
9660        /*
9661         * If we didn't find existing entry create a new one.
9662         */
9663        if (svntrp == NULL) {
9664                svntrp = kmem_cache_alloc(svntr_cache, KM_NOSLEEP);
9665                if (svntrp == NULL) {
9666                        mutex_exit(&svntr_hashtab[hash].tr_lock);
9667                        svd->tr_state = SEGVN_TR_OFF;
9668                        SEGVN_TR_ADDSTAT(nokmem);
9669                        return;
9670                }
9671 #ifdef DEBUG
9672                {
9673                        lgrp_id_t i;
9674                        for (i = 0; i < NLGRPS_MAX; i++) {
9675                                ASSERT(svntrp->tr_amp[i] == NULL);
9676                        }
9677                }
9678 #endif /* DEBUG */
9679                svntrp->tr_vp = vp;
9680                svntrp->tr_off = off;
9681                svntrp->tr_eoff = eoff;
9682                svntrp->tr_szc = szc;
9683                svntrp->tr_valid = 1;
9684                svntrp->tr_mtime = va.va_mtime;
9685                svntrp->tr_ctime = va.va_ctime;
9686                svntrp->tr_refcnt = 0;
9687                svntrp->tr_next = svntr_hashtab[hash].tr_head;
9688                svntr_hashtab[hash].tr_head = svntrp;
9689        }
9690        first = 1;
9691 again:
9692        /*
9693         * We want to pick a replica with pages on main thread's (t_tid = 1,
9694         * aka T1) lgrp. Currently text replication is only optimized for
9695         * workloads that either have all threads of a process on the same
9696         * lgrp or execute their large text primarily on main thread.
9697         */
9698        lgrp_id = p->p_t1_lgrpid;
9699        if (lgrp_id == LGRP_NONE) {
9700                /*
9701                 * In case exec() prefaults text on non main thread use
9702                 * current thread lgrpid.  It will become main thread anyway
9703                 * soon.
9704                 */
9705                lgrp_id = lgrp_home_id(curthread);
9706        }
9707        /*
9708         * Set p_tr_lgrpid to lgrpid if it hasn't been set yet.  Otherwise
9709         * just set it to NLGRPS_MAX if it's different from current process T1
```



```
9710         * home lgrp.  p_tr_lgrpid is used to detect if process uses text
9711         * replication and T1 new home is different from lgrp used for text
9712         * replication. When this happens asyncronous segvn thread rechecks if
9713         * segments should change lgrps used for text replication.  If we fail
9714         * to set p_tr_lgrpid with atomic_cas_32 then set it to NLGRPS_MAX
9715         * without cas if it's not already NLGRPS_MAX and not equal lgrp_id
9716         * we want to use.  We don't need to use cas in this case because
9717         * another thread that races in between our non atomic check and set
9718         * may only change p_tr_lgrpid to NLGRPS_MAX at this point.
9714         * to set p_tr_lgrpid with cas32 then set it to NLGRPS_MAX without cas
9715         * if it's not already NLGRPS_MAX and not equal lgrp_id we want to
9716         * use.  We don't need to use cas in this case because another thread
9717         * that races in between our non atomic check and set may only change
9718         * p_tr_lgrpid to NLGRPS_MAX at this point.
9719         */
9720        ASSERT(lgrp_id != LGRP_NONE && lgrp_id < NLGRPS_MAX);
9721        olid = p->p_tr_lgrpid;
9722        if (lgrp_id != olid && olid != NLGRPS_MAX) {
9723                lgrp_id_t nlid = (olid == LGRP_NONE) ? lgrp_id : NLGRPS_MAX;
9724                if (atomic_cas_32((uint32_t *)&p->p_tr_lgrpid, olid, nlid) !=
9725                    olid) {
9724                if (cas32((uint32_t *)&p->p_tr_lgrpid, olid, nlid) != olid) {
9726                        olid = p->p_tr_lgrpid;
9727                        ASSERT(olid != LGRP_NONE);
9728                        if (olid != lgrp_id && olid != NLGRPS_MAX) {
9729                                p->p_tr_lgrpid = NLGRPS_MAX;
9730                        }
9731                }
9732                ASSERT(p->p_tr_lgrpid != LGRP_NONE);
9733                membar_producer();
9734                /*
9735                 * lgrp_move_thread() won't schedule async recheck after
9736                 * p->p_t1_lgrpid update unless p->p_tr_lgrpid is not
9737                 * LGRP_NONE. Recheck p_t1_lgrpid once now that p->p_tr_lgrpid
9738                 * is not LGRP_NONE.
9739                 */
9740                if (first && p->p_t1_lgrpid != LGRP_NONE &&
9741                    p->p_t1_lgrpid != lgrp_id) {
9742                        first = 0;
9743                        goto again;
9744                }
9745        }
9746        /*
9747         * If no amp was created yet for lgrp_id create a new one as long as
9748         * we have enough memory to afford it.
9749         */
9750        if ((amp = svntrp->tr_amp[lgrp_id]) == NULL) {
9751                size_t trmem = atomic_add_long_nv(&segvn_textrepl_bytes, size);
9752                if (trmem > segvn_textrepl_max_bytes) {
9753                        SEGVN_TR_ADDSTAT(normem);
9754                        goto fail;
9755                }
9756                if (anon_try_resv_zone(size, NULL) == 0) {
9757                        SEGVN_TR_ADDSTAT(noanon);
9758                        goto fail;
9759                }
9760                amp = anonmap_alloc(size, size, ANON_NOSLEEP);
9761                if (amp == NULL) {
9762                        anon_unresv_zone(size, NULL);
9763                        SEGVN_TR_ADDSTAT(nokmem);
9764                        goto fail;
9765                }
9766                ASSERT(amp->refcnt == 1);
9767                amp->a_szc = szc;
9768                svntrp->tr_amp[lgrp_id] = amp;
9769                SEGVN_TR_ADDSTAT(newamp);
```

```
9770         }
9771         svntrp->tr_refcnt++;
9772         ASSERT(svd->svn_trnext == NULL);
9773         ASSERT(svd->svn_trprev == NULL);
9774         svd->svn_trnext = svntrp->tr_svnhead;
9775         svd->svn_trprev = NULL;
9776         if (svntrp->tr_svnhead != NULL) {
9777                 svntrp->tr_svnhead->svn_trprev = svd;
9778         }
9779         svntrp->tr_svnhead = svd;
9780         ASSERT(amp->a_szc == szc && amp->size == size && amp->swresv == size);
9781         ASSERT(amp->refcnt >= 1);
9782         svd->amp = amp;
9783         svd->anon_index = 0;
9784         svd->tr_policy_info.mem_policy = LGRP_MEM_POLICY_NEXT_SEG;
9785         svd->tr_policy_info.mem_lgrpid = lgrp_id;
9786         svd->tr_state = SEGVN_TR_ON;
9787         mutex_exit(&svntr_hashtab[hash].tr_lock);
9788         SEGVN_TR_ADDSTAT(repl);
9789         return;
9790 fail:
9791         ASSERT(segvn_textrepl_bytes >= size);
9792         atomic_add_long(&segvn_textrepl_bytes, -size);
9793         ASSERT(svntrp != NULL);
9794         ASSERT(svntrp->tr_amp[lgrp_id] == NULL);
9795         if (svntrp->tr_refcnt == 0) {
9796                 ASSERT(svntrp == svntr_hashtab[hash].tr_head);
9797                 svntr_hashtab[hash].tr_head = svntrp->tr_next;
9798                 mutex_exit(&svntr_hashtab[hash].tr_lock);
9799                 kmem_cache_free(svntr_cache, svntrp);
9800         } else {
9801                 mutex_exit(&svntr_hashtab[hash].tr_lock);
9802         }
9803         svd->tr_state = SEGVN_TR_OFF;
9804 }
```

_____***unchanged_portion_omitted_***

****
56114 Mon Jul 28 07:43:53 2014
new/usr/src/uts/i86pc/os/intr.c
5042 stop using deprecated atomic functions
****
_____***unchanged_portion_omitted_***

```
1199 /*
1200  * An interrupt thread is ending a time slice, so compute the interval it
1201  * ran for and update the statistic for its PIL.
1202  */
1203 void
1204 cpu_intr_swtch_enter(kthread_id_t t)
1205 {
1206         uint64_t        interval;
1207         uint64_t        start;
1208         cpu_t           *cpu;

1210         ASSERT((t->t_flag & T_INTR_THREAD) != 0);
1211         ASSERT(t->t_pil > 0 && t->t_pil <= LOCK_LEVEL);

1213         /*
1214          * We could be here with a zero timestamp. This could happen if:
1215          * an interrupt thread which no longer has a pinned thread underneath
1216          * it (i.e. it blocked at some point in its past) has finished running
1217          * its handler. intr_thread() updated the interrupt statistic for its
1218          * PIL and zeroed its timestamp. Since there was no pinned thread to
1219          * return to, swtch() gets called and we end up here.
1220          *
1221          * Note that we use atomic ops below (atomic_cas_64 and
1222          * atomic_add_64), which we don't use in the functions above,
1223          * because we're not called with interrupts blocked, but the
1224          * epilog/prolog functions are.
1221          * Note that we use atomic ops below (cas64 and atomic_add_64), which
1222          * we don't use in the functions above, because we're not called
1223          * with interrupts blocked, but the epilog/prolog functions are.
1225          */
1226         if (t->t_intr_start) {
1227                 do {
1228                         start = t->t_intr_start;
1229                         interval = tsc_read() - start;
1230                 } while (atomic_cas_64(&t->t_intr_start, start, 0) != start);
1229                 } while (cas64(&t->t_intr_start, start, 0) != start);
1231                 cpu = CPU;
1232                 cpu->cpu_m.intrstat[t->t_pil][0] += interval;

1234                 atomic_add_64((uint64_t *)&cpu->cpu_intracct[cpu->cpu_mstate],
1235                     interval);
1236         } else
1237                 ASSERT(t->t_intr == NULL);
1238 }

1240 /*
1241  * An interrupt thread is returning from swtch(). Place a starting timestamp
1242  * in its thread structure.
1243  */
1244 void
1245 cpu_intr_swtch_exit(kthread_id_t t)
1246 {
1247         uint64_t ts;

1249         ASSERT((t->t_flag & T_INTR_THREAD) != 0);
1250         ASSERT(t->t_pil > 0 && t->t_pil <= LOCK_LEVEL);

1252         do {
1253                 ts = t->t_intr_start;
```



```
1254         } while (atomic_cas_64(&t->t_intr_start, ts, tsc_read()) != ts);
1253         } while (cas64(&t->t_intr_start, ts, tsc_read()) != ts);
1255 }
```
_____***unchanged_portion_omitted_***

```
**********************************************************
    7131 Mon Jul 28 07:43:54 2014
new/usr/src/uts/i86pc/os/memnode.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #include <sys/systm.h>
  27 #include <sys/sysmacros.h>
  28 #include <sys/bootconf.h>
  29 #include <sys/atomic.h>
  30 #include <sys/lgrp.h>
  31 #include <sys/memlist.h>
  32 #include <sys/memnode.h>
  33 #include <sys/platform_module.h>
  34 #include <vm/vm_dep.h>

  36 int     max_mem_nodes = 1;

  38 struct mem_node_conf mem_node_config[MAX_MEM_NODES];
  39 int mem_node_pfn_shift;
  40 /*
  41  * num_memnodes should be updated atomically and always >=
  42  * the number of bits in memnodes_mask or the algorithm may fail.
  43  */
  44 uint16_t num_memnodes;
  45 mnodeset_t memnodes_mask; /* assumes 8*(sizeof(mnodeset_t)) >= MAX_MEM_NODES */

  47 /*
  48  * If set, mem_node_physalign should be a power of two, and
  49  * should reflect the minimum address alignment of each node.
  50  */
  51 uint64_t mem_node_physalign;

  53 /*
  54  * Platform hooks we will need.
  55  */

  57 #pragma weak plat_build_mem_nodes
  58 #pragma weak plat_slice_add
  59 #pragma weak plat_slice_del

  61 /*
```



```
  62  * Adjust the memnode config after a DR operation.
  63  *
  64  * It is rather tricky to do these updates since we can't
  65  * protect the memnode structures with locks, so we must
  66  * be mindful of the order in which updates and reads to
  67  * these values can occur.
  68  */

  70 void
  71 mem_node_add_slice(pfn_t start, pfn_t end)
  72 {
  73         int mnode;
  74         mnodeset_t newmask, oldmask;

  76         /*
  77          * DR will pass us the first pfn that is allocatable.
  78          * We need to round down to get the real start of
  79          * the slice.
  80          */
  81         if (mem_node_physalign) {
  82                 start &= ~(btop(mem_node_physalign) - 1);
  83                 end = roundup(end, btop(mem_node_physalign)) - 1;
  84         }

  86         mnode = PFN_2_MEM_NODE(start);
  87         ASSERT(mnode >= 0 && mnode < max_mem_nodes);

  89         if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
  89         if (cas32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
  90                 /*
  91                  * Add slice to existing node.
  92                  */
  93                 if (start < mem_node_config[mnode].physbase)
  94                         mem_node_config[mnode].physbase = start;
  95                 if (end > mem_node_config[mnode].physmax)
  96                         mem_node_config[mnode].physmax = end;
  97         } else {
  98                 mem_node_config[mnode].physbase = start;
  99                 mem_node_config[mnode].physmax = end;
 100                 atomic_add_16(&num_memnodes, 1);
 101                 do {
 102                         oldmask = memnodes_mask;
 103                         newmask = memnodes_mask | (1ull << mnode);
 104                 } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) !=
 105                     oldmask);
 104                 } while (cas64(&memnodes_mask, oldmask, newmask) != oldmask);
 106         }

 108         /*
 109          * Inform the common lgrp framework about the new memory
 110          */
 111         lgrp_config(LGRP_CONFIG_MEM_ADD, mnode, MEM_NODE_2_LGRPHAND(mnode));
 112 }

 114 /*
 115  * Remove a PFN range from a memnode.  On some platforms,
 116  * the memnode will be created with physbase at the first
 117  * allocatable PFN, but later deleted with the MC slice
 118  * base address converted to a PFN, in which case we need
 119  * to assume physbase and up.
 120  */
 121 void
 122 mem_node_del_slice(pfn_t start, pfn_t end)
 123 {
 124         int mnode;
 125         pgcnt_t delta_pgcnt, node_size;
```

```
126         mnodeset_t omask, nmask;

128         if (mem_node_physalign) {
129                 start &= ~(btop(mem_node_physalign) - 1);
130                 end = roundup(end, btop(mem_node_physalign)) - 1;
131         }
132         mnode = PFN_2_MEM_NODE(start);

134         ASSERT(mnode >= 0 && mnode < max_mem_nodes);
135         ASSERT(mem_node_config[mnode].exists == 1);

137         delta_pgcnt = end - start;
138         node_size = mem_node_config[mnode].physmax -
139             mem_node_config[mnode].physbase;

141         if (node_size > delta_pgcnt) {
142                 /*
143                  * Subtract the slice from the memnode.
144                  */
145                 if (start <= mem_node_config[mnode].physbase)
146                         mem_node_config[mnode].physbase = end + 1;
147                 ASSERT(end <= mem_node_config[mnode].physmax);
148                 if (end == mem_node_config[mnode].physmax)
149                         mem_node_config[mnode].physmax = start - 1;
150         } else {
151                 /*
152                  * Let the common lgrp framework know this mnode is
153                  * leaving
154                  */
155                 lgrp_config(LGRP_CONFIG_MEM_DEL,
156                     mnode, MEM_NODE_2_LGRPHAND(mnode));

158                 /*
159                  * Delete the whole node.
160                  */
161                 ASSERT(MNODE_PGCNT(mnode) == 0);
162                 do {
163                         omask = memnodes_mask;
164                         nmask = omask & ~(1ull << mnode);
165                 } while (atomic_cas_64(&memnodes_mask, omask, nmask) != omask);
164                 } while (cas64(&memnodes_mask, omask, nmask) != omask);
166                 atomic_add_16(&num_memnodes, -1);
167                 mem_node_config[mnode].exists = 0;
168         }
169 }
```

_____***unchanged_portion_omitted_***

```
219 /*
220  * Allocate an unassigned memnode.
221  */
222 int
223 mem_node_alloc()
224 {
225         int mnode;
226         mnodeset_t newmask, oldmask;

228         /*
229          * Find an unused memnode.  Update it atomically to prevent
230          * a first time memnode creation race.
231          */
232         for (mnode = 0; mnode < max_mem_nodes; mnode++)
233                 if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists,
232                 if (cas32((uint32_t *)&mem_node_config[mnode].exists,
234                     0, 1) == 0)
235                         break;
```



```
237         if (mnode >= max_mem_nodes)
238                 panic("Out of free memnodes\n");

240         mem_node_config[mnode].physbase = (pfn_t)-1l;
241         mem_node_config[mnode].physmax = 0;
242         atomic_add_16(&num_memnodes, 1);
243         do {
244                 oldmask = memnodes_mask;
245                 newmask = memnodes_mask | (1ull << mnode);
246         } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) != oldmask);
245         } while (cas64(&memnodes_mask, oldmask, newmask) != oldmask);

248         return (mnode);
249 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   18850 Mon Jul 28 07:43:54 2014
new/usr/src/uts/i86pc/os/x_call.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */
  25 /*
  26  * Copyright (c) 2010, Intel Corporation.
  27  * All rights reserved.
  28  */

  30 #include <sys/types.h>
  31 #include <sys/param.h>
  32 #include <sys/t_lock.h>
  33 #include <sys/thread.h>
  34 #include <sys/cpuvar.h>
  35 #include <sys/x_call.h>
  36 #include <sys/xc_levels.h>
  37 #include <sys/cpu.h>
  38 #include <sys/psw.h>
  39 #include <sys/sunddi.h>
  40 #include <sys/debug.h>
  41 #include <sys/systm.h>
  42 #include <sys/archsystm.h>
  43 #include <sys/machsystm.h>
  44 #include <sys/mutex_impl.h>
  45 #include <sys/stack.h>
  46 #include <sys/promif.h>
  47 #include <sys/x86_archext.h>

  49 /*
  50  * Implementation for cross-processor calls via interprocessor interrupts
  51  *
  52  * This implementation uses a message passing architecture to allow multiple
  53  * concurrent cross calls to be in flight at any given time. We use the cmpxchg
  54  * instruction, aka atomic_cas_ptr(), to implement simple efficient work
  55  * queues for message passing between CPUs with almost no need for regular
  56  * locking.  See xc_extract() and xc_insert() below.
  54  * instruction, aka casptr(), to implement simple efficient work queues for
  55  * message passing between CPUs with almost no need for regular locking.
  56  * See xc_extract() and xc_insert() below.
  57  *
  58  * The general idea is that initiating a cross call means putting a message
```



```
  59  * on a target(s) CPU's work queue. Any synchronization is handled by passing
  60  * the message back and forth between initiator and target(s).
  61  *
  62  * Every CPU has xc_work_cnt, which indicates it has messages to process.
  63  * This value is incremented as message traffic is initiated and decremented
  64  * with every message that finishes all processing.
  65  *
  66  * The code needs no mfence or other membar_*() calls. The uses of
  67  * atomic_cas_ptr(), atomic_cas_32() and atomic_dec_32() for the message
  68  * passing are implemented with LOCK prefix instructions which are
  69  * equivalent to mfence.
  67  * casptr(), cas32() and atomic_dec_32() for the message passing are
  68  * implemented with LOCK prefix instructions which are equivalent to mfence.
  70  *
  71  * One interesting aspect of this implmentation is that it allows 2 or more
  72  * CPUs to initiate cross calls to intersecting sets of CPUs at the same time.
  73  * The cross call processing by the CPUs will happen in any order with only
  74  * a guarantee, for xc_call() and xc_sync(), that an initiator won't return
  75  * from cross calls before all slaves have invoked the function.
  76  *
  77  * The reason for this asynchronous approach is to allow for fast global
  78  * TLB shootdowns. If all CPUs, say N, tried to do a global TLB invalidation
  79  * on a different Virtual Address at the same time. The old code required
  80  * N squared IPIs. With this method, depending on timing, it could happen
  81  * with just N IPIs.
  82  */

  84 /*
  85  * The default is to not enable collecting counts of IPI information, since
  86  * the updating of shared cachelines could cause excess bus traffic.
  87  */
  88 uint_t xc_collect_enable = 0;
  89 uint64_t xc_total_cnt = 0;      /* total #IPIs sent for cross calls */
  90 uint64_t xc_multi_cnt = 0;      /* # times we piggy backed on another IPI */

  92 /*
  93  * Values for message states. Here are the normal transitions. A transition
  94  * of "->" happens in the slave cpu and "=>" happens in the master cpu as
  95  * the messages are passed back and forth.
  96  *
  97  * FREE => ASYNC ->                       DONE => FREE
  98  * FREE => CALL ->                        DONE => FREE
  99  * FREE => SYNC -> WAITING => RELEASED -> DONE => FREE
 100  *
 101  * The interesing one above is ASYNC. You might ask, why not go directly
 102  * to FREE, instead of DONE. If it did that, it might be possible to exhaust
 103  * the master's xc_free list if a master can generate ASYNC messages faster
 104  * then the slave can process them. That could be handled with more complicated
 105  * handling. However since nothing important uses ASYNC, I've not bothered.
 106  */
 107 #define XC_MSG_FREE     (0)     /* msg in xc_free queue */
 108 #define XC_MSG_ASYNC    (1)     /* msg in slave xc_msgbox */
 109 #define XC_MSG_CALL     (2)     /* msg in slave xc_msgbox */
 110 #define XC_MSG_SYNC     (3)     /* msg in slave xc_msgbox */
 111 #define XC_MSG_WAITING  (4)     /* msg in master xc_msgbox or xc_waiters */
 112 #define XC_MSG_RELEASED (5)     /* msg in slave xc_msgbox */
 113 #define XC_MSG_DONE     (6)     /* msg in master xc_msgbox */

 115 /*
 116  * We allow for one high priority message at a time to happen in the system.
 117  * This is used for panic, kmdb, etc., so no locking is done.
 118  */
 119 static volatile cpuset_t xc_priority_set_store;
 120 static volatile ulong_t *xc_priority_set = CPUSET2BV(xc_priority_set_store);
 121 static xc_data_t xc_priority_data;
```

```
123 /*
124  * Wrappers to avoid C compiler warnings due to volatile. The atomic bit
125  * operations don't accept volatile bit vectors - which is a bit silly.
126  */
127 #define XC_BT_SET(vector, b)    BT_ATOMIC_SET((ulong_t *)(vector), (b))
128 #define XC_BT_CLEAR(vector, b)  BT_ATOMIC_CLEAR((ulong_t *)(vector), (b))

130 /*
131  * Decrement a CPU's work count
132  */
133 static void
134 xc_decrement(struct machcpu *mcpu)
135 {
136         atomic_dec_32(&mcpu->xc_work_cnt);
137 }

139 /*
140  * Increment a CPU's work count and return the old value
141  */
142 static int
143 xc_increment(struct machcpu *mcpu)
144 {
145         int old;
146         do {
147                 old = mcpu->xc_work_cnt;
148         } while (atomic_cas_32(&mcpu->xc_work_cnt, old, old + 1) != old);
147         } while (cas32((uint32_t *)&mcpu->xc_work_cnt, old, old + 1) != old);
149         return (old);
150 }

152 /*
153  * Put a message into a queue. The insertion is atomic no matter
154  * how many different inserts/extracts to the same queue happen.
155  */
156 static void
157 xc_insert(void *queue, xc_msg_t *msg)
158 {
159         xc_msg_t *old_head;

161         /*
162          * FREE messages should only ever be getting inserted into
163          * the xc_master CPUs xc_free queue.
164          */
165         ASSERT(msg->xc_command != XC_MSG_FREE ||
166             cpu[msg->xc_master] == NULL || /* possible only during init */
167             queue == &cpu[msg->xc_master]->cpu_m.xc_free);

169         do {
170                 old_head = (xc_msg_t *)*(volatile xc_msg_t **)queue;
171                 msg->xc_next = old_head;
172         } while (atomic_cas_ptr(queue, old_head, msg) != old_head);
171         } while (casptr(queue, old_head, msg) != old_head);
173 }

175 /*
176  * Extract a message from a queue. The extraction is atomic only
177  * when just one thread does extractions from the queue.
178  * If the queue is empty, NULL is returned.
179  */
180 static xc_msg_t *
181 xc_extract(xc_msg_t **queue)
182 {
183         xc_msg_t *old_head;

185         do {
186                 old_head = (xc_msg_t *)*(volatile xc_msg_t **)queue;
```



```
187                 if (old_head == NULL)
188                         return (old_head);
189         } while (atomic_cas_ptr(queue, old_head, old_head->xc_next) !=
190             old_head);
188         } while (casptr(queue, old_head, old_head->xc_next) != old_head);
191         old_head->xc_next = NULL;
192         return (old_head);
193 }
```

*unchanged_portion_omitted_*

```
544 /*
545  * Push out a priority cross call.
546  */
547 static void
548 xc_priority_common(
549         xc_func_t func,
550         xc_arg_t arg1,
551         xc_arg_t arg2,
552         xc_arg_t arg3,
553         ulong_t *set)
554 {
555         int i;
556         int c;
557         struct cpu *cpup;

559         /*
560          * Wait briefly for any previous xc_priority to have finished.
561          */
562         for (c = 0; c < max_ncpus; ++c) {
563                 cpup = cpu[c];
564                 if (cpup == NULL || !(cpup->cpu_flags & CPU_READY))
565                         continue;

567                 /*
568                  * The value of 40000 here is from old kernel code. It
569                  * really should be changed to some time based value, since
570                  * under a hypervisor, there's no guarantee a remote CPU
571                  * is even scheduled.
572                  */
573                 for (i = 0; BT_TEST(xc_priority_set, c) && i < 40000; ++i)
574                         SMT_PAUSE();

576                 /*
577                  * Some CPU did not respond to a previous priority request. It's
578                  * probably deadlocked with interrupts blocked or some such
579                  * problem. We'll just erase the previous request - which was
580                  * most likely a kmdb_enter that has already expired - and plow
581                  * ahead.
582                  */
583                 if (BT_TEST(xc_priority_set, c)) {
584                         XC_BT_CLEAR(xc_priority_set, c);
585                         if (cpup->cpu_m.xc_work_cnt > 0)
586                                 xc_decrement(&cpup->cpu_m);
587                 }
588         }

590         /*
591          * fill in cross call data
592          */
593         xc_priority_data.xc_func = func;
594         xc_priority_data.xc_a1 = arg1;
595         xc_priority_data.xc_a2 = arg2;
596         xc_priority_data.xc_a3 = arg3;

598         /*
599          * Post messages to all CPUs involved that are CPU_READY
```

```
600          * We'll always IPI, plus bang on the xc_msgbox for i86_mwait()
601          */
602         for (c = 0; c < max_ncpus; ++c) {
603                 if (!BT_TEST(set, c))
604                         continue;
605                 cpup = cpu[c];
606                 if (cpup == NULL || !(cpup->cpu_flags & CPU_READY) ||
607                     cpup == CPU)
608                         continue;
609                 (void) xc_increment(&cpup->cpu_m);
610                 XC_BT_SET(xc_priority_set, c);
611                 send_dirint(c, XC_HI_PIL);
612                 for (i = 0; i < 10; ++i) {
613                         (void) atomic_cas_ptr(&cpup->cpu_m.xc_msgbox,
611                         (void) casptr(&cpup->cpu_m.xc_msgbox,
614                             cpup->cpu_m.xc_msgbox, cpup->cpu_m.xc_msgbox);
615                 }
616         }
617 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  106512 Mon Jul 28 07:43:54 2014
new/usr/src/uts/i86pc/vm/hat_i86.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 945 /*
 946  * On 32 bit PAE mode, PTE's are 64 bits, but ordinary atomic memory references
 947  * are 32 bit, so for safety we must use cas64() to install these.
 948  */
 949 #ifdef __i386
 950 static void
 951 reload_pae32(hat_t *hat, cpu_t *cpu)
 952 {
 953         x86pte_t *src;
 954         x86pte_t *dest;
 955         x86pte_t pte;
 956         int i;

 958         /*
 959          * Load the 4 entries of the level 2 page table into this
 960          * cpu's range of the vlp_page and point cr3 at them.
 961          */
 962         ASSERT(mmu.pae_hat);
 963         src = hat->hat_vlp_ptes;
 964         dest = vlp_page + (cpu->cpu_id + 1) * VLP_NUM_PTES;
 965         for (i = 0; i < VLP_NUM_PTES; ++i) {
 966                 for (;;) {
 967                         pte = dest[i];
 968                         if (pte == src[i])
 969                                 break;
 970                         if (atomic_cas_64(dest + i, pte, src[i]) != src[i])
 970                         cas64(dest + i, pte, src[i]) != src[i])
 971                                 break;
 972                 }
 973         }
 974 }
______unchanged_portion_omitted_

1988 #define TLB_CPU_HALTED  (01ul)
1989 #define TLB_INVAL_ALL   (02ul)
1990 #define CAS_TLB_INFO(cpu, old, new)     \
1991         atomic_cas_ulong((ulong_t *)&(cpu)->cpu_m.mcpu_tlb_info, (old), (new))
1991         caslong((ulong_t *)&(cpu)->cpu_m.mcpu_tlb_info, (old), (new))

1993 /*
1994  * Record that a CPU is going idle
1995  */
1996 void
1997 tlb_going_idle(void)
1998 {
1999         atomic_or_ulong((ulong_t *)&CPU->cpu_m.mcpu_tlb_info, TLB_CPU_HALTED);
1999         atomic_or_long((ulong_t *)&CPU->cpu_m.mcpu_tlb_info, TLB_CPU_HALTED);
2000 }
______unchanged_portion_omitted_
```

```
**********************************************************
    8970 Mon Jul 28 07:43:54 2014
new/usr/src/uts/i86pc/vm/hat_pte.h
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * 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 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _VM_HAT_PTE_H
#define _VM_HAT_PTE_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#ifdef __cplusplus
extern "C" {
#endif

#include <sys/types.h>
#include <sys/mach_mmu.h>

/*
 * macros to get/set/clear the PTE fields
 */
#define PTE_SET(p, f)   ((p) |= (f))
#define PTE_CLR(p, f)   ((p) &= ~(x86pte_t)(f))
#define PTE_GET(p, f)   ((p) & (f))

/*
 * Handy macro to check if a pagetable entry or pointer is valid
 */
#define PTE_ISVALID(p)          PTE_GET(p, PT_VALID)

/*
 * Does a PTE map a large page.
 */
#define PTE_IS_LGPG(p, l)       ((l) > 0 && PTE_GET((p), PT_PAGESIZE))

/*
 * does this PTE represent a page (not a pointer to another page table)?
 */
#define PTE_ISPAGE(p, l)        \
        (PTE_ISVALID(p) && ((l) == 0 || PTE_GET(p, PT_PAGESIZE)))

/*
```



```
 * Handy macro to check if 2 PTE's are the same - ignores REF/MOD bits.
 * On the 64 bit hypervisor we also have to ignore the high order
 * software bits and the global/user bit which are set/cleared
 * capriciously (by the hypervisor!)
 */
#if defined(__amd64) && defined(__xpv)
#define PT_IGNORE       ((0x7ffUl << 52) | PT_GLOBAL | PT_USER)
#else
#define PT_IGNORE       (0)
#endif
#define PTE_EQUIV(a, b)  (((a) | (PT_IGNORE | PT_REF | PT_MOD)) == \
        ((b) | (PT_IGNORE | PT_REF | PT_MOD)))

/*
 * Shorthand for converting a PTE to it's pfn.
 */
#define PTE2MFN(p, l)   \
        mmu_btop(PTE_GET((p), PTE_IS_LGPG((p), (l)) ? PT_PADDR_LGPG : PT_PADDR))
#ifdef __xpv
#define PTE2PFN(p, l) pte2pfn(p, l)
#else
#define PTE2PFN(p, l) PTE2MFN(p, l)
#endif

#define PT_NX           (0x8000000000000000ull)
#define PT_PADDR        (0x000ffffffffff000ull)
#define PT_PADDR_LGPG   (0x000fffffffffe000ull) /* phys addr for large pages */

/*
 * Macros to create a PTP or PTE from the pfn and level
 */
#ifdef __xpv

/*
 * we use the highest order bit in physical address pfns to mark foreign mfns
 */
#ifdef _LP64
#define PFN_IS_FOREIGN_MFN (1ul << 51)
#else
#define PFN_IS_FOREIGN_MFN (1ul << 31)
#endif

#define MAKEPTP(pfn, l) \
        (pa_to_ma(pfn_to_pa(pfn)) | mmu.ptp_bits[(l) + 1])
#define MAKEPTE(pfn, l) \
        ((pfn & PFN_IS_FOREIGN_MFN) ? \
        ((pfn_to_pa(pfn & ~PFN_IS_FOREIGN_MFN) | mmu.pte_bits[l]) | \
        PT_FOREIGN | PT_REF | PT_MOD) : \
        (pa_to_ma(pfn_to_pa(pfn)) | mmu.pte_bits[l]))
#else
#define MAKEPTP(pfn, l) \
        (pfn_to_pa(pfn) | mmu.ptp_bits[(l) + 1])
#define MAKEPTE(pfn, l) \
        (pfn_to_pa(pfn) | mmu.pte_bits[l])
#endif

/*
 * The idea of "level" refers to the level where the page table is used in the
 * the hardware address translation steps. The level values correspond to the
 * following names of tables used in AMD/Intel architecture documents:
 *
 *      AMD/INTEL name          Level #
 *      ----------------------  -------
 *      Page Map Level 4           3
 *      Page Directory Pointer     2
 *      Page Directory             1
```

```
 *      Page Table                0
 *
 * The numbering scheme is such that the values of 0 and 1 can correspond to
 * the pagesize codes used for MPSS support. For now the Maximum level at
 * which you can have a large page is a constant, that may change in
 * future processors.
 *
 * The type of "level_t" is signed so that it can be used like:
 *      level_t l;
 *      ...
 *      while (--l >= 0)
 *              ...
 */
#define MAX_NUM_LEVEL           4
#define MAX_PAGE_LEVEL          2
typedef int8_t level_t;
#define LEVEL_SHIFT(l)  (mmu.level_shift[l])
#define LEVEL_SIZE(l)   (mmu.level_size[l])
#define LEVEL_OFFSET(l) (mmu.level_offset[l])
#define LEVEL_MASK(l)   (mmu.level_mask[l])

/*
 * Macros to:
 * Check for a PFN above 4Gig and 64Gig for 32 bit PAE support
 */
#define PFN_4G          (4ull * (1024 * 1024 * 1024 / MMU_PAGESIZE))
#define PFN_64G         (64ull * (1024 * 1024 * 1024 / MMU_PAGESIZE))
#define PFN_ABOVE4G(pfn) ((pfn) >= PFN_4G)
#define PFN_ABOVE64G(pfn) ((pfn) >= PFN_64G)

/*
 * The CR3 register holds the physical address of the top level page table.
 */
#define MAKECR3(pfn)    mmu_ptob(pfn)

/*
 * HAT/MMU parameters that depend on kernel mode and/or processor type
 */
struct htable;
struct hat_mmu_info {
        x86pte_t pt_nx;         /* either 0 or PT_NX */
        x86pte_t pt_global;     /* either 0 or PT_GLOBAL */

        pfn_t highest_pfn;

        uint_t num_level;       /* number of page table levels in use */
        uint_t max_level;       /* just num_level - 1 */
        uint_t max_page_level;  /* maximum level at which we can map a page */
        uint_t umax_page_level; /* max user page map level */
        uint_t ptes_per_table;  /* # of entries in lower level page tables */
        uint_t top_level_count; /* # of entries in top most level page table */

        uint_t  hash_cnt;       /* cnt of entries in htable_hash_cache */
        uint_t  vlp_hash_cnt;   /* cnt of entries in vlp htable_hash_cache */

        uint_t pae_hat;         /* either 0 or 1 */

        uintptr_t hole_start;   /* start of VA hole (or -1 if none) */
        uintptr_t hole_end;     /* end of VA hole (or 0 if none) */

        struct htable **kmap_htables; /* htables for segmap + 32 bit heap */
        x86pte_t *kmap_ptes;    /* mapping of pagetables that map kmap */
        uintptr_t kmap_addr;    /* start addr of kmap */
        uintptr_t kmap_eaddr;   /* end addr of kmap */

        uint_t pte_size;        /* either 4 or 8 */
```



```
        uint_t pte_size_shift;  /* either 2 or 3 */
        x86pte_t ptp_bits[MAX_NUM_LEVEL];       /* bits set for interior PTP */
        x86pte_t pte_bits[MAX_NUM_LEVEL];       /* bits set for leaf PTE */

        /*
         * A range of VA used to window pages in the i86pc/vm code.
         * See PWIN_XXX macros.
         */
        caddr_t pwin_base;
        caddr_t pwin_pte_va;
        paddr_t pwin_pte_pa;

        /*
         * The following tables are equivalent to PAGEXXXXX at different levels
         * in the page table hierarchy.
         */
        uint_t level_shift[MAX_NUM_LEVEL];      /* PAGESHIFT for given level */
        uintptr_t level_size[MAX_NUM_LEVEL];    /* PAGESIZE for given level */
        uintptr_t level_offset[MAX_NUM_LEVEL];  /* PAGEOFFSET for given level */
        uintptr_t level_mask[MAX_NUM_LEVEL];    /* PAGEMASK for given level */
};


#if defined(_KERNEL)

/*
 * Macros to access the HAT's private page windows. They're used for
 * accessing pagetables, ppcopy() and page_zero().
 * The 1st two macros are used to get an index for the particular use.
 * The next three give you:
 * - the virtual address of the window
 * - the virtual address of the pte that maps the window
 * - the physical address of the pte that map the window
 */
#define PWIN_TABLE(cpuid)       ((cpuid) * 2)
#define PWIN_SRC(cpuid)         ((cpuid) * 2 + 1)       /* for x86pte_copy() */
#define PWIN_VA(x)              (mmu.pwin_base + ((x) << MMU_PAGESHIFT))
#define PWIN_PTE_VA(x)          (mmu.pwin_pte_va + ((x) << mmu.pte_size_shift))
#define PWIN_PTE_PA(x)          (mmu.pwin_pte_pa + ((x) << mmu.pte_size_shift))

/*
 * The concept of a VA hole exists in AMD64. This might need to be made
 * model specific eventually.
 *
 * In the 64 bit kernel PTE loads are atomic, but need atomic_cas_64 on 32
 * bit kernel.
 * In the 64 bit kernel PTE loads are atomic, but need cas64 on 32 bit kernel.
 */
#if defined(__amd64)

#ifdef lint
#define IN_VA_HOLE(va)  (__lintzero)
#else
#define IN_VA_HOLE(va)  (mmu.hole_start <= (va) && (va) < mmu.hole_end)
#endif

#define FMT_PTE "0x%lx"
#define GET_PTE(ptr)            (*(x86pte_t *)(ptr))
#define SET_PTE(ptr, pte)       (*(x86pte_t *)(ptr) = pte)
#define CAS_PTE(ptr, x, y)      atomic_cas_64(ptr, x, y)
#define CAS_PTE(ptr, x, y)      cas64(ptr, x, y)

#elif defined(__i386)

#define IN_VA_HOLE(va)  (__lintzero)
```

```
256 #define FMT_PTE "0x%llx"

258 /* on 32 bit kernels, 64 bit loads aren't atomic, use get_pte64() */
259 extern x86pte_t get_pte64(x86pte_t *ptr);
260 #define GET_PTE(ptr)    (mmu.pae_hat ? get_pte64(ptr) : *(x86pte32_t *)(ptr))
261 #define SET_PTE(ptr, pte)                                       \
262         ((mmu.pae_hat ? ((x86pte32_t *)(ptr))[1] = (pte >> 32) : 0),    \
263         *(x86pte32_t *)(ptr) = pte)
264 #define CAS_PTE(ptr, x, y)                      \
265         (mmu.pae_hat ? atomic_cas_64(ptr, x, y) :       \
266         atomic_cas_32((uint32_t *)(ptr), (uint32_t)(x), (uint32_t)(y)))
266         (mmu.pae_hat ? cas64(ptr, x, y) :       \
267         cas32((uint32_t *)(ptr), (uint32_t)(x), (uint32_t)(y)))

268 #endif  /* __i386 */

270 /*
271  * Return a pointer to the pte entry at the given index within a page table.
272  */
273 #define PT_INDEX_PTR(p, x) \
274         ((x86pte_t *)((uintptr_t)(p) + ((x) << mmu.pte_size_shift)))

276 /*
277  * Return the physical address of the pte entry at the given index within a
278  * page table.
279  */
280 #define PT_INDEX_PHYSADDR(p, x) \
281         ((paddr_t)(p) + ((x) << mmu.pte_size_shift))

283 /*
284  * From pfn to bytes, careful not to lose bits on PAE.
285  */
286 #define pfn_to_pa(pfn) (mmu_ptob((paddr_t)(pfn)))

288 #ifdef __xpv
289 extern pfn_t pte2pfn(x86pte_t, level_t);
290 #endif

292 extern struct hat_mmu_info mmu;

294 #endif  /* _KERNEL */


297 #ifdef  __cplusplus
298 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    1293 Mon Jul 28 07:43:54 2014
new/usr/src/uts/intel/asm/atomic.h
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (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 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _ASM_ATOMIC_H
#define _ASM_ATOMIC_H

#include <sys/ccompile.h>
#include <sys/types.h>

#ifdef  __cplusplus
extern "C" {
#endif

#if !defined(__lint) && defined(__GNUC__)

#if defined(__amd64)

extern __GNU_INLINE void
atomic_or_long(ulong_t *target, ulong_t bits)
{
        __asm__ __volatile__(
            "lock; orq %1, (%0)"
            : /* no output */
            : "r" (target), "r" (bits));
}

extern __GNU_INLINE void
atomic_and_long(ulong_t *target, ulong_t bits)
{
        __asm__ __volatile__(
            "lock; andq %1, (%0)"
            : /* no output */
            : "r" (target), "r" (bits));
}

#ifdef notdef
extern __GNU_INLINE uint64_t
cas64(uint64_t *target, uint64_t cmp,
```



```
        uint64_t newval)
{
        uint64_t retval;

        __asm__ __volatile__(
            "movq %2, %%rax; lock; cmpxchgq %3, (%1)"
            : "=a" (retval)
            : "r" (target), "r" (cmp), "r" (newval));
        return (retval);
}
#endif

#elif defined(__i386)

extern __GNU_INLINE void
atomic_or_long(ulong_t *target, ulong_t bits)
{
        __asm__ __volatile__(
            "lock; orl %1, (%0)"
            : /* no output */
            : "r" (target), "r" (bits));
}

extern __GNU_INLINE void
atomic_and_long(ulong_t *target, ulong_t bits)
{
        __asm__ __volatile__(
            "lock; andl %1, (%0)"
            : /* no output */
            : "r" (target), "r" (bits));
}

#else
#error  "port me"
#endif

#endif /* !__lint && __GNUC__ */

#ifdef __cplusplus
}
```

_____unchanged_portion_omitted_

```
**********************************************************
    2530 Mon Jul 28 07:43:55 2014
new/usr/src/uts/intel/sys/synch32.h
5042 stop using deprecated atomic functions
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
  24  * Use is subject to license terms.
  25  */

  27 #ifndef _SYS_SYNCH32_H
  28 #define _SYS_SYNCH32_H

  30 #pragma ident   "%Z%%M% %I%     %E% SMI"

  30 #ifdef  __cplusplus
  31 extern "C" {
  32 #endif

  34 /* special defines for LWP mutexes */
  35 #define mutex_flag              flags.flag1
  36 #define mutex_ceiling           flags.ceiling
  37 #define mutex_type              flags.mbcp_type_un.mtype_rcount.count_type1
  38 #define mutex_rcount            flags.mbcp_type_un.mtype_rcount.count_type2
  39 #define mutex_magic             flags.magic
  40 #define mutex_owner             data
  41 /* used to atomically operate on whole word via cas or swap instruction */
  42 #define mutex_lockword          lock.lock32.lockword
  43 /* this requires atomic_cas_64 */
  45 /* this requires cas64 */
  44 #define mutex_lockword64        lock.owner64
  45 /* these are bytes */
  46 #define mutex_lockw             lock.lock64.pad[7]
  47 #define mutex_waiters           lock.lock64.pad[6]
  48 #define mutex_spinners          lock.lock64.pad[5]

  50 /* process-shared lock owner pid */
  51 #define mutex_ownerpid          lock.lock32.ownerpid

  53 /* Max. recusrion count for recursive mutexes */
  54 #define RECURSION_MAX           255

  56 /* special defines for LWP condition variables */
  57 #define cond_type               flags.type
  58 #define cond_magic              flags.magic
```



```
  59 #define cond_clockid            flags.flag[1]
  60 #define cond_waiters_user       flags.flag[2]
  61 #define cond_waiters_kernel     flags.flag[3]

  63 /* special defines for LWP semaphores */
  64 #define sema_count              count
  65 #define sema_type               type
  66 #define sema_waiters            flags[7]

  68 /* special defines for LWP rwlocks */
  69 #define rwlock_readers          readers
  70 #define rwlock_type             type
  71 #define rwlock_magic            magic
  72 #define rwlock_owner            readercv.data
  73 #define rwlock_ownerpid         writercv.data

  75 #define URW_HAS_WAITERS         0x80000000
  76 #define URW_WRITE_LOCKED        0x40000000
  77 #define URW_READERS_MASK        0x3fffffff

  79 #ifdef  __cplusplus
  80 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
  422765 Mon Jul 28 07:43:55 2014
new/usr/src/uts/sfmmu/vm/hat_sfmmu.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_
3612 #endif  /* VAC */

3614 /*
3615  * creates a large page shadow hmeblk for a tte.
3616  * The purpose of this routine is to allow us to do quick unloads because
3617  * the vm layer can easily pass a very large but sparsely populated range.
3618  */
3619 static struct hme_blk *
3620 sfmmu_shadow_hcreate(sfmmu_t *sfmmup, caddr_t vaddr, int ttesz, uint_t flags)
3621 {
3622         struct hmehash_bucket *hmebp;
3623         hmeblk_tag hblktag;
3624         int hmeshift, size, vshift;
3625         uint_t shw_mask, newshw_mask;
3626         struct hme_blk *hmeblkp;

3628         ASSERT(sfmmup != KHATID);
3629         if (mmu_page_sizes == max_mmu_page_sizes) {
3630                 ASSERT(ttesz < TTE256M);
3631         } else {
3632                 ASSERT(ttesz < TTE4M);
3633                 ASSERT(sfmmup->sfmmu_ttecnt[TTE32M] == 0);
3634                 ASSERT(sfmmup->sfmmu_ttecnt[TTE256M] == 0);
3635         }

3637         if (ttesz == TTE8K) {
3638                 size = TTE512K;
3639         } else {
3640                 size = ++ttesz;
3641         }

3643         hblktag.htag_id = sfmmup;
3644         hmeshift = HME_HASH_SHIFT(size);
3645         hblktag.htag_bspage = HME_HASH_BSPAGE(vaddr, hmeshift);
3646         hblktag.htag_rehash = HME_HASH_REHASH(size);
3647         hblktag.htag_rid = SFMMU_INVALID_SHMERID;
3648         hmebp = HME_HASH_FUNCTION(sfmmup, vaddr, hmeshift);

3650         SFMMU_HASH_LOCK(hmebp);

3652         HME_HASH_FAST_SEARCH(hmebp, hblktag, hmeblkp);
3653         ASSERT(hmeblkp != (struct hme_blk *)hblk_reserve);
3654         if (hmeblkp == NULL) {
3655                 hmeblkp = sfmmu_hblk_alloc(sfmmup, vaddr, hmebp, size,
3656                     hblktag, flags, SFMMU_INVALID_SHMERID);
3657         }
3658         ASSERT(hmeblkp);
3659         if (!hmeblkp->hblk_shw_mask) {
3660                 /*
3661                  * if this is a unused hblk it was just allocated or could
3662                  * potentially be a previous large page hblk so we need to
3663                  * set the shadow bit.
3664                  */
3665                 ASSERT(!hmeblkp->hblk_vcnt && !hmeblkp->hblk_hmecnt);
3666                 hmeblkp->hblk_shw_bit = 1;
3667         } else if (hmeblkp->hblk_shw_bit == 0) {
3668                 panic("sfmmu_shadow_hcreate: shw bit not set in hmeblkp 0x%p",
3669                     (void *)hmeblkp);
3670         }
3671         ASSERT(hmeblkp->hblk_shw_bit == 1);
```



```
3672         ASSERT(!hmeblkp->hblk_shared);
3673         vshift = vaddr_to_vshift(hblktag, vaddr, size);
3674         ASSERT(vshift < 8);
3675         /*
3676          * Atomically set shw mask bit
3677          */
3678         do {
3679                 shw_mask = hmeblkp->hblk_shw_mask;
3680                 newshw_mask = shw_mask | (1 << vshift);
3681                 newshw_mask = atomic_cas_32(&hmeblkp->hblk_shw_mask, shw_mask,
3681                 newshw_mask = cas32(&hmeblkp->hblk_shw_mask, shw_mask,
3682                     newshw_mask);
3683         } while (newshw_mask != shw_mask);

3685         SFMMU_HASH_UNLOCK(hmebp);

3687         return (hmeblkp);
3688 }
______unchanged_portion_omitted_

11616 /*
11617  * This routine does real work to prepare a hblk to be "stolen" by
11618  * unloading the mappings, updating shadow counts ....
11619  * It returns 1 if the block is ready to be reused (stolen), or 0
11620  * means the block cannot be stolen yet- pageunload is still working
11621  * on this hblk.
11622  */
11623 static int
11624 sfmmu_steal_this_hblk(struct hmehash_bucket *hmebp, struct hme_blk *hmeblkp,
11625         uint64_t hblkpa, struct hme_blk *pr_hblk)
11626 {
11627         int shw_size, vshift;
11628         struct hme_blk *shw_hblkp;
11629         caddr_t vaddr;
11630         uint_t shw_mask, newshw_mask;
11631         struct hme_blk *list = NULL;

11633         ASSERT(SFMMU_HASH_LOCK_ISHELD(hmebp));

11635         /*
11636          * check if the hmeblk is free, unload if necessary
11637          */
11638         if (hmeblkp->hblk_vcnt || hmeblkp->hblk_hmecnt) {
11639                 sfmmu_t *sfmmup;
11640                 demap_range_t dmr;

11642                 sfmmup = hblktosfmmu(hmeblkp);
11643                 if (hmeblkp->hblk_shared || sfmmup->sfmmu_ismhat) {
11644                         return (0);
11645                 }
11646                 DEMAP_RANGE_INIT(sfmmup, &dmr);
11647                 (void) sfmmu_hblk_unload(sfmmup, hmeblkp,
11648                     (caddr_t)get_hblk_base(hmeblkp),
11649                     get_hblk_endaddr(hmeblkp), &dmr, HAT_UNLOAD);
11650                 DEMAP_RANGE_FLUSH(&dmr);
11651                 if (hmeblkp->hblk_vcnt || hmeblkp->hblk_hmecnt) {
11652                         /*
11653                          * Pageunload is working on the same hblk.
11654                          */
11655                         return (0);
11656                 }

11658                 sfmmu_hblk_steal_unload_count++;
11659         }

11661         ASSERT(hmeblkp->hblk_lckcnt == 0);
```

```
11662         ASSERT(hmeblkp->hblk_vcnt == 0 && hmeblkp->hblk_hmecnt == 0);

11664         sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk, &list, 1);
11665         hmeblkp->hblk_nextpa = hblkpa;

11667         shw_hblkp = hmeblkp->hblk_shadow;
11668         if (shw_hblkp) {
11669                 ASSERT(!hmeblkp->hblk_shared);
11670                 shw_size = get_hblk_ttesz(shw_hblkp);
11671                 vaddr = (caddr_t)get_hblk_base(hmeblkp);
11672                 vshift = vaddr_to_vshift(shw_hblkp->hblk_tag, vaddr, shw_size);
11673                 ASSERT(vshift < 8);
11674                 /*
11675                  * Atomically clear shadow mask bit
11676                  */
11677                 do {
11678                         shw_mask = shw_hblkp->hblk_shw_mask;
11679                         ASSERT(shw_mask & (1 << vshift));
11680                         newshw_mask = shw_mask & ~(1 << vshift);
11681                         newshw_mask = atomic_cas_32(&shw_hblkp->hblk_shw_mask,
11681                         newshw_mask = cas32(&shw_hblkp->hblk_shw_mask,
11682                             shw_mask, newshw_mask);
11683                 } while (newshw_mask != shw_mask);
11684                 hmeblkp->hblk_shadow = NULL;
11685         }

11687         /*
11688          * remove shadow bit if we are stealing an unused shadow hmeblk.
11689          * sfmmu_hblk_alloc needs it that way, will set shadow bit later if
11690          * we are indeed allocating a shadow hmeblk.
11691          */
11692         hmeblkp->hblk_shw_bit = 0;

11694         if (hmeblkp->hblk_shared) {
11695                 sf_srd_t        *srdp;
11696                 sf_region_t     *rgnp;
11697                 uint_t          rid;

11699                 srdp = hblktosrd(hmeblkp);
11700                 ASSERT(srdp != NULL && srdp->srd_refcnt != 0);
11701                 rid = hmeblkp->hblk_tag.htag_rid;
11702                 ASSERT(SFMMU_IS_SHMERID_VALID(rid));
11703                 ASSERT(rid < SFMMU_MAX_HME_REGIONS);
11704                 rgnp = srdp->srd_hmergnp[rid];
11705                 ASSERT(rgnp != NULL);
11706                 SFMMU_VALIDATE_SHAREDHBLK(hmeblkp, srdp, rgnp, rid);
11707                 hmeblkp->hblk_shared = 0;
11708         }

11710         sfmmu_hblk_steal_count++;
11711         SFMMU_STAT(sf_steal_count);

11713         return (1);
11714 }
```

unchanged_portion_omitted_

```
15683 /*
15684  * This function is the first part of a 2 part process to remove an hmeblk
15685  * from the hash chain. In this phase we unlink the hmeblk from the hash chain
15686  * but leave the next physical pointer unchanged. The hmeblk is then linked onto
15687  * a per-cpu pending list using the virtual address pointer.
15688  *
15689  * TSB miss trap handlers that start after this phase will no longer see
15690  * this hmeblk. TSB miss handlers that still cache this hmeblk in a register
15691  * can still use it for further chain traversal because we haven't yet modifed
15692  * the next physical pointer or freed it.
```



```
15693  *
15694  * In the second phase of hmeblk removal we'll issue a barrier xcall before
15695  * we reuse or free this hmeblk. This will make sure all lingering references to
15696  * the hmeblk after first phase disappear before we finally reclaim it.
15697  * This scheme eliminates the need for TSB miss handlers to lock hmeblk chains
15698  * during their traversal.
15699  *
15700  * The hmehash_mutex must be held when calling this function.
15701  *
15702  * Input:
15703  *       hmebp - hme hash bucket pointer
15704  *       hmeblkp - address of hmeblk to be removed
15705  *       pr_hblk - virtual address of previous hmeblkp
15706  *       listp - pointer to list of hmeblks linked by virtual address
15707  *       free_now flag - indicates that a complete removal from the hash chains
15708  *                       is necessary.
15709  *
15710  * It is inefficient to use the free_now flag as a cross-call is required to
15711  * remove a single hmeblk from the hash chain but is necessary when hmeblks are
15712  * in short supply.
15713  */
15714 void
15715 sfmmu_hblk_hash_rm(struct hmehash_bucket *hmebp, struct hme_blk *hmeblkp,
15716     struct hme_blk *pr_hblk, struct hme_blk **listp,
15717     int free_now)
15718 {
15719         int shw_size, vshift;
15720         struct hme_blk *shw_hblkp;
15721         uint_t          shw_mask, newshw_mask;
15722         caddr_t         vaddr;
15723         int             size;
15724         cpuset_t cpuset = cpu_ready_set;

15726         ASSERT(SFMMU_HASH_LOCK_ISHELD(hmebp));

15728         if (hmebp->hmeblkp == hmeblkp) {
15729                 hmebp->hmeh_nextpa = hmeblkp->hblk_nextpa;
15730                 hmebp->hmeblkp = hmeblkp->hblk_next;
15731         } else {
15732                 pr_hblk->hblk_nextpa = hmeblkp->hblk_nextpa;
15733                 pr_hblk->hblk_next = hmeblkp->hblk_next;
15734         }

15736         size = get_hblk_ttesz(hmeblkp);
15737         shw_hblkp = hmeblkp->hblk_shadow;
15738         if (shw_hblkp) {
15739                 ASSERT(hblktosfmmu(hmeblkp) != KHATID);
15740                 ASSERT(!hmeblkp->hblk_shared);
15741 #ifdef  DEBUG
15742                 if (mmu_page_sizes == max_mmu_page_sizes) {
15743                         ASSERT(size < TTE256M);
15744                 } else {
15745                         ASSERT(size < TTE4M);
15746                 }
15747 #endif /* DEBUG */

15749                 shw_size = get_hblk_ttesz(shw_hblkp);
15750                 vaddr = (caddr_t)get_hblk_base(hmeblkp);
15751                 vshift = vaddr_to_vshift(shw_hblkp->hblk_tag, vaddr, shw_size);
15752                 ASSERT(vshift < 8);
15753                 /*
15754                  * Atomically clear shadow mask bit
15755                  */
15756                 do {
15757                         shw_mask = shw_hblkp->hblk_shw_mask;
15758                         ASSERT(shw_mask & (1 << vshift));
```

```
15759                         newshw_mask = shw_mask & ~(1 << vshift);
15760                         newshw_mask = atomic_cas_32(&shw_hblkp->hblk_shw_mask,
15760                         newshw_mask = cas32(&shw_hblkp->hblk_shw_mask,
15761                             shw_mask, newshw_mask);
15762                 } while (newshw_mask != shw_mask);
15763                 hmeblkp->hblk_shadow = NULL;
15764         }
15765         hmeblkp->hblk_shw_bit = 0;

15767         if (hmeblkp->hblk_shared) {
15768 #ifdef  DEBUG
15769                 sf_srd_t        *srdp;
15770                 sf_region_t     *rgnp;
15771                 uint_t          rid;

15773                 srdp = hblktosrd(hmeblkp);
15774                 ASSERT(srdp != NULL && srdp->srd_refcnt != 0);
15775                 rid = hmeblkp->hblk_tag.htag_rid;
15776                 ASSERT(SFMMU_IS_SHMERID_VALID(rid));
15777                 ASSERT(rid < SFMMU_MAX_HME_REGIONS);
15778                 rgnp = srdp->srd_hmergnp[rid];
15779                 ASSERT(rgnp != NULL);
15780                 SFMMU_VALIDATE_SHAREDHBLK(hmeblkp, srdp, rgnp, rid);
15781 #endif /* DEBUG */
15782                 hmeblkp->hblk_shared = 0;
15783         }
15784         if (free_now) {
15785                 kpreempt_disable();
15786                 CPUSET_DEL(cpuset, CPU->cpu_id);
15787                 xt_sync(cpuset);
15788                 xt_sync(cpuset);
15789                 kpreempt_enable();

15791                 hmeblkp->hblk_nextpa = HMEBLK_ENDPA;
15792                 hmeblkp->hblk_next = NULL;
15793         } else {
15794                 /* Append hmeblkp to listp for processing later. */
15795                 hmeblkp->hblk_next = *listp;
15796                 *listp = hmeblkp;
15797         }
15798 }
```

unchanged_portion_omitted_

```
**********************************************************
    2530 Mon Jul 28 07:43:55 2014
new/usr/src/uts/sparc/sys/synch32.h
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_SYNCH32_H
#define _SYS_SYNCH32_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#ifdef  __cplusplus
extern "C" {
#endif

/* special defines for LWP mutexes */
#define mutex_flag              flags.flag1
#define mutex_ceiling           flags.ceiling
#define mutex_type              flags.mbcp_type_un.mtype_rcount.count_type2
#define mutex_rcount            flags.mbcp_type_un.mtype_rcount.count_type1
#define mutex_magic             flags.magic
#define mutex_owner             data
/* used to atomically operate on whole word via cas or swap instruction */
#define mutex_lockword          lock.lock32.lockword
/* this requires atomic_cas_64 */
/* this requires cas64 */
#define mutex_lockword64        lock.owner64
/* these are bytes */
#define mutex_lockw             lock.lock64.pad[4]
#define mutex_waiters           lock.lock64.pad[7]
#define mutex_spinners          lock.lock64.pad[5]

/* process-shared lock owner pid */
#define mutex_ownerpid          lock.lock32.ownerpid

/* Max. recusrion count for recursive mutexes */
#define RECURSION_MAX           255

/* special defines for LWP condition variables */
#define cond_type               flags.type
#define cond_magic              flags.magic
```



```
#define cond_clockid            flags.flag[1]
#define cond_waiters_user       flags.flag[2]
#define cond_waiters_kernel     flags.flag[3]

/* special defines for LWP semaphores */
#define sema_count              count
#define sema_type               type
#define sema_waiters            flags[7]

/* special defines for LWP rwlocks */
#define rwlock_readers          readers
#define rwlock_type             type
#define rwlock_magic            magic
#define rwlock_owner            readercv.data
#define rwlock_ownerpid         writercv.data

#define URW_HAS_WAITERS         0x80000000
#define URW_WRITE_LOCKED        0x40000000
#define URW_READERS_MASK        0x3fffffff

#ifdef  __cplusplus
}
_____unchanged_portion_omitted_
```

```
**********************************************************
    6794 Mon Jul 28 07:43:56 2014
new/usr/src/uts/sparc/v9/syscall/install_utrap.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * 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 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/errno.h>
#include <sys/systm.h>
#include <sys/atomic.h>
#include <sys/kmem.h>
#include <sys/machpcb.h>
#include <sys/utrap.h>
#include <sys/model.h>

int
install_utrap(utrap_entry_t type, utrap_handler_t new_handler,
        utrap_handler_t *old_handlerp)
{
        struct proc *p = curthread->t_procp;
        utrap_handler_t *ov, *nv, *pv, *sv, *tmp;
        caddr32_t nv32;
        int idx;

        /*
         * Check trap number.
         */
        switch (type) {
        case UTRAP_V8P_FP_DISABLED:
#ifdef SF_ERRATA_30 /* call causes fp-disabled */
                {
                extern int spitfire_call_bug;

                if (spitfire_call_bug)
                        return ((int)set_errno(ENOSYS));
                }
#endif /* SF_ERRATA_30 */
                idx = UTRAP_V8P_FP_DISABLED;
                break;
        case UTRAP_V8P_MEM_ADDRESS_NOT_ALIGNED:
```



```
                idx = UTRAP_V8P_MEM_ADDRESS_NOT_ALIGNED;
                break;
        default:
                return ((int)set_errno(EINVAL));
        }
        if (get_udatamodel() == DATAMODEL_LP64)
                return ((int)set_errno(EINVAL));

        /*
         * Be sure handler address is word aligned.  The uintptr_t casts are
         * there to prevent warnings when using a certain compiler, and the
         * temporary 32 bit variable is intended to ensure proper code
         * generation and avoid a messy quadruple cast.
         */
        nv32 = (caddr32_t)(uintptr_t)new_handler;
        nv = (utrap_handler_t *)(uintptr_t)nv32;
        if (nv != UTRAP_UTH_NOCHANGE) {
                if (((uintptr_t)nv) & 0x3)
                        return ((int)set_errno(EINVAL));
        }
        /*
         * Allocate proc space for saving the addresses to these user
         * trap handlers, which must later be freed. Use atomic_cas_ptr to
         * trap handlers, which must later be freed. Use casptr to
         * do this atomically.
         */
        if (p->p_utraps == NULL) {
                pv = sv = kmem_zalloc((UT_PRECISE_MAXTRAPS+1) *
                    sizeof (utrap_handler_t *), KM_SLEEP);
                tmp = atomic_cas_ptr(&p->p_utraps, NULL, sv);
                tmp = casptr(&p->p_utraps, NULL, sv);
                if (tmp != NULL) {
                        kmem_free(pv, (UT_PRECISE_MAXTRAPS+1) *
                            sizeof (utrap_handler_t *));
                }
        }
        ASSERT(p->p_utraps != NULL);

        /*
         * Use atomic_cas_ptr to atomically install the handler.
         * Use casptr to atomically install the handler.
         */
        ov = p->p_utraps[idx];
        if (new_handler != (utrap_handler_t)UTRAP_UTH_NOCHANGE) {
                for (;;) {
                        tmp = atomic_cas_ptr(&p->p_utraps[idx], ov, nv);
                        tmp = casptr(&p->p_utraps[idx], ov, nv);
                        if (ov == tmp)
                                break;
                        ov = tmp;
                }
        }
        if (old_handlerp != NULL) {
                if (suword32(old_handlerp, (uint32_t)(uintptr_t)ov) == -1)
                        return ((int)set_errno(EINVAL));
        }
        return (0);
}
_____unchanged_portion_omitted_

/*
 * The code below supports the set of user traps which are required and
 * "must be provided by all ABI-conforming implementations", according to
 * 3.3.3 User Traps of the SPARC V9 ABI SUPPLEMENT, Delta Document 1.38.
 * There is only 1 deferred trap in Ultra I&II, the asynchronous error
 * traps, which are not required, so the deferred args are not used.
```

```
 */
/*ARGSUSED*/
int
sparc_utrap_install(utrap_entry_t type,
        utrap_handler_t new_precise, utrap_handler_t new_deferred,
        utrap_handler_t *old_precise, utrap_handler_t *old_deferred)
{
        struct proc *p = curthread->t_procp;
        utrap_handler_t *ov, *nvp, *pv, *sv, *tmp;
        int idx;

        /*
         * Check trap number.
         */
        switch (type) {
        case UT_ILLTRAP_INSTRUCTION:
                idx = UT_ILLTRAP_INSTRUCTION;
                break;
        case UT_FP_DISABLED:
#ifdef SF_ERRATA_30 /* call causes fp-disabled */
                {
                extern int spitfire_call_bug;

                if (spitfire_call_bug)
                        return ((int)set_errno(ENOSYS));
                }
#endif /* SF_ERRATA_30 */
                idx = UT_FP_DISABLED;
                break;
        case UT_FP_EXCEPTION_IEEE_754:
                idx = UT_FP_EXCEPTION_IEEE_754;
                break;
        case UT_TAG_OVERFLOW:
                idx = UT_TAG_OVERFLOW;
                break;
        case UT_DIVISION_BY_ZERO:
                idx = UT_DIVISION_BY_ZERO;
                break;
        case UT_MEM_ADDRESS_NOT_ALIGNED:
                idx = UT_MEM_ADDRESS_NOT_ALIGNED;
                break;
        case UT_PRIVILEGED_ACTION:
                idx = UT_PRIVILEGED_ACTION;
                break;
        case UT_TRAP_INSTRUCTION_16:
        case UT_TRAP_INSTRUCTION_17:
        case UT_TRAP_INSTRUCTION_18:
        case UT_TRAP_INSTRUCTION_19:
        case UT_TRAP_INSTRUCTION_20:
        case UT_TRAP_INSTRUCTION_21:
        case UT_TRAP_INSTRUCTION_22:
        case UT_TRAP_INSTRUCTION_23:
        case UT_TRAP_INSTRUCTION_24:
        case UT_TRAP_INSTRUCTION_25:
        case UT_TRAP_INSTRUCTION_26:
        case UT_TRAP_INSTRUCTION_27:
        case UT_TRAP_INSTRUCTION_28:
        case UT_TRAP_INSTRUCTION_29:
        case UT_TRAP_INSTRUCTION_30:
        case UT_TRAP_INSTRUCTION_31:
                idx = type;
                break;
        default:
                return ((int)set_errno(EINVAL));
        }
```



```
        if (get_udatamodel() == DATAMODEL_ILP32)
                return ((int)set_errno(EINVAL));

        /*
         * Be sure handler address is word aligned.
         * There are no deferred traps, so ignore them.
         */
        nvp = (utrap_handler_t *)new_precise;
        if (nvp != UTRAP_UTH_NOCHANGE) {
                if (((uintptr_t)nvp) & 0x3)
                        return ((int)set_errno(EINVAL));
        }

        /*
         * Allocate proc space for saving the addresses to these user
         * trap handlers, which must later be freed. Use atomic_cas_ptr to
         * trap handlers, which must later be freed. Use casptr to
         * do this atomically.
         */
        if (p->p_utraps == NULL) {
                pv = sv = kmem_zalloc((UT_PRECISE_MAXTRAPS+1) *
                    sizeof (utrap_handler_t *), KM_SLEEP);
                tmp = atomic_cas_ptr(&p->p_utraps, NULL, sv);
                tmp = casptr(&p->p_utraps, NULL, sv);
                if (tmp != NULL) {
                        kmem_free(pv, (UT_PRECISE_MAXTRAPS+1) *
                            sizeof (utrap_handler_t *));
                }
        }
        ASSERT(p->p_utraps != NULL);

        /*
         * Use atomic_cas_ptr to atomically install the handlers.
         * Use casptr to atomically install the handlers.
         */
        ov = p->p_utraps[idx];
        if (new_precise != (utrap_handler_t)UTH_NOCHANGE) {
                for (;;) {
                        tmp = atomic_cas_ptr(&p->p_utraps[idx], ov, nvp);
                        tmp = casptr(&p->p_utraps[idx], ov, nvp);
                        if (ov == tmp)
                                break;
                        ov = tmp;
                }
        }
        if (old_precise != NULL) {
                if (suword64(old_precise, (uint64_t)ov) == -1)
                        return ((int)set_errno(EINVAL));
        }
        return (0);
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   22487 Mon Jul 28 07:43:56 2014
new/usr/src/uts/sun4/os/machdep.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 523 /*
 524  * An interrupt thread is ending a time slice, so compute the interval it
 525  * ran for and update the statistic for its PIL.
 526  */
 527 void
 528 cpu_intr_swtch_enter(kthread_id_t t)
 529 {
 530         uint64_t        interval;
 531         uint64_t        start;
 532         cpu_t           *cpu;

 534         ASSERT((t->t_flag & T_INTR_THREAD) != 0);
 535         ASSERT(t->t_pil > 0 && t->t_pil <= LOCK_LEVEL);

 537         /*
 538          * We could be here with a zero timestamp. This could happen if:
 539          * an interrupt thread which no longer has a pinned thread underneath
 540          * it (i.e. it blocked at some point in its past) has finished running
 541          * its handler. intr_thread() updated the interrupt statistic for its
 542          * PIL and zeroed its timestamp. Since there was no pinned thread to
 543          * return to, swtch() gets called and we end up here.
 544          *
 545          * It can also happen if an interrupt thread in intr_thread() calls
 546          * preempt. It will have already taken care of updating stats. In
 547          * this event, the interrupt thread will be runnable.
 548          */
 549         if (t->t_intr_start) {
 550                 do {
 551                         start = t->t_intr_start;
 552                         interval = CLOCK_TICK_COUNTER() - start;
 553                 } while (atomic_cas_64(&t->t_intr_start, start, 0) != start);
 553                 } while (cas64(&t->t_intr_start, start, 0) != start);
 554                 cpu = CPU;
 555                 if (cpu->cpu_m.divisor > 1)
 556                         interval *= cpu->cpu_m.divisor;
 557                 cpu->cpu_m.intrstat[t->t_pil][0] += interval;

 559                 atomic_add_64((uint64_t *)&cpu->cpu_intracct[cpu->cpu_mstate],
 560                     interval);
 561         } else
 562                 ASSERT(t->t_intr == NULL || t->t_state == TS_RUN);
 563 }


 566 /*
 567  * An interrupt thread is returning from swtch(). Place a starting timestamp
 568  * in its thread structure.
 569  */
 570 void
 571 cpu_intr_swtch_exit(kthread_id_t t)
 572 {
 573         uint64_t ts;

 575         ASSERT((t->t_flag & T_INTR_THREAD) != 0);
 576         ASSERT(t->t_pil > 0 && t->t_pil <= LOCK_LEVEL);

 578         do {
 579                 ts = t->t_intr_start;
 580         } while (atomic_cas_64(&t->t_intr_start, ts, CLOCK_TICK_COUNTER()) !=
```



```
 581             ts);
 580         } while (cas64(&t->t_intr_start, ts, CLOCK_TICK_COUNTER()) != ts);
 582 }
______unchanged_portion_omitted_
```

```
**********************************************************
    8246 Mon Jul 28 07:43:56 2014
new/usr/src/uts/sun4/os/memnode.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #include <sys/systm.h>
  27 #include <sys/platform_module.h>
  28 #include <sys/sysmacros.h>
  29 #include <sys/atomic.h>
  30 #include <sys/memlist.h>
  31 #include <sys/memnode.h>
  32 #include <vm/vm_dep.h>

  34 int max_mem_nodes = 1;          /* max memory nodes on this system */

  36 struct mem_node_conf mem_node_config[MAX_MEM_NODES];
  37 int mem_node_pfn_shift;
  38 /*
  39  * num_memnodes should be updated atomically and always >=
  40  * the number of bits in memnodes_mask or the algorithm may fail.
  41  */
  42 uint16_t num_memnodes;
  43 mnodeset_t memnodes_mask; /* assumes 8*(sizeof(mnodeset_t)) >= MAX_MEM_NODES */

  45 /*
  46  * If set, mem_node_physalign should be a power of two, and
  47  * should reflect the minimum address alignment of each node.
  48  */
  49 uint64_t mem_node_physalign;

  51 /*
  52  * Platform hooks we will need.
  53  */

  55 #pragma weak plat_build_mem_nodes
  56 #pragma weak plat_slice_add
  57 #pragma weak plat_slice_del

  59 /*
  60  * Adjust the memnode config after a DR operation.
  61  *
```



```
  62  * It is rather tricky to do these updates since we can't
  63  * protect the memnode structures with locks, so we must
  64  * be mindful of the order in which updates and reads to
  65  * these values can occur.
  66  */
  67 void
  68 mem_node_add_slice(pfn_t start, pfn_t end)
  69 {
  70         int mnode;
  71         mnodeset_t newmask, oldmask;

  73         /*
  74          * DR will pass us the first pfn that is allocatable.
  75          * We need to round down to get the real start of
  76          * the slice.
  77          */
  78         if (mem_node_physalign) {
  79                 start &= ~(btop(mem_node_physalign) - 1);
  80                 end = roundup(end, btop(mem_node_physalign)) - 1;
  81         }

  83         mnode = PFN_2_MEM_NODE(start);
  84         ASSERT(mnode < max_mem_nodes);

  86         if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
  86         if (cas32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
  87                 /*
  88                  * Add slice to existing node.
  89                  */
  90                 if (start < mem_node_config[mnode].physbase)
  91                         mem_node_config[mnode].physbase = start;
  92                 if (end > mem_node_config[mnode].physmax)
  93                         mem_node_config[mnode].physmax = end;
  94         } else {
  95                 mem_node_config[mnode].physbase = start;
  96                 mem_node_config[mnode].physmax = end;
  97                 atomic_add_16(&num_memnodes, 1);
  98                 do {
  99                         oldmask = memnodes_mask;
 100                         newmask = memnodes_mask | (1ull << mnode);
 101                 } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) !=
 102                          oldmask);
 101                 } while (cas64(&memnodes_mask, oldmask, newmask) != oldmask);
 103         }
 104         /*
 105          * Let the common lgrp framework know about the new memory
 106          */
 107         lgrp_config(LGRP_CONFIG_MEM_ADD, mnode, MEM_NODE_2_LGRPHAND(mnode));
 108 }

 110 /*
 111  * Remove a PFN range from a memnode.  On some platforms,
 112  * the memnode will be created with physbase at the first
 113  * allocatable PFN, but later deleted with the MC slice
 114  * base address converted to a PFN, in which case we need
 115  * to assume physbase and up.
 116  */
 117 void
 118 mem_node_del_slice(pfn_t start, pfn_t end)
 119 {
 120         int mnode;
 121         pgcnt_t delta_pgcnt, node_size;
 122         mnodeset_t omask, nmask;

 124         if (mem_node_physalign) {
 125                 start &= ~(btop(mem_node_physalign) - 1);
```

```
126                 end = roundup(end, btop(mem_node_physalign)) - 1;
127         }
128         mnode = PFN_2_MEM_NODE(start);

130         ASSERT(mnode < max_mem_nodes);
131         ASSERT(mem_node_config[mnode].exists == 1);

133         delta_pgcnt = end - start;
134         node_size = mem_node_config[mnode].physmax -
135             mem_node_config[mnode].physbase;

137         if (node_size > delta_pgcnt) {
138                 /*
139                  * Subtract the slice from the memnode.
140                  */
141                 if (start <= mem_node_config[mnode].physbase)
142                         mem_node_config[mnode].physbase = end + 1;
143                 ASSERT(end <= mem_node_config[mnode].physmax);
144                 if (end == mem_node_config[mnode].physmax)
145                         mem_node_config[mnode].physmax = start - 1;
146         } else {

148                 /*
149                  * Let the common lgrp framework know the mnode is
150                  * leaving
151                  */
152                 lgrp_config(LGRP_CONFIG_MEM_DEL, mnode,
153                     MEM_NODE_2_LGRPHAND(mnode));

155                 /*
156                  * Delete the whole node.
157                  */
158                 ASSERT(MNODE_PGCNT(mnode) == 0);
159                 do {
160                         omask = memnodes_mask;
161                         nmask = omask & ~(1ull << mnode);
162                 } while (atomic_cas_64(&memnodes_mask, omask, nmask) != omask);
161                 } while (cas64(&memnodes_mask, omask, nmask) != omask);
163                 atomic_add_16(&num_memnodes, -1);
164                 mem_node_config[mnode].exists = 0;
165         }
166 }
```

_____*unchanged_portion_omitted_*

```
210 /*
211  * Allocate an unassigned memnode.
212  */
213 int
214 mem_node_alloc()
215 {
216         int mnode;
217         mnodeset_t newmask, oldmask;

219         /*
220          * Find an unused memnode.  Update it atomically to prevent
221          * a first time memnode creation race.
222          */
223         for (mnode = 0; mnode < max_mem_nodes; mnode++)
224                 if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists,
223                 if (cas32((uint32_t *)&mem_node_config[mnode].exists,
225                     0, 1) == 0)
226                         break;

228         if (mnode >= max_mem_nodes)
229                         panic("Out of free memnodes\n");
```



```
231         mem_node_config[mnode].physbase = (uint64_t)-1;
232         mem_node_config[mnode].physmax = 0;
233         atomic_add_16(&num_memnodes, 1);
234         do {
235                 oldmask = memnodes_mask;
236                 newmask = memnodes_mask | (1ull << mnode);
237         } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) != oldmask);
236         } while (cas64(&memnodes_mask, oldmask, newmask) != oldmask);

239         return (mnode);
240 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   16808 Mon Jul 28 07:43:56 2014
new/usr/src/uts/sun4/os/prom_subr.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #pragma ident   "%Z%%M% %I%     %E% SMI"

  26 #include <sys/types.h>
  27 #include <sys/param.h>
  28 #include <sys/cmn_err.h>
  29 #include <sys/mutex.h>
  30 #include <sys/systm.h>
  31 #include <sys/sysmacros.h>
  32 #include <sys/machsystm.h>
  33 #include <sys/archsystm.h>
  34 #include <sys/x_call.h>
  35 #include <sys/promif.h>
  36 #include <sys/prom_isa.h>
  37 #include <sys/privregs.h>
  38 #include <sys/vmem.h>
  39 #include <sys/atomic.h>
  40 #include <sys/panic.h>
  41 #include <sys/rwlock.h>
  42 #include <sys/reboot.h>
  43 #include <sys/kdi.h>
  44 #include <sys/kdi_machimpl.h>

  46 /*
  47  * We are called with a pointer to a cell-sized argument array.
  48  * The service name (the first element of the argument array) is
  49  * the name of the callback being invoked.  When called, we are
  50  * running on the firmwares trap table as a trusted subroutine
  51  * of the firmware.
  52  *
  53  * We define entry points to allow callback handlers to be dynamically
  54  * added and removed, to support obpsym, which is a separate module
  55  * and can be dynamically loaded and unloaded and registers its
  56  * callback handlers dynamically.
  57  *
  58  * Note: The actual callback handler we register, is the assembly lang.
  59  * glue, callback_handler, which takes care of switching from a 64
```



```
  60  * bit stack and environment to a 32 bit stack and environment, and
  61  * back again, if the callback handler returns. callback_handler calls
  62  * vx_handler to process the callback.
  63  */

  65 static kmutex_t vx_cmd_lock;    /* protect vx_cmd table */

  67 #define VX_CMD_MAX      10
  68 #define ENDADDR(a)      &a[sizeof (a) / sizeof (a[0])]
  69 #define vx_cmd_end      ((struct vx_cmd *)(ENDADDR(vx_cmd)))

  71 static struct vx_cmd {
  72         char    *service;       /* Service name */
  73         int     take_tba;       /* If Non-zero we take over the tba */
  74         void    (*func)(cell_t *argument_array);
  75 } vx_cmd[VX_CMD_MAX+1];
_____unchanged_portion_omitted_

 189 /*
 190  * PROM Locking Primitives
 191  *
 192  * These routines are called immediately before and immediately after calling
 193  * into the firmware.  The firmware is single-threaded and assumes that the
 194  * kernel will implement locking to prevent simultaneous service calls.  In
 195  * addition, some service calls (particularly character rendering) can be
 196  * slow, so we would like to sleep if we cannot acquire the lock to allow the
 197  * caller's CPU to continue to perform useful work in the interim.  Service
 198  * routines may also be called early in boot as part of slave CPU startup
 199  * when mutexes and cvs are not yet available (i.e. they are still running on
 200  * the prom's TLB handlers and cannot touch curthread).  Therefore, these
 201  * routines must reduce to a simple compare-and-swap spin lock when necessary.
 202  * Finally, kernel code may wish to acquire the firmware lock before executing
 203  * a block of code that includes service calls, so we also allow the firmware
 204  * lock to be acquired recursively by the owning CPU after disabling preemption.
 205  *
 206  * To meet these constraints, the lock itself is implemented as a compare-and-
 207  * swap spin lock on the global prom_cpu pointer.  We implement recursion by
 208  * atomically incrementing the integer prom_holdcnt after acquiring the lock.
 209  * If the current CPU is an "adult" (determined by testing cpu_m.mutex_ready),
 210  * we disable preemption before acquiring the lock and leave it disabled once
 211  * the lock is held.  The kern_postprom() routine then enables preemption if
 212  * we drop the lock and prom_holdcnt returns to zero.  If the current CPU is
 213  * an adult and the lock is held by another adult CPU, we can safely sleep
 214  * until the lock is released.  To do so, we acquire the adaptive prom_mutex
 215  * and then sleep on prom_cv.  Therefore, service routines must not be called
 216  * from above LOCK_LEVEL on any adult CPU.  Finally, if recursive entry is
 217  * attempted on an adult CPU, we must also verify that curthread matches the
 218  * saved prom_thread (the original owner) to ensure that low-level interrupt
 219  * threads do not step on other threads running on the same CPU.
 220  */

 222 static cpu_t *volatile prom_cpu;
 223 static kthread_t *volatile prom_thread;
 224 static uint32_t prom_holdcnt;
 225 static kmutex_t prom_mutex;
 226 static kcondvar_t prom_cv;

 228 /*
 229  * The debugger uses PROM services, and is thus unable to run if any of the
 230  * CPUs on the system are executing in the PROM at the time of debugger entry.
 231  * If a CPU is determined to be in the PROM when the debugger is entered,
 232  * prom_return_enter_debugger will be set, thus triggering a programmed debugger
 233  * entry when the given CPU returns from the PROM.  That CPU is then released by
 234  * the debugger, and is allowed to complete PROM-related work.
 235  */
 236 int prom_exit_enter_debugger;
```

```
238 void
239 kern_preprom(void)
240 {
241         for (;;) {
242                 /*
243                  * Load the current CPU pointer and examine the mutex_ready bit.
244                  * It doesn't matter if we are preempted here because we are
245                  * only trying to determine if we are in the *set* of mutex
246                  * ready CPUs.  We cannot disable preemption until we confirm
247                  * that we are running on a CPU in this set, since a call to
248                  * kpreempt_disable() requires access to curthread.
249                  */
250                 processorid_t cpuid = getprocessorid();
251                 cpu_t *cp = cpu[cpuid];
252                 cpu_t *prcp;

254                 if (panicstr)
255                         return; /* just return if we are currently panicking */

257                 if (CPU_IN_SET(cpu_ready_set, cpuid) && cp->cpu_m.mutex_ready) {
258                         /*
259                          * Disable premption, and reload the current CPU.  We
260                          * can't move from a mutex_ready cpu to a non-ready cpu
261                          * so we don't need to re-check cp->cpu_m.mutex_ready.
262                          */
263                         kpreempt_disable();
264                         cp = CPU;
265                         ASSERT(cp->cpu_m.mutex_ready);

267                         /*
268                          * Try the lock.  If we don't get the lock, re-enable
269                          * preemption and see if we should sleep.  If we are
270                          * already the lock holder, remove the effect of the
271                          * previous kpreempt_disable() before returning since
272                          * preemption was disabled by an earlier kern_preprom.
273                          */
274                         prcp = atomic_cas_ptr((void *)&prom_cpu, NULL, cp);
276                         prcp = casptr((void *)&prom_cpu, NULL, cp);
275                         if (prcp == NULL ||
276                             (prcp == cp && prom_thread == curthread)) {
277                                 if (prcp == cp)
278                                         kpreempt_enable();
279                                 break;
280                         }

282                         kpreempt_enable();

284                         /*
285                          * We have to be very careful here since both prom_cpu
286                          * and prcp->cpu_m.mutex_ready can be changed at any
287                          * time by a non mutex_ready cpu holding the lock.
288                          * If the owner is mutex_ready, holding prom_mutex
289                          * prevents kern_postprom() from completing.  If the
290                          * owner isn't mutex_ready, we only know it will clear
291                          * prom_cpu before changing cpu_m.mutex_ready, so we
292                          * issue a membar after checking mutex_ready and then
293                          * re-verify that prom_cpu is still held by the same
294                          * cpu before actually proceeding to cv_wait().
295                          */
296                         mutex_enter(&prom_mutex);
297                         prcp = prom_cpu;
298                         if (prcp != NULL && prcp->cpu_m.mutex_ready != 0) {
299                                 membar_consumer();
300                                 if (prcp == prom_cpu)
301                                         cv_wait(&prom_cv, &prom_mutex);
```



```
302                         }
303                         mutex_exit(&prom_mutex);

305                 } else {
306                         /*
307                          * If we are not yet mutex_ready, just attempt to grab
308                          * the lock.  If we get it or already hold it, break.
309                          */
310                         ASSERT(getpil() == PIL_MAX);
311                         prcp = atomic_cas_ptr((void *)&prom_cpu, NULL, cp);
313                         prcp = casptr((void *)&prom_cpu, NULL, cp);
312                         if (prcp == NULL || prcp == cp)
313                                 break;
314                 }
315         }

317         /*
318          * We now hold the prom_cpu lock.  Increment the hold count by one
319          * and assert our current state before returning to the caller.
320          */
321         atomic_add_32(&prom_holdcnt, 1);
322         ASSERT(prom_holdcnt >= 1);
323         prom_thread = curthread;
324 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   26474 Mon Jul 28 07:43:56 2014
new/usr/src/uts/sun4/vm/vm_dep.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 878 /*
 879  * To select our starting bin, we stride through the bins with a stride
 880  * of 337.  Why 337?  It's prime, it's largeish, and it performs well both
 881  * in simulation and practice for different workloads on varying cache sizes.
 882  */
 883 uint32_t color_start_current = 0;
 884 uint32_t color_start_stride = 337;
 885 int color_start_random = 0;

 887 /* ARGSUSED */
 888 uint_t
 889 get_color_start(struct as *as)
 890 {
 891         uint32_t old, new;

 893         if (consistent_coloring == 2 || color_start_random) {
 894                 return ((uint_t)(((gettick()) << (vac_shift - MMU_PAGESHIFT)) &
 895                     (hw_page_array[0].hp_colors - 1)));
 896         }

 898         do {
 899                 old = color_start_current;
 900                 new = old + (color_start_stride << (vac_shift - MMU_PAGESHIFT));
 901         } while (atomic_cas_32(&color_start_current, old, new) != old);
 901         } while (cas32(&color_start_current, old, new) != old);

 903         return ((uint_t)(new));
 904 }
______unchanged_portion_omitted_
```

```
**********************************************************
  129915 Mon Jul 28 07:43:56 2014
new/usr/src/uts/sun4u/cpu/spitfire.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

4278 /*
4279  * Legacy Correctable ECC Error Hash
4280  *
4281  * All of the code below this comment is used to implement a legacy array
4282  * which counted intermittent, persistent, and sticky CE errors by unum,
4283  * and then was later extended to publish the data as a kstat for SunVTS.
4284  * All of this code is replaced by FMA, and remains here until such time
4285  * that the UltraSPARC-I/II CPU code is converted to FMA, or is EOLed.
4286  *
4287  * Errors are saved in three buckets per-unum:
4288  * (1) sticky - scrub was unsuccessful, cannot be scrubbed
4289  *     This could represent a problem, and is immediately printed out.
4290  * (2) persistent - was successfully scrubbed
4291  *     These errors use the leaky bucket algorithm to determine
4292  *     if there is a serious problem.
4293  * (3) intermittent - may have originated from the cpu or upa/safari bus,
4294  *     and does not necessarily indicate any problem with the dimm itself,
4295  *     is critical information for debugging new hardware.
4296  *     Because we do not know if it came from the dimm, it would be
4297  *     inappropriate to include these in the leaky bucket counts.
4298  *
4299  * If the E$ line was modified before the scrub operation began, then the
4300  * displacement flush at the beginning of scrubphys() will cause the modified
4301  * line to be written out, which will clean up the CE.  Then, any subsequent
4302  * read will not cause an error, which will cause persistent errors to be
4303  * identified as intermittent.
4304  *
4305  * If a DIMM is going bad, it will produce true persistents as well as
4306  * false intermittents, so these intermittents can be safely ignored.
4307  *
4308  * If the error count is excessive for a DIMM, this function will return
4309  * PR_MCE, and the CPU module may then decide to remove that page from use.
4310  */
4311 static int
4312 ce_count_unum(int status, int len, char *unum)
4313 {
4314         int i;
4315         struct ce_info *psimm = mem_ce_simm;
4316         int page_status = PR_OK;

4318         ASSERT(psimm != NULL);

4320         if (len <= 0 ||
4321             (status & (ECC_STICKY | ECC_PERSISTENT | ECC_INTERMITTENT)) == 0)
4322                 return (page_status);

4324         /*
4325          * Initialize the leaky_bucket timeout
4326          */
4327         if (atomic_cas_ptr(&leaky_bucket_timeout_id,
4327         if (casptr(&leaky_bucket_timeout_id,
4328             TIMEOUT_NONE, TIMEOUT_SET) == TIMEOUT_NONE)
4329                 add_leaky_bucket_timeout();

4331         for (i = 0; i < mem_ce_simm_size; i++) {
4332                 if (psimm[i].name[0] == '\0') {
4333                         /*
4334                          * Hit the end of the valid entries, add
4335                          * a new one.
```



```
4336                          */
4337                         (void) strncpy(psimm[i].name, unum, len);
4338                         if (status & ECC_STICKY) {
4339                                 /*
4340                                  * Sticky - the leaky bucket is used to track
4341                                  * soft errors.  Since a sticky error is a
4342                                  * hard error and likely to be retired soon,
4343                                  * we do not count it in the leaky bucket.
4344                                  */
4345                                 psimm[i].leaky_bucket_cnt = 0;
4346                                 psimm[i].intermittent_total = 0;
4347                                 psimm[i].persistent_total = 0;
4348                                 psimm[i].sticky_total = 1;
4349                                 cmn_err(CE_NOTE,
4350                                     "[AFT0] Sticky Softerror encountered "
4351                                     "on Memory Module %s\n", unum);
4352                                 page_status = PR_MCE;
4353                         } else if (status & ECC_PERSISTENT) {
4354                                 psimm[i].leaky_bucket_cnt = 1;
4355                                 psimm[i].intermittent_total = 0;
4356                                 psimm[i].persistent_total = 1;
4357                                 psimm[i].sticky_total = 0;
4358                         } else {
4359                                 /*
4360                                  * Intermittent - Because the scrub operation
4361                                  * cannot find the error in the DIMM, we will
4362                                  * not count these in the leaky bucket
4363                                  */
4364                                 psimm[i].leaky_bucket_cnt = 0;
4365                                 psimm[i].intermittent_total = 1;
4366                                 psimm[i].persistent_total = 0;
4367                                 psimm[i].sticky_total = 0;
4368                         }
4369                         ecc_error_info_data.count.value.ui32++;
4370                         break;
4371                 } else if (strncmp(unum, psimm[i].name, len) == 0) {
4372                         /*
4373                          * Found an existing entry for the current
4374                          * memory module, adjust the counts.
4375                          */
4376                         if (status & ECC_STICKY) {
4377                                 psimm[i].sticky_total++;
4378                                 cmn_err(CE_NOTE,
4379                                     "[AFT0] Sticky Softerror encountered "
4380                                     "on Memory Module %s\n", unum);
4381                                 page_status = PR_MCE;
4382                         } else if (status & ECC_PERSISTENT) {
4383                                 int new_value;

4385                                 new_value = atomic_add_16_nv(
4386                                     &psimm[i].leaky_bucket_cnt, 1);
4387                                 psimm[i].persistent_total++;
4388                                 if (new_value > ecc_softerr_limit) {
4389                                         cmn_err(CE_NOTE, "[AFT0] Most recent %d"
4390                                             " soft errors from Memory Module"
4391                                             " %s exceed threshold (N=%d,"
4392                                             " T=%dh:%02dm) triggering page"
4393                                             " retire", new_value, unum,
4394                                             ecc_softerr_limit,
4395                                             ecc_softerr_interval / 60,
4396                                             ecc_softerr_interval % 60);
4397                                         atomic_add_16(
4398                                             &psimm[i].leaky_bucket_cnt, -1);
4399                                         page_status = PR_MCE;
4400                                 }
4401                         } else { /* Intermittent */
```

```
4402                                psimm[i].intermittent_total++;
4403                        }
4404                        break;
4405                }
4406        }

4408        if (i >= mem_ce_simm_size)
4409                cmn_err(CE_CONT, "[AFT0] Softerror: mem_ce_simm[] out of "
4410                    "space.\n");

4412        return (page_status);
4413 }
```

unchanged_portion_omitted_

```
**********************************************************
  210250 Mon Jul 28 07:43:57 2014
new/usr/src/uts/sun4u/cpu/us3_common.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 881 /*
 882  * Attempt to claim ownership, temporarily, of every cache line that a
 883  * non-responsive cpu might be using.  This might kick that cpu out of
 884  * this state.
 885  *
 886  * The return value indicates to the caller if we have exhausted all recovery
 887  * techniques. If 1 is returned, it is useless to call this function again
 888  * even for a different target CPU.
 889  */
 890 int
 891 mondo_recover(uint16_t cpuid, int bn)
 892 {
 893         struct memseg *seg;
 894         uint64_t begin_pa, end_pa, cur_pa;
 895         hrtime_t begin_hrt, end_hrt;
 896         int retval = 0;
 897         int pages_claimed = 0;
 898         cheetah_livelock_entry_t *histp;
 899         uint64_t idsr;

 901         if (atomic_cas_32(&sendmondo_in_recover, 0, 1) != 0) {
 901         if (cas32(&sendmondo_in_recover, 0, 1) != 0) {
 902                 /*
 903                  * Wait while recovery takes place
 904                  */
 905                 while (sendmondo_in_recover) {
 906                         drv_usecwait(1);
 907                 }
 908                 /*
 909                  * Assume we didn't claim the whole memory. If
 910                  * the target of this caller is not recovered,
 911                  * it will come back.
 912                  */
 913                 return (retval);
 914         }

 916         CHEETAH_LIVELOCK_ENTRY_NEXT(histp);
 917         CHEETAH_LIVELOCK_ENTRY_SET(histp, lbolt, LBOLT_WAITFREE);
 918         CHEETAH_LIVELOCK_ENTRY_SET(histp, cpuid, cpuid);
 919         CHEETAH_LIVELOCK_ENTRY_SET(histp, buddy, CPU->cpu_id);

 921         begin_hrt = gethrtime_waitfree();
 922         /*
 923          * First try to claim the lines in the TSB the target
 924          * may have been using.
 925          */
 926         if (mondo_recover_proc(cpuid, bn) == 1) {
 927                 /*
 928                  * Didn't claim the whole memory
 929                  */
 930                 goto done;
 931         }

 933         /*
 934          * We tried using the TSB. The target is still
 935          * not recovered. Check if complete memory scan is
 936          * enabled.
 937          */
 938         if (cheetah_sendmondo_fullscan == 0) {
```



```
 939                 /*
 940                  * Full memory scan is disabled.
 941                  */
 942                 retval = 1;
 943                 goto done;
 944         }

 946         /*
 947          * Try claiming the whole memory.
 948          */
 949         for (seg = memsegs; seg; seg = seg->next) {
 950                 begin_pa = (uint64_t)(seg->pages_base) << MMU_PAGESHIFT;
 951                 end_pa = (uint64_t)(seg->pages_end) << MMU_PAGESHIFT;
 952                 for (cur_pa = begin_pa; cur_pa < end_pa;
 953                     cur_pa += MMU_PAGESIZE) {
 954                         idsr = getidsr();
 955                         if ((idsr & (IDSR_NACK_BIT(bn) |
 956                             IDSR_BUSY_BIT(bn))) == 0) {
 957                                 /*
 958                                  * Didn't claim all memory
 959                                  */
 960                                 goto done;
 961                         }
 962                         claimlines(cur_pa, MMU_PAGESIZE,
 963                             CH_ECACHE_SUBBLK_SIZE);
 964                         if ((idsr & IDSR_BUSY_BIT(bn)) == 0) {
 965                                 shipit(cpuid, bn);
 966                         }
 967                         pages_claimed++;
 968                 }
 969         }

 971         /*
 972          * We did all we could.
 973          */
 974         retval = 1;

 976 done:
 977         /*
 978          * Update statistics
 979          */
 980         end_hrt = gethrtime_waitfree();
 981         CHEETAH_LIVELOCK_STAT(recovery);
 982         CHEETAH_LIVELOCK_MAXSTAT(hrt, (end_hrt - begin_hrt));
 983         CHEETAH_LIVELOCK_MAXSTAT(full_claimed, pages_claimed);
 984         CHEETAH_LIVELOCK_ENTRY_SET(histp, recovery_time, \
 985             (end_hrt -  begin_hrt));

 987         while (atomic_cas_32(&sendmondo_in_recover, 1, 0) != 1)
 987         while (cas32(&sendmondo_in_recover, 1, 0) != 1)
 988                 ;

 990         return (retval);
 991 }
______unchanged_portion_omitted_

6288 /*
6289  * Attempt a cpu logout for an error that we did not trap for, such
6290  * as a CE noticed with CEEN off.  It is assumed that we are still running
6291  * on the cpu that took the error and that we cannot migrate.  Returns
6292  * 0 on success, otherwise nonzero.
6293  */
6294 static int
6295 cpu_ce_delayed_ec_logout(uint64_t afar)
6296 {
6297         ch_cpu_logout_t *clop;
```

```
6299         if (CPU_PRIVATE(CPU) == NULL)
6300                 return (0);

6302         clop = CPU_PRIVATE_PTR(CPU, chpr_cecc_logout);
6303         if (atomic_cas_64(&clop->clo_data.chd_afar, LOGOUT_INVALID, afar) !=
6303         if (cas64(&clop->clo_data.chd_afar, LOGOUT_INVALID, afar) !=
6304             LOGOUT_INVALID)
6305                 return (0);

6307         cpu_delayed_logout(afar, clop);
6308         return (1);
6309 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   36934 Mon Jul 28 07:43:57 2014
new/usr/src/uts/sun4u/io/zuluvm.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #pragma ident   "%Z%%M% %I%     %E% SMI"

  26 /*
  27  * zuluvm module
  28  *
  29  * Provides services required by the XVR-4000 graphics accelerator (zulu)
  30  * that are not provided by the ddi. See PSARC 2002/231.
  31  *
  32  * Zulu has 2 dma engines with built in MMUs. zuluvm provides TLB miss
  33  * interrupt support obtaining virtual to physical address translations
  34  * using the XHAT interface PSARC/2003/517.
  35  *
  36  * The module has 3 components. This file, sun4u/vm/zulu_hat.c, and the
  37  * assembly language routines in sun4u/ml/zulu_asm.s and
  38  * sun4u/ml/zulu_hat_asm.s.
  39  *
  40  * The interrupt handler is a data bearing mondo interrupt handled at TL=1
  41  * If no translation is found in the zulu hat's tsb, or if the tsb is locked by
  42  * C code, the handler posts a soft interrupt which wakes up a parked
  43  * thread belonging to zuludaemon(1M).
  44  */

  46 #include <sys/conf.h>
  47 #include <sys/types.h>
  48 #include <sys/kmem.h>
  49 #include <sys/debug.h>
  50 #include <sys/modctl.h>
  51 #include <sys/autoconf.h>
  52 #include <sys/ddi_impldefs.h>
  53 #include <sys/ddi_subrdefs.h>
  54 #include <sys/intr.h>
  55 #include <sys/ddi.h>
  56 #include <sys/sunndi.h>
  57 #include <sys/proc.h>
  58 #include <sys/thread.h>
  59 #include <sys/machsystm.h>
```



```
  60 #include <sys/ivintr.h>
  61 #include <sys/tnf_probe.h>
  62 #include <sys/intreg.h>
  63 #include <sys/atomic.h>
  64 #include <vm/as.h>
  65 #include <vm/seg_enum.h>
  66 #include <vm/faultcode.h>
  67 #include <sys/dmv.h>
  68 #include <sys/zulumod.h>
  69 #include <sys/zulu_hat.h>

  71 #define ZULUVM_GET_PAGE(val) \
  72         (caddr_t)((uintptr_t)(val) & PAGEMASK)
  73 #define ZULUVM_GET_AS   curthread->t_procp->p_as

  75 #define ZULUVM_LOCK   mutex_enter(&(zdev->dev_lck))
  76 #define ZULUVM_UNLOCK mutex_exit(&(zdev->dev_lck))

  78 #define ZULUVM_SET_STATE(_z, b, c) \
  79         atomic_cas_32((uint32_t *)&((_z)->zvm.state), c, b)
  81         cas32((uint32_t *)&((_z)->zvm.state), c, b)
  80 #define ZULUVM_GET_STATE(_z) \
  81         (_z)->zvm.state
  82 #define ZULUVM_SET_IDLE(_z) \
  83         (_z)->zvm.state = ZULUVM_STATE_IDLE;

  85 #define ZULUVM_INO_MASK ((1<<INO_SIZE)-1)
  86 #define ZULUVM_IGN_MASK ((1<<IGN_SIZE)-1)
  87 #define ZULUVM_MONDO(_zdev, _n) \
  88         ((ZULUVM_IGN_MASK & _zdev->agentid) << INO_SIZE) | \
  89         (ZULUVM_INO_MASK & (_n))

  91 static void zuluvm_stop(zuluvm_state_t *, int, char *);
  92 static zuluvm_proc_t *zuluvm_find_proc(zuluvm_state_t *, struct as *);
  93 static int zuluvm_proc_release(zuluvm_state_t *zdev, zuluvm_proc_t *proc);
  94 static int zuluvm_get_intr_props(zuluvm_state_t *zdev, dev_info_t *devi);
  95 static int zuluvm_driver_attach(zuluvm_state_t *);
  96 static int zuluvm_driver_detach(zuluvm_state_t *);
  97 static void zuluvm_retarget_intr(void *arg);
  98 static void zuluvm_do_retarget(zuluvm_state_t *zdev);

 100 extern const unsigned int _mmu_pageshift;

 102 extern int zuluvm_base_pgsize;
 103 static int zuluvm_pagesizes[ZULUM_MAX_PG_SIZES + 1];

 105 int zuluvm_fast_tlb = 1;

 107 zuluvm_state_t *zuluvm_devtab[ZULUVM_MAX_DEV];
 108 kmutex_t zuluvm_lck;

 110 #ifdef DEBUG
 111 int zuluvm_debug_state = 0;
 112 #endif

 114 unsigned long zuluvm_ctx_locked = 0;

 116 /*
 117  * Module linkage information for the kernel.
 118  */
 119 extern struct mod_ops mod_miscops;

 121 static struct modlmisc modlmisc = {
 122         &mod_miscops,
 123         "sun4u support " ZULUVM_MOD_VERSION
 124 };
______unchanged_portion_omitted_
```

```
**********************************************************
   14501 Mon Jul 28 07:43:57 2014
new/usr/src/uts/sun4u/os/ppage.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/archsystm.h>
#include <sys/machsystm.h>
#include <sys/t_lock.h>
#include <sys/vmem.h>
#include <sys/mman.h>
#include <sys/vm.h>
#include <sys/cpu.h>
#include <sys/cmn_err.h>
#include <sys/cpuvar.h>
#include <sys/atomic.h>
#include <vm/as.h>
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/page.h>
#include <vm/seg.h>
#include <vm/seg_kmem.h>
#include <vm/seg_kpm.h>
#include <vm/hat_sfmmu.h>
#include <sys/debug.h>
#include <sys/cpu_module.h>
#include <sys/mem_cage.h>

/*
 * A quick way to generate a cache consistent address to map in a page.
 * users: ppcopy, pagezero, /proc, dev/mem
 *
 * The ppmapin/ppmapout routines provide a quick way of generating a cache
 * consistent address by reserving a given amount of kernel address space.
 * The base is PPMAPBASE and its size is PPMAPSIZE.  This memory is divided
 * into x number of sets, where x is the number of colors for the virtual
 * cache. The number of colors is how many times a page can be mapped
 * simulatenously in the cache.  For direct map caches this translates to
```



```
 * the number of pages in the cache.
 * Each set will be assigned a group of virtual pages from the reserved memory
 * depending on its virtual color.
 * When trying to assign a virtual address we will find out the color for the
 * physical page in question (if applicable).  Then we will try to find an
 * available virtual page from the set of the appropiate color.
 */

#define clsettoarray(color, set) ((color * nsets) + set)

int pp_slots = 4;               /* small default, tuned by cpu module */

/* tuned by cpu module, default is "safe" */
int pp_consistent_coloring = PPAGE_STORES_POLLUTE | PPAGE_LOADS_POLLUTE;

static caddr_t  ppmap_vaddrs[PPMAPSIZE / MMU_PAGESIZE];
static int      nsets;                  /* number of sets */
static int      ppmap_pages;            /* generate align mask */
static int      ppmap_shift;            /* set selector */

#ifdef PPDEBUG
#define         MAXCOLORS       16      /* for debug only */
static int      ppalloc_noslot = 0;     /* # of allocations from kernelmap */
static int      align_hits[MAXCOLORS];
static int      pp_allocs;              /* # of ppmapin requests */
#endif /* PPDEBUG */

/*
 * There are only 64 TLB entries on spitfire, 16 on cheetah
 * (fully-associative TLB) so we allow the cpu module to tune the
 * number to use here via pp_slots.
 */
static struct ppmap_va {
        caddr_t ppmap_slots[MAXPP_SLOTS];
} ppmap_va[NCPU];
_____unchanged_portion_omitted_

/*
 * Allocate a cache consistent virtual address to map a page, pp,
 * with protection, vprot; and map it in the MMU, using the most
 * efficient means possible.  The argument avoid is a virtual address
 * hint which when masked yields an offset into a virtual cache
 * that should be avoided when allocating an address to map in a
 * page.  An avoid arg of -1 means you don't care, for instance pagezero.
 *
 * machine dependent, depends on virtual address space layout,
 * understands that all kernel addresses have bit 31 set.
 *
 * NOTE: For sun4 platforms the meaning of the hint argument is opposite from
 * that found in other architectures.  In other architectures the hint
 * (called avoid) was used to ask ppmapin to NOT use the specified cache color.
 * This was used to avoid virtual cache trashing in the bcopy.  Unfortunately
 * in the case of a COW,  this later on caused a cache aliasing conflict.  In
 * sun4, the bcopy routine uses the block ld/st instructions so we don't have
 * to worry about virtual cache trashing.  Actually, by using the hint to choose
 * the right color we can almost guarantee a cache conflict will not occur.
 */

caddr_t
ppmapin(page_t *pp, uint_t vprot, caddr_t hint)
{
        int color, nset, index, start;
        caddr_t va;

#ifdef PPDEBUG
        pp_allocs++;
```

```
163 #endif /* PPDEBUG */
164         if (cache & CACHE_VAC) {
165                 color = sfmmu_get_ppvcolor(pp);
166                 if (color == -1) {
167                         if ((intptr_t)hint != -1L) {
168                                 color = addr_to_vcolor(hint);
169                         } else {
170                                 color = addr_to_vcolor(mmu_ptob(pp->p_pagenum));
171                         }
172                 }

174         } else {
175                 /*
176                  * For physical caches, we can pick any address we want.
177                  */
178                 color = 0;
179         }

181         start = color;
182         do {
183                 for (nset = 0; nset < nsets; nset++) {
184                         index = clsettoarray(color, nset);
185                         va = ppmap_vaddrs[index];
186                         if (va != NULL) {
187 #ifdef PPDEBUG
188                                 align_hits[color]++;
189 #endif /* PPDEBUG */
190                                 if (atomic_cas_ptr(&ppmap_vaddrs[index],
192                                 if (casptr(&ppmap_vaddrs[index],
191                                     va, NULL) == va) {
192                                         hat_memload(kas.a_hat, va, pp,
193                                             vprot | HAT_NOSYNC,
194                                             HAT_LOAD_LOCK);
195                                         return (va);
196                                 }
197                         }
198                 }
199                 /*
200                  * first pick didn't succeed, try another
201                  */
202                 if (++color == ppmap_pages)
203                         color = 0;
204         } while (color != start);

206 #ifdef PPDEBUG
207         ppalloc_noslot++;
208 #endif /* PPDEBUG */

210         /*
211          * No free slots; get a random one from the kernel heap area.
212          */
213         va = vmem_alloc(heap_arena, PAGESIZE, VM_SLEEP);

215         hat_memload(kas.a_hat, va, pp, vprot | HAT_NOSYNC, HAT_LOAD_LOCK);

217         return (va);

219 }
```

unchanged_portion_omitted_

```
251 #ifdef DEBUG
252 #define PP_STAT_ADD(stat)       (stat)++
253 uint_t pload, ploadfail;
254 uint_t ppzero, ppzero_short;
255 #else
256 #define PP_STAT_ADD(stat)
```



```
257 #endif /* DEBUG */

259 /*
260  * Find a slot in per CPU page copy area. Load up a locked TLB in the
261  * running cpu. We don't call hat layer to load up the tte since the
262  * mapping is only temporary. If the thread migrates it'll get a TLB
263  * miss trap and TLB/TSB miss handler will panic since there is no
264  * official hat record of this mapping.
265  */
266 static caddr_t
267 pp_load_tlb(processorid_t cpu, caddr_t **pslot, page_t *pp, uint_t prot)
268 {
269         struct ppmap_va *ppmap;
270         tte_t           tte;
271         caddr_t         *myslot;
272         caddr_t         va;
273         long            i, start, stride;
274         int             vcolor;
275         uint_t          flags, strict_flag;

277         PP_STAT_ADD(pload);

279         ppmap = &ppmap_va[cpu];
280         va = (caddr_t)(PPMAP_FAST_BASE + (MMU_PAGESIZE * MAXPP_SLOTS) * cpu);
281         myslot = ppmap->ppmap_slots;
282         ASSERT(addr_to_vcolor(va) == 0);

284         if (prot & TTE_HWWR_INT) {
285                 flags = PPAGE_STORE_VCOLORING | PPAGE_STORES_POLLUTE;
286                 strict_flag = PPAGE_STORES_POLLUTE;
287         } else {
288                 flags = PPAGE_LOAD_VCOLORING | PPAGE_LOADS_POLLUTE;
289                 strict_flag = PPAGE_LOADS_POLLUTE;
290         }

292         /*
293          * If consistent handling is required then keep the current
294          * vcolor of the page.  Furthermore, if loads or stores can
295          * pollute the VAC then using a "new" page (unassigned vcolor)
296          * won't work and we have to return a failure.
297          */
298         if (pp_consistent_coloring & flags) {
299                 vcolor = sfmmu_get_ppvcolor(pp);
300                 if ((vcolor == -1) &&
301                     (pp_consistent_coloring & strict_flag))
302                         return (NULL);
303                 /* else keep the current vcolor of the page */
304         } else {
305                 vcolor = -1;
306         }

308         if (vcolor != -1) {
309                 va += MMU_PAGESIZE * vcolor;
310                 start = vcolor;
311                 stride = ppmap_pages; /* number of colors */
312                 myslot += vcolor;
313         } else {
314                 start = 0;
315                 stride = 1;
316         }

318         for (i = start; i < pp_slots; i += stride) {
319                 if (*myslot == NULL) {
320                         if (atomic_cas_ptr(myslot, NULL, va) == NULL)
322                         if (casptr(myslot, NULL, va) == NULL)
321                                 break;
```

```
322                 }
323                 myslot += stride;
324                 va += MMU_PAGESIZE * stride;
325         }

327         if (i >= pp_slots) {
328                 PP_STAT_ADD(ploadfail);
329                 return (NULL);
330         }

332         ASSERT(vcolor == -1 || addr_to_vcolor(va) == vcolor);

334         /*
335          * Now we have a slot we can use, make the tte.
336          */
337         tte.tte_inthi = TTE_VALID_INT | TTE_PFN_INTHI(pp->p_pagenum);
338         tte.tte_intlo = TTE_PFN_INTLO(pp->p_pagenum) | TTE_CP_INT |
339             TTE_CV_INT | TTE_PRIV_INT | TTE_LCK_INT | prot;

341         ASSERT(CPU->cpu_id == cpu);
342         sfmmu_dtlb_ld_kva(va, &tte);

344         *pslot = myslot;        /* Return ptr to the slot we used. */

346         return (va);
347 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   36453 Mon Jul 28 07:43:58 2014
new/usr/src/uts/sun4u/vm/zulu_hat.c
5042 stop using deprecated atomic functions
**********************************************************
______unchanged_portion_omitted_

 255 /*
 256  * Lock the zulu tsb for a given zulu_hat.
 257  *
 258  * We're just protecting against the TLB trap handler here. Other operations
 259  * on the zulu_hat require entering the zhat's lock.
 260  */
 261 static void
 262 zulu_ctx_tsb_lock_enter(struct zulu_hat *zhat)
 263 {
 264         uint64_t        lck;
 265         uint64_t        *plck;

 267         ASSERT(mutex_owned(&zhat->lock));

 269         if (zhat->zulu_ctx < 0) {
 270                 return;
 271         }
 272         plck = (uint64_t *)&zulu_ctx_tab[zhat->zulu_ctx];

 274         for (; ; ) {
 275                 lck = *plck;
 276                 if (!(lck & ZULU_CTX_LOCK)) {
 277                         uint64_t old_lck, new_lck;

 279                         new_lck = lck | ZULU_CTX_LOCK;

 281                         old_lck = atomic_cas_64(plck, lck, new_lck);
 281                         old_lck = cas64(plck, lck, new_lck);

 283                         if (old_lck == lck) {
 284                                 /*
 285                                  * success
 286                                  */
 287                                 break;
 288                         }
 289                 }
 290         }
 291 }
______unchanged_portion_omitted_
```

```
**********************************************************
    9633 Mon Jul 28 07:43:58 2014
new/usr/src/uts/sun4v/os/ppage.c
5042 stop using deprecated atomic functions
**********************************************************
```

```
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/archsystm.h>
#include <sys/machsystm.h>
#include <sys/t_lock.h>
#include <sys/vmem.h>
#include <sys/mman.h>
#include <sys/vm.h>
#include <sys/cpu.h>
#include <sys/cmn_err.h>
#include <sys/cpuvar.h>
#include <sys/atomic.h>
#include <vm/as.h>
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/page.h>
#include <vm/seg.h>
#include <vm/seg_kmem.h>
#include <vm/seg_kpm.h>
#include <vm/hat_sfmmu.h>
#include <sys/debug.h>
#include <sys/cpu_module.h>

/*
 * A quick way to generate a cache consistent address to map in a page.
 * users: ppcopy, pagezero, /proc, dev/mem
 *
 * The ppmapin/ppmapout routines provide a quick way of generating a cache
 * consistent address by reserving a given amount of kernel address space.
 * The base is PPMAPBASE and its size is PPMAPSIZE.  This memory is divided
 * into x number of sets, where x is the number of colors for the virtual
 * cache. The number of colors is how many times a page can be mapped
 * simulatenously in the cache.  For direct map caches this translates to
 * the number of pages in the cache.
```



```
 * Each set will be assigned a group of virtual pages from the reserved memory
 * depending on its virtual color.
 * When trying to assign a virtual address we will find out the color for the
 * physical page in question (if applicable).  Then we will try to find an
 * available virtual page from the set of the appropiate color.
 */

int pp_slots = 4;               /* small default, tuned by cpu module */

/* tuned by cpu module, default is "safe" */
int pp_consistent_coloring = PPAGE_STORES_POLLUTE | PPAGE_LOADS_POLLUTE;

static caddr_t  ppmap_vaddrs[PPMAPSIZE / MMU_PAGESIZE];
static int      nsets;                  /* number of sets */
static int      ppmap_shift;            /* set selector */

#ifdef PPDEBUG
#define         MAXCOLORS       16      /* for debug only */
static int      ppalloc_noslot = 0;     /* # of allocations from kernelmap */
static int      align_hits;
static int      pp_allocs;              /* # of ppmapin requests */
#endif /* PPDEBUG */

/*
 * There are only 64 TLB entries on spitfire, 16 on cheetah
 * (fully-associative TLB) so we allow the cpu module to tune the
 * number to use here via pp_slots.
 */
static struct ppmap_va {
        caddr_t ppmap_slots[MAXPP_SLOTS];
} ppmap_va[NCPU];
_____unchanged_portion_omitted_

/*
 * Allocate a cache consistent virtual address to map a page, pp,
 * with protection, vprot; and map it in the MMU, using the most
 * efficient means possible.  The argument avoid is a virtual address
 * hint which when masked yields an offset into a virtual cache
 * that should be avoided when allocating an address to map in a
 * page.  An avoid arg of -1 means you don't care, for instance pagezero.
 *
 * machine dependent, depends on virtual address space layout,
 * understands that all kernel addresses have bit 31 set.
 *
 * NOTE: For sun4 platforms the meaning of the hint argument is opposite from
 * that found in other architectures.  In other architectures the hint
 * (called avoid) was used to ask ppmapin to NOT use the specified cache color.
 * This was used to avoid virtual cache trashing in the bcopy.  Unfortunately
 * in the case of a COW,  this later on caused a cache aliasing conflict.  In
 * sun4, the bcopy routine uses the block ld/st instructions so we don't have
 * to worry about virtual cache trashing.  Actually, by using the hint to choose
 * the right color we can almost guarantee a cache conflict will not occur.
 */

/*ARGSUSED2*/
caddr_t
ppmapin(page_t *pp, uint_t vprot, caddr_t hint)
{
        int nset;
        caddr_t va;

#ifdef PPDEBUG
        pp_allocs++;
#endif /* PPDEBUG */

        /*
```

```
153          * For sun4v caches are physical caches, we can pick any address
154          * we want.
155          */
156         for (nset = 0; nset < nsets; nset++) {
157                 va = ppmap_vaddrs[nset];
158                 if (va != NULL) {
159 #ifdef PPDEBUG
160                         align_hits++;
161 #endif /* PPDEBUG */
162                         if (atomic_cas_ptr(&ppmap_vaddrs[nset], va, NULL) ==
163                             va) {
164                         if (casptr(&ppmap_vaddrs[nset], va, NULL) == va) {
164                                 hat_memload(kas.a_hat, va, pp,
165                                     vprot | HAT_NOSYNC,
166                                     HAT_LOAD_LOCK);
167                                 return (va);
168                         }
169                 }
170         }

172 #ifdef PPDEBUG
173         ppalloc_noslot++;
174 #endif /* PPDEBUG */

176         /*
177          * No free slots; get a random one from the kernel heap area.
178          */
179         va = vmem_alloc(heap_arena, PAGESIZE, VM_SLEEP);

181         hat_memload(kas.a_hat, va, pp, vprot | HAT_NOSYNC, HAT_LOAD_LOCK);

183         return (va);

185 }
```

*unchanged_portion_omitted_*