```
**********************************************************
   34820 Fri Jan  3 22:11:52 2014
new/usr/src/uts/common/cpr/cpr_main.c
patch cpu-pause-func-deglobalize
**********************************************************
```

```
   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 /*
  27  * This module contains the guts of checkpoint-resume mechanism.
  28  * All code in this module is platform independent.
  29  */

  31 #include <sys/types.h>
  32 #include <sys/errno.h>
  33 #include <sys/callb.h>
  34 #include <sys/processor.h>
  35 #include <sys/machsystm.h>
  36 #include <sys/clock.h>
  37 #include <sys/vfs.h>
  38 #include <sys/kmem.h>
  39 #include <nfs/lm.h>
  40 #include <sys/systm.h>
  41 #include <sys/cpr.h>
  42 #include <sys/bootconf.h>
  43 #include <sys/cyclic.h>
  44 #include <sys/filio.h>
  45 #include <sys/fs/ufs_filio.h>
  46 #include <sys/epm.h>
  47 #include <sys/modctl.h>
  48 #include <sys/reboot.h>
  49 #include <sys/kdi.h>
  50 #include <sys/promif.h>
  51 #include <sys/srn.h>
  52 #include <sys/cpr_impl.h>

  54 #define PPM(dip) ((dev_info_t *)DEVI(dip)->devi_pm_ppm)

  56 extern struct cpr_terminator cpr_term;

  58 extern int cpr_alloc_statefile(int);
  59 extern void cpr_start_kernel_threads(void);
  60 extern void cpr_abbreviate_devpath(char *, char *);
  61 extern void cpr_convert_promtime(cpr_time_t *);
```



```
  62 extern void cpr_send_notice(void);
  63 extern void cpr_set_bitmap_size(void);
  64 extern void cpr_stat_init();
  65 extern void cpr_statef_close(void);
  66 extern void flush_windows(void);
  67 extern void (*srn_signal)(int, int);
  68 extern void init_cpu_syscall(struct cpu *);
  69 extern void i_cpr_pre_resume_cpus();
  70 extern void i_cpr_post_resume_cpus();
  71 extern int cpr_is_ufs(struct vfs *);

  73 extern int pm_powering_down;
  74 extern kmutex_t srn_clone_lock;
  75 extern int srn_inuse;

  77 static int cpr_suspend(int);
  78 static int cpr_resume(int);
  79 static void cpr_suspend_init(int);
  80 #if defined(__x86)
  81 static int cpr_suspend_cpus(void);
  82 static void cpr_resume_cpus(void);
  83 #endif
  84 static int cpr_all_online(void);
  85 static void cpr_restore_offline(void);

  87 cpr_time_t wholecycle_tv;
  88 int cpr_suspend_succeeded;
  89 pfn_t curthreadpfn;
  90 int curthreadremapped;

  92 extern cpuset_t cpu_ready_set;
  93 extern void *(*cpu_pause_func)(void *);

  94 extern processorid_t i_cpr_bootcpuid(void);
  95 extern cpu_t *i_cpr_bootcpu(void);
  96 extern void tsc_adjust_delta(hrtime_t tdelta);
  97 extern void tsc_resume(void);
  98 extern int tsc_resume_in_cyclic;

 100 /*
 101  * Set this variable to 1, to have device drivers resume in an
 102  * uniprocessor environment. This is to allow drivers that assume
 103  * that they resume on a UP machine to continue to work. Should be
 104  * deprecated once the broken drivers are fixed
 105  */
 106 int cpr_resume_uniproc = 0;

 108 /*
 109  * save or restore abort_enable;  this prevents a drop
 110  * to kadb or prom during cpr_resume_devices() when
 111  * there is no kbd present;  see abort_sequence_enter()
 112  */
 113 static void
 114 cpr_sae(int stash)
 115 {
 116         static int saved_ae = -1;

 118         if (stash) {
 119                 saved_ae = abort_enable;
 120                 abort_enable = 0;
 121         } else if (saved_ae != -1) {
 122                 abort_enable = saved_ae;
 123                 saved_ae = -1;
 124         }
 125 }
```

_____unchanged_portion_omitted_

```
384 int
385 cpr_suspend_cpus(void)
386 {
387         int     ret = 0;
388         extern void *i_cpr_save_context(void *arg);
389
390         mutex_enter(&cpu_lock);
391
392         /*
393          * the machine could not have booted without a bootcpu
394          */
395         ASSERT(i_cpr_bootcpu() != NULL);
396
397         /*
398          * bring all the offline cpus online
399          */
400         if ((ret = cpr_all_online())) {
401                 mutex_exit(&cpu_lock);
402                 return (ret);
403         }
404
405         /*
406          * Set the affinity to be the boot processor
407          * This is cleared in either cpr_resume_cpus() or cpr_unpause_cpus()
408          */
409         affinity_set(i_cpr_bootcpuid());
410
411         ASSERT(CPU->cpu_id == 0);
412
413         PMD(PMD_SX, ("curthread running on bootcpu\n"))
414
415         /*
416          * pause all other running CPUs and save the CPU state at the sametime
417          */
418         pause_cpus(NULL, i_cpr_save_context);
419         cpu_pause_func = i_cpr_save_context;
420         pause_cpus(NULL);
421
422         mutex_exit(&cpu_lock);
423
424         return (0);
425 }
_____unchanged_portion_omitted_

764 void
765 cpr_resume_cpus(void)
766 {
767         /*
768          * this is a cut down version of start_other_cpus()
769          * just do the initialization to wake the other cpus
770          */
771
772 #if defined(__x86)
773         /*
774          * Initialize our syscall handlers
775          */
776         init_cpu_syscall(CPU);
777
778 #endif
779
780         i_cpr_pre_resume_cpus();
781
782         /*
783          * Restart the paused cpus
784          */
```



```
785         mutex_enter(&cpu_lock);
786         start_cpus();
787         mutex_exit(&cpu_lock);
788
789         i_cpr_post_resume_cpus();
790
791         mutex_enter(&cpu_lock);
792         /*
793          * Restore this cpu to use the regular cpu_pause(), so that
794          * online and offline will work correctly
795          */
796         cpu_pause_func = NULL;
797
798         /*
799          * clear the affinity set in cpr_suspend_cpus()
800          */
801         affinity_clear();
802
803         /*
804          * offline all the cpus that were brought online during suspend
805          */
806         cpr_restore_offline();
807
808         mutex_exit(&cpu_lock);
809 }
810
811 void
812 cpr_unpause_cpus(void)
813 {
814         /*
815          * Now restore the system back to what it was before we suspended
816          */
817
818         PMD(PMD_SX, ("cpr_unpause_cpus: restoring system\n"))
819
820         mutex_enter(&cpu_lock);
821
822         /*
823          * Restore this cpu to use the regular cpu_pause(), so that
824          * online and offline will work correctly
825          */
826         cpu_pause_func = NULL;
827
828         /*
829          * Restart the paused cpus
830          */
831         start_cpus();
832
833         /*
834          * clear the affinity set in cpr_suspend_cpus()
835          */
836         affinity_clear();
837
838         /*
839          * offline all the cpus that were brought online during suspend
840          */
841         cpr_restore_offline();
842
843         mutex_exit(&cpu_lock);
844 }
845
846 /*
847  * Bring the system back up from a checkpoint, at this point
848  * the VM has been minimally restored by boot, the following
849  * are executed sequentially:
850  *
```

```
 838  *    - machdep setup and enable interrupts (mp startup if it's mp)
 839  *    - resume all devices
 840  *    - restart daemons
 841  *    - put all threads back on run queue
 842  */
 843 static int
 844 cpr_resume(int sleeptype)
 845 {
 846         cpr_time_t pwron_tv, *ctp;
 847         char *str;
 848         int rc = 0;

 850         /*
 851          * The following switch is used to resume the system
 852          * that was suspended to a different level.
 853          */
 854         CPR_DEBUG(CPR_DEBUG1, "\nEntering cpr_resume...\n");
 855         PMD(PMD_SX, ("cpr_resume %x\n", sleeptype))

 857         /*
 858          * Note:
 859          *
 860          * The rollback labels rb_xyz do not represent the cpr resume
 861          * state when event 'xyz' has happened. Instead they represent
 862          * the state during cpr suspend when event 'xyz' was being
 863          * entered (and where cpr suspend failed). The actual call that
 864          * failed may also need to be partially rolled back, since they
 865          * aren't atomic in most cases.  In other words, rb_xyz means
 866          * "roll back all cpr suspend events that happened before 'xyz',
 867          * and the one that caused the failure, if necessary."
 868          */
 869         switch (CPR->c_substate) {
 870 #if defined(__sparc)
 871         case C_ST_DUMP:
 872                 /*
 873                  * This is most likely a full-fledged cpr_resume after
 874                  * a complete and successful cpr suspend. Just roll back
 875                  * everything.
 876                  */
 877                 ASSERT(sleeptype == CPR_TODISK);
 878                 break;

 880         case C_ST_REUSABLE:
 881         case C_ST_DUMP_NOSPC:
 882         case C_ST_SETPROPS_0:
 883         case C_ST_SETPROPS_1:
 884                 /*
 885                  * C_ST_REUSABLE and C_ST_DUMP_NOSPC are the only two
 886                  * special switch cases here. The other two do not have
 887                  * any state change during cpr_suspend() that needs to
 888                  * be rolled back. But these are exit points from
 889                  * cpr_suspend, so theoretically (or in the future), it
 890                  * is possible that a need for roll back of a state
 891                  * change arises between these exit points.
 892                  */
 893                 ASSERT(sleeptype == CPR_TODISK);
 894                 goto rb_dump;
 895 #endif

 897         case C_ST_NODUMP:
 898                 PMD(PMD_SX, ("cpr_resume: NODUMP\n"))
 899                 goto rb_nodump;

 901         case C_ST_STOP_KERNEL_THREADS:
 902                 PMD(PMD_SX, ("cpr_resume: STOP_KERNEL_THREADS\n"))
 903                 goto rb_stop_kernel_threads;
```



```
 905         case C_ST_SUSPEND_DEVICES:
 906                 PMD(PMD_SX, ("cpr_resume: SUSPEND_DEVICES\n"))
 907                 goto rb_suspend_devices;

 909 #if defined(__sparc)
 910         case C_ST_STATEF_ALLOC:
 911                 ASSERT(sleeptype == CPR_TODISK);
 912                 goto rb_statef_alloc;

 914         case C_ST_DISABLE_UFS_LOGGING:
 915                 ASSERT(sleeptype == CPR_TODISK);
 916                 goto rb_disable_ufs_logging;
 917 #endif

 919         case C_ST_PM_REATTACH_NOINVOL:
 920                 PMD(PMD_SX, ("cpr_resume: REATTACH_NOINVOL\n"))
 921                 goto rb_pm_reattach_noinvol;

 923         case C_ST_STOP_USER_THREADS:
 924                 PMD(PMD_SX, ("cpr_resume: STOP_USER_THREADS\n"))
 925                 goto rb_stop_user_threads;

 927 #if defined(__sparc)
 928         case C_ST_MP_OFFLINE:
 929                 PMD(PMD_SX, ("cpr_resume: MP_OFFLINE\n"))
 930                 goto rb_mp_offline;
 931 #endif

 933 #if defined(__x86)
 934         case C_ST_MP_PAUSED:
 935                 PMD(PMD_SX, ("cpr_resume: MP_PAUSED\n"))
 936                 goto rb_mp_paused;
 937 #endif


 940         default:
 941                 PMD(PMD_SX, ("cpr_resume: others\n"))
 942                 goto rb_others;
 943         }

 945 rb_all:
 946         /*
 947          * perform platform-dependent initialization
 948          */
 949         if (cpr_suspend_succeeded)
 950                 i_cpr_machdep_setup();

 952         /*
 953          * system did not really go down if we jump here
 954          */
 955 rb_dump:
 956         /*
 957          * IMPORTANT:  SENSITIVE RESUME SEQUENCE
 958          *
 959          * DO NOT ADD ANY INITIALIZATION STEP BEFORE THIS POINT!!
 960          */
 961 rb_nodump:
 962         /*
 963          * If we did suspend to RAM, we didn't generate a dump
 964          */
 965         PMD(PMD_SX, ("cpr_resume: CPR DMA callback\n"))
 966         (void) callb_execute_class(CB_CL_CPR_DMA, CB_CODE_CPR_RESUME);
 967         if (cpr_suspend_succeeded) {
 968                 PMD(PMD_SX, ("cpr_resume: CPR RPC callback\n"))
 969                 (void) callb_execute_class(CB_CL_CPR_RPC, CB_CODE_CPR_RESUME);
```

```
 970         }

 972         prom_resume_prepost();
 973 #if !defined(__sparc)
 974         /*
 975          * Need to sync the software clock with the hardware clock.
 976          * On Sparc, this occurs in the sparc-specific cbe.  However
 977          * on x86 this needs to be handled _before_ we bring other cpu's
 978          * back online.  So we call a resume function in timestamp.c
 979          */
 980         if (tsc_resume_in_cyclic == 0)
 981                 tsc_resume();

 983 #endif

 985 #if defined(__sparc)
 986         if (cpr_suspend_succeeded && (boothowto & RB_DEBUG))
 987                 kdi_dvec_cpr_restart();
 988 #endif


 991 #if defined(__x86)
 992 rb_mp_paused:
 993         PT(PT_RMPO);
 994         PMD(PMD_SX, ("resume aux cpus\n"))

 996         if (cpr_suspend_succeeded) {
 997                 cpr_resume_cpus();
 998         } else {
 999                 cpr_unpause_cpus();
1000         }
1001 #endif

1003         /*
1004          * let the tmp callout catch up.
1005          */
1006         PMD(PMD_SX, ("cpr_resume: CPR CALLOUT callback\n"))
1007         (void) callb_execute_class(CB_CL_CPR_CALLOUT, CB_CODE_CPR_RESUME);

1009         i_cpr_enable_intr();

1011         mutex_enter(&cpu_lock);
1012         PMD(PMD_SX, ("cpr_resume: cyclic resume\n"))
1013         cyclic_resume();
1014         mutex_exit(&cpu_lock);

1016         PMD(PMD_SX, ("cpr_resume: handle xc\n"))
1017         i_cpr_handle_xc(0);     /* turn it off to allow xc assertion */

1019         PMD(PMD_SX, ("cpr_resume: CPR POST KERNEL callback\n"))
1020         (void) callb_execute_class(CB_CL_CPR_POST_KERNEL, CB_CODE_CPR_RESUME);

1022         /*
1023          * statistics gathering
1024          */
1025         if (cpr_suspend_succeeded) {
1026                 /*
1027                  * Prevent false alarm in tod_validate() due to tod
1028                  * value change between suspend and resume
1029                  */
1030                 cpr_tod_status_set(TOD_CPR_RESUME_DONE);

1032                 cpr_convert_promtime(&pwron_tv);

1034                 ctp = &cpr_term.tm_shutdown;
1035                 if (sleeptype == CPR_TODISK)
```



```
1036                         CPR_STAT_EVENT_END_TMZ("  write statefile", ctp);
1037                 CPR_STAT_EVENT_END_TMZ("Suspend Total", ctp);

1039                 CPR_STAT_EVENT_START_TMZ("Resume Total", &pwron_tv);

1041                 str = "  prom time";
1042                 CPR_STAT_EVENT_START_TMZ(str, &pwron_tv);
1043                 ctp = &cpr_term.tm_cprboot_start;
1044                 CPR_STAT_EVENT_END_TMZ(str, ctp);

1046                 str = "  read statefile";
1047                 CPR_STAT_EVENT_START_TMZ(str, ctp);
1048                 ctp = &cpr_term.tm_cprboot_end;
1049                 CPR_STAT_EVENT_END_TMZ(str, ctp);
1050         }

1052 rb_stop_kernel_threads:
1053         /*
1054          * Put all threads back to where they belong; get the kernel
1055          * daemons straightened up too. Note that the callback table
1056          * locked during cpr_stop_kernel_threads() is released only
1057          * in cpr_start_kernel_threads(). Ensure modunloading is
1058          * disabled before starting kernel threads, we don't want
1059          * modunload thread to start changing device tree underneath.
1060          */
1061         PMD(PMD_SX, ("cpr_resume: modunload disable\n"))
1062         modunload_disable();
1063         PMD(PMD_SX, ("cpr_resume: start kernel threads\n"))
1064         cpr_start_kernel_threads();

1066 rb_suspend_devices:
1067         CPR_DEBUG(CPR_DEBUG1, "resuming devices...");
1068         CPR_STAT_EVENT_START("  start drivers");

1070         PMD(PMD_SX,
1071             ("cpr_resume: rb_suspend_devices: cpr_resume_uniproc = %d\n",
1072             cpr_resume_uniproc))

1074 #if defined(__x86)
1075         /*
1076          * If cpr_resume_uniproc is set, then pause all the other cpus
1077          * apart from the current cpu, so that broken drivers that think
1078          * that they are on a uniprocessor machine will resume
1079          */
1080         if (cpr_resume_uniproc) {
1081                 mutex_enter(&cpu_lock);
1082                 pause_cpus(NULL, NULL);
1097                 pause_cpus(NULL);
1083                 mutex_exit(&cpu_lock);
1084         }
1085 #endif

1087         /*
1088          * The policy here is to continue resume everything we can if we did
1089          * not successfully finish suspend; and panic if we are coming back
1090          * from a fully suspended system.
1091          */
1092         PMD(PMD_SX, ("cpr_resume: resume devices\n"))
1093         rc = cpr_resume_devices(ddi_root_node(), 0);

1095         cpr_sae(0);

1097         str = "Failed to resume one or more devices.";

1099         if (rc) {
1100                 if (CPR->c_substate == C_ST_DUMP ||
```

```
1101                    (sleeptype == CPR_TORAM &&
1102                    CPR->c_substate == C_ST_NODUMP)) {
1103                        if (cpr_test_point == FORCE_SUSPEND_TO_RAM) {
1104                                PMD(PMD_SX, ("cpr_resume: resume device "
1105                                    "warn\n"))
1106                                cpr_err(CE_WARN, str);
1107                        } else {
1108                                PMD(PMD_SX, ("cpr_resume: resume device "
1109                                    "panic\n"))
1110                                cpr_err(CE_PANIC, str);
1111                        }
1112                } else {
1113                        PMD(PMD_SX, ("cpr_resume: resume device warn\n"))
1114                        cpr_err(CE_WARN, str);
1115                }
1116        }

1118        CPR_STAT_EVENT_END("  start drivers");
1119        CPR_DEBUG(CPR_DEBUG1, "done\n");

1121 #if defined(__x86)
1122        /*
1123         * If cpr_resume_uniproc is set, then unpause all the processors
1124         * that were paused before resuming the drivers
1125         */
1126        if (cpr_resume_uniproc) {
1127                mutex_enter(&cpu_lock);
1128                start_cpus();
1129                mutex_exit(&cpu_lock);
1130        }
1131 #endif

1133        /*
1134         * If we had disabled modunloading in this cpr resume cycle (i.e. we
1135         * resumed from a state earlier than C_ST_SUSPEND_DEVICES), re-enable
1136         * modunloading now.
1137         */
1138        if (CPR->c_substate != C_ST_SUSPEND_DEVICES) {
1139                PMD(PMD_SX, ("cpr_resume: modload enable\n"))
1140                modunload_enable();
1141        }

1143        /*
1144         * Hooks needed by lock manager prior to resuming.
1145         * Refer to code for more comments.
1146         */
1147        PMD(PMD_SX, ("cpr_resume: lock mgr\n"))
1148        cpr_lock_mgr(lm_cprresume);

1150 #if defined(__sparc)
1151        /*
1152         * This is a partial (half) resume during cpr suspend, we
1153         * haven't yet given up on the suspend. On return from here,
1154         * cpr_suspend() will try to reallocate and retry the suspend.
1155         */
1156        if (CPR->c_substate == C_ST_DUMP_NOSPC) {
1157                return (0);
1158        }

1160        if (sleeptype == CPR_TODISK) {
1161 rb_statef_alloc:
1162                cpr_statef_close();

1164 rb_disable_ufs_logging:
1165                /*
1166                 * if ufs logging was disabled, re-enable
```



```
1167                 */
1168                (void) cpr_ufs_logging(1);
1169        }
1170 #endif

1172 rb_pm_reattach_noinvol:
1173        /*
1174         * When pm_reattach_noinvol() succeeds, modunload_thread will
1175         * remain disabled until after cpr suspend passes the
1176         * C_ST_STOP_KERNEL_THREADS state. If any failure happens before
1177         * cpr suspend reaches this state, we'll need to enable modunload
1178         * thread during rollback.
1179         */
1180        if (CPR->c_substate == C_ST_DISABLE_UFS_LOGGING ||
1181            CPR->c_substate == C_ST_STATEF_ALLOC ||
1182            CPR->c_substate == C_ST_SUSPEND_DEVICES ||
1183            CPR->c_substate == C_ST_STOP_KERNEL_THREADS) {
1184                PMD(PMD_SX, ("cpr_resume: reattach noinvol fini\n"))
1185                pm_reattach_noinvol_fini();
1186        }

1188        PMD(PMD_SX, ("cpr_resume: CPR POST USER callback\n"))
1189        (void) callb_execute_class(CB_CL_CPR_POST_USER, CB_CODE_CPR_RESUME);
1190        PMD(PMD_SX, ("cpr_resume: CPR PROMPRINTF callback\n"))
1191        (void) callb_execute_class(CB_CL_CPR_PROMPRINTF, CB_CODE_CPR_RESUME);

1193        PMD(PMD_SX, ("cpr_resume: restore direct levels\n"))
1194        pm_restore_direct_levels();

1196 rb_stop_user_threads:
1197        CPR_DEBUG(CPR_DEBUG1, "starting user threads...");
1198        PMD(PMD_SX, ("cpr_resume: starting user threads\n"))
1199        cpr_start_user_threads();
1200        CPR_DEBUG(CPR_DEBUG1, "done\n");
1201        /*
1202         * Ask Xorg to resume the frame buffer, and wait for it to happen
1203         */
1204        mutex_enter(&srn_clone_lock);
1205        if (srn_signal) {
1206                PMD(PMD_SX, ("cpr_suspend: (*srn_signal)(..., "
1207                    "SRN_NORMAL_RESUME)\n"))
1208                srn_inuse = 1;          /* because (*srn_signal) cv_waits */
1209                (*srn_signal)(SRN_TYPE_APM, SRN_NORMAL_RESUME);
1210                srn_inuse = 0;
1211        } else {
1212                PMD(PMD_SX, ("cpr_suspend: srn_signal NULL\n"))
1213        }
1214        mutex_exit(&srn_clone_lock);

1216 #if defined(__sparc)
1217 rb_mp_offline:
1218        if (cpr_mp_online())
1219                cpr_err(CE_WARN, "Failed to online all the processors.");
1220 #endif

1222 rb_others:
1223        PMD(PMD_SX, ("cpr_resume: dep thread\n"))
1224        pm_dispatch_to_dep_thread(PM_DEP_WK_CPR_RESUME, NULL, NULL,
1225            PM_DEP_WAIT, NULL, 0);

1227        PMD(PMD_SX, ("cpr_resume: CPR PM callback\n"))
1228        (void) callb_execute_class(CB_CL_CPR_PM, CB_CODE_CPR_RESUME);

1230        if (cpr_suspend_succeeded) {
1231                cpr_stat_record_events();
1232        }
```

```
#if defined(__sparc)
        if (sleeptype == CPR_TODISK && !cpr_reusable_mode)
                cpr_clear_definfo();
#endif

        i_cpr_free_cpus();
        CPR_DEBUG(CPR_DEBUG1, "Sending SIGTHAW...");
        PMD(PMD_SX, ("cpr_resume: SIGTHAW\n"))
        cpr_signal_user(SIGTHAW);
        CPR_DEBUG(CPR_DEBUG1, "done\n");

        CPR_STAT_EVENT_END("Resume Total");

        CPR_STAT_EVENT_START_TMZ("WHOLE CYCLE", &wholecycle_tv);
        CPR_STAT_EVENT_END("WHOLE CYCLE");

        if (cpr_debug & CPR_DEBUG1)
                cmn_err(CE_CONT, "\nThe system is back where you left!\n");

        CPR_STAT_EVENT_START("POST CPR DELAY");

#ifdef CPR_STAT
        ctp = &cpr_term.tm_shutdown;
        CPR_STAT_EVENT_START_TMZ("PWROFF TIME", ctp);
        CPR_STAT_EVENT_END_TMZ("PWROFF TIME", &pwron_tv);

        CPR_STAT_EVENT_PRINT();
#endif /* CPR_STAT */

        PMD(PMD_SX, ("cpr_resume returns %x\n", rc))
        return (rc);
}
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   52333 Fri Jan  3 22:11:52 2014
new/usr/src/uts/common/disp/cmt.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_
```

```
 320 /*
 321  * Promote PG above it's current parent.
 322  * This is only legal if PG has an equal or greater number of CPUs than its
 323  * parent.
 324  *
 325  * This routine operates on the CPU specific processor group data (for the CPUs
 326  * in the PG being promoted), and may be invoked from a context where one CPU's
 327  * PG data is under construction. In this case the argument "pgdata", if not
 328  * NULL, is a reference to the CPU's under-construction PG data.
 329  */
 330 static void
 331 cmt_hier_promote(pg_cmt_t *pg, cpu_pg_t *pgdata)
 332 {
 333         pg_cmt_t        *parent;
 334         group_t         *children;
 335         cpu_t           *cpu;
 336         group_iter_t    iter;
 337         pg_cpu_itr_t    cpu_iter;
 338         int             r;
 339         int             err;
 340         int             nchildren;

 342         ASSERT(MUTEX_HELD(&cpu_lock));

 344         parent = pg->cmt_parent;
 345         if (parent == NULL) {
 346                 /*
 347                  * Nothing to do
 348                  */
 349                 return;
 350         }

 352         ASSERT(PG_NUM_CPUS((pg_t *)pg) >= PG_NUM_CPUS((pg_t *)parent));

 354         /*
 355          * We're changing around the hierarchy, which is actively traversed
 356          * by the dispatcher. Pause CPUS to ensure exclusivity.
 357          */
 358         pause_cpus(NULL, NULL);
 358         pause_cpus(NULL);

 360         /*
 361          * If necessary, update the parent's sibling set, replacing parent
 362          * with PG.
 363          */
 364         if (parent->cmt_siblings) {
 365                 if (group_remove(parent->cmt_siblings, parent, GRP_NORESIZE)
 366                     != -1) {
 367                         r = group_add(parent->cmt_siblings, pg, GRP_NORESIZE);
 368                         ASSERT(r != -1);
 369                 }
 370         }

 372         /*
 373          * If the parent is at the top of the hierarchy, replace it's entry
 374          * in the root lgroup's group of top level PGs.
 375          */
 376         if (parent->cmt_parent == NULL &&
 377             parent->cmt_siblings != &cmt_root->cl_pgs) {
```



```
 378                 if (group_remove(&cmt_root->cl_pgs, parent, GRP_NORESIZE)
 379                     != -1) {
 380                         r = group_add(&cmt_root->cl_pgs, pg, GRP_NORESIZE);
 381                         ASSERT(r != -1);
 382                 }
 383         }

 385         /*
 386          * We assume (and therefore assert) that the PG being promoted is an
 387          * only child of it's parent. Update the parent's children set
 388          * replacing PG's entry with the parent (since the parent is becoming
 389          * the child). Then have PG and the parent swap children sets and
 390          * children counts.
 391          */
 392         ASSERT(GROUP_SIZE(parent->cmt_children) <= 1);
 393         if (group_remove(parent->cmt_children, pg, GRP_NORESIZE) != -1) {
 394                 r = group_add(parent->cmt_children, parent, GRP_NORESIZE);
 395                 ASSERT(r != -1);
 396         }

 398         children = pg->cmt_children;
 399         pg->cmt_children = parent->cmt_children;
 400         parent->cmt_children = children;

 402         nchildren = pg->cmt_nchildren;
 403         pg->cmt_nchildren = parent->cmt_nchildren;
 404         parent->cmt_nchildren = nchildren;

 406         /*
 407          * Update the sibling references for PG and it's parent
 408          */
 409         pg->cmt_siblings = parent->cmt_siblings;
 410         parent->cmt_siblings = pg->cmt_children;

 412         /*
 413          * Update any cached lineages in the per CPU pg data.
 414          */
 415         PG_CPU_ITR_INIT(pg, cpu_iter);
 416         while ((cpu = pg_cpu_next(&cpu_iter)) != NULL) {
 417                 int             idx;
 418                 int             sz;
 419                 pg_cmt_t        *cpu_pg;
 420                 cpu_pg_t        *pgd;   /* CPU's PG data */

 422                 /*
 423                  * The CPU's whose lineage is under construction still
 424                  * references the bootstrap CPU PG data structure.
 425                  */
 426                 if (pg_cpu_is_bootstrapped(cpu))
 427                         pgd = pgdata;
 428                 else
 429                         pgd = cpu->cpu_pg;

 431                 /*
 432                  * Iterate over the CPU's PGs updating the children
 433                  * of the PG being promoted, since they have a new parent.
 434                  */
 435                 group_iter_init(&iter);
 436                 while ((cpu_pg = group_iterate(&pgd->cmt_pgs, &iter)) != NULL) {
 437                         if (cpu_pg->cmt_parent == pg) {
 438                                 cpu_pg->cmt_parent = parent;
 439                         }
 440                 }

 442                 /*
 443                  * Update the CMT load balancing lineage
```

```
444                  */
445                 if ((idx = group_find(&pgd->cmt_pgs, (void *)pg)) == -1) {
446                         /*
447                          * Unless this is the CPU who's lineage is being
448                          * constructed, the PG being promoted should be
449                          * in the lineage.
450                          */
451                         ASSERT(pg_cpu_is_bootstrapped(cpu));
452                         continue;
453                 }

455                 ASSERT(idx > 0);
456                 ASSERT(GROUP_ACCESS(&pgd->cmt_pgs, idx - 1) == parent);

458                 /*
459                  * Have the child and the parent swap places in the CPU's
460                  * lineage
461                  */
462                 group_remove_at(&pgd->cmt_pgs, idx);
463                 group_remove_at(&pgd->cmt_pgs, idx - 1);
464                 err = group_add_at(&pgd->cmt_pgs, parent, idx);
465                 ASSERT(err == 0);
466                 err = group_add_at(&pgd->cmt_pgs, pg, idx - 1);
467                 ASSERT(err == 0);

469                 /*
470                  * Ensure cmt_lineage references CPU's leaf PG.
471                  * Since cmt_pgs is top-down ordered, the bottom is the last
472                  * element.
473                  */
474                 if ((sz = GROUP_SIZE(&pgd->cmt_pgs)) > 0)
475                         pgd->cmt_lineage = GROUP_ACCESS(&pgd->cmt_pgs, sz - 1);
476         }

478         /*
479          * Update the parent references for PG and it's parent
480          */
481         pg->cmt_parent = parent->cmt_parent;
482         parent->cmt_parent = pg;

484         start_cpus();
485 }
_____unchanged_portion_omitted_

1455 /*
1456  * Prune PG, and all other instances of PG's hardware sharing relationship
1457  * from the CMT PG hierarchy.
1458  *
1459  * This routine operates on the CPU specific processor group data (for the CPUs
1460  * in the PG being pruned), and may be invoked from a context where one CPU's
1461  * PG data is under construction. In this case the argument "pgdata", if not
1462  * NULL, is a reference to the CPU's under-construction PG data.
1463  */
1464 static int
1465 pg_cmt_prune(pg_cmt_t *pg_bad, pg_cmt_t **lineage, int *sz, cpu_pg_t *pgdata)
1466 {
1467         group_t         *hwset, *children;
1468         int             i, j, r, size = *sz;
1469         group_iter_t    hw_iter, child_iter;
1470         pg_cpu_itr_t    cpu_iter;
1471         pg_cmt_t        *pg, *child;
1472         cpu_t           *cpu;
1473         int             cap_needed;
1474         pghw_type_t     hw;

1476         ASSERT(MUTEX_HELD(&cpu_lock));
```



```
1478         /*
1479          * Inform pghw layer that this PG is pruned.
1480          */
1481         pghw_cmt_fini((pghw_t *)pg_bad);

1483         hw = ((pghw_t *)pg_bad)->pghw_hw;

1485         if (hw == PGHW_POW_ACTIVE) {
1486                 cmn_err(CE_NOTE, "!Active CPUPM domain groups look suspect. "
1487                     "Event Based CPUPM Unavailable");
1488         } else if (hw == PGHW_POW_IDLE) {
1489                 cmn_err(CE_NOTE, "!Idle CPUPM domain groups look suspect. "
1490                     "Dispatcher assisted CPUPM disabled.");
1491         }

1493         /*
1494          * Find and eliminate the PG from the lineage.
1495          */
1496         for (i = 0; i < size; i++) {
1497                 if (lineage[i] == pg_bad) {
1498                         for (j = i; j < size - 1; j++)
1499                                 lineage[j] = lineage[j + 1];
1500                         *sz = size - 1;
1501                         break;
1502                 }
1503         }

1505         /*
1506          * We'll prune all instances of the hardware sharing relationship
1507          * represented by pg. But before we do that (and pause CPUs) we need
1508          * to ensure the hierarchy's groups are properly sized.
1509          */
1510         hwset = pghw_set_lookup(hw);

1512         /*
1513          * Blacklist the hardware so future processor groups of this type won't
1514          * participate in CMT thread placement.
1515          *
1516          * XXX
1517          * For heterogeneous system configurations, this might be overkill.
1518          * We may only need to blacklist the illegal PGs, and other instances
1519          * of this hardware sharing relationship may be ok.
1520          */
1521         cmt_hw_blacklisted[hw] = 1;

1523         /*
1524          * For each of the PGs being pruned, ensure sufficient capacity in
1525          * the siblings set for the PG's children
1526          */
1527         group_iter_init(&hw_iter);
1528         while ((pg = group_iterate(hwset, &hw_iter)) != NULL) {
1529                 /*
1530                  * PG is being pruned, but if it is bringing up more than
1531                  * one child, ask for more capacity in the siblings group.
1532                  */
1533                 cap_needed = 0;
1534                 if (pg->cmt_children &&
1535                     GROUP_SIZE(pg->cmt_children) > 1) {
1536                         cap_needed = GROUP_SIZE(pg->cmt_children) - 1;

1538                         group_expand(pg->cmt_siblings,
1539                             GROUP_SIZE(pg->cmt_siblings) + cap_needed);

1541                         /*
1542                          * If this is a top level group, also ensure the
```

```
1543                         * capacity in the root lgrp level CMT grouping.
1544                         */
1545                        if (pg->cmt_parent == NULL &&
1546                            pg->cmt_siblings != &cmt_root->cl_pgs) {
1547                                group_expand(&cmt_root->cl_pgs,
1548                                    GROUP_SIZE(&cmt_root->cl_pgs) + cap_needed);
1549                                cmt_root->cl_npgs += cap_needed;
1550                        }
1551                }
1552        }

1554        /*
1555         * We're operating on the PG hierarchy. Pause CPUs to ensure
1556         * exclusivity with respect to the dispatcher.
1557         */
1558        pause_cpus(NULL, NULL);
1558        pause_cpus(NULL);

1560        /*
1561         * Prune all PG instances of the hardware sharing relationship
1562         * represented by pg.
1563         */
1564        group_iter_init(&hw_iter);
1565        while ((pg = group_iterate(hwset, &hw_iter)) != NULL) {

1567                /*
1568                 * Remove PG from it's group of siblings, if it's there.
1569                 */
1570                if (pg->cmt_siblings) {
1571                        (void) group_remove(pg->cmt_siblings, pg, GRP_NORESIZE);
1572                }
1573                if (pg->cmt_parent == NULL &&
1574                    pg->cmt_siblings != &cmt_root->cl_pgs) {
1575                        (void) group_remove(&cmt_root->cl_pgs, pg,
1576                            GRP_NORESIZE);
1577                }

1579                /*
1580                 * Indicate that no CMT policy will be implemented across
1581                 * this PG.
1582                 */
1583                pg->cmt_policy = CMT_NO_POLICY;

1585                /*
1586                 * Move PG's children from it's children set to it's parent's
1587                 * children set. Note that the parent's children set, and PG's
1588                 * siblings set are the same thing.
1589                 *
1590                 * Because we are iterating over the same group that we are
1591                 * operating on (removing the children), first add all of PG's
1592                 * children to the parent's children set, and once we are done
1593                 * iterating, empty PG's children set.
1594                 */
1595                if (pg->cmt_children != NULL) {
1596                        children = pg->cmt_children;

1598                        group_iter_init(&child_iter);
1599                        while ((child = group_iterate(children, &child_iter))
1600                            != NULL) {
1601                                if (pg->cmt_siblings != NULL) {
1602                                        r = group_add(pg->cmt_siblings, child,
1603                                            GRP_NORESIZE);
1604                                        ASSERT(r == 0);

1606                                        if (pg->cmt_parent == NULL &&
1607                                            pg->cmt_siblings !=
```



```
1608                                            &cmt_root->cl_pgs) {
1609                                                r = group_add(&cmt_root->cl_pgs,
1610                                                    child, GRP_NORESIZE);
1611                                                ASSERT(r == 0);
1612                                        }
1613                                }
1614                        }
1615                        group_empty(pg->cmt_children);
1616                }

1618                /*
1619                 * Reset the callbacks to the defaults
1620                 */
1621                pg_callback_set_defaults((pg_t *)pg);

1623                /*
1624                 * Update all the CPU lineages in each of PG's CPUs
1625                 */
1626                PG_CPU_ITR_INIT(pg, cpu_iter);
1627                while ((cpu = pg_cpu_next(&cpu_iter)) != NULL) {
1628                        pg_cmt_t        *cpu_pg;
1629                        group_iter_t    liter;  /* Iterator for the lineage */
1630                        cpu_pg_t        *cpd;   /* CPU's PG data */

1632                        /*
1633                         * The CPU's lineage is under construction still
1634                         * references the bootstrap CPU PG data structure.
1635                         */
1636                        if (pg_cpu_is_bootstrapped(cpu))
1637                                cpd = pgdata;
1638                        else
1639                                cpd = cpu->cpu_pg;

1641                        /*
1642                         * Iterate over the CPU's PGs updating the children
1643                         * of the PG being promoted, since they have a new
1644                         * parent and siblings set.
1645                         */
1646                        group_iter_init(&liter);
1647                        while ((cpu_pg = group_iterate(&cpd->pgs,
1648                            &liter)) != NULL) {
1649                                if (cpu_pg->cmt_parent == pg) {
1650                                        cpu_pg->cmt_parent = pg->cmt_parent;
1651                                        cpu_pg->cmt_siblings = pg->cmt_siblings;
1652                                }
1653                        }

1655                        /*
1656                         * Update the CPU's lineages
1657                         *
1658                         * Remove the PG from the CPU's group used for CMT
1659                         * scheduling.
1660                         */
1661                        (void) group_remove(&cpd->cmt_pgs, pg, GRP_NORESIZE);
1662                }
1663        }
1664        start_cpus();
1665        return (0);
1666 }

1668 /*
1669  * Disable CMT scheduling
1670  */
1671 static void
1672 pg_cmt_disable(void)
1673 {
```

```
1674         cpu_t           *cpu;

1676         ASSERT(MUTEX_HELD(&cpu_lock));

1678         pause_cpus(NULL, NULL);
1678         pause_cpus(NULL);
1679         cpu = cpu_list;

1681         do {
1682                 if (cpu->cpu_pg)
1683                         group_empty(&cpu->cpu_pg->cmt_pgs);
1684         } while ((cpu = cpu->cpu_next) != cpu_list);

1686         cmt_sched_disabled = 1;
1687         start_cpus();
1688         cmn_err(CE_NOTE, "!CMT thread placement optimizations unavailable");
1689 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   30551 Fri Jan  3 22:11:52 2014
new/usr/src/uts/common/disp/cpupart.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_


static int
cpupart_move_cpu(cpu_t *cp, cpupart_t *newpp, int forced)
{
        cpupart_t *oldpp;
        cpu_t   *ncp, *newlist;
        kthread_t *t;
        int     move_threads = 1;
        lgrp_id_t lgrpid;
        proc_t  *p;
        int lgrp_diff_lpl;
        lpl_t   *cpu_lpl;
        int     ret;
        boolean_t unbind_all_threads = (forced != 0);

        ASSERT(MUTEX_HELD(&cpu_lock));
        ASSERT(newpp != NULL);

        oldpp = cp->cpu_part;
        ASSERT(oldpp != NULL);
        ASSERT(oldpp->cp_ncpus > 0);

        if (newpp == oldpp) {
                /*
                 * Don't need to do anything.
                 */
                return (0);
        }

        cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_OUT);

        if (!disp_bound_partition(cp, 0)) {
                /*
                 * Don't need to move threads if there are no threads in
                 * the partition.  Note that threads can't enter the
                 * partition while we're holding cpu_lock.
                 */
                move_threads = 0;
        } else if (oldpp->cp_ncpus == 1) {
                /*
                 * The last CPU is removed from a partition which has threads
                 * running in it. Some of these threads may be bound to this
                 * CPU.
                 *
                 * Attempt to unbind threads from the CPU and from the processor
                 * set. Note that no threads should be bound to this CPU since
                 * cpupart_move_threads will refuse to move bound threads to
                 * other CPUs.
                 */
                (void) cpu_unbind(oldpp->cp_cpulist->cpu_id, B_FALSE);
                (void) cpupart_unbind_threads(oldpp, B_FALSE);

                if (!disp_bound_partition(cp, 0)) {
                        /*
                         * No bound threads in this partition any more
                         */
                        move_threads = 0;
                } else {
                        /*
```



```
                         * There are still threads bound to the partition
                         */
                        cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_IN);
                        return (EBUSY);
                }
        }

        /*
         * If forced flag is set unbind any threads from this CPU.
         * Otherwise unbind soft-bound threads only.
         */
        if ((ret = cpu_unbind(cp->cpu_id, unbind_all_threads)) != 0) {
                cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_IN);
                return (ret);
        }

        /*
         * Stop further threads weak binding to this cpu.
         */
        cpu_inmotion = cp;
        membar_enter();

        /*
         * Notify the Processor Groups subsystem that the CPU
         * will be moving cpu partitions. This is done before
         * CPUs are paused to provide an opportunity for any
         * needed memory allocations.
         */
        pg_cpupart_out(cp, oldpp);
        pg_cpupart_in(cp, newpp);

again:
        if (move_threads) {
                int loop_count;
                /*
                 * Check for threads strong or weak bound to this CPU.
                 */
                for (loop_count = 0; disp_bound_threads(cp, 0); loop_count++) {
                        if (loop_count >= 5) {
                                cpu_state_change_notify(cp->cpu_id,
                                    CPU_CPUPART_IN);
                                pg_cpupart_out(cp, newpp);
                                pg_cpupart_in(cp, oldpp);
                                cpu_inmotion = NULL;
                                return (EBUSY); /* some threads still bound */
                        }
                        delay(1);
                }
        }

        /*
         * Before we actually start changing data structures, notify
         * the cyclic subsystem that we want to move this CPU out of its
         * partition.
         */
        if (!cyclic_move_out(cp)) {
                /*
                 * This CPU must be the last CPU in a processor set with
                 * a bound cyclic.
                 */
                cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_IN);
                pg_cpupart_out(cp, newpp);
                pg_cpupart_in(cp, oldpp);
                cpu_inmotion = NULL;
                return (EBUSY);
        }
```

```
444         pause_cpus(cp, NULL);
444         pause_cpus(cp);

446         if (move_threads) {
447                 /*
448                  * The thread on cpu before the pause thread may have read
449                  * cpu_inmotion before we raised the barrier above.  Check
450                  * again.
451                  */
452                 if (disp_bound_threads(cp, 1)) {
453                         start_cpus();
454                         goto again;
455                 }

457         }

459         /*
460          * Now that CPUs are paused, let the PG subsystem perform
461          * any necessary data structure updates.
462          */
463         pg_cpupart_move(cp, oldpp, newpp);

465         /* save this cpu's lgroup -- it'll be the same in the new partition */
466         lgrpid = cp->cpu_lpl->lpl_lgrpid;

468         cpu_lpl = cp->cpu_lpl;
469         /*
470          * let the lgroup framework know cp has left the partition
471          */
472         lgrp_config(LGRP_CONFIG_CPUPART_DEL, (uintptr_t)cp, lgrpid);

474         /* move out of old partition */
475         oldpp->cp_ncpus--;
476         if (oldpp->cp_ncpus > 0) {

478                 ncp = cp->cpu_prev_part->cpu_next_part = cp->cpu_next_part;
479                 cp->cpu_next_part->cpu_prev_part = cp->cpu_prev_part;
480                 if (oldpp->cp_cpulist == cp) {
481                         oldpp->cp_cpulist = ncp;
482                 }
483         } else {
484                 ncp = oldpp->cp_cpulist = NULL;
485                 cp_numparts_nonempty--;
486                 ASSERT(cp_numparts_nonempty != 0);
487         }
488         oldpp->cp_gen++;

490         /* move into new partition */
491         newlist = newpp->cp_cpulist;
492         if (newlist == NULL) {
493                 newpp->cp_cpulist = cp->cpu_next_part = cp->cpu_prev_part = cp;
494                 cp_numparts_nonempty++;
495                 ASSERT(cp_numparts_nonempty != 0);
496         } else {
497                 cp->cpu_next_part = newlist;
498                 cp->cpu_prev_part = newlist->cpu_prev_part;
499                 newlist->cpu_prev_part->cpu_next_part = cp;
500                 newlist->cpu_prev_part = cp;
501         }
502         cp->cpu_part = newpp;
503         newpp->cp_ncpus++;
504         newpp->cp_gen++;

506         ASSERT(bitset_is_null(&newpp->cp_haltset));
507         ASSERT(bitset_is_null(&oldpp->cp_haltset));
```



```
509         /*
510          * let the lgroup framework know cp has entered the partition
511          */
512         lgrp_config(LGRP_CONFIG_CPUPART_ADD, (uintptr_t)cp, lgrpid);

514         /*
515          * If necessary, move threads off processor.
516          */
517         if (move_threads) {
518                 ASSERT(ncp != NULL);

520                 /*
521                  * Walk thru the active process list to look for
522                  * threads that need to have a new home lgroup,
523                  * or the last CPU they run on is the same CPU
524                  * being moved out of the partition.
525                  */

527                 for (p = practive; p != NULL; p = p->p_next) {

529                         t = p->p_tlist;

531                         if (t == NULL)
532                                 continue;

534                         lgrp_diff_lpl = 0;

536                         do {

538                                 ASSERT(t->t_lpl != NULL);

540                                 /*
541                                  * Update the count of how many threads are
542                                  * in this CPU's lgroup but have a different lpl
543                                  */

545                                 if (t->t_lpl != cpu_lpl &&
546                                     t->t_lpl->lpl_lgrpid == lgrpid)
547                                         lgrp_diff_lpl++;
548                                 /*
549                                  * If the lgroup that t is assigned to no
550                                  * longer has any CPUs in t's partition,
551                                  * we'll have to choose a new lgroup for t.
552                                  */

554                                 if (!LGRP_CPUS_IN_PART(t->t_lpl->lpl_lgrpid,
555                                     t->t_cpupart)) {
556                                         lgrp_move_thread(t,
557                                             lgrp_choose(t, t->t_cpupart), 0);
558                                 }

560                                 /*
561                                  * make sure lpl points to our own partition
562                                  */
563                                 ASSERT(t->t_lpl >= t->t_cpupart->cp_lgrploads &&
564                                     (t->t_lpl < t->t_cpupart->cp_lgrploads +
565                                     t->t_cpupart->cp_nlgrploads));

567                                 ASSERT(t->t_lpl->lpl_ncpu > 0);

569                                 /* Update CPU last ran on if it was this CPU */
570                                 if (t->t_cpu == cp && t->t_cpupart == oldpp &&
571                                     t->t_bound_cpu != cp) {
572                                         t->t_cpu = disp_lowpri_cpu(ncp,
573                                             t->t_lpl, t->t_pri, NULL);
```

```
574                                 }
575                                 t = t->t_forw;
576                         } while (t != p->p_tlist);

578                         /*
579                          * Didn't find any threads in the same lgroup as this
580                          * CPU with a different lpl, so remove the lgroup from
581                          * the process lgroup bitmask.
582                          */

584                         if (lgrp_diff_lpl)
585                                 klgrpset_del(p->p_lgrpset, lgrpid);
586                 }

588                 /*
589                  * Walk thread list looking for threads that need to be
590                  * rehomed, since there are some threads that are not in
591                  * their process's p_tlist.
592                  */

594                 t = curthread;

596                 do {
597                         ASSERT(t != NULL && t->t_lpl != NULL);

599                         /*
600                          * If the lgroup that t is assigned to no
601                          * longer has any CPUs in t's partition,
602                          * we'll have to choose a new lgroup for t.
603                          * Also, choose best lgroup for home when
604                          * thread has specified lgroup affinities,
605                          * since there may be an lgroup with more
606                          * affinity available after moving CPUs
607                          * around.
608                          */
609                         if (!LGRP_CPUS_IN_PART(t->t_lpl->lpl_lgrpid,
610                             t->t_cpupart) || t->t_lgrp_affinity) {
611                                 lgrp_move_thread(t,
612                                     lgrp_choose(t, t->t_cpupart), 1);
613                         }

615                         /* make sure lpl points to our own partition */
616                         ASSERT((t->t_lpl >= t->t_cpupart->cp_lgrploads) &&
617                             (t->t_lpl < t->t_cpupart->cp_lgrploads +
618                             t->t_cpupart->cp_nlgrploads));

620                         ASSERT(t->t_lpl->lpl_ncpu > 0);

622                         /* Update CPU last ran on if it was this CPU */
623                         if (t->t_cpu == cp && t->t_cpupart == oldpp &&
624                             t->t_bound_cpu != cp) {
625                                 t->t_cpu = disp_lowpri_cpu(ncp, t->t_lpl,
626                                     t->t_pri, NULL);
627                         }

629                         t = t->t_next;
630                 } while (t != curthread);

632                 /*
633                  * Clear off the CPU's run queue, and the kp queue if the
634                  * partition is now empty.
635                  */
636                 disp_cpu_inactive(cp);

638                 /*
639                  * Make cp switch to a thread from the new partition.
```



```
640                  */
641                 cp->cpu_runrun = 1;
642                 cp->cpu_kprunrun = 1;
643         }

645         cpu_inmotion = NULL;
646         start_cpus();

648         /*
649          * Let anyone interested know that cpu has been added to the set.
650          */
651         cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_IN);

653         /*
654          * Now let the cyclic subsystem know that it can reshuffle cyclics
655          * bound to the new processor set.
656          */
657         cyclic_move_in(cp);

659         return (0);
660 }
```

*unchanged_portion_omitted_*

```
812 /*
813  * Create a new partition.  On MP systems, this also allocates a
814  * kpreempt disp queue for that partition.
815  */
816 int
817 cpupart_create(psetid_t *psid)
818 {
819         cpupart_t       *pp;

821         ASSERT(pool_lock_held());

823         pp = kmem_zalloc(sizeof (cpupart_t), KM_SLEEP);
824         pp->cp_nlgrploads = lgrp_plat_max_lgrps();
825         pp->cp_lgrploads = kmem_zalloc(sizeof (lpl_t) * pp->cp_nlgrploads,
826             KM_SLEEP);

828         mutex_enter(&cpu_lock);
829         if (cp_numparts == cp_max_numparts) {
830                 mutex_exit(&cpu_lock);
831                 kmem_free(pp->cp_lgrploads, sizeof (lpl_t) * pp->cp_nlgrploads);
832                 pp->cp_lgrploads = NULL;
833                 kmem_free(pp, sizeof (cpupart_t));
834                 return (ENOMEM);
835         }
836         cp_numparts++;
837         /* find the next free partition ID */
838         while (cpupart_find(CPTOPS(cp_id_next)) != NULL)
839                 cp_id_next++;
840         pp->cp_id = cp_id_next++;
841         pp->cp_ncpus = 0;
842         pp->cp_cpulist = NULL;
843         pp->cp_attr = 0;
844         klgrpset_clear(pp->cp_lgrpset);
845         pp->cp_kp_queue.disp_maxrunpri = -1;
846         pp->cp_kp_queue.disp_max_unbound_pri = -1;
847         pp->cp_kp_queue.disp_cpu = NULL;
848         pp->cp_gen = 0;
849         DISP_LOCK_INIT(&pp->cp_kp_queue.disp_lock);
850         *psid = CPTOPS(pp->cp_id);
851         disp_kp_alloc(&pp->cp_kp_queue, v.v_nglobpris);
852         cpupart_kstat_create(pp);
853         cpupart_lpl_initialize(pp);
```

```
855         bitset_init(&pp->cp_cmt_pgs);

857         /*
858          * Initialize and size the partition's bitset of halted CPUs.
859          */
860         bitset_init_fanout(&pp->cp_haltset, cp_haltset_fanout);
861         bitset_resize(&pp->cp_haltset, max_ncpus);

863         /*
864          * Pause all CPUs while changing the partition list, to make sure
865          * the clock thread (which traverses the list without holding
866          * cpu_lock) isn't running.
867          */
868         pause_cpus(NULL, NULL);
868         pause_cpus(NULL);
869         pp->cp_next = cp_list_head;
870         pp->cp_prev = cp_list_head->cp_prev;
871         cp_list_head->cp_prev->cp_next = pp;
872         cp_list_head->cp_prev = pp;
873         start_cpus();
874         mutex_exit(&cpu_lock);

876         return (0);
877 }
_____unchanged_portion_omitted_

949 /*
950  * Destroy a partition.
951  */
952 int
953 cpupart_destroy(psetid_t psid)
954 {
955         cpu_t   *cp, *first_cp;
956         cpupart_t *pp, *newpp;
957         int     err = 0;

959         ASSERT(pool_lock_held());
960         mutex_enter(&cpu_lock);

962         pp = cpupart_find(psid);
963         if (pp == NULL || pp == &cp_default) {
964                 mutex_exit(&cpu_lock);
965                 return (EINVAL);
966         }

968         /*
969          * Unbind all the threads currently bound to the partition.
970          */
971         err = cpupart_unbind_threads(pp, B_TRUE);
972         if (err) {
973                 mutex_exit(&cpu_lock);
974                 return (err);
975         }

977         newpp = &cp_default;
978         while ((cp = pp->cp_cpulist) != NULL) {
979                 if (err = cpupart_move_cpu(cp, newpp, 0)) {
980                         mutex_exit(&cpu_lock);
981                         return (err);
982                 }
983         }

985         ASSERT(bitset_is_null(&pp->cp_cmt_pgs));
986         ASSERT(bitset_is_null(&pp->cp_haltset));
```



```
988         /*
989          * Teardown the partition's group of active CMT PGs and halted
990          * CPUs now that they have all left.
991          */
992         bitset_fini(&pp->cp_cmt_pgs);
993         bitset_fini(&pp->cp_haltset);

995         /*
996          * Reset the pointers in any offline processors so they won't
997          * try to rejoin the destroyed partition when they're turned
998          * online.
999          */
1000        first_cp = cp = CPU;
1001        do {
1002                if (cp->cpu_part == pp) {
1003                        ASSERT(cp->cpu_flags & CPU_OFFLINE);
1004                        cp->cpu_part = newpp;
1005                }
1006                cp = cp->cpu_next;
1007        } while (cp != first_cp);

1009        /*
1010         * Pause all CPUs while changing the partition list, to make sure
1011         * the clock thread (which traverses the list without holding
1012         * cpu_lock) isn't running.
1013         */
1014        pause_cpus(NULL, NULL);
1014        pause_cpus(NULL);
1015        pp->cp_prev->cp_next = pp->cp_next;
1016        pp->cp_next->cp_prev = pp->cp_prev;
1017        if (cp_list_head == pp)
1018                cp_list_head = pp->cp_next;
1019        start_cpus();

1021        if (cp_id_next > pp->cp_id)
1022                cp_id_next = pp->cp_id;

1024        if (pp->cp_kstat)
1025                kstat_delete(pp->cp_kstat);

1027        cp_numparts--;

1029        disp_kp_free(&pp->cp_kp_queue);

1031        cpupart_lpl_teardown(pp);

1033        kmem_free(pp, sizeof (cpupart_t));
1034        mutex_exit(&cpu_lock);

1036        return (err);
1037 }
______unchanged_portion_omitted_
```

```
**********************************************************
   68046 Fri Jan  3 22:11:52 2014
new/usr/src/uts/common/disp/disp.c
patch setfrontbackdq
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_


 312 /*
 313  * For each CPU, allocate new dispatch queues
 314  * with the stated number of priorities.
 315  */
 316 static void
 317 cpu_dispqalloc(int numpris)
 318 {
 319         cpu_t   *cpup;
 320         struct disp_queue_info  *disp_mem;
 321         int i, num;

 323         ASSERT(MUTEX_HELD(&cpu_lock));

 325         disp_mem = kmem_zalloc(NCPU *
 326             sizeof (struct disp_queue_info), KM_SLEEP);

 328         /*
 329          * This routine must allocate all of the memory before stopping
 330          * the cpus because it must not sleep in kmem_alloc while the
 331          * CPUs are stopped.  Locks they hold will not be freed until they
 332          * are restarted.
 333          */
 334         i = 0;
 335         cpup = cpu_list;
 336         do {
 337                 disp_dq_alloc(&disp_mem[i], numpris, cpup->cpu_disp);
 338                 i++;
 339                 cpup = cpup->cpu_next;
 340         } while (cpup != cpu_list);
 341         num = i;

 343         pause_cpus(NULL, NULL);
 343         pause_cpus(NULL);
 344         for (i = 0; i < num; i++)
 345                 disp_dq_assign(&disp_mem[i], numpris);
 346         start_cpus();

 348         /*
 349          * I must free all of the memory after starting the cpus because
 350          * I can not risk sleeping in kmem_free while the cpus are stopped.
 351          */
 352         for (i = 0; i < num; i++)
 353                 disp_dq_free(&disp_mem[i]);

 355         kmem_free(disp_mem, NCPU * sizeof (struct disp_queue_info));
 356 }
______unchanged_portion_omitted_

1151 /*
1152  * setbackdq() keeps runqs balanced such that the difference in length
1153  * between the chosen runq and the next one is no more than RUNQ_MAX_DIFF.
1154  * For threads with priorities below RUNQ_MATCH_PRI levels, the runq's lengths
1155  * must match.  When per-thread TS_RUNQMATCH flag is set, setbackdq() will
1156  * try to keep runqs perfectly balanced regardless of the thread priority.
1157  */
1158 #define RUNQ_MATCH_PRI  16      /* pri below which queue lengths must match */
1159 #define RUNQ_MAX_DIFF   2       /* maximum runq length difference */
```



```
1160 #define RUNQ_LEN(cp, pri)       ((cp)->cpu_disp->disp_q[pri].dq_sruncnt)

1162 /*
1163  * Macro that evaluates to true if it is likely that the thread has cache
1164  * warmth. This is based on the amount of time that has elapsed since the
1165  * thread last ran. If that amount of time is less than "rechoose_interval"
1166  * ticks, then we decide that the thread has enough cache warmth to warrant
1167  * some affinity for t->t_cpu.
1168  */
1169 #define THREAD_HAS_CACHE_WARMTH(thread) \
1170         ((thread == curthread) ||       \
1171         ((ddi_get_lbolt() - thread->t_disp_time) <= rechoose_interval))

1173 #endif /* ! codereview */
1174 /*
1175  * Put the specified thread on the front/back of the dispatcher queue
1176  * corresponding to its current priority.
1172  * Put the specified thread on the back of the dispatcher
1173  * queue corresponding to its current priority.
1177  *
1178  * Called with the thread in transition, onproc or stopped state and locked
1179  * (transition implies locked) and at high spl.  Returns with the thread in
1180  * TS_RUN state and still locked.
1175  * Called with the thread in transition, onproc or stopped state
1176  * and locked (transition implies locked) and at high spl.
1177  * Returns with the thread in TS_RUN state and still locked.
1181  */
1182 static void
1183 setfrontbackdq(kthread_t *tp, boolean_t front)
1179 void
1180 setbackdq(kthread_t *tp)
1184 {
1185         dispq_t         *dq;
1186         disp_t          *dp;
1187         cpu_t           *cp;
1188         pri_t           tpri;
1189         boolean_t       bound;
1186         int             bound;
1190         boolean_t       self;

1192         ASSERT(THREAD_LOCK_HELD(tp));
1193         ASSERT((tp->t_schedflag & TS_ALLSTART) == 0);
1194         ASSERT(!thread_on_queue(tp));   /* make sure tp isn't on a runq */

1196         /*
1197          * If thread is "swapped" or on the swap queue don't
1198          * queue it, but wake sched.
1199          */
1200         if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD) {
1201                 disp_swapped_setrun(tp);
1202                 return;
1203         }

1205         self  = (tp == curthread);
1206         bound = (tp->t_bound_cpu || tp->t_weakbound_cpu);

1204         if (tp->t_bound_cpu || tp->t_weakbound_cpu)
1205                 bound = 1;
1206         else
1207                 bound = 0;

1208         tpri = DISP_PRIO(tp);
1209         if (ncpus == 1)
1210                 cp = tp->t_cpu;
1211         else if (!bound) {
1212                 if (tpri >= kpqpri) {
```

```
                        setkpdq(tp, front ? SETKP_FRONT : SETKP_BACK);
                        setkpdq(tp, SETKP_BACK);
                        return;
                }

                cp = tp->t_cpu;

                if (!front) {
#endif /* ! codereview */
                        /*
                         * We'll generally let this thread continue to run where
                         * it last ran...but will consider migration if:
                         * - We thread probably doesn't have much cache warmth.
                         * - The CPU where it last ran is the target of an offli
                         *   request.
                         * - The thread last ran outside it's home lgroup.
                         */
                        if ((!THREAD_HAS_CACHE_WARMTH(tp)) || (cp == cpu_inmotio
                if ((!THREAD_HAS_CACHE_WARMTH(tp)) ||
                    (tp->t_cpu == cpu_inmotion)) {
                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
                        } else if (!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, tp->t
                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
                                    self ? tp->t_cpu : NULL);
                        }

                } else {
                        cp = tp->t_cpu;
                }

                if (tp->t_cpupart == cp->cpu_part) {
                        if (front) {
                                /*
                                 * We'll generally let this thread continue to r
                                 * where it last ran, but will consider migratio
                                 * - The thread last ran outside it's home lgrou
                                 * - The CPU where it last ran is the target of
                                 *   offline request (a thread_nomigrate() on th
                                 *   motion CPU relies on this when forcing a pr
                                 * - The thread isn't the highest priority threa
                                 *   it last ran, and it is considered not likel
                                 *   have significant cache warmth.
                                 */
                                if ((!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, cp)
                                    (cp == cpu_inmotion)) {
                                        cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lp
                                            self ? cp : NULL);
                                } else if ((tpri < cp->cpu_disp->disp_maxrunpri)
                                    (!THREAD_HAS_CACHE_WARMTH(tp))) {
                                        cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lp
                                            NULL);
                                }
                        } else {
#endif /* ! codereview */
                                int     qlen;

                                /*
                                 * Perform any CMT load balancing
                                 */
                                cp = cmt_balance(tp, cp);

                                /*
                                 * Balance across the run queues
                                 */
                                qlen = RUNQ_LEN(cp, tpri);
                                if (tpri >= RUNQ_MATCH_PRI &&
```



```
                                    !(tp->t_schedflag & TS_RUNQMATCH))
                                        qlen -= RUNQ_MAX_DIFF;
                                if (qlen > 0) {
                                        cpu_t *newcp;

                                        if (tp->t_lpl->lpl_lgrpid == LGRP_ROOTID
                                                newcp = cp->cpu_next_part;
                                        } else if ((newcp = cp->cpu_next_lpl) ==
                                                newcp = cp->cpu_next_part;
                                        }

                                        if (RUNQ_LEN(newcp, tpri) < qlen) {
                                                DTRACE_PROBE3(runq__balance,
                                                    kthread_t *, tp,
                                                    cpu_t *, cp, cpu_t *, newcp)
                                                cp = newcp;
                                        }
                                }
                        }
#endif /* ! codereview */
                } else {
                        /*
                         * Migrate to a cpu in the new partition.
                         */
                        cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
                            tp->t_lpl, tp->t_pri, NULL);
                }

#endif /* ! codereview */
                ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
        } else {
                /*
                 * It is possible that t_weakbound_cpu != t_bound_cpu (for
                 * a short time until weak binding that existed when the
                 * strong binding was established has dropped) so we must
                 * favour weak binding over strong.
                 */
                cp = tp->t_weakbound_cpu ?
                    tp->t_weakbound_cpu : tp->t_bound_cpu;
        }

#endif /* ! codereview */
        /*
         * A thread that is ONPROC may be temporarily placed on the run queue
         * but then chosen to run again by disp.  If the thread we're placing on
         * the queue is in TS_ONPROC state, don't set its t_waitrq until a
         * replacement process is actually scheduled in swtch().  In this
         * situation, curthread is the only thread that could be in the ONPROC
         * state.
         */
        if ((!self) && (tp->t_waitrq == 0)) {
                hrtime_t curtime;

                curtime = gethrtime_unscaled();
                (void) cpu_update_pct(tp, curtime);
                tp->t_waitrq = curtime;
        } else {
                (void) cpu_update_pct(tp, gethrtime_unscaled());
        }

        dp = cp->cpu_disp;
        disp_lock_enter_high(&dp->disp_lock);

        DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, front);
        if (front) {
                TRACE_2(TR_FAC_DISP, TR_FRONTQ, "frontq:pri %d tid %p", tpri,
```

```
1340                        tp);
1341        } else {
1229        DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 0);
1342                TRACE_3(TR_FAC_DISP, TR_BACKQ, "setbackdq:pri %d cpu %p tid %p",
1343                        tpri, cp, tp);
1344        }
1345 #endif /* ! codereview */

1347 #ifndef NPROBE
1348        /* Kernel probe */
1349        if (tnf_tracing_active)
1350                tnf_thread_queue(tp, cp, tpri);
1351 #endif /* NPROBE */

1353        ASSERT(tpri >= 0 && tpri < dp->disp_npri);

1355        THREAD_RUN(tp, &dp->disp_lock);         /* set t_state to TS_RUN */
1356        tp->t_disp_queue = dp;
1357        tp->t_link = NULL;

1359        dq = &dp->disp_q[tpri];
1360        dp->disp_nrunnable++;
1361        if (!bound)
1362                dp->disp_steal = 0;
1363        membar_enter();

1365        if (dq->dq_sruncnt++ != 0) {
1366                if (front) {
1367                        ASSERT(dq->dq_last != NULL);
1368                        tp->t_link = dq->dq_first;
1369                        dq->dq_first = tp;
1370                } else {
1371 #endif /* ! codereview */
1372                        ASSERT(dq->dq_first != NULL);
1373                        dq->dq_last->t_link = tp;
1374                        dq->dq_last = tp;
1375                }
1376 #endif /* ! codereview */
1377        } else {
1378                ASSERT(dq->dq_first == NULL);
1379                ASSERT(dq->dq_last == NULL);
1380                dq->dq_first = dq->dq_last = tp;
1381                BT_SET(dp->disp_qactmap, tpri);
1382                if (tpri > dp->disp_maxrunpri) {
1383                        dp->disp_maxrunpri = tpri;
1384                        membar_enter();
1385                        cpu_resched(cp, tpri);
1386                }
1387        }

1389        if (!bound && tpri > dp->disp_max_unbound_pri) {
1390                if (self && dp->disp_max_unbound_pri == -1 && cp == CPU) {
1391                        /*
1392                         * If there are no other unbound threads on the
1393                         * run queue, don't allow other CPUs to steal
1394                         * this thread while we are in the middle of a
1395                         * context switch. We may just switch to it
1396                         * again right away. CPU_DISP_DONTSTEAL is cleared
1397                         * in swtch and swtch_to.
1398                         */
1399                        cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
1400                }
1401                dp->disp_max_unbound_pri = tpri;
1402        }

1404 #endif /* ! codereview */
```



```
1405        (*disp_enq_thread)(cp, bound);
1406 }

1408 /*
1409  * Put the specified thread on the back of the dispatcher
1410  * queue corresponding to its current priority.
1411  *
1412  * Called with the thread in transition, onproc or stopped state
1413  * and locked (transition implies locked) and at high spl.
1414  * Returns with the thread in TS_RUN state and still locked.
1415  */
1416 void
1417 setbackdq(kthread_t *tp)
1418 {
1419        setfrontbackdq(tp, B_FALSE);
1420 }

1422 /*
1423 #endif /* ! codereview */
1424  * Put the specified thread on the front of the dispatcher
1425  * queue corresponding to its current priority.
1426  *
1427  * Called with the thread in transition, onproc or stopped state
1428  * and locked (transition implies locked) and at high spl.
1429  * Returns with the thread in TS_RUN state and still locked.
1430  */
1431 void
1432 setfrontdq(kthread_t *tp)
1433 {
1434        setfrontbackdq(tp, B_TRUE);
1232        disp_t          *dp;
1233        dispq_t         *dq;
1234        cpu_t           *cp;
1235        pri_t           tpri;
1236        int             bound;

1238        ASSERT(THREAD_LOCK_HELD(tp));
1239        ASSERT((tp->t_schedflag & TS_ALLSTART) == 0);
1240        ASSERT(!thread_on_queue(tp));   /* make sure tp isn't on a runq */

1242        /*
1243         * If thread is "swapped" or on the swap queue don't
1244         * queue it, but wake sched.
1245         */
1246        if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD) {
1247                disp_swapped_setrun(tp);
1248                return;
1249        }

1251        if (tp->t_bound_cpu || tp->t_weakbound_cpu)
1252                bound = 1;
1253        else
1254                bound = 0;

1256        tpri = DISP_PRIO(tp);
1257        if (ncpus == 1)
1258                cp = tp->t_cpu;
1259        else if (!bound) {
1260                if (tpri >= kpqpri) {
1261                        setkpdq(tp, SETKP_FRONT);
1262                        return;
1263                }
1264                cp = tp->t_cpu;
1265                if (tp->t_cpupart == cp->cpu_part) {
1266                        /*
1267                         * We'll generally let this thread continue to run
```

```
1268                         * where it last ran, but will consider migration if:
1269                         * - The thread last ran outside it's home lgroup.
1270                         * - The CPU where it last ran is the target of an
1271                         *   offline request (a thread_nomigrate() on the in
1272                         *   motion CPU relies on this when forcing a preempt).
1273                         * - The thread isn't the highest priority thread where
1274                         *   it last ran, and it is considered not likely to
1275                         *   have significant cache warmth.
1276                         */
1277                        if ((!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, cp)) ||
1278                            (cp == cpu_inmotion)) {
1279                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1280                                    (tp == curthread) ? cp : NULL);
1281                        } else if ((tpri < cp->cpu_disp->disp_maxrunpri) &&
1282                            (!THREAD_HAS_CACHE_WARMTH(tp))) {
1283                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1284                                    NULL);
1285                        }
1286                } else {
1287                        /*
1288                         * Migrate to a cpu in the new partition.
1289                         */
1290                        cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
1291                            tp->t_lpl, tp->t_pri, NULL);
1292                }
1293                ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
1294        } else {
1295                /*
1296                 * It is possible that t_weakbound_cpu != t_bound_cpu (for
1297                 * a short time until weak binding that existed when the
1298                 * strong binding was established has dropped) so we must
1299                 * favour weak binding over strong.
1300                 */
1301                cp = tp->t_weakbound_cpu ?
1302                    tp->t_weakbound_cpu : tp->t_bound_cpu;
1303        }
1304
1305        /*
1306         * A thread that is ONPROC may be temporarily placed on the run queue
1307         * but then chosen to run again by disp.  If the thread we're placing on
1308         * the queue is in TS_ONPROC state, don't set its t_waitrq until a
1309         * replacement process is actually scheduled in swtch().  In this
1310         * situation, curthread is the only thread that could be in the ONPROC
1311         * state.
1312         */
1313        if ((tp != curthread) && (tp->t_waitrq == 0)) {
1314                hrtime_t curtime;
1315
1316                curtime = gethrtime_unscaled();
1317                (void) cpu_update_pct(tp, curtime);
1318                tp->t_waitrq = curtime;
1319        } else {
1320                (void) cpu_update_pct(tp, gethrtime_unscaled());
1321        }
1322
1323        dp = cp->cpu_disp;
1324        disp_lock_enter_high(&dp->disp_lock);
1325
1326        TRACE_2(TR_FAC_DISP, TR_FRONTQ, "frontq:pri %d tid %p", tpri, tp);
1327        DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 1);
1328
1329 #ifndef NPROBE
1330        /* Kernel probe */
1331        if (tnf_tracing_active)
1332                tnf_thread_queue(tp, cp, tpri);
1333 #endif /* NPROBE */
```



```
1335        ASSERT(tpri >= 0 && tpri < dp->disp_npri);
1336
1337        THREAD_RUN(tp, &dp->disp_lock);         /* set TS_RUN state and lock */
1338        tp->t_disp_queue = dp;
1339
1340        dq = &dp->disp_q[tpri];
1341        dp->disp_nrunnable++;
1342        if (!bound)
1343                dp->disp_steal = 0;
1344        membar_enter();
1345
1346        if (dq->dq_sruncnt++ != 0) {
1347                ASSERT(dq->dq_last != NULL);
1348                tp->t_link = dq->dq_first;
1349                dq->dq_first = tp;
1350        } else {
1351                ASSERT(dq->dq_last == NULL);
1352                ASSERT(dq->dq_first == NULL);
1353                tp->t_link = NULL;
1354                dq->dq_first = dq->dq_last = tp;
1355                BT_SET(dp->disp_qactmap, tpri);
1356                if (tpri > dp->disp_maxrunpri) {
1357                        dp->disp_maxrunpri = tpri;
1358                        membar_enter();
1359                        cpu_resched(cp, tpri);
1360                }
1361        }
1362
1363        if (!bound && tpri > dp->disp_max_unbound_pri) {
1364                if (tp == curthread && dp->disp_max_unbound_pri == -1 &&
1365                    cp == CPU) {
1366                        /*
1367                         * If there are no other unbound threads on the
1368                         * run queue, don't allow other CPUs to steal
1369                         * this thread while we are in the middle of a
1370                         * context switch. We may just switch to it
1371                         * again right away. CPU_DISP_DONTSTEAL is cleared
1372                         * in swtch and swtch_to.
1373                         */
1374                        cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
1375                }
1376                dp->disp_max_unbound_pri = tpri;
1377        }
1378        (*disp_enq_thread)(cp, bound);
1379 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   94655 Fri Jan  3 22:11:53 2014
new/usr/src/uts/common/os/cpu.c
patch cpu-pause-func-deglobalize
**********************************************************
```

```
   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 (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright (c) 2012 by Delphix. All rights reserved.
  24  */

  26 /*
  27  * Architecture-independent CPU control functions.
  28  */

  30 #include <sys/types.h>
  31 #include <sys/param.h>
  32 #include <sys/var.h>
  33 #include <sys/thread.h>
  34 #include <sys/cpuvar.h>
  35 #include <sys/cpu_event.h>
  36 #include <sys/kstat.h>
  37 #include <sys/uadmin.h>
  38 #include <sys/systm.h>
  39 #include <sys/errno.h>
  40 #include <sys/cmn_err.h>
  41 #include <sys/procset.h>
  42 #include <sys/processor.h>
  43 #include <sys/debug.h>
  44 #include <sys/cpupart.h>
  45 #include <sys/lgrp.h>
  46 #include <sys/pset.h>
  47 #include <sys/pghw.h>
  48 #include <sys/kmem.h>
  49 #include <sys/kmem_impl.h>      /* to set per-cpu kmem_cache offset */
  50 #include <sys/atomic.h>
  51 #include <sys/callb.h>
  52 #include <sys/vtrace.h>
  53 #include <sys/cyclic.h>
  54 #include <sys/bitmap.h>
  55 #include <sys/nvpair.h>
  56 #include <sys/pool_pset.h>
  57 #include <sys/msacct.h>
  58 #include <sys/time.h>
  59 #include <sys/archsystm.h>
  60 #include <sys/sdt.h>
  61 #if defined(__x86) || defined(__amd64)
```



```
  62 #include <sys/x86_archext.h>
  63 #endif
  64 #include <sys/callo.h>

  66 extern int      mp_cpu_start(cpu_t *);
  67 extern int      mp_cpu_stop(cpu_t *);
  68 extern int      mp_cpu_poweron(cpu_t *);
  69 extern int      mp_cpu_poweroff(cpu_t *);
  70 extern int      mp_cpu_configure(int);
  71 extern int      mp_cpu_unconfigure(int);
  72 extern void     mp_cpu_faulted_enter(cpu_t *);
  73 extern void     mp_cpu_faulted_exit(cpu_t *);

  75 extern int cmp_cpu_to_chip(processorid_t cpuid);
  76 #ifdef __sparcv9
  77 extern char *cpu_fru_fmri(cpu_t *cp);
  78 #endif

  80 static void cpu_add_active_internal(cpu_t *cp);
  81 static void cpu_remove_active(cpu_t *cp);
  82 static void cpu_info_kstat_create(cpu_t *cp);
  83 static void cpu_info_kstat_destroy(cpu_t *cp);
  84 static void cpu_stats_kstat_create(cpu_t *cp);
  85 static void cpu_stats_kstat_destroy(cpu_t *cp);

  87 static int cpu_sys_stats_ks_update(kstat_t *ksp, int rw);
  88 static int cpu_vm_stats_ks_update(kstat_t *ksp, int rw);
  89 static int cpu_stat_ks_update(kstat_t *ksp, int rw);
  90 static int cpu_state_change_hooks(int, cpu_setup_t, cpu_setup_t);

  92 /*
  93  * cpu_lock protects ncpus, ncpus_online, cpu_flag, cpu_list, cpu_active,
  94  * max_cpu_seqid_ever, and dispatch queue reallocations.  The lock ordering with
  95  * respect to related locks is:
  96  *
  97  *      cpu_lock --> thread_free_lock  --->  p_lock  --->  thread_lock()
  98  *
  99  * Warning:  Certain sections of code do not use the cpu_lock when
 100  * traversing the cpu_list (e.g. mutex_vector_enter(), clock()).  Since
 101  * all cpus are paused during modifications to this list, a solution
 102  * to protect the list is too either disable kernel preemption while
 103  * walking the list, *or* recheck the cpu_next pointer at each
 104  * iteration in the loop.  Note that in no cases can any cached
 105  * copies of the cpu pointers be kept as they may become invalid.
 106  */
 107 kmutex_t        cpu_lock;
 108 cpu_t           *cpu_list;              /* list of all CPUs */
 109 cpu_t           *clock_cpu_list;        /* used by clock to walk CPUs */
 110 cpu_t           *cpu_active;            /* list of active CPUs */
 111 static cpuset_t cpu_available;          /* set of available CPUs */
 112 cpuset_t        cpu_seqid_inuse;        /* which cpu_seqids are in use */

 114 cpu_t           **cpu_seq;              /* ptrs to CPUs, indexed by seq_id */

 116 /*
 117  * max_ncpus keeps the max cpus the system can have. Initially
 118  * it's NCPU, but since most archs scan the devtree for cpus
 119  * fairly early on during boot, the real max can be known before
 120  * ncpus is set (useful for early NCPU based allocations).
 121  */
 122 int max_ncpus = NCPU;
 123 /*
 124  * platforms that set max_ncpus to maxiumum number of cpus that can be
 125  * dynamically added will set boot_max_ncpus to the number of cpus found
 126  * at device tree scan time during boot.
 127  */
```

```
128 int boot_max_ncpus = -1;
129 int boot_ncpus = -1;
130 /*
131  * Maximum possible CPU id.  This can never be >= NCPU since NCPU is
132  * used to size arrays that are indexed by CPU id.
133  */
134 processorid_t max_cpuid = NCPU - 1;

136 /*
137  * Maximum cpu_seqid was given. This number can only grow and never shrink. It
138  * can be used to optimize NCPU loops to avoid going through CPUs which were
139  * never on-line.
140  */
141 processorid_t max_cpu_seqid_ever = 0;

143 int ncpus = 1;
144 int ncpus_online = 1;

146 /*
147  * CPU that we're trying to offline.  Protected by cpu_lock.
148  */
149 cpu_t *cpu_inmotion;

151 /*
152  * Can be raised to suppress further weakbinding, which are instead
153  * satisfied by disabling preemption.  Must be raised/lowered under cpu_lock,
154  * while individual thread weakbinding synchronization is done under thread
155  * lock.
156  */
157 int weakbindingbarrier;

159 /*
160  * Variables used in pause_cpus().
161  */
162 static volatile char safe_list[NCPU];

164 static struct _cpu_pause_info {
165         int             cp_spl;         /* spl saved in pause_cpus() */
166         volatile int    cp_go;          /* Go signal sent after all ready */
167         int             cp_count;       /* # of CPUs to pause */
168         ksema_t         cp_sem;         /* synch pause_cpus & cpu_pause */
169         kthread_id_t    cp_paused;
170         void            *(*cp_func)(void *);
171 #endif /* ! codereview */
172 } cpu_pause_info;

174 static kmutex_t pause_free_mutex;
175 static kcondvar_t pause_free_cv;

170 void *(*cpu_pause_func)(void *) = NULL;


178 static struct cpu_sys_stats_ks_data {
179         kstat_named_t cpu_ticks_idle;
180         kstat_named_t cpu_ticks_user;
181         kstat_named_t cpu_ticks_kernel;
182         kstat_named_t cpu_ticks_wait;
183         kstat_named_t cpu_nsec_idle;
184         kstat_named_t cpu_nsec_user;
185         kstat_named_t cpu_nsec_kernel;
186         kstat_named_t cpu_nsec_dtrace;
187         kstat_named_t cpu_nsec_intr;
188         kstat_named_t cpu_load_intr;
189         kstat_named_t wait_ticks_io;
190         kstat_named_t dtrace_probes;
191         kstat_named_t bread;
```



```
192         kstat_named_t bwrite;
193         kstat_named_t lread;
194         kstat_named_t lwrite;
195         kstat_named_t phread;
196         kstat_named_t phwrite;
197         kstat_named_t pswitch;
198         kstat_named_t trap;
199         kstat_named_t intr;
200         kstat_named_t syscall;
201         kstat_named_t sysread;
202         kstat_named_t syswrite;
203         kstat_named_t sysfork;
204         kstat_named_t sysvfork;
205         kstat_named_t sysexec;
206         kstat_named_t readch;
207         kstat_named_t writech;
208         kstat_named_t rcvint;
209         kstat_named_t xmtint;
210         kstat_named_t mdmint;
211         kstat_named_t rawch;
212         kstat_named_t canch;
213         kstat_named_t outch;
214         kstat_named_t msg;
215         kstat_named_t sema;
216         kstat_named_t namei;
217         kstat_named_t ufsiget;
218         kstat_named_t ufsdirblk;
219         kstat_named_t ufsipage;
220         kstat_named_t ufsinopage;
221         kstat_named_t procovf;
222         kstat_named_t intrthread;
223         kstat_named_t intrblk;
224         kstat_named_t intrunpin;
225         kstat_named_t idlethread;
226         kstat_named_t inv_swtch;
227         kstat_named_t nthreads;
228         kstat_named_t cpumigrate;
229         kstat_named_t xcalls;
230         kstat_named_t mutex_adenters;
231         kstat_named_t rw_rdfails;
232         kstat_named_t rw_wrfails;
233         kstat_named_t modload;
234         kstat_named_t modunload;
235         kstat_named_t bawrite;
236         kstat_named_t iowait;
237 } cpu_sys_stats_ks_data_template = {
_____unchanged_portion_omitted_

751 /*
752  * This routine is called to place the CPUs in a safe place so that
753  * one of them can be taken off line or placed on line.  What we are
754  * trying to do here is prevent a thread from traversing the list
755  * of active CPUs while we are changing it or from getting placed on
756  * the run queue of a CPU that has just gone off line.  We do this by
757  * creating a thread with the highest possible prio for each CPU and
758  * having it call this routine.  The advantage of this method is that
759  * we can eliminate all checks for CPU_ACTIVE in the disp routines.
760  * This makes disp faster at the expense of making p_online() slower
761  * which is a good trade off.
762  */
763 static void
764 cpu_pause(int index)
765 {
766         int s;
767         struct _cpu_pause_info *cpi = &cpu_pause_info;
768         volatile char *safe = &safe_list[index];
```

```
 769         long    lindex = index;

 771         ASSERT((curthread->t_bound_cpu != NULL) || (*safe == PAUSE_DIE));

 773         while (*safe != PAUSE_DIE) {
 774                 *safe = PAUSE_READY;
 775                 membar_enter();         /* make sure stores are flushed */
 776                 sema_v(&cpi->cp_sem);   /* signal requesting thread */

 778                 /*
 779                  * Wait here until all pause threads are running.  That
 780                  * indicates that it's safe to do the spl.  Until
 781                  * cpu_pause_info.cp_go is set, we don't want to spl
 782                  * because that might block clock interrupts needed
 783                  * to preempt threads on other CPUs.
 784                  */
 785                 while (cpi->cp_go == 0)
 786                         ;
 787                 /*
 788                  * Even though we are at the highest disp prio, we need
 789                  * to block out all interrupts below LOCK_LEVEL so that
 790                  * an intr doesn't come in, wake up a thread, and call
 791                  * setbackdq/setfrontdq.
 792                  */
 793                 s = splhigh();
 794                 /*
 795                  * if cp_func has been set then call it using index as the
 796                  * argument, currently only used by cpr_suspend_cpus().
 797                  * This function is used as the code to execute on the
 798                  * "paused" cpu's when a machine comes out of a sleep state
 799                  * and CPU's were powered off.  (could also be used for
 800                  * hotplugging CPU's).
 790                  * if cpu_pause_func() has been set then call it using
 791                  * index as the argument, currently only used by
 792                  * cpr_suspend_cpus().  This function is used as the
 793                  * code to execute on the "paused" cpu's when a machine
 794                  * comes out of a sleep state and CPU's were powered off.
 795                  * (could also be used for hotplugging CPU's).
 801                  */
 802                 if (cpi->cp_func != NULL)
 803                         (*cpi->cp_func)((void *)lindex);
 797                 if (cpu_pause_func != NULL)
 798                         (*cpu_pause_func)((void *)lindex);

 805                 mach_cpu_pause(safe);

 807                 splx(s);
 808                 /*
 809                  * Waiting is at an end. Switch out of cpu_pause
 810                  * loop and resume useful work.
 811                  */
 812                 swtch();
 813         }

 815         mutex_enter(&pause_free_mutex);
 816         *safe = PAUSE_DEAD;
 817         cv_broadcast(&pause_free_cv);
 818         mutex_exit(&pause_free_mutex);
 819 }
```

unchanged_portion_omitted_

```
 974 /*
 975  * Pause all of the CPUs except the one we are on by creating a high
 976  * priority thread bound to those CPUs.
 977  *
```



```
 978  * Note that one must be extremely careful regarding code
 979  * executed while CPUs are paused.  Since a CPU may be paused
 980  * while a thread scheduling on that CPU is holding an adaptive
 981  * lock, code executed with CPUs paused must not acquire adaptive
 982  * (or low-level spin) locks.  Also, such code must not block,
 983  * since the thread that is supposed to initiate the wakeup may
 984  * never run.
 985  *
 986  * With a few exceptions, the restrictions on code executed with CPUs
 987  * paused match those for code executed at high-level interrupt
 988  * context.
 989  */
 990 void
 991 pause_cpus(cpu_t *off_cp, void *(*func)(void *))
 986 pause_cpus(cpu_t *off_cp)
 992 {
 993         processorid_t   cpu_id;
 994         int             i;
 995         struct _cpu_pause_info  *cpi = &cpu_pause_info;

 997         ASSERT(MUTEX_HELD(&cpu_lock));
 998         ASSERT(cpi->cp_paused == NULL);
 999         cpi->cp_count = 0;
1000         cpi->cp_go = 0;
1001         for (i = 0; i < NCPU; i++)
1002                 safe_list[i] = PAUSE_IDLE;
1003         kpreempt_disable();

1005         cpi->cp_func = func;

1007 #endif /* ! codereview */
1008         /*
1009          * If running on the cpu that is going offline, get off it.
1010          * This is so that it won't be necessary to rechoose a CPU
1011          * when done.
1012          */
1013         if (CPU == off_cp)
1014                 cpu_id = off_cp->cpu_next_part->cpu_id;
1015         else
1016                 cpu_id = CPU->cpu_id;
1017         affinity_set(cpu_id);

1019         /*
1020          * Start the pause threads and record how many were started
1021          */
1022         cpi->cp_count = cpu_pause_start(cpu_id);

1024         /*
1025          * Now wait for all CPUs to be running the pause thread.
1026          */
1027         while (cpi->cp_count > 0) {
1028                 /*
1029                  * Spin reading the count without grabbing the disp
1030                  * lock to make sure we don't prevent the pause
1031                  * threads from getting the lock.
1032                  */
1033                 while (sema_held(&cpi->cp_sem))
1034                         ;
1035                 if (sema_tryp(&cpi->cp_sem))
1036                         --cpi->cp_count;
1037         }
1038         cpi->cp_go = 1;                 /* all have reached cpu_pause */

1040         /*
1041          * Now wait for all CPUs to spl. (Transition from PAUSE_READY
1042          * to PAUSE_WAIT.)
```

```
1043          */
1044         for (i = 0; i < NCPU; i++) {
1045                 while (safe_list[i] != PAUSE_WAIT)
1046                         ;
1047         }
1048         cpi->cp_spl = splhigh();        /* block dispatcher on this CPU */
1049         cpi->cp_paused = curthread;
1050 }

1052 /*
1053  * Check whether the current thread has CPUs paused
1054  */
1055 int
1056 cpus_paused(void)
1057 {
1058         if (cpu_pause_info.cp_paused != NULL) {
1059                 ASSERT(cpu_pause_info.cp_paused == curthread);
1060                 return (1);
1061         }
1062         return (0);
1063 }

1065 static cpu_t *
1066 cpu_get_all(processorid_t cpun)
1067 {
1068         ASSERT(MUTEX_HELD(&cpu_lock));

1070         if (cpun >= NCPU || cpun < 0 || !CPU_IN_SET(cpu_available, cpun))
1071                 return (NULL);
1072         return (cpu[cpun]);
1073 }

1075 /*
1076  * Check whether cpun is a valid processor id and whether it should be
1077  * visible from the current zone. If it is, return a pointer to the
1078  * associated CPU structure.
1079  */
1080 cpu_t *
1081 cpu_get(processorid_t cpun)
1082 {
1083         cpu_t *c;

1085         ASSERT(MUTEX_HELD(&cpu_lock));
1086         c = cpu_get_all(cpun);
1087         if (c != NULL && !INGLOBALZONE(curproc) && pool_pset_enabled() &&
1088             zone_pset_get(curproc->p_zone) != cpupart_query_cpu(c))
1089                 return (NULL);
1090         return (c);
1091 }

1093 /*
1094  * The following functions should be used to check CPU states in the kernel.
1095  * They should be invoked with cpu_lock held.  Kernel subsystems interested
1096  * in CPU states should *not* use cpu_get_state() and various P_ONLINE/etc
1097  * states.  Those are for user-land (and system call) use only.
1098  */

1100 /*
1101  * Determine whether the CPU is online and handling interrupts.
1102  */
1103 int
1104 cpu_is_online(cpu_t *cpu)
1105 {
1106         ASSERT(MUTEX_HELD(&cpu_lock));
1107         return (cpu_flagged_online(cpu->cpu_flags));
1108 }
```



```
1110 /*
1111  * Determine whether the CPU is offline (this includes spare and faulted).
1112  */
1113 int
1114 cpu_is_offline(cpu_t *cpu)
1115 {
1116         ASSERT(MUTEX_HELD(&cpu_lock));
1117         return (cpu_flagged_offline(cpu->cpu_flags));
1118 }

1120 /*
1121  * Determine whether the CPU is powered off.
1122  */
1123 int
1124 cpu_is_poweredoff(cpu_t *cpu)
1125 {
1126         ASSERT(MUTEX_HELD(&cpu_lock));
1127         return (cpu_flagged_poweredoff(cpu->cpu_flags));
1128 }

1130 /*
1131  * Determine whether the CPU is handling interrupts.
1132  */
1133 int
1134 cpu_is_nointr(cpu_t *cpu)
1135 {
1136         ASSERT(MUTEX_HELD(&cpu_lock));
1137         return (cpu_flagged_nointr(cpu->cpu_flags));
1138 }

1140 /*
1141  * Determine whether the CPU is active (scheduling threads).
1142  */
1143 int
1144 cpu_is_active(cpu_t *cpu)
1145 {
1146         ASSERT(MUTEX_HELD(&cpu_lock));
1147         return (cpu_flagged_active(cpu->cpu_flags));
1148 }

1150 /*
1151  * Same as above, but these require cpu_flags instead of cpu_t pointers.
1152  */
1153 int
1154 cpu_flagged_online(cpu_flag_t cpu_flags)
1155 {
1156         return (cpu_flagged_active(cpu_flags) &&
1157             (cpu_flags & CPU_ENABLE));
1158 }

1160 int
1161 cpu_flagged_offline(cpu_flag_t cpu_flags)
1162 {
1163         return (((cpu_flags & CPU_POWEROFF) == 0) &&
1164             ((cpu_flags & (CPU_READY | CPU_OFFLINE)) != CPU_READY));
1165 }

1167 int
1168 cpu_flagged_poweredoff(cpu_flag_t cpu_flags)
1169 {
1170         return ((cpu_flags & CPU_POWEROFF) == CPU_POWEROFF);
1171 }

1173 int
1174 cpu_flagged_nointr(cpu_flag_t cpu_flags)
```

```
1175 {
1176         return (cpu_flagged_active(cpu_flags) &&
1177             (cpu_flags & CPU_ENABLE) == 0);
1178 }

1180 int
1181 cpu_flagged_active(cpu_flag_t cpu_flags)
1182 {
1183         return (((cpu_flags & (CPU_POWEROFF | CPU_FAULTED | CPU_SPARE)) == 0) &&
1184             ((cpu_flags & (CPU_READY | CPU_OFFLINE)) == CPU_READY));
1185 }

1187 /*
1188  * Bring the indicated CPU online.
1189  */
1190 int
1191 cpu_online(cpu_t *cp)
1192 {
1193         int     error = 0;

1195         /*
1196          * Handle on-line request.
1197          *      This code must put the new CPU on the active list before
1198          *      starting it because it will not be paused, and will start
1199          *      using the active list immediately.  The real start occurs
1200          *      when the CPU_QUIESCED flag is turned off.
1201          */

1203         ASSERT(MUTEX_HELD(&cpu_lock));

1205         /*
1206          * Put all the cpus into a known safe place.
1207          * No mutexes can be entered while CPUs are paused.
1208          */
1209         error = mp_cpu_start(cp);       /* arch-dep hook */
1210         if (error == 0) {
1211                 pg_cpupart_in(cp, cp->cpu_part);
1212                 pause_cpus(NULL, NULL);
1000                 pause_cpus(NULL);
1213                 cpu_add_active_internal(cp);
1214                 if (cp->cpu_flags & CPU_FAULTED) {
1215                         cp->cpu_flags &= ~CPU_FAULTED;
1216                         mp_cpu_faulted_exit(cp);
1217                 }
1218                 cp->cpu_flags &= ~(CPU_QUIESCED | CPU_OFFLINE | CPU_FROZEN |
1219                     CPU_SPARE);
1220                 CPU_NEW_GENERATION(cp);
1221                 start_cpus();
1222                 cpu_stats_kstat_create(cp);
1223                 cpu_create_intrstat(cp);
1224                 lgrp_kstat_create(cp);
1225                 cpu_state_change_notify(cp->cpu_id, CPU_ON);
1226                 cpu_intr_enable(cp);    /* arch-dep hook */
1227                 cpu_state_change_notify(cp->cpu_id, CPU_INTR_ON);
1228                 cpu_set_state(cp);
1229                 cyclic_online(cp);
1230                 /*
1231                  * This has to be called only after cyclic_online(). This
1232                  * function uses cyclics.
1233                  */
1234                 callout_cpu_online(cp);
1235                 poke_cpu(cp->cpu_id);
1236         }

1238         return (error);
1239 }
```



```
1241 /*
1242  * Take the indicated CPU offline.
1243  */
1244 int
1245 cpu_offline(cpu_t *cp, int flags)
1246 {
1247         cpupart_t *pp;
1248         int     error = 0;
1249         cpu_t   *ncp;
1250         int     intr_enable;
1251         int     cyclic_off = 0;
1252         int     callout_off = 0;
1253         int     loop_count;
1254         int     no_quiesce = 0;
1255         int     (*bound_func)(struct cpu *, int);
1256         kthread_t *t;
1257         lpl_t   *cpu_lpl;
1258         proc_t  *p;
1259         int     lgrp_diff_lpl;
1260         boolean_t unbind_all_threads = (flags & CPU_FORCED) != 0;

1262         ASSERT(MUTEX_HELD(&cpu_lock));

1264         /*
1265          * If we're going from faulted or spare to offline, just
1266          * clear these flags and update CPU state.
1267          */
1268         if (cp->cpu_flags & (CPU_FAULTED | CPU_SPARE)) {
1269                 if (cp->cpu_flags & CPU_FAULTED) {
1270                         cp->cpu_flags &= ~CPU_FAULTED;
1271                         mp_cpu_faulted_exit(cp);
1272                 }
1273                 cp->cpu_flags &= ~CPU_SPARE;
1274                 cpu_set_state(cp);
1275                 return (0);
1276         }

1278         /*
1279          * Handle off-line request.
1280          */
1281         pp = cp->cpu_part;
1282         /*
1283          * Don't offline last online CPU in partition
1284          */
1285         if (ncpus_online <= 1 || pp->cp_ncpus <= 1 || cpu_intr_count(cp) < 2)
1286                 return (EBUSY);
1287         /*
1288          * Unbind all soft-bound threads bound to our CPU and hard bound threads
1289          * if we were asked to.
1290          */
1291         error = cpu_unbind(cp->cpu_id, unbind_all_threads);
1292         if (error != 0)
1293                 return (error);
1294         /*
1295          * We shouldn't be bound to this CPU ourselves.
1296          */
1297         if (curthread->t_bound_cpu == cp)
1298                 return (EBUSY);

1300         /*
1301          * Tell interested parties that this CPU is going offline.
1302          */
1303         CPU_NEW_GENERATION(cp);
1304         cpu_state_change_notify(cp->cpu_id, CPU_OFF);
```

```
1306         /*
1307          * Tell the PG subsystem that the CPU is leaving the partition
1308          */
1309         pg_cpupart_out(cp, pp);

1311         /*
1312          * Take the CPU out of interrupt participation so we won't find
1313          * bound kernel threads.  If the architecture cannot completely
1314          * shut off interrupts on the CPU, don't quiesce it, but don't
1315          * run anything but interrupt thread... this is indicated by
1316          * the CPU_OFFLINE flag being on but the CPU_QUIESCE flag being
1317          * off.
1318          */
1319         intr_enable = cp->cpu_flags & CPU_ENABLE;
1320         if (intr_enable)
1321                 no_quiesce = cpu_intr_disable(cp);

1323         /*
1324          * Record that we are aiming to offline this cpu.  This acts as
1325          * a barrier to further weak binding requests in thread_nomigrate
1326          * and also causes cpu_choose, disp_lowpri_cpu and setfrontdq to
1327          * lean away from this cpu.  Further strong bindings are already
1328          * avoided since we hold cpu_lock.  Since threads that are set
1329          * runnable around now and others coming off the target cpu are
1330          * directed away from the target, existing strong and weak bindings
1331          * (especially the latter) to the target cpu stand maximum chance of
1332          * being able to unbind during the short delay loop below (if other
1333          * unbound threads compete they may not see cpu in time to unbind
1334          * even if they would do so immediately.
1335          */
1336         cpu_inmotion = cp;
1337         membar_enter();

1339         /*
1340          * Check for kernel threads (strong or weak) bound to that CPU.
1341          * Strongly bound threads may not unbind, and we'll have to return
1342          * EBUSY.  Weakly bound threads should always disappear - we've
1343          * stopped more weak binding with cpu_inmotion and existing
1344          * bindings will drain imminently (they may not block).  Nonetheless
1345          * we will wait for a fixed period for all bound threads to disappear.
1346          * Inactive interrupt threads are OK (they'll be in TS_FREE
1347          * state).  If test finds some bound threads, wait a few ticks
1348          * to give short-lived threads (such as interrupts) chance to
1349          * complete.  Note that if no_quiesce is set, i.e. this cpu
1350          * is required to service interrupts, then we take the route
1351          * that permits interrupt threads to be active (or bypassed).
1352          */
1353         bound_func = no_quiesce ? disp_bound_threads : disp_bound_anythreads;

1355 again:  for (loop_count = 0; (*bound_func)(cp, 0); loop_count++) {
1356                 if (loop_count >= 5) {
1357                         error = EBUSY;  /* some threads still bound */
1358                         break;
1359                 }

1361                 /*
1362                  * If some threads were assigned, give them
1363                  * a chance to complete or move.
1364                  *
1365                  * This assumes that the clock_thread is not bound
1366                  * to any CPU, because the clock_thread is needed to
1367                  * do the delay(hz/100).
1368                  *
1369                  * Note: we still hold the cpu_lock while waiting for
1370                  * the next clock tick.  This is OK since it isn't
1371                  * needed for anything else except processor_bind(2),
```



```
1372                  * and system initialization.  If we drop the lock,
1373                  * we would risk another p_online disabling the last
1374                  * processor.
1375                  */
1376                 delay(hz/100);
1377         }

1379         if (error == 0 && callout_off == 0) {
1380                 callout_cpu_offline(cp);
1381                 callout_off = 1;
1382         }

1384         if (error == 0 && cyclic_off == 0) {
1385                 if (!cyclic_offline(cp)) {
1386                         /*
1387                          * We must have bound cyclics...
1388                          */
1389                         error = EBUSY;
1390                         goto out;
1391                 }
1392                 cyclic_off = 1;
1393         }

1395         /*
1396          * Call mp_cpu_stop() to perform any special operations
1397          * needed for this machine architecture to offline a CPU.
1398          */
1399         if (error == 0)
1400                 error = mp_cpu_stop(cp);        /* arch-dep hook */

1402         /*
1403          * If that all worked, take the CPU offline and decrement
1404          * ncpus_online.
1405          */
1406         if (error == 0) {
1407                 /*
1408                  * Put all the cpus into a known safe place.
1409                  * No mutexes can be entered while CPUs are paused.
1410                  */
1411                 pause_cpus(cp, NULL);
1199                 pause_cpus(cp);
1412                 /*
1413                  * Repeat the operation, if necessary, to make sure that
1414                  * all outstanding low-level interrupts run to completion
1415                  * before we set the CPU_QUIESCED flag.  It's also possible
1416                  * that a thread has weak bound to the cpu despite our raising
1417                  * cpu_inmotion above since it may have loaded that
1418                  * value before the barrier became visible (this would have
1419                  * to be the thread that was on the target cpu at the time
1420                  * we raised the barrier).
1421                  */
1422                 if ((!no_quiesce && cp->cpu_intr_actv != 0) ||
1423                     (*bound_func)(cp, 1)) {
1424                         start_cpus();
1425                         (void) mp_cpu_start(cp);
1426                         goto again;
1427                 }
1428                 ncp = cp->cpu_next_part;
1429                 cpu_lpl = cp->cpu_lpl;
1430                 ASSERT(cpu_lpl != NULL);

1432                 /*
1433                  * Remove the CPU from the list of active CPUs.
1434                  */
1435                 cpu_remove_active(cp);
```

```
1437                 /*
1438                  * Walk the active process list and look for threads
1439                  * whose home lgroup needs to be updated, or
1440                  * the last CPU they run on is the one being offlined now.
1441                  */

1443                 ASSERT(curthread->t_cpu != cp);
1444                 for (p = practive; p != NULL; p = p->p_next) {

1446                         t = p->p_tlist;

1448                         if (t == NULL)
1449                                 continue;

1451                         lgrp_diff_lpl = 0;

1453                         do {
1454                                 ASSERT(t->t_lpl != NULL);
1455                                 /*
1456                                  * Taking last CPU in lpl offline
1457                                  * Rehome thread if it is in this lpl
1458                                  * Otherwise, update the count of how many
1459                                  * threads are in this CPU's lgroup but have
1460                                  * a different lpl.
1461                                  */

1463                                 if (cpu_lpl->lpl_ncpu == 0) {
1464                                         if (t->t_lpl == cpu_lpl)
1465                                                 lgrp_move_thread(t,
1466                                                     lgrp_choose(t,
1467                                                     t->t_cpupart), 0);
1468                                         else if (t->t_lpl->lpl_lgrpid ==
1469                                             cpu_lpl->lpl_lgrpid)
1470                                                 lgrp_diff_lpl++;
1471                                 }
1472                                 ASSERT(t->t_lpl->lpl_ncpu > 0);

1474                                 /*
1475                                  * Update CPU last ran on if it was this CPU
1476                                  */
1477                                 if (t->t_cpu == cp && t->t_bound_cpu != cp)
1478                                         t->t_cpu = disp_lowpri_cpu(ncp,
1479                                             t->t_lpl, t->t_pri, NULL);
1480                                 ASSERT(t->t_cpu != cp || t->t_bound_cpu == cp ||
1481                                     t->t_weakbound_cpu == cp);

1483                                 t = t->t_forw;
1484                         } while (t != p->p_tlist);

1486                         /*
1487                          * Didn't find any threads in the same lgroup as this
1488                          * CPU with a different lpl, so remove the lgroup from
1489                          * the process lgroup bitmask.
1490                          */

1492                         if (lgrp_diff_lpl == 0)
1493                                 klgrpset_del(p->p_lgrpset, cpu_lpl->lpl_lgrpid);
1494                 }

1496                 /*
1497                  * Walk thread list looking for threads that need to be
1498                  * rehomed, since there are some threads that are not in
1499                  * their process's p_tlist.
1500                  */

1502                 t = curthread;
```



```
1503                 do {
1504                         ASSERT(t != NULL && t->t_lpl != NULL);

1506                         /*
1507                          * Rehome threads with same lpl as this CPU when this
1508                          * is the last CPU in the lpl.
1509                          */

1511                         if ((cpu_lpl->lpl_ncpu == 0) && (t->t_lpl == cpu_lpl))
1512                                 lgrp_move_thread(t,
1513                                     lgrp_choose(t, t->t_cpupart), 1);

1515                         ASSERT(t->t_lpl->lpl_ncpu > 0);

1517                         /*
1518                          * Update CPU last ran on if it was this CPU
1519                          */

1521                         if (t->t_cpu == cp && t->t_bound_cpu != cp) {
1522                                 t->t_cpu = disp_lowpri_cpu(ncp,
1523                                     t->t_lpl, t->t_pri, NULL);
1524                         }
1525                         ASSERT(t->t_cpu != cp || t->t_bound_cpu == cp ||
1526                             t->t_weakbound_cpu == cp);
1527                         t = t->t_next;

1529                 } while (t != curthread);
1530                 ASSERT((cp->cpu_flags & (CPU_FAULTED | CPU_SPARE)) == 0);
1531                 cp->cpu_flags |= CPU_OFFLINE;
1532                 disp_cpu_inactive(cp);
1533                 if (!no_quiesce)
1534                         cp->cpu_flags |= CPU_QUIESCED;
1535                 ncpus_online--;
1536                 cpu_set_state(cp);
1537                 cpu_inmotion = NULL;
1538                 start_cpus();
1539                 cpu_stats_kstat_destroy(cp);
1540                 cpu_delete_intrstat(cp);
1541                 lgrp_kstat_destroy(cp);
1542         }

1544 out:
1545         cpu_inmotion = NULL;

1547         /*
1548          * If we failed, re-enable interrupts.
1549          * Do this even if cpu_intr_disable returned an error, because
1550          * it may have partially disabled interrupts.
1551          */
1552         if (error && intr_enable)
1553                 cpu_intr_enable(cp);

1555         /*
1556          * If we failed, but managed to offline the cyclic subsystem on this
1557          * CPU, bring it back online.
1558          */
1559         if (error && cyclic_off)
1560                 cyclic_online(cp);

1562         /*
1563          * If we failed, but managed to offline callouts on this CPU,
1564          * bring it back online.
1565          */
1566         if (error && callout_off)
1567                 callout_cpu_online(cp);
```

```
1569         /*
1570          * If we failed, tell the PG subsystem that the CPU is back
1571          */
1572         pg_cpupart_in(cp, pp);

1574         /*
1575          * If we failed, we need to notify everyone that this CPU is back on.
1576          */
1577         if (error != 0) {
1578                 CPU_NEW_GENERATION(cp);
1579                 cpu_state_change_notify(cp->cpu_id, CPU_ON);
1580                 cpu_state_change_notify(cp->cpu_id, CPU_INTR_ON);
1581         }

1583         return (error);
1584 }
______unchanged_portion_omitted_

1731 /*
1732  * Insert a CPU into the list of available CPUs.
1733  */
1734 void
1735 cpu_add_unit(cpu_t *cp)
1736 {
1737         int seqid;

1739         ASSERT(MUTEX_HELD(&cpu_lock));
1740         ASSERT(cpu_list != NULL);       /* list started in cpu_list_init */

1742         lgrp_config(LGRP_CONFIG_CPU_ADD, (uintptr_t)cp, 0);

1744         /*
1745          * Note: most users of the cpu_list will grab the
1746          * cpu_lock to insure that it isn't modified.  However,
1747          * certain users can't or won't do that.  To allow this
1748          * we pause the other cpus.  Users who walk the list
1749          * without cpu_lock, must disable kernel preemption
1750          * to insure that the list isn't modified underneath
1751          * them.  Also, any cached pointers to cpu structures
1752          * must be revalidated by checking to see if the
1753          * cpu_next pointer points to itself.  This check must
1754          * be done with the cpu_lock held or kernel preemption
1755          * disabled.  This check relies upon the fact that
1756          * old cpu structures are not free'ed or cleared after
1757          * then are removed from the cpu_list.
1758          *
1759          * Note that the clock code walks the cpu list dereferencing
1760          * the cpu_part pointer, so we need to initialize it before
1761          * adding the cpu to the list.
1762          */
1763         cp->cpu_part = &cp_default;
1764         (void) pause_cpus(NULL, NULL);
1552         (void) pause_cpus(NULL);
1765         cp->cpu_next = cpu_list;
1766         cp->cpu_prev = cpu_list->cpu_prev;
1767         cpu_list->cpu_prev->cpu_next = cp;
1768         cpu_list->cpu_prev = cp;
1769         start_cpus();

1771         for (seqid = 0; CPU_IN_SET(cpu_seqid_inuse, seqid); seqid++)
1772                 continue;
1773         CPUSET_ADD(cpu_seqid_inuse, seqid);
1774         cp->cpu_seqid = seqid;

1776         if (seqid > max_cpu_seqid_ever)
1777                 max_cpu_seqid_ever = seqid;
```



```
1779         ASSERT(ncpus < max_ncpus);
1780         ncpus++;
1781         cp->cpu_cache_offset = KMEM_CPU_CACHE_OFFSET(cp->cpu_seqid);
1782         cpu[cp->cpu_id] = cp;
1783         CPUSET_ADD(cpu_available, cp->cpu_id);
1784         cpu_seq[cp->cpu_seqid] = cp;

1786         /*
1787          * allocate a pause thread for this CPU.
1788          */
1789         cpu_pause_alloc(cp);

1791         /*
1792          * So that new CPUs won't have NULL prev_onln and next_onln pointers,
1793          * link them into a list of just that CPU.
1794          * This is so that disp_lowpri_cpu will work for thread_create in
1795          * pause_cpus() when called from the startup thread in a new CPU.
1796          */
1797         cp->cpu_next_onln = cp;
1798         cp->cpu_prev_onln = cp;
1799         cpu_info_kstat_create(cp);
1800         cp->cpu_next_part = cp;
1801         cp->cpu_prev_part = cp;

1803         init_cpu_mstate(cp, CMS_SYSTEM);

1805         pool_pset_mod = gethrtime();
1806 }

1808 /*
1809  * Do the opposite of cpu_add_unit().
1810  */
1811 void
1812 cpu_del_unit(int cpuid)
1813 {
1814         struct cpu      *cp, *cpnext;

1816         ASSERT(MUTEX_HELD(&cpu_lock));
1817         cp = cpu[cpuid];
1818         ASSERT(cp != NULL);

1820         ASSERT(cp->cpu_next_onln == cp);
1821         ASSERT(cp->cpu_prev_onln == cp);
1822         ASSERT(cp->cpu_next_part == cp);
1823         ASSERT(cp->cpu_prev_part == cp);

1825         /*
1826          * Tear down the CPU's physical ID cache, and update any
1827          * processor groups
1828          */
1829         pg_cpu_fini(cp, NULL);
1830         pghw_physid_destroy(cp);

1832         /*
1833          * Destroy kstat stuff.
1834          */
1835         cpu_info_kstat_destroy(cp);
1836         term_cpu_mstate(cp);
1837         /*
1838          * Free up pause thread.
1839          */
1840         cpu_pause_free(cp);
1841         CPUSET_DEL(cpu_available, cp->cpu_id);
1842         cpu[cp->cpu_id] = NULL;
1843         cpu_seq[cp->cpu_seqid] = NULL;
```

```
1845         /*
1846          * The clock thread and mutex_vector_enter cannot hold the
1847          * cpu_lock while traversing the cpu list, therefore we pause
1848          * all other threads by pausing the other cpus. These, and any
1849          * other routines holding cpu pointers while possibly sleeping
1850          * must be sure to call kpreempt_disable before processing the
1851          * list and be sure to check that the cpu has not been deleted
1852          * after any sleeps (check cp->cpu_next != NULL). We guarantee
1853          * to keep the deleted cpu structure around.
1854          *
1855          * Note that this MUST be done AFTER cpu_available
1856          * has been updated so that we don't waste time
1857          * trying to pause the cpu we're trying to delete.
1858          */
1859         (void) pause_cpus(NULL, NULL);
1647         (void) pause_cpus(NULL);

1861         cpnext = cp->cpu_next;
1862         cp->cpu_prev->cpu_next = cp->cpu_next;
1863         cp->cpu_next->cpu_prev = cp->cpu_prev;
1864         if (cp == cpu_list)
1865                 cpu_list = cpnext;

1867         /*
1868          * Signals that the cpu has been deleted (see above).
1869          */
1870         cp->cpu_next = NULL;
1871         cp->cpu_prev = NULL;

1873         start_cpus();

1875         CPUSET_DEL(cpu_seqid_inuse, cp->cpu_seqid);
1876         ncpus--;
1877         lgrp_config(LGRP_CONFIG_CPU_DEL, (uintptr_t)cp, 0);

1879         pool_pset_mod = gethrtime();
1880 }
```

_____***unchanged_portion_omitted_***

```
1922 /*
1923  * Add a CPU to the list of active CPUs.
1924  *      This is called from machine-dependent layers when a new CPU is started.
1925  */
1926 void
1927 cpu_add_active(cpu_t *cp)
1928 {
1929         pg_cpupart_in(cp, cp->cpu_part);

1931         pause_cpus(NULL, NULL);
1719         pause_cpus(NULL);
1932         cpu_add_active_internal(cp);
1933         start_cpus();

1935         cpu_stats_kstat_create(cp);
1936         cpu_create_intrstat(cp);
1937         lgrp_kstat_create(cp);
1938         cpu_state_change_notify(cp->cpu_id, CPU_INIT);
1939 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   30588 Fri Jan  3 22:11:53 2014
new/usr/src/uts/common/os/cpu_event.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 368 static void
 369 cpu_idle_insert_callback(cpu_idle_cb_impl_t *cip)
 370 {
 371         int unlock = 0, unpause = 0;
 372         int i, cnt_new = 0, cnt_old = 0;
 373         char *buf_new = NULL, *buf_old = NULL;

 375         ASSERT(MUTEX_HELD(&cpu_idle_cb_lock));

 377         /*
 378          * Expand array if it's full.
 379          * Memory must be allocated out of pause/start_cpus() scope because
 380          * kmem_zalloc() can't be called with KM_SLEEP flag within that scope.
 381          */
 382         if (cpu_idle_cb_curr == cpu_idle_cb_max) {
 383                 cnt_new = cpu_idle_cb_max + CPU_IDLE_ARRAY_CAPACITY_INC;
 384                 buf_new = (char *)kmem_zalloc(cnt_new *
 385                     sizeof (cpu_idle_cb_item_t), KM_SLEEP);
 386         }

 388         /* Try to acquire cpu_lock if not held yet. */
 389         if (!MUTEX_HELD(&cpu_lock)) {
 390                 mutex_enter(&cpu_lock);
 391                 unlock = 1;
 392         }
 393         /*
 394          * Pause all other CPUs (and let them run pause thread).
 395          * It's guaranteed that no other threads will access cpu_idle_cb_array
 396          * after pause_cpus().
 397          */
 398         if (!cpus_paused()) {
 399                 pause_cpus(NULL, NULL);
 399                 pause_cpus(NULL);
 400                 unpause = 1;
 401         }

 403         /* Copy content to new buffer if needed. */
 404         if (buf_new != NULL) {
 405                 buf_old = (char *)cpu_idle_cb_array;
 406                 cnt_old = cpu_idle_cb_max;
 407                 if (buf_old != NULL) {
 408                         ASSERT(cnt_old != 0);
 409                         bcopy(cpu_idle_cb_array, buf_new,
 410                             sizeof (cpu_idle_cb_item_t) * cnt_old);
 411                 }
 412                 cpu_idle_cb_array = (cpu_idle_cb_item_t *)buf_new;
 413                 cpu_idle_cb_max = cnt_new;
 414         }

 416         /* Insert into array according to priority. */
 417         ASSERT(cpu_idle_cb_curr < cpu_idle_cb_max);
 418         for (i = cpu_idle_cb_curr; i > 0; i--) {
 419                 if (cpu_idle_cb_array[i - 1].impl->priority >= cip->priority) {
 420                         break;
 421                 }
 422                 cpu_idle_cb_array[i] = cpu_idle_cb_array[i - 1];
 423         }
 424         cpu_idle_cb_array[i].arg = cip->argument;
 425         cpu_idle_cb_array[i].enter = cip->callback->idle_enter;
```



```
 426         cpu_idle_cb_array[i].exit = cip->callback->idle_exit;
 427         cpu_idle_cb_array[i].impl = cip;
 428         cpu_idle_cb_curr++;

 430         /* Resume other CPUs from paused state if needed. */
 431         if (unpause) {
 432                 start_cpus();
 433         }
 434         if (unlock) {
 435                 mutex_exit(&cpu_lock);
 436         }

 438         /* Free old resource if needed. */
 439         if (buf_old != NULL) {
 440                 ASSERT(cnt_old != 0);
 441                 kmem_free(buf_old, cnt_old * sizeof (cpu_idle_cb_item_t));
 442         }
 443 }

 445 static void
 446 cpu_idle_remove_callback(cpu_idle_cb_impl_t *cip)
 447 {
 448         int i, found = 0;
 449         int unlock = 0, unpause = 0;
 450         cpu_idle_cb_state_t *sp;

 452         ASSERT(MUTEX_HELD(&cpu_idle_cb_lock));

 454         /* Try to acquire cpu_lock if not held yet. */
 455         if (!MUTEX_HELD(&cpu_lock)) {
 456                 mutex_enter(&cpu_lock);
 457                 unlock = 1;
 458         }
 459         /*
 460          * Pause all other CPUs.
 461          * It's guaranteed that no other threads will access cpu_idle_cb_array
 462          * after pause_cpus().
 463          */
 464         if (!cpus_paused()) {
 465                 pause_cpus(NULL, NULL);
 465                 pause_cpus(NULL);
 466                 unpause = 1;
 467         }

 469         /* Remove cip from array. */
 470         for (i = 0; i < cpu_idle_cb_curr; i++) {
 471                 if (found == 0) {
 472                         if (cpu_idle_cb_array[i].impl == cip) {
 473                                 found = 1;
 474                         }
 475                 } else {
 476                         cpu_idle_cb_array[i - 1] = cpu_idle_cb_array[i];
 477                 }
 478         }
 479         ASSERT(found != 0);
 480         cpu_idle_cb_curr--;

 482         /*
 483          * Reset property ready flag for all CPUs if no registered callback
 484          * left because cpu_idle_enter/exit will stop updating property if
 485          * there's no callback registered.
 486          */
 487         if (cpu_idle_cb_curr == 0) {
 488                 for (sp = cpu_idle_cb_state, i = 0; i < max_ncpus; i++, sp++) {
 489                         sp->v.ready = B_FALSE;
 490                 }
```

```
 491         }

 493         /* Resume other CPUs from paused state if needed. */
 494         if (unpause) {
 495                 start_cpus();
 496         }
 497         if (unlock) {
 498                 mutex_exit(&cpu_lock);
 499         }
 500 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   21527 Fri Jan  3 22:11:53 2014
new/usr/src/uts/common/os/cpu_pm.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 172 int
 173 cpupm_set_policy(cpupm_policy_t new_policy)
 174 {
 175         static int      gov_init = 0;
 176         int             result = 0;

 178         mutex_enter(&cpu_lock);
 179         if (new_policy == cpupm_policy) {
 180                 mutex_exit(&cpu_lock);
 181                 return (result);
 182         }

 184         /*
 185          * Pausing CPUs causes a high priority thread to be scheduled
 186          * on all other CPUs (besides the current one). This locks out
 187          * other CPUs from making CPUPM state transitions.
 188          */
 189         switch (new_policy) {
 190         case CPUPM_POLICY_DISABLED:
 191                 pause_cpus(NULL, NULL);
 191                 pause_cpus(NULL);
 192                 cpupm_policy = CPUPM_POLICY_DISABLED;
 193                 start_cpus();

 195                 result = cmt_pad_disable(PGHW_POW_ACTIVE);

 197                 /*
 198                  * Once PAD has been enabled, it should always be possible
 199                  * to disable it.
 200                  */
 201                 ASSERT(result == 0);

 203                 /*
 204                  * Bring all the active power domains to the maximum
 205                  * performance state.
 206                  */
 207                 cpupm_state_change_global(CPUPM_DTYPE_ACTIVE,
 208                     CPUPM_STATE_MAX_PERF);

 210                 break;
 211         case CPUPM_POLICY_ELASTIC:

 213                 result = cmt_pad_enable(PGHW_POW_ACTIVE);
 214                 if (result < 0) {
 215                         /*
 216                          * Failed to enable PAD across the active power
 217                          * domains, which may well be because none were
 218                          * enumerated.
 219                          */
 220                         break;
 221                 }

 223                 /*
 224                  * Initialize the governor parameters the first time through.
 225                  */
 226                 if (gov_init == 0) {
 227                         cpupm_governor_initialize();
 228                         gov_init = 1;
 229                 }
```



```
 231                 pause_cpus(NULL, NULL);
 231                 pause_cpus(NULL);
 232                 cpupm_policy = CPUPM_POLICY_ELASTIC;
 233                 start_cpus();

 235                 break;
 236         default:
 237                 cmn_err(CE_WARN, "Attempt to set unknown CPUPM policy %d\n",
 238                     new_policy);
 239                 ASSERT(0);
 240                 break;
 241         }
 242         mutex_exit(&cpu_lock);

 244         return (result);
 245 }
______unchanged_portion_omitted_
```

```
**********************************************************
  119440 Fri Jan  3 22:11:53 2014
new/usr/src/uts/common/os/lgrp.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

1225 /*
1226  * Called to indicate that the lgrp with platform handle "hand" now
1227  * contains the memory identified by "mnode".
1228  *
1229  * LOCKING for this routine is a bit tricky. Usually it is called without
1230  * cpu_lock and it must must grab cpu_lock here to prevent racing with other
1231  * callers. During DR of the board containing the caged memory it may be called
1232  * with cpu_lock already held and CPUs paused.
1233  *
1234  * If the insertion is part of the DR copy-rename and the inserted mnode (and
1235  * only this mnode) is already present in the lgrp_root->lgrp_mnodes set, we are
1236  * dealing with the special case of DR copy-rename described in
1237  * lgrp_mem_rename().
1238  */
1239 void
1240 lgrp_mem_init(int mnode, lgrp_handle_t hand, boolean_t is_copy_rename)
1241 {
1242         klgrpset_t      changed;
1243         int             count;
1244         int             i;
1245         lgrp_t          *my_lgrp;
1246         lgrp_id_t       lgrpid;
1247         mnodeset_t      mnodes_mask = ((mnodeset_t)1 << mnode);
1248         boolean_t       drop_lock = B_FALSE;
1249         boolean_t       need_synch = B_FALSE;

1251         /*
1252          * Grab CPU lock (if we haven't already)
1253          */
1254         if (!MUTEX_HELD(&cpu_lock)) {
1255                 mutex_enter(&cpu_lock);
1256                 drop_lock = B_TRUE;
1257         }

1259         /*
1260          * This routine may be called from a context where we already
1261          * hold cpu_lock, and have already paused cpus.
1262          */
1263         if (!cpus_paused())
1264                 need_synch = B_TRUE;

1266         /*
1267          * Check if this mnode is already configured and return immediately if
1268          * it is.
1269          *
1270          * NOTE: in special case of copy-rename of the only remaining mnode,
1271          * lgrp_mem_fini() refuses to remove the last mnode from the root, so we
1272          * recognize this case and continue as usual, but skip the update to
1273          * the lgrp_mnodes and the lgrp_nmnodes. This restores the inconsistency
1274          * in topology, temporarily introduced by lgrp_mem_fini().
1275          */
1276         if (! (is_copy_rename && (lgrp_root->lgrp_mnodes == mnodes_mask)) &&
1277             lgrp_root->lgrp_mnodes & mnodes_mask) {
1278                 if (drop_lock)
1279                         mutex_exit(&cpu_lock);
1280                 return;
1281         }

1283         /*
```



```
1284          * Update lgroup topology with new memory resources, keeping track of
1285          * which lgroups change
1286          */
1287         count = 0;
1288         klgrpset_clear(changed);
1289         my_lgrp = lgrp_hand_to_lgrp(hand);
1290         if (my_lgrp == NULL) {
1291                 /* new lgrp */
1292                 my_lgrp = lgrp_create();
1293                 lgrpid = my_lgrp->lgrp_id;
1294                 my_lgrp->lgrp_plathand = hand;
1295                 my_lgrp->lgrp_latency = lgrp_plat_latency(hand, hand);
1296                 klgrpset_add(my_lgrp->lgrp_leaves, lgrpid);
1297                 klgrpset_add(my_lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);

1299                 if (need_synch)
1300                         pause_cpus(NULL, NULL);
1300                         pause_cpus(NULL);
1301                 count = lgrp_leaf_add(my_lgrp, lgrp_table, lgrp_alloc_max + 1,
1302                     &changed);
1303                 if (need_synch)
1304                         start_cpus();
1305         } else if (my_lgrp->lgrp_latency == 0 && lgrp_plat_latency(hand, hand)
1306             > 0) {
1307                 /*
1308                  * Leaf lgroup was created, but latency wasn't available
1309                  * then.  So, set latency for it and fill in rest of lgroup
1310                  * topology  now that we know how far it is from other leaf
1311                  * lgroups.
1312                  */
1313                 klgrpset_clear(changed);
1314                 lgrpid = my_lgrp->lgrp_id;
1315                 if (!klgrpset_ismember(my_lgrp->lgrp_set[LGRP_RSRC_MEM],
1316                     lgrpid))
1317                         klgrpset_add(my_lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
1318                 if (need_synch)
1319                         pause_cpus(NULL, NULL);
1319                         pause_cpus(NULL);
1320                 count = lgrp_leaf_add(my_lgrp, lgrp_table, lgrp_alloc_max + 1,
1321                     &changed);
1322                 if (need_synch)
1323                         start_cpus();
1324         } else if (!klgrpset_ismember(my_lgrp->lgrp_set[LGRP_RSRC_MEM],
1325             my_lgrp->lgrp_id)) {
1326                 /*
1327                  * Add new lgroup memory resource to existing lgroup
1328                  */
1329                 lgrpid = my_lgrp->lgrp_id;
1330                 klgrpset_add(my_lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
1331                 klgrpset_add(changed, lgrpid);
1332                 count++;
1333                 for (i = 0; i <= lgrp_alloc_max; i++) {
1334                         lgrp_t          *lgrp;

1336                         lgrp = lgrp_table[i];
1337                         if (!LGRP_EXISTS(lgrp) ||
1338                             !lgrp_rsets_member(lgrp->lgrp_set, lgrpid))
1339                                 continue;

1341                         klgrpset_add(lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
1342                         klgrpset_add(changed, lgrp->lgrp_id);
1343                         count++;
1344                 }
1345         }

1347         /*
```

```
1348          * Add memory node to lgroup and remove lgroup from ones that need
1349          * to be updated
1350          */
1351         if (!(my_lgrp->lgrp_mnodes & mnodes_mask)) {
1352                 my_lgrp->lgrp_mnodes |= mnodes_mask;
1353                 my_lgrp->lgrp_nmnodes++;
1354         }
1355         klgrpset_del(changed, lgrpid);

1357         /*
1358          * Update memory node information for all lgroups that changed and
1359          * contain new memory node as a resource
1360          */
1361         if (count)
1362                 (void) lgrp_mnode_update(changed, NULL);

1364         if (drop_lock)
1365                 mutex_exit(&cpu_lock);
1366 }

1368 /*
1369  * Called to indicate that the lgroup associated with the platform
1370  * handle "hand" no longer contains given memory node
1371  *
1372  * LOCKING for this routine is a bit tricky. Usually it is called without
1373  * cpu_lock and it must must grab cpu_lock here to prevent racing with other
1374  * callers. During DR of the board containing the caged memory it may be called
1375  * with cpu_lock already held and CPUs paused.
1376  *
1377  * If the deletion is part of the DR copy-rename and the deleted mnode is the
1378  * only one present in the lgrp_root->lgrp_mnodes, all the topology is updated,
1379  * but lgrp_root->lgrp_mnodes is left intact. Later, lgrp_mem_init() will insert
1380  * the same mnode back into the topology. See lgrp_mem_rename() and
1381  * lgrp_mem_init() for additional details.
1382  */
1383 void
1384 lgrp_mem_fini(int mnode, lgrp_handle_t hand, boolean_t is_copy_rename)
1385 {
1386         klgrpset_t      changed;
1387         int             count;
1388         int             i;
1389         lgrp_t          *my_lgrp;
1390         lgrp_id_t       lgrpid;
1391         mnodeset_t      mnodes_mask;
1392         boolean_t       drop_lock = B_FALSE;
1393         boolean_t       need_synch = B_FALSE;

1395         /*
1396          * Grab CPU lock (if we haven't already)
1397          */
1398         if (!MUTEX_HELD(&cpu_lock)) {
1399                 mutex_enter(&cpu_lock);
1400                 drop_lock = B_TRUE;
1401         }

1403         /*
1404          * This routine may be called from a context where we already
1405          * hold cpu_lock and have already paused cpus.
1406          */
1407         if (!cpus_paused())
1408                 need_synch = B_TRUE;

1410         my_lgrp = lgrp_hand_to_lgrp(hand);

1412         /*
1413          * The lgrp *must* be pre-existing
```



```
1414          */
1415         ASSERT(my_lgrp != NULL);

1417         /*
1418          * Delete memory node from lgroups which contain it
1419          */
1420         mnodes_mask = ((mnodeset_t)1 << mnode);
1421         for (i = 0; i <= lgrp_alloc_max; i++) {
1422                 lgrp_t *lgrp = lgrp_table[i];
1423                 /*
1424                  * Skip any non-existent lgroups and any lgroups that don't
1425                  * contain leaf lgroup of memory as a memory resource
1426                  */
1427                 if (!LGRP_EXISTS(lgrp) ||
1428                     !(lgrp->lgrp_mnodes & mnodes_mask))
1429                         continue;

1431                 /*
1432                  * Avoid removing the last mnode from the root in the DR
1433                  * copy-rename case. See lgrp_mem_rename() for details.
1434                  */
1435                 if (is_copy_rename &&
1436                     (lgrp == lgrp_root) && (lgrp->lgrp_mnodes == mnodes_mask))
1437                         continue;

1439                 /*
1440                  * Remove memory node from lgroup.
1441                  */
1442                 lgrp->lgrp_mnodes &= ~mnodes_mask;
1443                 lgrp->lgrp_nmnodes--;
1444                 ASSERT(lgrp->lgrp_nmnodes >= 0);
1445         }
1446         ASSERT(lgrp_root->lgrp_nmnodes > 0);

1448         /*
1449          * Don't need to update lgroup topology if this lgroup still has memory.
1450          *
1451          * In the special case of DR copy-rename with the only mnode being
1452          * removed, the lgrp_mnodes for the root is always non-zero, but we
1453          * still need to update the lgroup topology.
1454          */
1455         if ((my_lgrp->lgrp_nmnodes > 0) &&
1456             !(is_copy_rename && (my_lgrp == lgrp_root) &&
1457             (my_lgrp->lgrp_mnodes == mnodes_mask))) {
1458                 if (drop_lock)
1459                         mutex_exit(&cpu_lock);
1460                 return;
1461         }

1463         /*
1464          * This lgroup does not contain any memory now
1465          */
1466         klgrpset_clear(my_lgrp->lgrp_set[LGRP_RSRC_MEM]);

1468         /*
1469          * Remove this lgroup from lgroup topology if it does not contain any
1470          * resources now
1471          */
1472         lgrpid = my_lgrp->lgrp_id;
1473         count = 0;
1474         klgrpset_clear(changed);
1475         if (lgrp_rsets_empty(my_lgrp->lgrp_set)) {
1476                 /*
1477                  * Delete lgroup when no more resources
1478                  */
1479                 if (need_synch)
```

```
1480                        pause_cpus(NULL, NULL);
1480                        pause_cpus(NULL);
1481                count = lgrp_leaf_delete(my_lgrp, lgrp_table,
1482                    lgrp_alloc_max + 1, &changed);
1483                ASSERT(count > 0);
1484                if (need_synch)
1485                        start_cpus();
1486        } else {
1487                /*
1488                 * Remove lgroup from memory resources of any lgroups that
1489                 * contain it as such
1490                 */
1491                for (i = 0; i <= lgrp_alloc_max; i++) {
1492                        lgrp_t          *lgrp;

1494                        lgrp = lgrp_table[i];
1495                        if (!LGRP_EXISTS(lgrp) ||
1496                            !klgrpset_ismember(lgrp->lgrp_set[LGRP_RSRC_MEM],
1497                            lgrpid))
1498                                continue;

1500                        klgrpset_del(lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
1501                }
1502        }
1503        if (drop_lock)
1504                mutex_exit(&cpu_lock);
1505 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   36945 Fri Jan  3 22:11:54 2014
new/usr/src/uts/common/os/lgrp_topo.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_


1448 /*
1449  * Update lgroup topology for any leaves that don't have their latency set
1450  *
1451  * This may happen on some machines when the lgroup platform support doesn't
1452  * know the latencies between nodes soon enough to provide it when the
1453  * resources are being added.  If the lgroup platform code needs to probe
1454  * memory to determine the latencies between nodes, it must wait until the
1455  * CPUs become active so at least one CPU in each node can probe memory in
1456  * each node.
1457  */
1458 int
1459 lgrp_topo_update(lgrp_t **lgrps, int lgrp_count, klgrpset_t *changed)
1460 {
1461         klgrpset_t      changes;
1462         int             count;
1463         int             i;
1464         lgrp_t          *lgrp;

1466         count = 0;
1467         if (changed)
1468                 klgrpset_clear(*changed);

1470         /*
1471          * For UMA machines, make sure that root lgroup contains all
1472          * resources.  The root lgrp should also name itself as its own leaf
1473          */
1474         if (nlgrps == 1) {
1475                 for (i = 0; i < LGRP_RSRC_COUNT; i++)
1476                         klgrpset_add(lgrp_root->lgrp_set[i],
1477                             lgrp_root->lgrp_id);
1478                 klgrpset_add(lgrp_root->lgrp_leaves, lgrp_root->lgrp_id);
1479                 return (0);
1480         }

1482         mutex_enter(&cpu_lock);
1483         pause_cpus(NULL, NULL);
1483         pause_cpus(NULL);

1485         /*
1486          * Look for any leaf lgroup without its latency set, finish adding it
1487          * to the lgroup topology assuming that it exists and has the root
1488          * lgroup as its parent, and update the memory nodes of all lgroups
1489          * that have it as a memory resource.
1490          */
1491         for (i = 0; i < lgrp_count; i++) {
1492                 lgrp = lgrps[i];

1494                 /*
1495                  * Skip non-existent and non-leaf lgroups and any lgroup
1496                  * with its latency set already
1497                  */
1498                 if (lgrp == NULL || lgrp->lgrp_id == LGRP_NONE ||
1499                     lgrp->lgrp_childcnt != 0 || lgrp->lgrp_latency != 0)
1500                         continue;

1502 #ifdef  DEBUG
1503                 if (lgrp_topo_debug > 1) {
1504                         prom_printf("\nlgrp_topo_update: updating lineage "
```



```
1505                             "of lgrp %d at 0x%p\n", lgrp->lgrp_id,
1506                             (void *)lgrp);
1507                 }
1508 #endif  /* DEBUG */

1510                 count += lgrp_leaf_add(lgrp, lgrps, lgrp_count, &changes);
1511                 if (changed)
1512                         klgrpset_or(*changed, changes);

1514                 if (!klgrpset_isempty(changes))
1515                         (void) lgrp_mnode_update(changes, NULL);

1517 #ifdef  DEBUG
1518                 if (lgrp_topo_debug > 1 && changed)
1519                         prom_printf("lgrp_topo_update: changed %d lgrps: "
1520                             "0x%llx\n",
1521                             count, (u_longlong_t)*changed);
1522 #endif  /* DEBUG */
1523         }

1525         if (lgrp_topo_levels < LGRP_TOPO_LEVELS && lgrp_topo_levels == 2) {
1526                 count += lgrp_topo_flatten(2, lgrps, lgrp_count, changed);
1527                 (void) lpl_topo_flatten(2);
1528         }

1530         start_cpus();
1531         mutex_exit(&cpu_lock);

1533         return (count);
1534 }
______unchanged_portion_omitted_
```

```
**********************************************************
   82758 Fri Jan  3 22:11:54 2014
new/usr/src/uts/common/os/mem_config.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

3295 /*
3296  * Invalidate memseg pointers in cpu private vm data caches.
3297  */
3298 static void
3299 memseg_cpu_vm_flush()
3300 {
3301         cpu_t *cp;
3302         vm_cpu_data_t *vc;

3304         mutex_enter(&cpu_lock);
3305         pause_cpus(NULL, NULL);
3305         pause_cpus(NULL);

3307         cp = cpu_list;
3308         do {
3309                 vc = cp->cpu_vm_data;
3310                 vc->vc_pnum_memseg = NULL;
3311                 vc->vc_pnext_memseg = NULL;

3313         } while ((cp = cp->cpu_next) != cpu_list);

3315         start_cpus();
3316         mutex_exit(&cpu_lock);
3317 }
______unchanged_portion_omitted_
```

```
**********************************************************
   30244 Fri Jan  3 22:11:54 2014
new/usr/src/uts/common/sys/cpuvar.h
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 611 #define CPU_STATS(cp, stat)                                     \
 612         ((cp)->cpu_stats.stat)

 614 /*
 615  * Increment CPU generation value.
 616  * This macro should be called whenever CPU goes on-line or off-line.
 617  * Updates to cpu_generation should be protected by cpu_lock.
 618  */
 619 #define CPU_NEW_GENERATION(cp)  ((cp)->cpu_generation++)

 621 #endif /* _KERNEL || _KMEMUSER */

 623 /*
 624  * CPU support routines.
 625  */
 626 #if     defined(_KERNEL) && defined(__STDC__)   /* not for genassym.c */

 628 struct zone;

 630 void    cpu_list_init(cpu_t *);
 631 void    cpu_add_unit(cpu_t *);
 632 void    cpu_del_unit(int cpuid);
 633 void    cpu_add_active(cpu_t *);
 634 void    cpu_kstat_init(cpu_t *);
 635 void    cpu_visibility_add(cpu_t *, struct zone *);
 636 void    cpu_visibility_remove(cpu_t *, struct zone *);
 637 void    cpu_visibility_configure(cpu_t *, struct zone *);
 638 void    cpu_visibility_unconfigure(cpu_t *, struct zone *);
 639 void    cpu_visibility_online(cpu_t *, struct zone *);
 640 void    cpu_visibility_offline(cpu_t *, struct zone *);
 641 void    cpu_create_intrstat(cpu_t *);
 642 void    cpu_delete_intrstat(cpu_t *);
 643 int     cpu_kstat_intrstat_update(kstat_t *, int);
 644 void    cpu_intr_swtch_enter(kthread_t *);
 645 void    cpu_intr_swtch_exit(kthread_t *);

 647 void    mbox_lock_init(void);   /* initialize cross-call locks */
 648 void    mbox_init(int cpun);    /* initialize cross-calls */
 649 void    poke_cpu(int cpun);     /* interrupt another CPU (to preempt) */

 651 /*
 652  * values for safe_list.  Pause state that CPUs are in.
 653  */
 654 #define PAUSE_IDLE      0               /* normal state */
 655 #define PAUSE_READY     1               /* paused thread ready to spl */
 656 #define PAUSE_WAIT      2               /* paused thread is spl-ed high */
 657 #define PAUSE_DIE       3               /* tell pause thread to leave */
 658 #define PAUSE_DEAD      4               /* pause thread has left */

 660 void    mach_cpu_pause(volatile char *);

 662 void    pause_cpus(cpu_t *off_cp, void *(*func)(void *));
 662 void    pause_cpus(cpu_t *off_cp);
 663 void    start_cpus(void);
 664 int     cpus_paused(void);

 666 void    cpu_pause_init(void);
 667 cpu_t   *cpu_get(processorid_t cpun);   /* get the CPU struct associated */
```



```
 669 int     cpu_online(cpu_t *cp);                  /* take cpu online */
 670 int     cpu_offline(cpu_t *cp, int flags);      /* take cpu offline */
 671 int     cpu_spare(cpu_t *cp, int flags);        /* take cpu to spare */
 672 int     cpu_faulted(cpu_t *cp, int flags);      /* take cpu to faulted */
 673 int     cpu_poweron(cpu_t *cp);         /* take powered-off cpu to offline */
 674 int     cpu_poweroff(cpu_t *cp);        /* take offline cpu to powered-off */

 676 cpu_t   *cpu_intr_next(cpu_t *cp);      /* get next online CPU taking intrs */
 677 int     cpu_intr_count(cpu_t *cp);      /* count # of CPUs handling intrs */
 678 int     cpu_intr_on(cpu_t *cp);         /* CPU taking I/O interrupts? */
 679 void    cpu_intr_enable(cpu_t *cp);     /* enable I/O interrupts */
 680 int     cpu_intr_disable(cpu_t *cp);    /* disable I/O interrupts */
 681 void    cpu_intr_alloc(cpu_t *cp, int n); /* allocate interrupt threads */

 683 /*
 684  * Routines for checking CPU states.
 685  */
 686 int     cpu_is_online(cpu_t *);         /* check if CPU is online */
 687 int     cpu_is_nointr(cpu_t *);         /* check if CPU can service intrs */
 688 int     cpu_is_active(cpu_t *);         /* check if CPU can run threads */
 689 int     cpu_is_offline(cpu_t *);        /* check if CPU is offline */
 690 int     cpu_is_poweredoff(cpu_t *);     /* check if CPU is powered off */

 692 int     cpu_flagged_online(cpu_flag_t); /* flags show CPU is online */
 693 int     cpu_flagged_nointr(cpu_flag_t); /* flags show CPU not handling intrs */
 694 int     cpu_flagged_active(cpu_flag_t); /* flags show CPU scheduling threads */
 695 int     cpu_flagged_offline(cpu_flag_t); /* flags show CPU is offline */
 696 int     cpu_flagged_poweredoff(cpu_flag_t); /* flags show CPU is powered off */

 698 /*
 699  * The processor_info(2) state of a CPU is a simplified representation suitable
 700  * for use by an application program.  Kernel subsystems should utilize the
 701  * internal per-CPU state as given by the cpu_flags member of the cpu structure,
 702  * as this information may include platform- or architecture-specific state
 703  * critical to a subsystem's disposition of a particular CPU.
 704  */
 705 void    cpu_set_state(cpu_t *);         /* record/timestamp current state */
 706 int     cpu_get_state(cpu_t *);         /* get current cpu state */
 707 const char *cpu_get_state_str(cpu_t *); /* get current cpu state as string */


 710 void    cpu_set_curr_clock(uint64_t);   /* indicate the current CPU's freq */
 711 void    cpu_set_supp_freqs(cpu_t *, const char *); /* set the CPU supported */
 712                                                 /* frequencies */

 714 int     cpu_configure(int);
 715 int     cpu_unconfigure(int);
 716 void    cpu_destroy_bound_threads(cpu_t *cp);

 718 extern int cpu_bind_thread(kthread_t *tp, processorid_t bind,
 719     processorid_t *obind, int *error);
 720 extern int cpu_unbind(processorid_t cpu_id, boolean_t force);
 721 extern void thread_affinity_set(kthread_t *t, int cpu_id);
 722 extern void thread_affinity_clear(kthread_t *t);
 723 extern void affinity_set(int cpu_id);
 724 extern void affinity_clear(void);
 725 extern void init_cpu_mstate(struct cpu *, int);
 726 extern void term_cpu_mstate(struct cpu *);
 727 extern void new_cpu_mstate(int, hrtime_t);
 728 extern void get_cpu_mstate(struct cpu *, hrtime_t *);
 729 extern void thread_nomigrate(void);
 730 extern void thread_allowmigrate(void);
 731 extern void weakbinding_stop(void);
 732 extern void weakbinding_start(void);

 734 /*
```

```
 735  * The following routines affect the CPUs participation in interrupt processing,
 736  * if that is applicable on the architecture.  This only affects interrupts
 737  * which aren't directed at the processor (not cross calls).
 738  *
 739  * cpu_disable_intr returns non-zero if interrupts were previously enabled.
 740  */
 741 int     cpu_disable_intr(struct cpu *cp); /* stop issuing interrupts to cpu */
 742 void    cpu_enable_intr(struct cpu *cp); /* start issuing interrupts to cpu */

 744 /*
 745  * The mutex cpu_lock protects cpu_flags for all CPUs, as well as the ncpus
 746  * and ncpus_online counts.
 747  */
 748 extern kmutex_t cpu_lock;       /* lock protecting CPU data */

 750 /*
 751  * CPU state change events
 752  *
 753  * Various subsystems need to know when CPUs change their state. They get this
 754  * information by registering  CPU state change callbacks using
 755  * register_cpu_setup_func(). Whenever any CPU changes its state, the callback
 756  * function is called. The callback function is passed three arguments:
 757  *
 758  *   Event, described by cpu_setup_t
 759  *   CPU ID
 760  *   Transparent pointer passed when registering the callback
 761  *
 762  * The callback function is called with cpu_lock held. The return value from the
 763  * callback function is usually ignored, except for CPU_CONFIG and CPU_UNCONFIG
 764  * events. For these two events, non-zero return value indicates a failure and
 765  * prevents successful completion of the operation.
 766  *
 767  * New events may be added in the future. Callback functions should ignore any
 768  * events that they do not understand.
 769  *
 770  * The following events provide notification callbacks:
 771  *
 772  *  CPU_INIT    A new CPU is started and added to the list of active CPUs
 773  *                This event is only used during boot
 774  *
 775  *  CPU_CONFIG  A newly inserted CPU is prepared for starting running code
 776  *                This event is called by DR code
 777  *
 778  *  CPU_UNCONFIG CPU has been powered off and needs cleanup
 779  *                This event is called by DR code
 780  *
 781  *  CPU_ON      CPU is enabled but does not run anything yet
 782  *
 783  *  CPU_INTR_ON CPU is enabled and has interrupts enabled
 784  *
 785  *  CPU_OFF     CPU is going offline but can still run threads
 786  *
 787  *  CPU_CPUPART_OUT     CPU is going to move out of its partition
 788  *
 789  *  CPU_CPUPART_IN      CPU is going to move to a new partition
 790  *
 791  *  CPU_SETUP   CPU is set up during boot and can run threads
 792  */
 793 typedef enum {
 794         CPU_INIT,
 795         CPU_CONFIG,
 796         CPU_UNCONFIG,
 797         CPU_ON,
 798         CPU_OFF,
 799         CPU_CPUPART_IN,
 800         CPU_CPUPART_OUT,
```



```
 801         CPU_SETUP,
 802         CPU_INTR_ON
 803 } cpu_setup_t;
```

*unchanged_portion_omitted_*

```
**********************************************************
   18246 Fri Jan  3 22:11:54 2014
new/usr/src/uts/i86pc/i86hvm/io/xpv/xpv_support.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 429 /*
 430  * Top level routine to direct suspend/resume of a domain.
 431  */
 432 void
 433 xen_suspend_domain(void)
 434 {
 435         extern void rtcsync(void);
 436         extern void ec_resume(void);
 437         extern kmutex_t ec_lock;
 438         struct xen_add_to_physmap xatp;
 439         ulong_t flags;
 440         int err;

 442         cmn_err(CE_NOTE, "Domain suspending for save/migrate");

 444         SUSPEND_DEBUG("xen_suspend_domain\n");

 446         /*
 447          * We only want to suspend the PV devices, since the emulated devices
 448          * are suspended by saving the emulated device state.  The PV devices
 449          * are all children of the xpvd nexus device.  So we search the
 450          * device tree for the xpvd node to use as the root of the tree to
 451          * be suspended.
 452          */
 453         if (xpvd_dip == NULL)
 454                 ddi_walk_devs(ddi_root_node(), check_xpvd, NULL);

 456         /*
 457          * suspend interrupts and devices
 458          */
 459         if (xpvd_dip != NULL)
 460                 (void) xen_suspend_devices(ddi_get_child(xpvd_dip));
 461         else
 462                 cmn_err(CE_WARN, "No PV devices found to suspend");
 463         SUSPEND_DEBUG("xenbus_suspend\n");
 464         xenbus_suspend();

 466         mutex_enter(&cpu_lock);

 468         /*
 469          * Suspend on vcpu 0
 470          */
 471         thread_affinity_set(curthread, 0);
 472         kpreempt_disable();

 474         if (ncpus > 1)
 475                 pause_cpus(NULL, NULL);
 475                 pause_cpus(NULL);
 476         /*
 477          * We can grab the ec_lock as it's a spinlock with a high SPL. Hence
 478          * any holder would have dropped it to get through pause_cpus().
 479          */
 480         mutex_enter(&ec_lock);

 482         /*
 483          * From here on in, we can't take locks.
 484          */

 486         flags = intr_clear();
```



```
 488         SUSPEND_DEBUG("HYPERVISOR_suspend\n");
 489         /*
 490          * At this point we suspend and sometime later resume.
 491          * Note that this call may return with an indication of a cancelled
 492          * for now no matter ehat the return we do a full resume of all
 493          * suspended drivers, etc.
 494          */
 495         (void) HYPERVISOR_shutdown(SHUTDOWN_suspend);

 497         /*
 498          * Point HYPERVISOR_shared_info to the proper place.
 499          */
 500         xatp.domid = DOMID_SELF;
 501         xatp.idx = 0;
 502         xatp.space = XENMAPSPACE_shared_info;
 503         xatp.gpfn = xen_shared_info_frame;
 504         if ((err = HYPERVISOR_memory_op(XENMEM_add_to_physmap, &xatp)) != 0)
 505                 panic("Could not set shared_info page. error: %d", err);

 507         SUSPEND_DEBUG("gnttab_resume\n");
 508         gnttab_resume();

 510         SUSPEND_DEBUG("ec_resume\n");
 511         ec_resume();

 513         intr_restore(flags);

 515         if (ncpus > 1)
 516                 start_cpus();

 518         mutex_exit(&ec_lock);
 519         mutex_exit(&cpu_lock);

 521         /*
 522          * Now we can take locks again.
 523          */

 525         rtcsync();

 527         SUSPEND_DEBUG("xenbus_resume\n");
 528         xenbus_resume();
 529         SUSPEND_DEBUG("xen_resume_devices\n");
 530         if (xpvd_dip != NULL)
 531                 (void) xen_resume_devices(ddi_get_child(xpvd_dip), 0);

 533         thread_affinity_clear(curthread);
 534         kpreempt_enable();

 536         SUSPEND_DEBUG("finished xen_suspend_domain\n");

 538         cmn_err(CE_NOTE, "domain restore/migrate completed");
 539 }
______unchanged_portion_omitted_
```

```
**********************************************************
   23453 Fri Jan  3 22:11:54 2014
new/usr/src/uts/i86pc/io/dr/dr_quiesce.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 785 int
 786 dr_suspend(dr_sr_handle_t *srh)
 787 {
 788         dr_handle_t     *handle;
 789         int             force;
 790         int             dev_errs_idx;
 791         uint64_t        dev_errs[DR_MAX_ERR_INT];
 792         int             rc = DDI_SUCCESS;

 794         handle = srh->sr_dr_handlep;

 796         force = dr_cmd_flags(handle) & SBD_FLAG_FORCE;

 798         prom_printf("\nDR: suspending user threads...\n");
 799         srh->sr_suspend_state = DR_SRSTATE_USER;
 800         if (((rc = dr_stop_user_threads(srh)) != DDI_SUCCESS) &&
 801             dr_check_user_stop_result) {
 802                 dr_resume(srh);
 803                 return (rc);
 804         }

 806         if (!force) {
 807                 struct dr_ref drc = {0};

 809                 prom_printf("\nDR: checking devices...\n");
 810                 dev_errs_idx = 0;

 812                 drc.arr = dev_errs;
 813                 drc.idx = &dev_errs_idx;
 814                 drc.len = DR_MAX_ERR_INT;

 816                 /*
 817                  * Since the root node can never go away, it
 818                  * doesn't have to be held.
 819                  */
 820                 ddi_walk_devs(ddi_root_node(), dr_check_unsafe_major, &drc);
 821                 if (dev_errs_idx) {
 822                         handle->h_err = drerr_int(ESBD_UNSAFE, dev_errs,
 823                             dev_errs_idx, 1);
 824                         dr_resume(srh);
 825                         return (DDI_FAILURE);
 826                 }
 827                 PR_QR("done\n");
 828         } else {
 829                 prom_printf("\nDR: dr_suspend invoked with force flag\n");
 830         }

 832 #ifndef SKIP_SYNC
 833         /*
 834          * This sync swap out all user pages
 835          */
 836         vfs_sync(SYNC_ALL);
 837 #endif

 839         /*
 840          * special treatment for lock manager
 841          */
 842         lm_cprsuspend();
```



```
 844 #ifndef SKIP_SYNC
 845         /*
 846          * sync the file system in case we never make it back
 847          */
 848         sync();
 849 #endif

 851         /*
 852          * now suspend drivers
 853          */
 854         prom_printf("DR: suspending drivers...\n");
 855         srh->sr_suspend_state = DR_SRSTATE_DRIVER;
 856         srh->sr_err_idx = 0;
 857         /* No parent to hold busy */
 858         if ((rc = dr_suspend_devices(ddi_root_node(), srh)) != DDI_SUCCESS) {
 859                 if (srh->sr_err_idx && srh->sr_dr_handlep) {
 860                         (srh->sr_dr_handlep)->h_err = drerr_int(ESBD_SUSPEND,
 861                             srh->sr_err_ints, srh->sr_err_idx, 1);
 862                 }
 863                 dr_resume(srh);
 864                 return (rc);
 865         }

 867         drmach_suspend_last();

 869         /*
 870          * finally, grab all cpus
 871          */
 872         srh->sr_suspend_state = DR_SRSTATE_FULL;

 874         mutex_enter(&cpu_lock);
 875         pause_cpus(NULL, NULL);
 875         pause_cpus(NULL);
 876         dr_stop_intr();

 878         return (rc);
 879 }
______unchanged_portion_omitted_
```

```
**********************************************************
   34720 Fri Jan  3 22:11:55 2014
new/usr/src/uts/i86pc/io/pcplusmp/apic.c
patch apic-simplify
patch remove-apic_cr8pri
patch spacing-fix
patch apic-task-reg-write-dup
**********************************************************
______unchanged_portion_omitted_
 137         /*
 138          * The ipl of an ISR at vector X is apic_vectortoipl[X>>4]
 139          * NOTE that this is vector as passed into intr_enter which is
 140          * programmed vector - 0x20 (APIC_BASE_VECT)
 141          */

 143 uchar_t apic_ipltopri[MAXIPL + 1];      /* unix ipl to apic pri */
 144         /* The taskpri to be programmed into apic to mask given ipl */

 146 #if defined(__amd64)
 147 uchar_t apic_cr8pri[MAXIPL + 1];        /* unix ipl to cr8 pri  */
 148 #endif

 146 /*
 147  * Correlation of the hardware vector to the IPL in use, initialized
 148  * from apic_vectortoipl[] in apic_init().  The final IPLs may not correlate
 149  * to the IPLs in apic_vectortoipl on some systems that share interrupt lines
 150  * connected to errata-stricken IOAPICs
 151  */
 152 uchar_t apic_ipls[APIC_AVAIL_VECTOR];

 154 /*
 155  * Patchable global variables.
 156  */
 157 int     apic_enable_hwsoftint = 0;      /* 0 - disable, 1 - enable      */
 158 int     apic_enable_bind_log = 1;       /* 1 - display interrupt binding log */

 160 /*
 161  *      Local static data
 162  */
 163 static struct   psm_ops apic_ops = {
 164         apic_probe,

 166         apic_init,
 167         apic_picinit,
 168         apic_intr_enter,
 169         apic_intr_exit,
 170         apic_setspl,
 171         apic_addspl,
 172         apic_delspl,
 173         apic_disable_intr,
 174         apic_enable_intr,
 175         (int (*)(int))NULL,             /* psm_softlvl_to_irq */
 176         (void (*)(int))NULL,            /* psm_set_softintr */

 178         apic_set_idlecpu,
 179         apic_unset_idlecpu,

 181         apic_clkinit,
 182         apic_getclkirq,
 183         (void (*)(void))NULL,           /* psm_hrtimeinit */
 184         apic_gethrtime,

 186         apic_get_next_processorid,
 187         apic_cpu_start,
 188         apic_post_cpu_start,
 189         apic_shutdown,
```



```
 190         apic_get_ipivect,
 191         apic_send_ipi,

 193         (int (*)(dev_info_t *, int))NULL,       /* psm_translate_irq */
 194         (void (*)(int, char *))NULL,    /* psm_notify_error */
 195         (void (*)(int))NULL,            /* psm_notify_func */
 196         apic_timer_reprogram,
 197         apic_timer_enable,
 198         apic_timer_disable,
 199         apic_post_cyclic_setup,
 200         apic_preshutdown,
 201         apic_intr_ops,                  /* Advanced DDI Interrupt framework */
 202         apic_state,                     /* save, restore apic state for S3 */
 203         apic_cpu_ops,                   /* CPU control interface. */
 204 };
______unchanged_portion_omitted_

 274 void
 275 apic_init(void)
 276 {
 277         int i;
 278         int     j = 1;

 280         psm_get_ioapicid = apic_get_ioapicid;
 281         psm_get_localapicid = apic_get_localapicid;
 282         psm_xlate_vector_by_irq = apic_xlate_vector_by_irq;

 284         apic_ipltopri[0] = APIC_VECTOR_PER_IPL; /* leave 0 for idle */
 285         for (i = 0; i < (APIC_AVAIL_VECTOR / APIC_VECTOR_PER_IPL); i++) {
 286                 if ((i < ((APIC_AVAIL_VECTOR / APIC_VECTOR_PER_IPL) - 1)) &&
 287                     (apic_vectortoipl[i + 1] == apic_vectortoipl[i]))
 288                         /* get to highest vector at the same ipl */
 289                         continue;
 290                 for (; j <= apic_vectortoipl[i]; j++) {
 291                         apic_ipltopri[j] = (i << APIC_IPL_SHIFT) +
 292                             APIC_BASE_VECT;
 293                 }
 294         }
 295         for (; j < MAXIPL + 1; j++)
 296                 /* fill up any empty ipltopri slots */
 297                 apic_ipltopri[j] = (i << APIC_IPL_SHIFT) + APIC_BASE_VECT;
 298         apic_init_common();
 299 #ifndef __amd64
 303 #if defined(__amd64)
 304         /*
 305          * Make cpu-specific interrupt info point to cr8pri vector
 306          */
 307         for (i = 0; i <= MAXIPL; i++)
 308                 apic_cr8pri[i] = apic_ipltopri[i] >> APIC_IPL_SHIFT;
 309         CPU->cpu_pri_data = apic_cr8pri;
 310 #else
 300         if (cpuid_have_cr8access(CPU))
 301                 apic_have_32bit_cr8 = 1;
 302 #endif  /* !__amd64 */
 313 #endif  /* __amd64 */
 303 }
______unchanged_portion_omitted_
 528 #endif /* DEBUG */

 530 /*
 531  * platform_intr_enter
 532  *
 533  *      Called at the beginning of the interrupt service routine to
 534  *      mask all level equal to and below the interrupt priority
 535  *      of the interrupting vector.  An EOI should be given to
 536  *      the interrupt controller to enable other HW interrupts.
```

```
 *
 *      Return -1 for spurious interrupts
 *
 */
/*ARGSUSED*/
static int
apic_intr_enter(int ipl, int *vectorp)
{
        uchar_t vector;
        int nipl;
        int irq;
        ulong_t iflag;
        apic_cpus_info_t *cpu_infop;

        /*
         * The real vector delivered is (*vectorp + 0x20), but our caller
         * subtracts 0x20 from the vector before passing it to us.
         * (That's why APIC_BASE_VECT is 0x20.)
         */
        vector = (uchar_t)*vectorp;

        /* if interrupted by the clock, increment apic_nsec_since_boot */
        if (vector == apic_clkvect) {
                if (!apic_oneshot) {
                        /* NOTE: this is not MT aware */
                        apic_hrtime_stamp++;
                        apic_nsec_since_boot += apic_nsec_per_intr;
                        apic_hrtime_stamp++;
                        last_count_read = apic_hertz_count;
                        apic_redistribute_compute();
                }

                /* We will avoid all the book keeping overhead for clock */
                nipl = apic_ipls[vector];

                *vectorp = apic_vector_to_irq[vector + APIC_BASE_VECT];

                apic_reg_ops->apic_write_task_reg(apic_ipltopri[nipl]);
                apic_reg_ops->apic_send_eoi(0);
                if (apic_mode == LOCAL_APIC) {
#if defined(__amd64)
                        setcr8((ulong_t)(apic_ipltopri[nipl] >>
                            APIC_IPL_SHIFT));
#else
                        if (apic_have_32bit_cr8)
                                setcr8((ulong_t)(apic_ipltopri[nipl] >>
                                    APIC_IPL_SHIFT));
                        else
                                LOCAL_APIC_WRITE_REG(APIC_TASK_REG,
                                    (uint32_t)apic_ipltopri[nipl]);
#endif
                        LOCAL_APIC_WRITE_REG(APIC_EOI_REG, 0);
                } else {
                        X2APIC_WRITE(APIC_TASK_REG, apic_ipltopri[nipl]);
                        X2APIC_WRITE(APIC_EOI_REG, 0);
                }

                return (nipl);
        }

        cpu_infop = &apic_cpus[psm_get_cpu_id()];

        if (vector == (APIC_SPUR_INTR - APIC_BASE_VECT)) {
                cpu_infop->aci_spur_cnt++;
                return (APIC_INT_SPURIOUS);
        }
```



```
        /* Check if the vector we got is really what we need */
        if (apic_revector_pending) {
                /*
                 * Disable interrupts for the duration of
                 * the vector translation to prevent a self-race for
                 * the apic_revector_lock.  This cannot be done
                 * in apic_xlate_vector because it is recursive and
                 * we want the vector translation to be atomic with
                 * respect to other (higher-priority) interrupts.
                 */
                iflag = intr_clear();
                vector = apic_xlate_vector(vector + APIC_BASE_VECT) -
                    APIC_BASE_VECT;
                intr_restore(iflag);
        }

        nipl = apic_ipls[vector];
        *vectorp = irq = apic_vector_to_irq[vector + APIC_BASE_VECT];

        apic_reg_ops->apic_write_task_reg(apic_ipltopri[nipl]);
        if (apic_mode == LOCAL_APIC) {
#if defined(__amd64)
                setcr8((ulong_t)(apic_ipltopri[nipl] >> APIC_IPL_SHIFT));
#else
                if (apic_have_32bit_cr8)
                        setcr8((ulong_t)(apic_ipltopri[nipl] >>
                            APIC_IPL_SHIFT));
                else
                        LOCAL_APIC_WRITE_REG(APIC_TASK_REG,
                            (uint32_t)apic_ipltopri[nipl]);
#endif
        } else {
                X2APIC_WRITE(APIC_TASK_REG, apic_ipltopri[nipl]);
        }

        cpu_infop->aci_current[nipl] = (uchar_t)irq;
        cpu_infop->aci_curipl = (uchar_t)nipl;
        cpu_infop->aci_ISR_in_progress |= 1 << nipl;

        /*
         * apic_level_intr could have been assimilated into the irq struct.
         * but, having it as a character array is more efficient in terms of
         * cache usage. So, we leave it as is.
         */
        if (!apic_level_intr[irq]) {
                apic_reg_ops->apic_send_eoi(0);
                if (apic_mode == LOCAL_APIC) {
                        LOCAL_APIC_WRITE_REG(APIC_EOI_REG, 0);
                } else {
                        X2APIC_WRITE(APIC_EOI_REG, 0);
                }
        }

#ifdef  DEBUG
        APIC_DEBUG_BUF_PUT(vector);
        APIC_DEBUG_BUF_PUT(irq);
        APIC_DEBUG_BUF_PUT(nipl);
        APIC_DEBUG_BUF_PUT(psm_get_cpu_id());
        if ((apic_stretch_interrupts) && (apic_stretch_ISR & (1 << nipl)))
                drv_usecwait(apic_stretch_interrupts);

        if (apic_break_on_cpu == psm_get_cpu_id())
                apic_break();
#endif /* DEBUG */
        return (nipl);
```

```
 633 }
______unchanged_portion_omitted_

 649 /*
 650  * Any changes made to this function must also change X2APIC
 651  * version of intr_exit.
 652  */
 653 void
 654 apic_intr_exit(int prev_ipl, int irq)
 655 {
 656         apic_cpus_info_t *cpu_infop;

 658         local_apic_write_task_reg(apic_ipltopri[prev_ipl]);
 700 #if defined(__amd64)
 701         setcr8((ulong_t)apic_cr8pri[prev_ipl]);
 702 #else
 703         if (apic_have_32bit_cr8)
 704                 setcr8((ulong_t)(apic_ipltopri[prev_ipl] >> APIC_IPL_SHIFT));
 705         else
 706                 apicadr[APIC_TASK_REG] = apic_ipltopri[prev_ipl];
 707 #endif

 660         APIC_INTR_EXIT();
 661 }
______unchanged_portion_omitted_

 685 /*
 686  * Mask all interrupts below or equal to the given IPL.
 687  * Any changes made to this function must also change X2APIC
 688  * version of setspl.
 689  */
 690 static void
 691 apic_setspl(int ipl)
 692 {
 693         local_apic_write_task_reg(apic_ipltopri[ipl]);
 742 #if defined(__amd64)
 743         setcr8((ulong_t)apic_cr8pri[ipl]);
 744 #else
 745         if (apic_have_32bit_cr8)
 746                 setcr8((ulong_t)(apic_ipltopri[ipl] >> APIC_IPL_SHIFT));
 747         else
 748                 apicadr[APIC_TASK_REG] = apic_ipltopri[ipl];
 749 #endif

 695         /* interrupts at ipl above this cannot be in progress */
 696         apic_cpus[psm_get_cpu_id()].aci_ISR_in_progress &= (2 << ipl) - 1;
 697         /*
 698          * this is a patch fix for the ALR QSMP P5 machine, so that interrupts
 699          * have enough time to come in before the priority is raised again
 700          * during the idle() loop.
 701          */
 702         if (apic_setspl_delay)
 703                 (void) apic_reg_ops->apic_get_pri();
 704 }
______unchanged_portion_omitted_
```

```
**********************************************************
    9758 Fri Jan  3 22:11:55 2014
new/usr/src/uts/i86pc/io/pcplusmp/apic_regops.c
patch use-x2apic-feature
patch apic-simplify
patch apic-task-reg-write-dup
**********************************************************
______unchanged_portion_omitted_

  86 int apic_have_32bit_cr8 = 0;

  88 /* The default ops is local APIC (Memory Mapped IO) */
  89 apic_reg_ops_t *apic_reg_ops = &local_apic_regs_ops;

  91 /*
  92  * APIC register ops related data sturctures and functions.
  93  */
  94 void apic_send_EOI();
  95 void apic_send_directed_EOI(uint32_t irq);

  97 #define X2APIC_CPUID_BIT        21
  97 #define X2APIC_ENABLE_BIT       10

  99 /*
 100  * Local APIC Implementation
 101  */
 102 static uint64_t
 103 local_apic_read(uint32_t reg)
 104 {
 105         return ((uint32_t)apicadr[reg]);
 106 }

 108 static void
 109 local_apic_write(uint32_t reg, uint64_t value)
 110 {
 111         LOCAL_APIC_WRITE_REG(reg, (uint32_t)value);
 112         apicadr[reg] = (uint32_t)value;
 112 }
______unchanged_portion_omitted_

 126 static void
 127 local_apic_write_task_reg(uint64_t value)
 128 {
 129 #if defined(__amd64)
 130         setcr8((ulong_t)(value >> APIC_IPL_SHIFT));
 131 #else
 132         if (apic_have_32bit_cr8)
 133                 setcr8((ulong_t)(value >> APIC_IPL_SHIFT));
 134         else
 135                 LOCAL_APIC_WRITE_REG(APIC_TASK_REG, (uint32_t)value);
 136                 apicadr[APIC_TASK_REG] = (uint32_t)value;
 136 #endif
 137 }
______unchanged_portion_omitted_

 232 int
 233 apic_detect_x2apic(void)
 234 {
 236         struct cpuid_regs cp;

 235         if (x2apic_enable == 0)
 236                 return (0);

 238         return is_x86_feature(x86_featureset, X86FSET_X2APIC);
 241         cp.cp_eax = 1;
 242         (void) __cpuid_insn(&cp);
```



```
 244         return ((cp.cp_ecx & (0x1 << X2APIC_CPUID_BIT)) ? 1 : 0);
 239 }
______unchanged_portion_omitted_
```

```
**********************************************************
    4291 Fri Jan  3 22:11:55 2014
new/usr/src/uts/i86pc/io/ppm/acpisleep.c
patch cpu-pause-func-deglobalize
**********************************************************
```

```
/*
 * 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.
 */

#include <sys/types.h>
#include <sys/smp_impldefs.h>
#include <sys/promif.h>

#include <sys/kmem.h>
#include <sys/archsystm.h>
#include <sys/cpuvar.h>
#include <sys/pte.h>
#include <vm/seg_kmem.h>
#include <sys/epm.h>
#include <sys/cpr.h>
#include <sys/machsystm.h>
#include <sys/clock.h>

#include <sys/cpr_wakecode.h>
#include <sys/acpi/acpi.h>

#ifdef OLDPMCODE
#include "acpi.h"
#endif

#include        <sys/x86_archext.h>
#include        <sys/reboot.h>
#include        <sys/cpu_module.h>
#include        <sys/kdi.h>

/*
 * S3 stuff
 */

int acpi_rtc_wake = 0x0;                /* wake in N seconds */

#if 0   /* debug */
static uint8_t  branchbuf[64 * 1024];   /* for the HDT branch trace stuff */
#endif  /* debug */
```



```
extern int boothowto;

#define BOOTCPU 0       /* cpu 0 is always the boot cpu */

extern void             kernel_wc_code(void);
extern tod_ops_t        *tod_ops;
extern int flushes_require_xcalls;
extern int tsc_gethrtime_enable;

extern cpuset_t cpu_ready_set;
extern void *(*cpu_pause_func)(void *);


/*
 * This is what we've all been waiting for!
 */
int
acpi_enter_sleepstate(s3a_t *s3ap)
{
        ACPI_PHYSICAL_ADDRESS   wakephys = s3ap->s3a_wakephys;
        caddr_t                 wakevirt = rm_platter_va;
        /*LINTED*/
        wakecode_t              *wp = (wakecode_t *)wakevirt;
        uint_t                  Sx = s3ap->s3a_state;

        PT(PT_SWV);
        /* Set waking vector */
        if (AcpiSetFirmwareWakingVector(wakephys) != AE_OK) {
                PT(PT_SWV_FAIL);
                PMD(PMD_SX, ("Can't SetFirmwareWakingVector(%lx)\n",
                    (long)wakephys))
                goto insomnia;
        }

        PT(PT_EWE);
        /* Enable wake events */
        if (AcpiEnableEvent(ACPI_EVENT_POWER_BUTTON, 0) != AE_OK) {
                PT(PT_EWE_FAIL);
                PMD(PMD_SX, ("Can't EnableEvent(POWER_BUTTON)\n"))
        }
        if (acpi_rtc_wake > 0) {
                /* clear the RTC bit first */
                (void) AcpiWriteBitRegister(ACPI_BITREG_RT_CLOCK_STATUS, 1);
                PT(PT_RTCW);
                if (AcpiEnableEvent(ACPI_EVENT_RTC, 0) != AE_OK) {
                        PT(PT_RTCW_FAIL);
                        PMD(PMD_SX, ("Can't EnableEvent(RTC)\n"))
                }

                /*
                 * Set RTC to wake us in a wee while.
                 */
                mutex_enter(&tod_lock);
                PT(PT_TOD);
                TODOP_SETWAKE(tod_ops, acpi_rtc_wake);
                mutex_exit(&tod_lock);
        }

        /*
         * Prepare for sleep ... could've done this earlier?
         */
        PT(PT_SXP);
        PMD(PMD_SX, ("Calling AcpiEnterSleepStatePrep(%d) ...\n", Sx))
        if (AcpiEnterSleepStatePrep(Sx) != AE_OK) {
```

```
126                 PMD(PMD_SX, ("... failed\n!"))
127                 goto insomnia;
128         }

130         switch (s3ap->s3a_test_point) {
131         case DEVICE_SUSPEND_TO_RAM:
132         case FORCE_SUSPEND_TO_RAM:
133         case LOOP_BACK_PASS:
134                 return (0);
135         case LOOP_BACK_FAIL:
136                 return (1);
137         default:
138                 ASSERT(s3ap->s3a_test_point == LOOP_BACK_NONE);
139         }

141         /*
142          * Tell the hardware to sleep.
143          */
144         PT(PT_SXE);
145         PMD(PMD_SX, ("Calling AcpiEnterSleepState(%d) ...\n", Sx))
146         if (AcpiEnterSleepState(Sx) != AE_OK) {
147                 PT(PT_SXE_FAIL);
148                 PMD(PMD_SX, ("... failed!\n"))
149         }

151 insomnia:
152         PT(PT_INSOM);
153         /* cleanup is done in the caller */
154         return (1);
155 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   27105 Fri Jan  3 22:11:55 2014
new/usr/src/uts/i86pc/os/cpr_impl.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 724 /*
 725  * Stop all other cpu's before halting or rebooting. We pause the cpu's
 726  * instead of sending a cross call.
 727  * Stolen from sun4/os/mp_states.c
 728  */

 730 static int cpu_are_paused;      /* sic */

 732 void
 733 i_cpr_stop_other_cpus(void)
 734 {
 735         mutex_enter(&cpu_lock);
 736         if (cpu_are_paused) {
 737                 mutex_exit(&cpu_lock);
 738                 return;
 739         }
 740         pause_cpus(NULL, NULL);
 740         pause_cpus(NULL);
 741         cpu_are_paused = 1;

 743         mutex_exit(&cpu_lock);
 744 }
______unchanged_portion_omitted_
```

```
**********************************************************
  122611 Fri Jan  3 22:11:55 2014
new/usr/src/uts/i86pc/os/cpuid.c
patch x2apic-x86fset
patch remove-unused-vars
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011 by Delphix. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2014 Josef "Jeff" Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */
/*
 * Copyright (c) 2010, Intel Corporation.
 * All rights reserved.
 */
/*
 * Portions Copyright 2009 Advanced Micro Devices, Inc.
 */
/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */
/*
 * Various routines to handle identification
 * and classification of x86 processors.
 */

#include <sys/types.h>
#include <sys/archsystm.h>
#include <sys/x86_archext.h>
#include <sys/kmem.h>
#include <sys/systm.h>
#include <sys/cmn_err.h>
#include <sys/sunddi.h>
#include <sys/sunndi.h>
#include <sys/cpuvar.h>
#include <sys/processor.h>
#include <sys/sysmacros.h>
#include <sys/pg.h>
#include <sys/fp.h>
#include <sys/controlregs.h>
#include <sys/bitmap.h>
#include <sys/auxv_386.h>
#include <sys/memnode.h>
#include <sys/pci_cfgspace.h>
```



```
#ifdef __xpv
#include <sys/hypervisor.h>
#else
#include <sys/ontrap.h>
#endif

/*
 * Pass 0 of cpuid feature analysis happens in locore. It contains special code
 * to recognize Cyrix processors that are not cpuid-compliant, and to deal with
 * them accordingly. For most modern processors, feature detection occurs here
 * in pass 1.
 *
 * Pass 1 of cpuid feature analysis happens just at the beginning of mlsetup()
 * for the boot CPU and does the basic analysis that the early kernel needs.
 * x86_featureset is set based on the return value of cpuid_pass1() of the boot
 * CPU.
 *
 * Pass 1 includes:
 *
 *      o Determining vendor/model/family/stepping and setting x86_type and
 *        x86_vendor accordingly.
 *      o Processing the feature flags returned by the cpuid instruction while
 *        applying any workarounds or tricks for the specific processor.
 *      o Mapping the feature flags into Solaris feature bits (X86_*).
 *      o Processing extended feature flags if supported by the processor,
 *        again while applying specific processor knowledge.
 *      o Determining the CMT characteristics of the system.
 *
 * Pass 1 is done on non-boot CPUs during their initialization and the results
 * are used only as a meager attempt at ensuring that all processors within the
 * system support the same features.
 *
 * Pass 2 of cpuid feature analysis happens just at the beginning
 * of startup().  It just copies in and corrects the remainder
 * of the cpuid data we depend on: standard cpuid functions that we didn't
 * need for pass1 feature analysis, and extended cpuid functions beyond the
 * simple feature processing done in pass1.
 *
 * Pass 3 of cpuid analysis is invoked after basic kernel services; in
 * particular kernel memory allocation has been made available. It creates a
 * readable brand string based on the data collected in the first two passes.
 *
 * Pass 4 of cpuid analysis is invoked after post_startup() when all
 * the support infrastructure for various hardware features has been
 * initialized. It determines which processor features will be reported
 * to userland via the aux vector.
 *
 * All passes are executed on all CPUs, but only the boot CPU determines what
 * features the kernel will use.
 *
 * Much of the worst junk in this file is for the support of processors
 * that didn't really implement the cpuid instruction properly.
 *
 * NOTE: The accessor functions (cpuid_get*) are aware of, and ASSERT upon,
 * the pass numbers.  Accordingly, changes to the pass code may require changes
 * to the accessor code.
 */

uint_t x86_vendor = X86_VENDOR_IntelClone;
uint_t x86_type = X86_TYPE_OTHER;
uint_t x86_clflush_size = 0;

uint_t pentiumpro_bug4046376;
uint_t pentiumpro_bug4064495;
```

```
126 uchar_t x86_featureset[BT_SIZEOFMAP(NUM_X86_FEATURES)];

128 static char *x86_feature_names[NUM_X86_FEATURES] = {
129         "lgpg",
130         "tsc",
131         "msr",
132         "mtrr",
133         "pge",
134         "de",
135         "cmov",
136         "mmx",
137         "mca",
138         "pae",
139         "cv8",
140         "pat",
141         "sep",
142         "sse",
143         "sse2",
144         "htt",
145         "asysc",
146         "nx",
147         "sse3",
148         "cx16",
149         "cmp",
150         "tscp",
151         "mwait",
152         "sse4a",
153         "cpuid",
154         "ssse3",
155         "sse4_1",
156         "sse4_2",
157         "1gpg",
158         "clfsh",
159         "64",
160         "aes",
161         "pclmulqdq",
162         "xsave",
163         "avx",
164         "vmx",
165         "svm",
166         "topoext",
167         "f16c",
168         "rdrand",
169         "x2apic",
 69         "rdrand"
170 };
_____unchanged_portion_omitted_

119 uint_t enable486;

219 static size_t xsave_state_size = 0;
220 uint64_t xsave_bv_all = (XFEATURE_LEGACY_FP | XFEATURE_SSE);
221 boolean_t xsave_force_disable = B_FALSE;

223 /*
224  * This is set to platform type we are running on.
225  */
226 static int platform_type = -1;

228 #if !defined(__xpv)
229 /*
230  * Variable to patch if hypervisor platform detection needs to be
231  * disabled (e.g. platform_type will always be HW_NATIVE if this is 0).
232  */
233 int enable_platform_detection = 1;
234 #endif
```



```
236 /*
237  * monitor/mwait info.
238  *
239  * size_actual and buf_actual are the real address and size allocated to get
240  * proper mwait_buf alignement.  buf_actual and size_actual should be passed
241  * to kmem_free().  Currently kmem_alloc() and mwait happen to both use
242  * processor cache-line alignment, but this is not guarantied in the furture.
243  */
244 struct mwait_info {
245         size_t          mon_min;        /* min size to avoid missed wakeups */
246         size_t          mon_max;        /* size to avoid false wakeups */
247         size_t          size_actual;    /* size actually allocated */
248         void            *buf_actual;    /* memory actually allocated */
249         uint32_t        support;        /* processor support of monitor/mwait */
250 };
_____unchanged_portion_omitted_

936 void
937 cpuid_pass1(cpu_t *cpu, uchar_t *featureset)
938 {
939         uint32_t mask_ecx, mask_edx;
940         struct cpuid_info *cpi;
941         struct cpuid_regs *cp;
942         int xcpuid;
943 #if !defined(__xpv)
944         extern int idle_cpu_prefer_mwait;
945 #endif

947         /*
948          * Space statically allocated for BSP, ensure pointer is set
949          */
950         if (cpu->cpu_id == 0) {
951                 if (cpu->cpu_m.mcpu_cpi == NULL)
952                         cpu->cpu_m.mcpu_cpi = &cpuid_info0;
953         }

955         add_x86_feature(featureset, X86FSET_CPUID);

957         cpi = cpu->cpu_m.mcpu_cpi;
958         ASSERT(cpi != NULL);
959         cp = &cpi->cpi_std[0];
960         cp->cp_eax = 0;
961         cpi->cpi_maxeax = __cpuid_insn(cp);
962         {
963                 uint32_t *iptr = (uint32_t *)cpi->cpi_vendorstr;
964                 *iptr++ = cp->cp_ebx;
965                 *iptr++ = cp->cp_edx;
966                 *iptr++ = cp->cp_ecx;
967                 *(char *)&cpi->cpi_vendorstr[12] = '\0';
968         }

970         cpi->cpi_vendor = _cpuid_vendorstr_to_vendorcode(cpi->cpi_vendorstr);
971         x86_vendor = cpi->cpi_vendor; /* for compatibility */

973         /*
974          * Limit the range in case of weird hardware
975          */
976         if (cpi->cpi_maxeax > CPI_MAXEAX_MAX)
977                 cpi->cpi_maxeax = CPI_MAXEAX_MAX;
978         if (cpi->cpi_maxeax < 1)
979                 goto pass1_done;

981         cp = &cpi->cpi_std[1];
982         cp->cp_eax = 1;
983         (void) __cpuid_insn(cp);
```

```
 985         /*
 986          * Extract identifying constants for easy access.
 987          */
 988         cpi->cpi_model = CPI_MODEL(cpi);
 989         cpi->cpi_family = CPI_FAMILY(cpi);

 991         if (cpi->cpi_family == 0xf)
 992                 cpi->cpi_family += CPI_FAMILY_XTD(cpi);

 994         /*
 995          * Beware: AMD uses "extended model" iff base *FAMILY* == 0xf.
 996          * Intel, and presumably everyone else, uses model == 0xf, as
 997          * one would expect (max value means possible overflow).  Sigh.
 998          */

1000         switch (cpi->cpi_vendor) {
1001         case X86_VENDOR_Intel:
1002                 if (IS_EXTENDED_MODEL_INTEL(cpi))
1003                         cpi->cpi_model += CPI_MODEL_XTD(cpi) << 4;
1004                 break;
1005         case X86_VENDOR_AMD:
1006                 if (CPI_FAMILY(cpi) == 0xf)
1007                         cpi->cpi_model += CPI_MODEL_XTD(cpi) << 4;
1008                 break;
1009         default:
1010                 if (cpi->cpi_model == 0xf)
1011                         cpi->cpi_model += CPI_MODEL_XTD(cpi) << 4;
1012                 break;
1013         }

1015         cpi->cpi_step = CPI_STEP(cpi);
1016         cpi->cpi_brandid = CPI_BRANDID(cpi);

1018         /*
1019          * *default* assumptions:
1020          * - believe %edx feature word
1021          * - ignore %ecx feature word
1022          * - 32-bit virtual and physical addressing
1023          */
1024         mask_edx = 0xffffffff;
1025         mask_ecx = 0;

1027         cpi->cpi_pabits = cpi->cpi_vabits = 32;

1029         switch (cpi->cpi_vendor) {
1030         case X86_VENDOR_Intel:
1031                 if (cpi->cpi_family == 5)
1032                         x86_type = X86_TYPE_P5;
1033                 else if (IS_LEGACY_P6(cpi)) {
1034                         x86_type = X86_TYPE_P6;
1035                         pentiumpro_bug4046376 = 1;
 938                         pentiumpro_bug4064495 = 1;
1036                         /*
1037                          * Clear the SEP bit when it was set erroneously
1038                          */
1039                         if (cpi->cpi_model < 3 && cpi->cpi_step < 3)
1040                                 cp->cp_edx &= ~CPUID_INTC_EDX_SEP;
1041                 } else if (IS_NEW_F6(cpi) || cpi->cpi_family == 0xf) {
1042                         x86_type = X86_TYPE_P4;
1043                         /*
1044                          * We don't currently depend on any of the %ecx
1045                          * features until Prescott, so we'll only check
1046                          * this from P4 onwards.  We might want to revisit
1047                          * that idea later.
1048                          */
```



```
1049                         mask_ecx = 0xffffffff;
1050                 } else if (cpi->cpi_family > 0xf)
1051                         mask_ecx = 0xffffffff;
1052                 /*
1053                  * We don't support MONITOR/MWAIT if leaf 5 is not available
1054                  * to obtain the monitor linesize.
1055                  */
1056                 if (cpi->cpi_maxeax < 5)
1057                         mask_ecx &= ~CPUID_INTC_ECX_MON;
1058                 break;
1059         case X86_VENDOR_IntelClone:
1060         default:
1061                 break;
1062         case X86_VENDOR_AMD:
1063 #if defined(OPTERON_ERRATUM_108)
1064                 if (cpi->cpi_family == 0xf && cpi->cpi_model == 0xe) {
1065                         cp->cp_eax = (0xf0f & cp->cp_eax) | 0xc0;
1066                         cpi->cpi_model = 0xc;
1067                 } else
1068 #endif
1069                 if (cpi->cpi_family == 5) {
1070                         /*
1071                          * AMD K5 and K6
1072                          *
1073                          * These CPUs have an incomplete implementation
1074                          * of MCA/MCE which we mask away.
1075                          */
1076                         mask_edx &= ~(CPUID_INTC_EDX_MCE | CPUID_INTC_EDX_MCA);

1078                         /*
1079                          * Model 0 uses the wrong (APIC) bit
1080                          * to indicate PGE.  Fix it here.
1081                          */
1082                         if (cpi->cpi_model == 0) {
1083                                 if (cp->cp_edx & 0x200) {
1084                                         cp->cp_edx &= ~0x200;
1085                                         cp->cp_edx |= CPUID_INTC_EDX_PGE;
1086                                 }
1087                         }

1089                         /*
1090                          * Early models had problems w/ MMX; disable.
1091                          */
1092                         if (cpi->cpi_model < 6)
1093                                 mask_edx &= ~CPUID_INTC_EDX_MMX;
1094                 }

1096                 /*
1097                  * For newer families, SSE3 and CX16, at least, are valid;
1098                  * enable all
1099                  */
1100                 if (cpi->cpi_family >= 0xf)
1101                         mask_ecx = 0xffffffff;
1102                 /*
1103                  * We don't support MONITOR/MWAIT if leaf 5 is not available
1104                  * to obtain the monitor linesize.
1105                  */
1106                 if (cpi->cpi_maxeax < 5)
1107                         mask_ecx &= ~CPUID_INTC_ECX_MON;

1109 #if !defined(__xpv)
1110                 /*
1111                  * Do not use MONITOR/MWAIT to halt in the idle loop on any AMD
1112                  * processors.  AMD does not intend MWAIT to be used in the cpu
1113                  * idle loop on current and future processors.  10h and future
1114                  * AMD processors use more power in MWAIT than HLT.
```

```
1115                  * Pre-family-10h Opterons do not have the MWAIT instruction.
1116                  */
1117                 idle_cpu_prefer_mwait = 0;
1118 #endif

1120                 break;
1121         case X86_VENDOR_TM:
1122                 /*
1123                  * workaround the NT workaround in CMS 4.1
1124                  */
1125                 if (cpi->cpi_family == 5 && cpi->cpi_model == 4 &&
1126                     (cpi->cpi_step == 2 || cpi->cpi_step == 3))
1127                         cp->cp_edx |= CPUID_INTC_EDX_CX8;
1128                 break;
1129         case X86_VENDOR_Centaur:
1130                 /*
1131                  * workaround the NT workarounds again
1132                  */
1133                 if (cpi->cpi_family == 6)
1134                         cp->cp_edx |= CPUID_INTC_EDX_CX8;
1135                 break;
1136         case X86_VENDOR_Cyrix:
1137                 /*
1138                  * We rely heavily on the probing in locore
1139                  * to actually figure out what parts, if any,
1140                  * of the Cyrix cpuid instruction to believe.
1141                  */
1142                 switch (x86_type) {
1143                 case X86_TYPE_CYRIX_486:
1144                         mask_edx = 0;
1145                         break;
1146                 case X86_TYPE_CYRIX_6x86:
1147                         mask_edx = 0;
1148                         break;
1149                 case X86_TYPE_CYRIX_6x86L:
1150                         mask_edx =
1151                             CPUID_INTC_EDX_DE |
1152                             CPUID_INTC_EDX_CX8;
1153                         break;
1154                 case X86_TYPE_CYRIX_6x86MX:
1155                         mask_edx =
1156                             CPUID_INTC_EDX_DE |
1157                             CPUID_INTC_EDX_MSR |
1158                             CPUID_INTC_EDX_CX8 |
1159                             CPUID_INTC_EDX_PGE |
1160                             CPUID_INTC_EDX_CMOV |
1161                             CPUID_INTC_EDX_MMX;
1162                         break;
1163                 case X86_TYPE_CYRIX_GXm:
1164                         mask_edx =
1165                             CPUID_INTC_EDX_MSR |
1166                             CPUID_INTC_EDX_CX8 |
1167                             CPUID_INTC_EDX_CMOV |
1168                             CPUID_INTC_EDX_MMX;
1169                         break;
1170                 case X86_TYPE_CYRIX_MediaGX:
1171                         break;
1172                 case X86_TYPE_CYRIX_MII:
1173                 case X86_TYPE_VIA_CYRIX_III:
1174                         mask_edx =
1175                             CPUID_INTC_EDX_DE |
1176                             CPUID_INTC_EDX_TSC |
1177                             CPUID_INTC_EDX_MSR |
1178                             CPUID_INTC_EDX_CX8 |
1179                             CPUID_INTC_EDX_PGE |
1180                             CPUID_INTC_EDX_CMOV |
```



```
1181                             CPUID_INTC_EDX_MMX;
1182                         break;
1183                 default:
1184                         break;
1185                 }
1186                 break;
1187         }

1189 #if defined(__xpv)
1190         /*
1191          * Do not support MONITOR/MWAIT under a hypervisor
1192          */
1193         mask_ecx &= ~CPUID_INTC_ECX_MON;
1194         /*
1195          * Do not support XSAVE under a hypervisor for now
1196          */
1197         xsave_force_disable = B_TRUE;

1199 #endif  /* __xpv */

1201         if (xsave_force_disable) {
1202                 mask_ecx &= ~CPUID_INTC_ECX_XSAVE;
1203                 mask_ecx &= ~CPUID_INTC_ECX_AVX;
1204                 mask_ecx &= ~CPUID_INTC_ECX_F16C;
1205         }

1207         /*
1208          * Now we've figured out the masks that determine
1209          * which bits we choose to believe, apply the masks
1210          * to the feature words, then map the kernel's view
1211          * of these feature words into its feature word.
1212          */
1213         cp->cp_edx &= mask_edx;
1214         cp->cp_ecx &= mask_ecx;

1216         /*
1217          * apply any platform restrictions (we don't call this
1218          * immediately after __cpuid_insn here, because we need the
1219          * workarounds applied above first)
1220          */
1221         platform_cpuid_mangle(cpi->cpi_vendor, 1, cp);

1223         /*
1224          * fold in overrides from the "eeprom" mechanism
1225          */
1226         cp->cp_edx |= cpuid_feature_edx_include;
1227         cp->cp_edx &= ~cpuid_feature_edx_exclude;

1229         cp->cp_ecx |= cpuid_feature_ecx_include;
1230         cp->cp_ecx &= ~cpuid_feature_ecx_exclude;

1232         if (cp->cp_edx & CPUID_INTC_EDX_PSE) {
1233                 add_x86_feature(featureset, X86FSET_LARGEPAGE);
1234         }
1235         if (cp->cp_edx & CPUID_INTC_EDX_TSC) {
1236                 add_x86_feature(featureset, X86FSET_TSC);
1237         }
1238         if (cp->cp_edx & CPUID_INTC_EDX_MSR) {
1239                 add_x86_feature(featureset, X86FSET_MSR);
1240         }
1241         if (cp->cp_edx & CPUID_INTC_EDX_MTRR) {
1242                 add_x86_feature(featureset, X86FSET_MTRR);
1243         }
1244         if (cp->cp_edx & CPUID_INTC_EDX_PGE) {
1245                 add_x86_feature(featureset, X86FSET_PGE);
1246         }
```

```
1247         if (cp->cp_edx & CPUID_INTC_EDX_CMOV) {
1248                 add_x86_feature(featureset, X86FSET_CMOV);
1249         }
1250         if (cp->cp_edx & CPUID_INTC_EDX_MMX) {
1251                 add_x86_feature(featureset, X86FSET_MMX);
1252         }
1253         if ((cp->cp_edx & CPUID_INTC_EDX_MCE) != 0 &&
1254             (cp->cp_edx & CPUID_INTC_EDX_MCA) != 0) {
1255                 add_x86_feature(featureset, X86FSET_MCA);
1256         }
1257         if (cp->cp_edx & CPUID_INTC_EDX_PAE) {
1258                 add_x86_feature(featureset, X86FSET_PAE);
1259         }
1260         if (cp->cp_edx & CPUID_INTC_EDX_CX8) {
1261                 add_x86_feature(featureset, X86FSET_CX8);
1262         }
1263         if (cp->cp_ecx & CPUID_INTC_ECX_CX16) {
1264                 add_x86_feature(featureset, X86FSET_CX16);
1265         }
1266         if (cp->cp_edx & CPUID_INTC_EDX_PAT) {
1267                 add_x86_feature(featureset, X86FSET_PAT);
1268         }
1269         if (cp->cp_edx & CPUID_INTC_EDX_SEP) {
1270                 add_x86_feature(featureset, X86FSET_SEP);
1271         }
1272         if (cp->cp_edx & CPUID_INTC_EDX_FXSR) {
1273                 /*
1274                  * In our implementation, fxsave/fxrstor
1275                  * are prerequisites before we'll even
1276                  * try and do SSE things.
1277                  */
1278                 if (cp->cp_edx & CPUID_INTC_EDX_SSE) {
1279                         add_x86_feature(featureset, X86FSET_SSE);
1280                 }
1281                 if (cp->cp_edx & CPUID_INTC_EDX_SSE2) {
1282                         add_x86_feature(featureset, X86FSET_SSE2);
1283                 }
1284                 if (cp->cp_ecx & CPUID_INTC_ECX_SSE3) {
1285                         add_x86_feature(featureset, X86FSET_SSE3);
1286                 }
1287                 if (cp->cp_ecx & CPUID_INTC_ECX_SSSE3) {
1288                         add_x86_feature(featureset, X86FSET_SSSE3);
1289                 }
1290                 if (cp->cp_ecx & CPUID_INTC_ECX_SSE4_1) {
1291                         add_x86_feature(featureset, X86FSET_SSE4_1);
1292                 }
1293                 if (cp->cp_ecx & CPUID_INTC_ECX_SSE4_2) {
1294                         add_x86_feature(featureset, X86FSET_SSE4_2);
1295                 }
1296                 if (cp->cp_ecx & CPUID_INTC_ECX_AES) {
1297                         add_x86_feature(featureset, X86FSET_AES);
1298                 }
1299                 if (cp->cp_ecx & CPUID_INTC_ECX_PCLMULQDQ) {
1300                         add_x86_feature(featureset, X86FSET_PCLMULQDQ);
1301                 }

1303                 if (cp->cp_ecx & CPUID_INTC_ECX_XSAVE) {
1304                         add_x86_feature(featureset, X86FSET_XSAVE);

1306                         /* We only test AVX when there is XSAVE */
1307                         if (cp->cp_ecx & CPUID_INTC_ECX_AVX) {
1308                                 add_x86_feature(featureset,
1309                                     X86FSET_AVX);

1311                                 if (cp->cp_ecx & CPUID_INTC_ECX_F16C)
1312                                         add_x86_feature(featureset,
```



```
1313                                             X86FSET_F16C);
1314                         }
1315                 }
1316         }
1317         if (cp->cp_ecx & CPUID_INTC_ECX_X2APIC) {
1318                 add_x86_feature(featureset, X86FSET_X2APIC);
1319         }
1320 #endif /* ! codereview */
1321         if (cp->cp_edx & CPUID_INTC_EDX_DE) {
1322                 add_x86_feature(featureset, X86FSET_DE);
1323         }
1324 #if !defined(__xpv)
1325         if (cp->cp_ecx & CPUID_INTC_ECX_MON) {

1327                 /*
1328                  * We require the CLFLUSH instruction for erratum workaround
1329                  * to use MONITOR/MWAIT.
1330                  */
1331                 if (cp->cp_edx & CPUID_INTC_EDX_CLFSH) {
1332                         cpi->cpi_mwait.support |= MWAIT_SUPPORT;
1333                         add_x86_feature(featureset, X86FSET_MWAIT);
1334                 } else {
1335                         extern int idle_cpu_assert_cflush_monitor;

1337                         /*
1338                          * All processors we are aware of which have
1339                          * MONITOR/MWAIT also have CLFLUSH.
1340                          */
1341                         if (idle_cpu_assert_cflush_monitor) {
1342                                 ASSERT((cp->cp_ecx & CPUID_INTC_ECX_MON) &&
1343                                     (cp->cp_edx & CPUID_INTC_EDX_CLFSH));
1344                         }
1345                 }
1346         }
1347 #endif  /* __xpv */

1349         if (cp->cp_ecx & CPUID_INTC_ECX_VMX) {
1350                 add_x86_feature(featureset, X86FSET_VMX);
1351         }

1353         if (cp->cp_ecx & CPUID_INTC_ECX_RDRAND)
1354                 add_x86_feature(featureset, X86FSET_RDRAND);

1356         /*
1357          * Only need it first time, rest of the cpus would follow suit.
1358          * we only capture this for the bootcpu.
1359          */
1360         if (cp->cp_edx & CPUID_INTC_EDX_CLFSH) {
1361                 add_x86_feature(featureset, X86FSET_CLFSH);
1362                 x86_clflush_size = (BITX(cp->cp_ebx, 15, 8) * 8);
1363         }
1364         if (is_x86_feature(featureset, X86FSET_PAE))
1365                 cpi->cpi_pabits = 36;

1367         /*
1368          * Hyperthreading configuration is slightly tricky on Intel
1369          * and pure clones, and even trickier on AMD.
1370          *
1371          * (AMD chose to set the HTT bit on their CMP processors,
1372          * even though they're not actually hyperthreaded.  Thus it
1373          * takes a bit more work to figure out what's really going
1374          * on ... see the handling of the CMP_LGCY bit below)
1375          */
1376         if (cp->cp_edx & CPUID_INTC_EDX_HTT) {
1377                 cpi->cpi_ncpu_per_chip = CPI_CPU_COUNT(cpi);
1378                 if (cpi->cpi_ncpu_per_chip > 1)
```

```
1379                        add_x86_feature(featureset, X86FSET_HTT);
1380        } else {
1381                cpi->cpi_ncpu_per_chip = 1;
1382        }

1384        /*
1385         * Work on the "extended" feature information, doing
1386         * some basic initialization for cpuid_pass2()
1387         */
1388        xcpuid = 0;
1389        switch (cpi->cpi_vendor) {
1390        case X86_VENDOR_Intel:
1391                if (IS_NEW_F6(cpi) || cpi->cpi_family >= 0xf)
1392                        xcpuid++;
1393                break;
1394        case X86_VENDOR_AMD:
1395                if (cpi->cpi_family > 5 ||
1396                    (cpi->cpi_family == 5 && cpi->cpi_model >= 1))
1397                        xcpuid++;
1398                break;
1399        case X86_VENDOR_Cyrix:
1400                /*
1401                 * Only these Cyrix CPUs are -known- to support
1402                 * extended cpuid operations.
1403                 */
1404                if (x86_type == X86_TYPE_VIA_CYRIX_III ||
1405                    x86_type == X86_TYPE_CYRIX_GXm)
1406                        xcpuid++;
1407                break;
1408        case X86_VENDOR_Centaur:
1409        case X86_VENDOR_TM:
1410        default:
1411                xcpuid++;
1412                break;
1413        }

1415        if (xcpuid) {
1416                cp = &cpi->cpi_extd[0];
1417                cp->cp_eax = 0x80000000;
1418                cpi->cpi_xmaxeax = __cpuid_insn(cp);
1419        }

1421        if (cpi->cpi_xmaxeax & 0x80000000) {

1423                if (cpi->cpi_xmaxeax > CPI_XMAXEAX_MAX)
1424                        cpi->cpi_xmaxeax = CPI_XMAXEAX_MAX;

1426                switch (cpi->cpi_vendor) {
1427                case X86_VENDOR_Intel:
1428                case X86_VENDOR_AMD:
1429                        if (cpi->cpi_xmaxeax < 0x80000001)
1430                                break;
1431                        cp = &cpi->cpi_extd[1];
1432                        cp->cp_eax = 0x80000001;
1433                        (void) __cpuid_insn(cp);

1435                        if (cpi->cpi_vendor == X86_VENDOR_AMD &&
1436                            cpi->cpi_family == 5 &&
1437                            cpi->cpi_model == 6 &&
1438                            cpi->cpi_step == 6) {
1439                                /*
1440                                 * K6 model 6 uses bit 10 to indicate SYSC
1441                                 * Later models use bit 11. Fix it here.
1442                                 */
1443                                if (cp->cp_edx & 0x400) {
1444                                        cp->cp_edx &= ~0x400;
```



```
1445                                        cp->cp_edx |= CPUID_AMD_EDX_SYSC;
1446                                }
1447                        }

1449                        platform_cpuid_mangle(cpi->cpi_vendor, 0x80000001, cp);

1451                        /*
1452                         * Compute the additions to the kernel's feature word.
1453                         */
1454                        if (cp->cp_edx & CPUID_AMD_EDX_NX) {
1455                                add_x86_feature(featureset, X86FSET_NX);
1456                        }

1458                        /*
1459                         * Regardless whether or not we boot 64-bit,
1460                         * we should have a way to identify whether
1461                         * the CPU is capable of running 64-bit.
1462                         */
1463                        if (cp->cp_edx & CPUID_AMD_EDX_LM) {
1464                                add_x86_feature(featureset, X86FSET_64);
1465                        }

1467 #if defined(__amd64)
1468                        /* 1 GB large page - enable only for 64 bit kernel */
1469                        if (cp->cp_edx & CPUID_AMD_EDX_1GPG) {
1470                                add_x86_feature(featureset, X86FSET_1GPG);
1471                        }
1472 #endif

1474                        if ((cpi->cpi_vendor == X86_VENDOR_AMD) &&
1475                            (cpi->cpi_std[1].cp_edx & CPUID_INTC_EDX_FXSR) &&
1476                            (cp->cp_ecx & CPUID_AMD_ECX_SSE4A)) {
1477                                add_x86_feature(featureset, X86FSET_SSE4A);
1478                        }

1480                        /*
1481                         * If both the HTT and CMP_LGCY bits are set,
1482                         * then we're not actually HyperThreaded.  Read
1483                         * "AMD CPUID Specification" for more details.
1484                         */
1485                        if (cpi->cpi_vendor == X86_VENDOR_AMD &&
1486                            is_x86_feature(featureset, X86FSET_HTT) &&
1487                            (cp->cp_ecx & CPUID_AMD_ECX_CMP_LGCY)) {
1488                                remove_x86_feature(featureset, X86FSET_HTT);
1489                                add_x86_feature(featureset, X86FSET_CMP);
1490                        }
1491 #if defined(__amd64)
1492                        /*
1493                         * It's really tricky to support syscall/sysret in
1494                         * the i386 kernel; we rely on sysenter/sysexit
1495                         * instead.  In the amd64 kernel, things are -way-
1496                         * better.
1497                         */
1498                        if (cp->cp_edx & CPUID_AMD_EDX_SYSC) {
1499                                add_x86_feature(featureset, X86FSET_ASYSC);
1500                        }

1502                        /*
1503                         * While we're thinking about system calls, note
1504                         * that AMD processors don't support sysenter
1505                         * in long mode at all, so don't try to program them.
1506                         */
1507                        if (x86_vendor == X86_VENDOR_AMD) {
1508                                remove_x86_feature(featureset, X86FSET_SEP);
1509                        }
1510 #endif
```

```
1511                        if (cp->cp_edx & CPUID_AMD_EDX_TSCP) {
1512                                add_x86_feature(featureset, X86FSET_TSCP);
1513                        }

1515                        if (cp->cp_ecx & CPUID_AMD_ECX_SVM) {
1516                                add_x86_feature(featureset, X86FSET_SVM);
1517                        }

1519                        if (cp->cp_ecx & CPUID_AMD_ECX_TOPOEXT) {
1520                                add_x86_feature(featureset, X86FSET_TOPOEXT);
1521                        }
1522                        break;
1523                default:
1524                        break;
1525                }

1527                /*
1528                 * Get CPUID data about processor cores and hyperthreads.
1529                 */
1530                switch (cpi->cpi_vendor) {
1531                case X86_VENDOR_Intel:
1532                        if (cpi->cpi_maxeax >= 4) {
1533                                cp = &cpi->cpi_std[4];
1534                                cp->cp_eax = 4;
1535                                cp->cp_ecx = 0;
1536                                (void) __cpuid_insn(cp);
1537                                platform_cpuid_mangle(cpi->cpi_vendor, 4, cp);
1538                        }
1539                        /*FALLTHROUGH*/
1540                case X86_VENDOR_AMD:
1541                        if (cpi->cpi_xmaxeax < 0x80000008)
1542                                break;
1543                        cp = &cpi->cpi_extd[8];
1544                        cp->cp_eax = 0x80000008;
1545                        (void) __cpuid_insn(cp);
1546                        platform_cpuid_mangle(cpi->cpi_vendor, 0x80000008, cp);

1548                        /*
1549                         * Virtual and physical address limits from
1550                         * cpuid override previously guessed values.
1551                         */
1552                        cpi->cpi_pabits = BITX(cp->cp_eax, 7, 0);
1553                        cpi->cpi_vabits = BITX(cp->cp_eax, 15, 8);
1554                        break;
1555                default:
1556                        break;
1557                }

1559                /*
1560                 * Derive the number of cores per chip
1561                 */
1562                switch (cpi->cpi_vendor) {
1563                case X86_VENDOR_Intel:
1564                        if (cpi->cpi_maxeax < 4) {
1565                                cpi->cpi_ncore_per_chip = 1;
1566                                break;
1567                        } else {
1568                                cpi->cpi_ncore_per_chip =
1569                                    BITX((cpi)->cpi_std[4].cp_eax, 31, 26) + 1;
1570                        }
1571                        break;
1572                case X86_VENDOR_AMD:
1573                        if (cpi->cpi_xmaxeax < 0x80000008) {
1574                                cpi->cpi_ncore_per_chip = 1;
1575                                break;
1576                        } else {
```



```
1577                                /*
1578                                 * On family 0xf cpuid fn 2 ECX[7:0] "NC" is
1579                                 * 1 less than the number of physical cores on
1580                                 * the chip.  In family 0x10 this value can
1581                                 * be affected by "downcoring" - it reflects
1582                                 * 1 less than the number of cores actually
1583                                 * enabled on this node.
1584                                 */
1585                                cpi->cpi_ncore_per_chip =
1586                                    BITX((cpi)->cpi_extd[8].cp_ecx, 7, 0) + 1;
1587                        }
1588                        break;
1589                default:
1590                        cpi->cpi_ncore_per_chip = 1;
1591                        break;
1592                }

1594                /*
1595                 * Get CPUID data about TSC Invariance in Deep C-State.
1596                 */
1597                switch (cpi->cpi_vendor) {
1598                case X86_VENDOR_Intel:
1599                        if (cpi->cpi_maxeax >= 7) {
1600                                cp = &cpi->cpi_extd[7];
1601                                cp->cp_eax = 0x80000007;
1602                                cp->cp_ecx = 0;
1603                                (void) __cpuid_insn(cp);
1604                        }
1605                        break;
1606                default:
1607                        break;
1608                }
1609        } else {
1610                cpi->cpi_ncore_per_chip = 1;
1611        }

1613        /*
1614         * If more than one core, then this processor is CMP.
1615         */
1616        if (cpi->cpi_ncore_per_chip > 1) {
1617                add_x86_feature(featureset, X86FSET_CMP);
1618        }

1620        /*
1621         * If the number of cores is the same as the number
1622         * of CPUs, then we cannot have HyperThreading.
1623         */
1624        if (cpi->cpi_ncpu_per_chip == cpi->cpi_ncore_per_chip) {
1625                remove_x86_feature(featureset, X86FSET_HTT);
1626        }

1628        cpi->cpi_apicid = CPI_APIC_ID(cpi);
1629        cpi->cpi_procnodes_per_pkg = 1;
1630        cpi->cpi_cores_per_compunit = 1;
1631        if (is_x86_feature(featureset, X86FSET_HTT) == B_FALSE &&
1632            is_x86_feature(featureset, X86FSET_CMP) == B_FALSE) {
1633                /*
1634                 * Single-core single-threaded processors.
1635                 */
1636                cpi->cpi_chipid = -1;
1637                cpi->cpi_clogid = 0;
1638                cpi->cpi_coreid = cpu->cpu_id;
1639                cpi->cpi_pkgcoreid = 0;
1640                if (cpi->cpi_vendor == X86_VENDOR_AMD)
1641                        cpi->cpi_procnodeid = BITX(cpi->cpi_apicid, 3, 0);
1642                else
```

```
1643                        cpi->cpi_procnodeid = cpi->cpi_chipid;
1644        } else if (cpi->cpi_ncpu_per_chip > 1) {
1645                if (cpi->cpi_vendor == X86_VENDOR_Intel)
1646                        cpuid_intel_getids(cpu, featureset);
1647                else if (cpi->cpi_vendor == X86_VENDOR_AMD)
1648                        cpuid_amd_getids(cpu);
1649                else {
1650                        /*
1651                         * All other processors are currently
1652                         * assumed to have single cores.
1653                         */
1654                        cpi->cpi_coreid = cpi->cpi_chipid;
1655                        cpi->cpi_pkgcoreid = 0;
1656                        cpi->cpi_procnodeid = cpi->cpi_chipid;
1657                        cpi->cpi_compunitid = cpi->cpi_chipid;
1658                }
1659        }

1661        /*
1662         * Synthesize chip "revision" and socket type
1663         */
1664        cpi->cpi_chiprev = _cpuid_chiprev(cpi->cpi_vendor, cpi->cpi_family,
1665            cpi->cpi_model, cpi->cpi_step);
1666        cpi->cpi_chiprevstr = _cpuid_chiprevstr(cpi->cpi_vendor,
1667            cpi->cpi_family, cpi->cpi_model, cpi->cpi_step);
1668        cpi->cpi_socket = _cpuid_skt(cpi->cpi_vendor, cpi->cpi_family,
1669            cpi->cpi_model, cpi->cpi_step);

1671 pass1_done:
1672        cpi->cpi_pass = 1;
1673 }

1675 /*
1676  * Make copies of the cpuid table entries we depend on, in
1677  * part for ease of parsing now, in part so that we have only
1678  * one place to correct any of it, in part for ease of
1679  * later export to userland, and in part so we can look at
1680  * this stuff in a crash dump.
1681  */

1683 /*ARGSUSED*/
1684 void
1685 cpuid_pass2(cpu_t *cpu)
1686 {
1687        uint_t n, nmax;
1688        int i;
1689        struct cpuid_regs *cp;
1690        uint8_t *dp;
1691        uint32_t *iptr;
1692        struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;

1694        ASSERT(cpi->cpi_pass == 1);

1696        if (cpi->cpi_maxeax < 1)
1697                goto pass2_done;

1699        if ((nmax = cpi->cpi_maxeax + 1) > NMAX_CPI_STD)
1700                nmax = NMAX_CPI_STD;
1701        /*
1702         * (We already handled n == 0 and n == 1 in pass 1)
1703         */
1704        for (n = 2, cp = &cpi->cpi_std[2]; n < nmax; n++, cp++) {
1705                cp->cp_eax = n;

1707                /*
1708                 * CPUID function 4 expects %ecx to be initialized
```



```
1709                 * with an index which indicates which cache to return
1710                 * information about. The OS is expected to call function 4
1711                 * with %ecx set to 0, 1, 2, ... until it returns with
1712                 * EAX[4:0] set to 0, which indicates there are no more
1713                 * caches.
1714                 *
1715                 * Here, populate cpi_std[4] with the information returned by
1716                 * function 4 when %ecx == 0, and do the rest in cpuid_pass3()
1717                 * when dynamic memory allocation becomes available.
1718                 *
1719                 * Note: we need to explicitly initialize %ecx here, since
1720                 * function 4 may have been previously invoked.
1721                 */
1722                if (n == 4)
1723                        cp->cp_ecx = 0;

1725                (void) __cpuid_insn(cp);
1726                platform_cpuid_mangle(cpi->cpi_vendor, n, cp);
1727                switch (n) {
1728                case 2:
1729                        /*
1730                         * "the lower 8 bits of the %eax register
1731                         * contain a value that identifies the number
1732                         * of times the cpuid [instruction] has to be
1733                         * executed to obtain a complete image of the
1734                         * processor's caching systems."
1735                         *
1736                         * How *do* they make this stuff up?
1737                         */
1738                        cpi->cpi_ncache = sizeof (*cp) *
1739                            BITX(cp->cp_eax, 7, 0);
1740                        if (cpi->cpi_ncache == 0)
1741                                break;
1742                        cpi->cpi_ncache--;      /* skip count byte */

1744                        /*
1745                         * Well, for now, rather than attempt to implement
1746                         * this slightly dubious algorithm, we just look
1747                         * at the first 15 ..
1748                         */
1749                        if (cpi->cpi_ncache > (sizeof (*cp) - 1))
1750                                cpi->cpi_ncache = sizeof (*cp) - 1;

1752                        dp = cpi->cpi_cacheinfo;
1753                        if (BITX(cp->cp_eax, 31, 31) == 0) {
1754                                uint8_t *p = (void *)&cp->cp_eax;
1755                                for (i = 1; i < 4; i++)
1756                                        if (p[i] != 0)
1757                                                *dp++ = p[i];
1758                        }
1759                        if (BITX(cp->cp_ebx, 31, 31) == 0) {
1760                                uint8_t *p = (void *)&cp->cp_ebx;
1761                                for (i = 0; i < 4; i++)
1762                                        if (p[i] != 0)
1763                                                *dp++ = p[i];
1764                        }
1765                        if (BITX(cp->cp_ecx, 31, 31) == 0) {
1766                                uint8_t *p = (void *)&cp->cp_ecx;
1767                                for (i = 0; i < 4; i++)
1768                                        if (p[i] != 0)
1769                                                *dp++ = p[i];
1770                        }
1771                        if (BITX(cp->cp_edx, 31, 31) == 0) {
1772                                uint8_t *p = (void *)&cp->cp_edx;
1773                                for (i = 0; i < 4; i++)
1774                                        if (p[i] != 0)
```

```
1775                                            *dp++ = p[i];
1776                    }
1777                    break;

1779            case 3: /* Processor serial number, if PSN supported */
1780                    break;

1782            case 4: /* Deterministic cache parameters */
1783                    break;

1785            case 5: /* Monitor/Mwait parameters */
1786                    {
1787                            size_t mwait_size;

1789                            /*
1790                             * check cpi_mwait.support which was set in cpuid_pass1
1791                             */
1792                            if (!(cpi->cpi_mwait.support & MWAIT_SUPPORT))
1793                                    break;

1795                            /*
1796                             * Protect ourself from insane mwait line size.
1797                             * Workaround for incomplete hardware emulator(s).
1798                             */
1799                            mwait_size = (size_t)MWAIT_SIZE_MAX(cpi);
1800                            if (mwait_size < sizeof (uint32_t) ||
1801                                !ISP2(mwait_size)) {
1802 #if DEBUG
1803                                    cmn_err(CE_NOTE, "Cannot handle cpu %d mwait "
1804                                        "size %ld", cpu->cpu_id, (long)mwait_size);
1805 #endif
1806                                    break;
1807                            }

1809                            cpi->cpi_mwait.mon_min = (size_t)MWAIT_SIZE_MIN(cpi);
1810                            cpi->cpi_mwait.mon_max = mwait_size;
1811                            if (MWAIT_EXTENSION(cpi)) {
1812                                    cpi->cpi_mwait.support |= MWAIT_EXTENSIONS;
1813                                    if (MWAIT_INT_ENABLE(cpi))
1814                                            cpi->cpi_mwait.support |=
1815                                                MWAIT_ECX_INT_ENABLE;
1816                            }
1817                            break;
1818                    }
1819            default:
1820                    break;
1821            }
1822    }

1824    if (cpi->cpi_maxeax >= 0xB && cpi->cpi_vendor == X86_VENDOR_Intel) {
1825            struct cpuid_regs regs;

1827            cp = &regs;
1828            cp->cp_eax = 0xB;
1829            cp->cp_edx = cp->cp_ebx = cp->cp_ecx = 0;

1831            (void) __cpuid_insn(cp);

1833            /*
1834             * Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero, which
1835             * indicates that the extended topology enumeration leaf is
1836             * available.
1837             */
1838            if (cp->cp_ebx) {
1839                    uint32_t x2apic_id;
1840                    uint_t coreid_shift = 0;
```



```
1841                    uint_t ncpu_per_core = 1;
1842                    uint_t chipid_shift = 0;
1843                    uint_t ncpu_per_chip = 1;
1844                    uint_t i;
1845                    uint_t level;

1847                    for (i = 0; i < CPI_FNB_ECX_MAX; i++) {
1848                            cp->cp_eax = 0xB;
1849                            cp->cp_ecx = i;

1851                            (void) __cpuid_insn(cp);
1852                            level = CPI_CPU_LEVEL_TYPE(cp);

1854                            if (level == 1) {
1855                                    x2apic_id = cp->cp_edx;
1856                                    coreid_shift = BITX(cp->cp_eax, 4, 0);
1857                                    ncpu_per_core = BITX(cp->cp_ebx, 15, 0);
1858                            } else if (level == 2) {
1859                                    x2apic_id = cp->cp_edx;
1860                                    chipid_shift = BITX(cp->cp_eax, 4, 0);
1861                                    ncpu_per_chip = BITX(cp->cp_ebx, 15, 0);
1862                            }
1863                    }

1865                    cpi->cpi_apicid = x2apic_id;
1866                    cpi->cpi_ncpu_per_chip = ncpu_per_chip;
1867                    cpi->cpi_ncore_per_chip = ncpu_per_chip /
1868                        ncpu_per_core;
1869                    cpi->cpi_chipid = x2apic_id >> chipid_shift;
1870                    cpi->cpi_clogid = x2apic_id & ((1 << chipid_shift) - 1);
1871                    cpi->cpi_coreid = x2apic_id >> coreid_shift;
1872                    cpi->cpi_pkgcoreid = cpi->cpi_clogid >> coreid_shift;
1873            }

1875            /* Make cp NULL so that we don't stumble on others */
1876            cp = NULL;
1877    }

1879    /*
1880     * XSAVE enumeration
1881     */
1882    if (cpi->cpi_maxeax >= 0xD) {
1883            struct cpuid_regs regs;
1884            boolean_t cpuid_d_valid = B_TRUE;

1886            cp = &regs;
1887            cp->cp_eax = 0xD;
1888            cp->cp_edx = cp->cp_ebx = cp->cp_ecx = 0;

1890            (void) __cpuid_insn(cp);

1892            /*
1893             * Sanity checks for debug
1894             */
1895            if ((cp->cp_eax & XFEATURE_LEGACY_FP) == 0 ||
1896                (cp->cp_eax & XFEATURE_SSE) == 0) {
1897                    cpuid_d_valid = B_FALSE;
1898            }

1900            cpi->cpi_xsave.xsav_hw_features_low = cp->cp_eax;
1901            cpi->cpi_xsave.xsav_hw_features_high = cp->cp_edx;
1902            cpi->cpi_xsave.xsav_max_size = cp->cp_ecx;

1904            /*
1905             * If the hw supports AVX, get the size and offset in the save
1906             * area for the ymm state.
```

```
1907                 */
1908                if (cpi->cpi_xsave.xsav_hw_features_low & XFEATURE_AVX) {
1909                        cp->cp_eax = 0xD;
1910                        cp->cp_ecx = 2;
1911                        cp->cp_edx = cp->cp_ebx = 0;

1913                        (void) __cpuid_insn(cp);

1915                        if (cp->cp_ebx != CPUID_LEAFD_2_YMM_OFFSET ||
1916                            cp->cp_eax != CPUID_LEAFD_2_YMM_SIZE) {
1917                                cpuid_d_valid = B_FALSE;
1918                        }

1920                        cpi->cpi_xsave.ymm_size = cp->cp_eax;
1921                        cpi->cpi_xsave.ymm_offset = cp->cp_ebx;
1922                }

1924                if (is_x86_feature(x86_featureset, X86FSET_XSAVE)) {
1925                        xsave_state_size = 0;
1926                } else if (cpuid_d_valid) {
1927                        xsave_state_size = cpi->cpi_xsave.xsav_max_size;
1928                } else {
1929                        /* Broken CPUID 0xD, probably in HVM */
1930                        cmn_err(CE_WARN, "cpu%d: CPUID.0xD returns invalid "
1931                            "value: hw_low = %d, hw_high = %d, xsave_size = %d"
1932                            ", ymm_size = %d, ymm_offset = %d\n",
1933                            cpu->cpu_id, cpi->cpi_xsave.xsav_hw_features_low,
1934                            cpi->cpi_xsave.xsav_hw_features_high,
1935                            (int)cpi->cpi_xsave.xsav_max_size,
1936                            (int)cpi->cpi_xsave.ymm_size,
1937                            (int)cpi->cpi_xsave.ymm_offset);

1939                        if (xsave_state_size != 0) {
1940                                /*
1941                                 * This must be a non-boot CPU. We cannot
1942                                 * continue, because boot cpu has already
1943                                 * enabled XSAVE.
1944                                 */
1945                                ASSERT(cpu->cpu_id != 0);
1946                                cmn_err(CE_PANIC, "cpu%d: we have already "
1947                                    "enabled XSAVE on boot cpu, cannot "
1948                                    "continue.", cpu->cpu_id);
1949                        } else {
1950                                /*
1951                                 * Must be from boot CPU, OK to disable XSAVE.
1952                                 */
1953                                ASSERT(cpu->cpu_id == 0);
1954                                remove_x86_feature(x86_featureset,
1955                                    X86FSET_XSAVE);
1956                                remove_x86_feature(x86_featureset, X86FSET_AVX);
1957                                CPI_FEATURES_ECX(cpi) &= ~CPUID_INTC_ECX_XSAVE;
1958                                CPI_FEATURES_ECX(cpi) &= ~CPUID_INTC_ECX_AVX;
1959                                CPI_FEATURES_ECX(cpi) &= ~CPUID_INTC_ECX_F16C;
1960                                xsave_force_disable = B_TRUE;
1961                        }
1962                }
1963        }


1966        if ((cpi->cpi_xmaxeax & 0x80000000) == 0)
1967                goto pass2_done;

1969        if ((nmax = cpi->cpi_xmaxeax - 0x80000000 + 1) > NMAX_CPI_EXTD)
1970                nmax = NMAX_CPI_EXTD;
1971        /*
1972         * Copy the extended properties, fixing them as we go.
```



```
1973         * (We already handled n == 0 and n == 1 in pass 1)
1974         */
1975        iptr = (void *)cpi->cpi_brandstr;
1976        for (n = 2, cp = &cpi->cpi_extd[2]; n < nmax; cp++, n++) {
1977                cp->cp_eax = 0x80000000 + n;
1978                (void) __cpuid_insn(cp);
1979                platform_cpuid_mangle(cpi->cpi_vendor, 0x80000000 + n, cp);
1980                switch (n) {
1981                case 2:
1982                case 3:
1983                case 4:
1984                        /*
1985                         * Extract the brand string
1986                         */
1987                        *iptr++ = cp->cp_eax;
1988                        *iptr++ = cp->cp_ebx;
1989                        *iptr++ = cp->cp_ecx;
1990                        *iptr++ = cp->cp_edx;
1991                        break;
1992                case 5:
1993                        switch (cpi->cpi_vendor) {
1994                        case X86_VENDOR_AMD:
1995                                /*
1996                                 * The Athlon and Duron were the first
1997                                 * parts to report the sizes of the
1998                                 * TLB for large pages. Before then,
1999                                 * we don't trust the data.
2000                                 */
2001                                if (cpi->cpi_family < 6 ||
2002                                    (cpi->cpi_family == 6 &&
2003                                    cpi->cpi_model < 1))
2004                                        cp->cp_eax = 0;
2005                                break;
2006                        default:
2007                                break;
2008                        }
2009                        break;
2010                case 6:
2011                        switch (cpi->cpi_vendor) {
2012                        case X86_VENDOR_AMD:
2013                                /*
2014                                 * The Athlon and Duron were the first
2015                                 * AMD parts with L2 TLB's.
2016                                 * Before then, don't trust the data.
2017                                 */
2018                                if (cpi->cpi_family < 6 ||
2019                                    cpi->cpi_family == 6 &&
2020                                    cpi->cpi_model < 1)
2021                                        cp->cp_eax = cp->cp_ebx = 0;
2022                                /*
2023                                 * AMD Duron rev A0 reports L2
2024                                 * cache size incorrectly as 1K
2025                                 * when it is really 64K
2026                                 */
2027                                if (cpi->cpi_family == 6 &&
2028                                    cpi->cpi_model == 3 &&
2029                                    cpi->cpi_step == 0) {
2030                                        cp->cp_ecx &= 0xffff;
2031                                        cp->cp_ecx |= 0x400000;
2032                                }
2033                                break;
2034                        case X86_VENDOR_Cyrix:  /* VIA C3 */
2035                                /*
2036                                 * VIA C3 processors are a bit messed
2037                                 * up w.r.t. encoding cache sizes in %ecx
2038                                 */
```

```
2039                                if (cpi->cpi_family != 6)
2040                                        break;
2041                                /*
2042                                 * model 7 and 8 were incorrectly encoded
2043                                 *
2044                                 * xxx is model 8 really broken?
2045                                 */
2046                                if (cpi->cpi_model == 7 ||
2047                                    cpi->cpi_model == 8)
2048                                        cp->cp_ecx =
2049                                            BITX(cp->cp_ecx, 31, 24) << 16 |
2050                                            BITX(cp->cp_ecx, 23, 16) << 12 |
2051                                            BITX(cp->cp_ecx, 15, 8) << 8 |
2052                                            BITX(cp->cp_ecx, 7, 0);
2053                                /*
2054                                 * model 9 stepping 1 has wrong associativity
2055                                 */
2056                                if (cpi->cpi_model == 9 && cpi->cpi_step == 1)
2057                                        cp->cp_ecx |= 8 << 12;
2058                                break;
2059                        case X86_VENDOR_Intel:
2060                                /*
2061                                 * Extended L2 Cache features function.
2062                                 * First appeared on Prescott.
2063                                 */
2064                        default:
2065                                break;
2066                        }
2067                        break;
2068                default:
2069                        break;
2070                }
2071        }

2073 pass2_done:
2074        cpi->cpi_pass = 2;
2075 }

2077 static const char *
2078 intel_cpubrand(const struct cpuid_info *cpi)
2079 {
2080        int i;

2082        if (!is_x86_feature(x86_featureset, X86FSET_CPUID) ||
2083            cpi->cpi_maxeax < 1 || cpi->cpi_family < 5)
2084                return ("i486");

2086        switch (cpi->cpi_family) {
2087        case 5:
2088                return ("Intel Pentium(r)");
2089        case 6:
2090                switch (cpi->cpi_model) {
2091                        uint_t celeron, xeon;
2092                        const struct cpuid_regs *cp;
2093                case 0:
2094                case 1:
2095                case 2:
2096                        return ("Intel Pentium(r) Pro");
2097                case 3:
2098                case 4:
2099                        return ("Intel Pentium(r) II");
2100                case 6:
2101                        return ("Intel Celeron(r)");
2102                case 5:
2103                case 7:
2104                        celeron = xeon = 0;
```



```
2105                        cp = &cpi->cpi_std[2];  /* cache info */

2107                        for (i = 1; i < 4; i++) {
2108                                uint_t tmp;

2110                                tmp = (cp->cp_eax >> (8 * i)) & 0xff;
2111                                if (tmp == 0x40)
2112                                        celeron++;
2113                                if (tmp >= 0x44 && tmp <= 0x45)
2114                                        xeon++;
2115                        }

2117                        for (i = 0; i < 2; i++) {
2118                                uint_t tmp;

2120                                tmp = (cp->cp_ebx >> (8 * i)) & 0xff;
2121                                if (tmp == 0x40)
2122                                        celeron++;
2123                                else if (tmp >= 0x44 && tmp <= 0x45)
2124                                        xeon++;
2125                        }

2127                        for (i = 0; i < 4; i++) {
2128                                uint_t tmp;

2130                                tmp = (cp->cp_ecx >> (8 * i)) & 0xff;
2131                                if (tmp == 0x40)
2132                                        celeron++;
2133                                else if (tmp >= 0x44 && tmp <= 0x45)
2134                                        xeon++;
2135                        }

2137                        for (i = 0; i < 4; i++) {
2138                                uint_t tmp;

2140                                tmp = (cp->cp_edx >> (8 * i)) & 0xff;
2141                                if (tmp == 0x40)
2142                                        celeron++;
2143                                else if (tmp >= 0x44 && tmp <= 0x45)
2144                                        xeon++;
2145                        }

2147                        if (celeron)
2148                                return ("Intel Celeron(r)");
2149                        if (xeon)
2150                                return (cpi->cpi_model == 5 ?
2151                                    "Intel Pentium(r) II Xeon(tm)" :
2152                                    "Intel Pentium(r) III Xeon(tm)");
2153                        return (cpi->cpi_model == 5 ?
2154                            "Intel Pentium(r) II or Pentium(r) II Xeon(tm)" :
2155                            "Intel Pentium(r) III or Pentium(r) III Xeon(tm)");
2156                default:
2157                        break;
2158                }
2159        default:
2160                break;
2161        }

2163        /* BrandID is present if the field is nonzero */
2164        if (cpi->cpi_brandid != 0) {
2165                static const struct {
2166                        uint_t bt_bid;
2167                        const char *bt_str;
2168                } brand_tbl[] = {
2169                        { 0x1,  "Intel(r) Celeron(r)" },
2170                        { 0x2,  "Intel(r) Pentium(r) III" },
```

```
2171                        { 0x3,  "Intel(r) Pentium(r) III Xeon(tm)" },
2172                        { 0x4,  "Intel(r) Pentium(r) III" },
2173                        { 0x6,  "Mobile Intel(r) Pentium(r) III" },
2174                        { 0x7,  "Mobile Intel(r) Celeron(r)" },
2175                        { 0x8,  "Intel(r) Pentium(r) 4" },
2176                        { 0x9,  "Intel(r) Pentium(r) 4" },
2177                        { 0xa,  "Intel(r) Celeron(r)" },
2178                        { 0xb,  "Intel(r) Xeon(tm)" },
2179                        { 0xc,  "Intel(r) Xeon(tm) MP" },
2180                        { 0xe,  "Mobile Intel(r) Pentium(r) 4" },
2181                        { 0xf,  "Mobile Intel(r) Celeron(r)" },
2182                        { 0x11, "Mobile Genuine Intel(r)" },
2183                        { 0x12, "Intel(r) Celeron(r) M" },
2184                        { 0x13, "Mobile Intel(r) Celeron(r)" },
2185                        { 0x14, "Intel(r) Celeron(r)" },
2186                        { 0x15, "Mobile Genuine Intel(r)" },
2187                        { 0x16, "Intel(r) Pentium(r) M" },
2188                        { 0x17, "Mobile Intel(r) Celeron(r)" }
2189                };
2190                uint_t btblmax = sizeof (brand_tbl) / sizeof (brand_tbl[0]);
2191                uint_t sgn;

2193                sgn = (cpi->cpi_family << 8) |
2194                    (cpi->cpi_model << 4) | cpi->cpi_step;

2196                for (i = 0; i < btblmax; i++)
2197                        if (brand_tbl[i].bt_bid == cpi->cpi_brandid)
2198                                break;
2199                if (i < btblmax) {
2200                        if (sgn == 0x6b1 && cpi->cpi_brandid == 3)
2201                                return ("Intel(r) Celeron(r)");
2202                        if (sgn < 0xf13 && cpi->cpi_brandid == 0xb)
2203                                return ("Intel(r) Xeon(tm) MP");
2204                        if (sgn < 0xf13 && cpi->cpi_brandid == 0xe)
2205                                return ("Intel(r) Xeon(tm)");
2206                        return (brand_tbl[i].bt_str);
2207                }
2208        }

2210        return (NULL);
2211 }

2213 static const char *
2214 amd_cpubrand(const struct cpuid_info *cpi)
2215 {
2216        if (!is_x86_feature(x86_featureset, X86FSET_CPUID) ||
2217            cpi->cpi_maxeax < 1 || cpi->cpi_family < 5)
2218                return ("i486 compatible");

2220        switch (cpi->cpi_family) {
2221        case 5:
2222                switch (cpi->cpi_model) {
2223                case 0:
2224                case 1:
2225                case 2:
2226                case 3:
2227                case 4:
2228                case 5:
2229                        return ("AMD-K5(r)");
2230                case 6:
2231                case 7:
2232                        return ("AMD-K6(r)");
2233                case 8:
2234                        return ("AMD-K6(r)-2");
2235                case 9:
2236                        return ("AMD-K6(r)-III");
```



```
2237                default:
2238                        return ("AMD (family 5)");
2239                }
2240        case 6:
2241                switch (cpi->cpi_model) {
2242                case 1:
2243                        return ("AMD-K7(tm)");
2244                case 0:
2245                case 2:
2246                case 4:
2247                        return ("AMD Athlon(tm)");
2248                case 3:
2249                case 7:
2250                        return ("AMD Duron(tm)");
2251                case 6:
2252                case 8:
2253                case 10:
2254                        /*
2255                         * Use the L2 cache size to distinguish
2256                         */
2257                        return ((cpi->cpi_extd[6].cp_ecx >> 16) >= 256 ?
2258                            "AMD Athlon(tm)" : "AMD Duron(tm)");
2259                default:
2260                        return ("AMD (family 6)");
2261                }
2262        default:
2263                break;
2264        }

2266        if (cpi->cpi_family == 0xf && cpi->cpi_model == 5 &&
2267            cpi->cpi_brandid != 0) {
2268                switch (BITX(cpi->cpi_brandid, 7, 5)) {
2269                case 3:
2270                        return ("AMD Opteron(tm) UP 1xx");
2271                case 4:
2272                        return ("AMD Opteron(tm) DP 2xx");
2273                case 5:
2274                        return ("AMD Opteron(tm) MP 8xx");
2275                default:
2276                        return ("AMD Opteron(tm)");
2277                }
2278        }

2280        return (NULL);
2281 }

2283 static const char *
2284 cyrix_cpubrand(struct cpuid_info *cpi, uint_t type)
2285 {
2286        if (!is_x86_feature(x86_featureset, X86FSET_CPUID) ||
2287            cpi->cpi_maxeax < 1 || cpi->cpi_family < 5 ||
2288            type == X86_TYPE_CYRIX_486)
2289                return ("i486 compatible");

2291        switch (type) {
2292        case X86_TYPE_CYRIX_6x86:
2293                return ("Cyrix 6x86");
2294        case X86_TYPE_CYRIX_6x86L:
2295                return ("Cyrix 6x86L");
2296        case X86_TYPE_CYRIX_6x86MX:
2297                return ("Cyrix 6x86MX");
2298        case X86_TYPE_CYRIX_GXm:
2299                return ("Cyrix GXm");
2300        case X86_TYPE_CYRIX_MediaGX:
2301                return ("Cyrix MediaGX");
2302        case X86_TYPE_CYRIX_MII:
```

```
2303                 return ("Cyrix M2");
2304         case X86_TYPE_VIA_CYRIX_III:
2305                 return ("VIA Cyrix M3");
2306         default:
2307                 /*
2308                  * Have another wild guess ..
2309                  */
2310                 if (cpi->cpi_family == 4 && cpi->cpi_model == 9)
2311                         return ("Cyrix 5x86");
2312                 else if (cpi->cpi_family == 5) {
2313                         switch (cpi->cpi_model) {
2314                         case 2:
2315                                 return ("Cyrix 6x86");  /* Cyrix M1 */
2316                         case 4:
2317                                 return ("Cyrix MediaGX");
2318                         default:
2319                                 break;
2320                         }
2321                 } else if (cpi->cpi_family == 6) {
2322                         switch (cpi->cpi_model) {
2323                         case 0:
2324                                 return ("Cyrix 6x86MX"); /* Cyrix M2? */
2325                         case 5:
2326                         case 6:
2327                         case 7:
2328                         case 8:
2329                         case 9:
2330                                 return ("VIA C3");
2331                         default:
2332                                 break;
2333                         }
2334                 }
2335                 break;
2336         }
2337         return (NULL);
2338 }

2340 /*
2341  * This only gets called in the case that the CPU extended
2342  * feature brand string (0x80000002, 0x80000003, 0x80000004)
2343  * aren't available, or contain null bytes for some reason.
2344  */
2345 static void
2346 fabricate_brandstr(struct cpuid_info *cpi)
2347 {
2348         const char *brand = NULL;

2350         switch (cpi->cpi_vendor) {
2351         case X86_VENDOR_Intel:
2352                 brand = intel_cpubrand(cpi);
2353                 break;
2354         case X86_VENDOR_AMD:
2355                 brand = amd_cpubrand(cpi);
2356                 break;
2357         case X86_VENDOR_Cyrix:
2358                 brand = cyrix_cpubrand(cpi, x86_type);
2359                 break;
2360         case X86_VENDOR_NexGen:
2361                 if (cpi->cpi_family == 5 && cpi->cpi_model == 0)
2362                         brand = "NexGen Nx586";
2363                 break;
2364         case X86_VENDOR_Centaur:
2365                 if (cpi->cpi_family == 5)
2366                         switch (cpi->cpi_model) {
2367                         case 4:
2368                                 brand = "Centaur C6";
```



```
2369                                 break;
2370                         case 8:
2371                                 brand = "Centaur C2";
2372                                 break;
2373                         case 9:
2374                                 brand = "Centaur C3";
2375                                 break;
2376                         default:
2377                                 break;
2378                         }
2379                 break;
2380         case X86_VENDOR_Rise:
2381                 if (cpi->cpi_family == 5 &&
2382                     (cpi->cpi_model == 0 || cpi->cpi_model == 2))
2383                         brand = "Rise mP6";
2384                 break;
2385         case X86_VENDOR_SiS:
2386                 if (cpi->cpi_family == 5 && cpi->cpi_model == 0)
2387                         brand = "SiS 55x";
2388                 break;
2389         case X86_VENDOR_TM:
2390                 if (cpi->cpi_family == 5 && cpi->cpi_model == 4)
2391                         brand = "Transmeta Crusoe TM3x00 or TM5x00";
2392                 break;
2393         case X86_VENDOR_NSC:
2394         case X86_VENDOR_UMC:
2395         default:
2396                 break;
2397         }
2398         if (brand) {
2399                 (void) strcpy((char *)cpi->cpi_brandstr, brand);
2400                 return;
2401         }

2403         /*
2404          * If all else fails ...
2405          */
2406         (void) snprintf(cpi->cpi_brandstr, sizeof (cpi->cpi_brandstr),
2407             "%s %d.%d.%d", cpi->cpi_vendorstr, cpi->cpi_family,
2408             cpi->cpi_model, cpi->cpi_step);
2409 }

2411 /*
2412  * This routine is called just after kernel memory allocation
2413  * becomes available on cpu0, and as part of mp_startup() on
2414  * the other cpus.
2415  *
2416  * Fixup the brand string, and collect any information from cpuid
2417  * that requires dynamically allocated storage to represent.
2418  */
2419 /*ARGSUSED*/
2420 void
2421 cpuid_pass3(cpu_t *cpu)
2422 {
2423         int     i, max, shft, level, size;
2424         struct cpuid_regs regs;
2425         struct cpuid_regs *cp;
2426         struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;

2428         ASSERT(cpi->cpi_pass == 2);

2430         /*
2431          * Function 4: Deterministic cache parameters
2432          *
2433          * Take this opportunity to detect the number of threads
2434          * sharing the last level cache, and construct a corresponding
```

```
2435          * cache id. The respective cpuid_info members are initialized
2436          * to the default case of "no last level cache sharing".
2437          */
2438         cpi->cpi_ncpu_shr_last_cache = 1;
2439         cpi->cpi_last_lvl_cacheid = cpu->cpu_id;

2441         if (cpi->cpi_maxeax >= 4 && cpi->cpi_vendor == X86_VENDOR_Intel) {

2443                 /*
2444                  * Find the # of elements (size) returned by fn 4, and along
2445                  * the way detect last level cache sharing details.
2446                  */
2447                 bzero(&regs, sizeof (regs));
2448                 cp = &regs;
2449                 for (i = 0, max = 0; i < CPI_FN4_ECX_MAX; i++) {
2450                         cp->cp_eax = 4;
2451                         cp->cp_ecx = i;

2453                         (void) __cpuid_insn(cp);

2455                         if (CPI_CACHE_TYPE(cp) == 0)
2456                                 break;
2457                         level = CPI_CACHE_LVL(cp);
2458                         if (level > max) {
2459                                 max = level;
2460                                 cpi->cpi_ncpu_shr_last_cache =
2461                                     CPI_NTHR_SHR_CACHE(cp) + 1;
2462                         }
2463                 }
2464                 cpi->cpi_std_4_size = size = i;

2466                 /*
2467                  * Allocate the cpi_std_4 array. The first element
2468                  * references the regs for fn 4, %ecx == 0, which
2469                  * cpuid_pass2() stashed in cpi->cpi_std[4].
2470                  */
2471                 if (size > 0) {
2472                         cpi->cpi_std_4 =
2473                             kmem_alloc(size * sizeof (cp), KM_SLEEP);
2474                         cpi->cpi_std_4[0] = &cpi->cpi_std[4];

2476                         /*
2477                          * Allocate storage to hold the additional regs
2478                          * for function 4, %ecx == 1 .. cpi_std_4_size.
2479                          *
2480                          * The regs for fn 4, %ecx == 0 has already
2481                          * been allocated as indicated above.
2482                          */
2483                         for (i = 1; i < size; i++) {
2484                                 cp = cpi->cpi_std_4[i] =
2485                                     kmem_zalloc(sizeof (regs), KM_SLEEP);
2486                                 cp->cp_eax = 4;
2487                                 cp->cp_ecx = i;

2489                                 (void) __cpuid_insn(cp);
2490                         }
2491                 }
2492                 /*
2493                  * Determine the number of bits needed to represent
2494                  * the number of CPUs sharing the last level cache.
2495                  *
2496                  * Shift off that number of bits from the APIC id to
2497                  * derive the cache id.
2498                  */
2499                 shft = 0;
2500                 for (i = 1; i < cpi->cpi_ncpu_shr_last_cache; i <<= 1)
```



```
2501                         shft++;
2502                 cpi->cpi_last_lvl_cacheid = cpi->cpi_apicid >> shft;
2503         }

2505         /*
2506          * Now fixup the brand string
2507          */
2508         if ((cpi->cpi_xmaxeax & 0x80000000) == 0) {
2509                 fabricate_brandstr(cpi);
2510         } else {

2512                 /*
2513                  * If we successfully extracted a brand string from the cpuid
2514                  * instruction, clean it up by removing leading spaces and
2515                  * similar junk.
2516                  */
2517                 if (cpi->cpi_brandstr[0]) {
2518                         size_t maxlen = sizeof (cpi->cpi_brandstr);
2519                         char *src, *dst;

2521                         dst = src = (char *)cpi->cpi_brandstr;
2522                         src[maxlen - 1] = '\0';
2523                         /*
2524                          * strip leading spaces
2525                          */
2526                         while (*src == ' ')
2527                                 src++;
2528                         /*
2529                          * Remove any 'Genuine' or "Authentic" prefixes
2530                          */
2531                         if (strncmp(src, "Genuine ", 8) == 0)
2532                                 src += 8;
2533                         if (strncmp(src, "Authentic ", 10) == 0)
2534                                 src += 10;

2536                         /*
2537                          * Now do an in-place copy.
2538                          * Map (R) to (r) and (TM) to (tm).
2539                          * The era of teletypes is long gone, and there's
2540                          * -really- no need to shout.
2541                          */
2542                         while (*src != '\0') {
2543                                 if (src[0] == '(') {
2544                                         if (strncmp(src + 1, "R)", 2) == 0) {
2545                                                 (void) strncpy(dst, "(r)", 3);
2546                                                 src += 3;
2547                                                 dst += 3;
2548                                                 continue;
2549                                         }
2550                                         if (strncmp(src + 1, "TM)", 3) == 0) {
2551                                                 (void) strncpy(dst, "(tm)", 4);
2552                                                 src += 4;
2553                                                 dst += 4;
2554                                                 continue;
2555                                         }
2556                                 }
2557                                 *dst++ = *src++;
2558                         }
2559                         *dst = '\0';

2561                         /*
2562                          * Finally, remove any trailing spaces
2563                          */
2564                         while (--dst > cpi->cpi_brandstr)
2565                                 if (*dst == ' ')
2566                                         *dst = '\0';
```

```
2567                                    else
2568                                            break;
2569                    } else
2570                            fabricate_brandstr(cpi);
2571            }
2572            cpi->cpi_pass = 3;
2573    }

2575    /*
2576     * This routine is called out of bind_hwcap() much later in the life
2577     * of the kernel (post_startup()).  The job of this routine is to resolve
2578     * the hardware feature support and kernel support for those features into
2579     * what we're actually going to tell applications via the aux vector.
2580     */
2581    void
2582    cpuid_pass4(cpu_t *cpu, uint_t *hwcap_out)
2583    {
2584            struct cpuid_info *cpi;
2585            uint_t hwcap_flags = 0, hwcap_flags_2 = 0;

2587            if (cpu == NULL)
2588                    cpu = CPU;
2589            cpi = cpu->cpu_m.mcpu_cpi;

2591            ASSERT(cpi->cpi_pass == 3);

2593            if (cpi->cpi_maxeax >= 1) {
2594                    uint32_t *edx = &cpi->cpi_support[STD_EDX_FEATURES];
2595                    uint32_t *ecx = &cpi->cpi_support[STD_ECX_FEATURES];

2597                    *edx = CPI_FEATURES_EDX(cpi);
2598                    *ecx = CPI_FEATURES_ECX(cpi);

2600                    /*
2601                     * [these require explicit kernel support]
2602                     */
2603                    if (!is_x86_feature(x86_featureset, X86FSET_SEP))
2604                            *edx &= ~CPUID_INTC_EDX_SEP;

2606                    if (!is_x86_feature(x86_featureset, X86FSET_SSE))
2607                            *edx &= ~(CPUID_INTC_EDX_FXSR|CPUID_INTC_EDX_SSE);
2608                    if (!is_x86_feature(x86_featureset, X86FSET_SSE2))
2609                            *edx &= ~CPUID_INTC_EDX_SSE2;

2611                    if (!is_x86_feature(x86_featureset, X86FSET_HTT))
2612                            *edx &= ~CPUID_INTC_EDX_HTT;

2614                    if (!is_x86_feature(x86_featureset, X86FSET_SSE3))
2615                            *ecx &= ~CPUID_INTC_ECX_SSE3;

2617                    if (!is_x86_feature(x86_featureset, X86FSET_SSSE3))
2618                            *ecx &= ~CPUID_INTC_ECX_SSSE3;
2619                    if (!is_x86_feature(x86_featureset, X86FSET_SSE4_1))
2620                            *ecx &= ~CPUID_INTC_ECX_SSE4_1;
2621                    if (!is_x86_feature(x86_featureset, X86FSET_SSE4_2))
2622                            *ecx &= ~CPUID_INTC_ECX_SSE4_2;
2623                    if (!is_x86_feature(x86_featureset, X86FSET_AES))
2624                            *ecx &= ~CPUID_INTC_ECX_AES;
2625                    if (!is_x86_feature(x86_featureset, X86FSET_PCLMULQDQ))
2626                            *ecx &= ~CPUID_INTC_ECX_PCLMULQDQ;
2627                    if (!is_x86_feature(x86_featureset, X86FSET_XSAVE))
2628                            *ecx &= ~(CPUID_INTC_ECX_XSAVE |
2629                                CPUID_INTC_ECX_OSXSAVE);
2630                    if (!is_x86_feature(x86_featureset, X86FSET_AVX))
2631                            *ecx &= ~CPUID_INTC_ECX_AVX;
2632                    if (!is_x86_feature(x86_featureset, X86FSET_F16C))
```



```
2633                            *ecx &= ~CPUID_INTC_ECX_F16C;

2635                    /*
2636                     * [no explicit support required beyond x87 fp context]
2637                     */
2638                    if (!fpu_exists)
2639                            *edx &= ~(CPUID_INTC_EDX_FPU | CPUID_INTC_EDX_MMX);

2641                    /*
2642                     * Now map the supported feature vector to things that we
2643                     * think userland will care about.
2644                     */
2645                    if (*edx & CPUID_INTC_EDX_SEP)
2646                            hwcap_flags |= AV_386_SEP;
2647                    if (*edx & CPUID_INTC_EDX_SSE)
2648                            hwcap_flags |= AV_386_FXSR | AV_386_SSE;
2649                    if (*edx & CPUID_INTC_EDX_SSE2)
2650                            hwcap_flags |= AV_386_SSE2;
2651                    if (*ecx & CPUID_INTC_ECX_SSE3)
2652                            hwcap_flags |= AV_386_SSE3;
2653                    if (*ecx & CPUID_INTC_ECX_SSSE3)
2654                            hwcap_flags |= AV_386_SSSE3;
2655                    if (*ecx & CPUID_INTC_ECX_SSE4_1)
2656                            hwcap_flags |= AV_386_SSE4_1;
2657                    if (*ecx & CPUID_INTC_ECX_SSE4_2)
2658                            hwcap_flags |= AV_386_SSE4_2;
2659                    if (*ecx & CPUID_INTC_ECX_MOVBE)
2660                            hwcap_flags |= AV_386_MOVBE;
2661                    if (*ecx & CPUID_INTC_ECX_AES)
2662                            hwcap_flags |= AV_386_AES;
2663                    if (*ecx & CPUID_INTC_ECX_PCLMULQDQ)
2664                            hwcap_flags |= AV_386_PCLMULQDQ;
2665                    if ((*ecx & CPUID_INTC_ECX_XSAVE) &&
2666                        (*ecx & CPUID_INTC_ECX_OSXSAVE)) {
2667                            hwcap_flags |= AV_386_XSAVE;

2669                            if (*ecx & CPUID_INTC_ECX_AVX) {
2670                                    hwcap_flags |= AV_386_AVX;
2671                                    if (*ecx & CPUID_INTC_ECX_F16C)
2672                                            hwcap_flags_2 |= AV_386_2_F16C;
2673                            }
2674                    }
2675                    if (*ecx & CPUID_INTC_ECX_VMX)
2676                            hwcap_flags |= AV_386_VMX;
2677                    if (*ecx & CPUID_INTC_ECX_POPCNT)
2678                            hwcap_flags |= AV_386_POPCNT;
2679                    if (*edx & CPUID_INTC_EDX_FPU)
2680                            hwcap_flags |= AV_386_FPU;
2681                    if (*edx & CPUID_INTC_EDX_MMX)
2682                            hwcap_flags |= AV_386_MMX;

2684                    if (*edx & CPUID_INTC_EDX_TSC)
2685                            hwcap_flags |= AV_386_TSC;
2686                    if (*edx & CPUID_INTC_EDX_CX8)
2687                            hwcap_flags |= AV_386_CX8;
2688                    if (*edx & CPUID_INTC_EDX_CMOV)
2689                            hwcap_flags |= AV_386_CMOV;
2690                    if (*ecx & CPUID_INTC_ECX_CX16)
2691                            hwcap_flags |= AV_386_CX16;

2693                    if (*ecx & CPUID_INTC_ECX_RDRAND)
2694                            hwcap_flags_2 |= AV_386_2_RDRAND;
2695            }

2697            if (cpi->cpi_xmaxeax < 0x80000001)
2698                    goto pass4_done;
```

```
2700         switch (cpi->cpi_vendor) {
2701                 struct cpuid_regs cp;
2702                 uint32_t *edx, *ecx;
2703
2704         case X86_VENDOR_Intel:
2705                 /*
2706                  * Seems like Intel duplicated what we necessary
2707                  * here to make the initial crop of 64-bit OS's work.
2708                  * Hopefully, those are the only "extended" bits
2709                  * they'll add.
2710                  */
2711                 /*FALLTHROUGH*/
2712
2713         case X86_VENDOR_AMD:
2714                 edx = &cpi->cpi_support[AMD_EDX_FEATURES];
2715                 ecx = &cpi->cpi_support[AMD_ECX_FEATURES];
2716
2717                 *edx = CPI_FEATURES_XTD_EDX(cpi);
2718                 *ecx = CPI_FEATURES_XTD_ECX(cpi);
2719
2720                 /*
2721                  * [these features require explicit kernel support]
2722                  */
2723                 switch (cpi->cpi_vendor) {
2724                 case X86_VENDOR_Intel:
2725                         if (!is_x86_feature(x86_featureset, X86FSET_TSCP))
2726                                 *edx &= ~CPUID_AMD_EDX_TSCP;
2727                         break;
2728
2729                 case X86_VENDOR_AMD:
2730                         if (!is_x86_feature(x86_featureset, X86FSET_TSCP))
2731                                 *edx &= ~CPUID_AMD_EDX_TSCP;
2732                         if (!is_x86_feature(x86_featureset, X86FSET_SSE4A))
2733                                 *ecx &= ~CPUID_AMD_ECX_SSE4A;
2734                         break;
2735
2736                 default:
2737                         break;
2738                 }
2739
2740                 /*
2741                  * [no explicit support required beyond
2742                  * x87 fp context and exception handlers]
2743                  */
2744                 if (!fpu_exists)
2745                         *edx &= ~(CPUID_AMD_EDX_MMXamd |
2746                             CPUID_AMD_EDX_3DNow | CPUID_AMD_EDX_3DNowx);
2747
2748                 if (!is_x86_feature(x86_featureset, X86FSET_NX))
2749                         *edx &= ~CPUID_AMD_EDX_NX;
2750 #if !defined(__amd64)
2751                 *edx &= ~CPUID_AMD_EDX_LM;
2752 #endif
2753                 /*
2754                  * Now map the supported feature vector to
2755                  * things that we think userland will care about.
2756                  */
2757 #if defined(__amd64)
2758                 if (*edx & CPUID_AMD_EDX_SYSC)
2759                         hwcap_flags |= AV_386_AMD_SYSC;
2760 #endif
2761                 if (*edx & CPUID_AMD_EDX_MMXamd)
2762                         hwcap_flags |= AV_386_AMD_MMX;
2763                 if (*edx & CPUID_AMD_EDX_3DNow)
2764                         hwcap_flags |= AV_386_AMD_3DNow;
```



```
2765                 if (*edx & CPUID_AMD_EDX_3DNowx)
2766                         hwcap_flags |= AV_386_AMD_3DNowx;
2767                 if (*ecx & CPUID_AMD_ECX_SVM)
2768                         hwcap_flags |= AV_386_AMD_SVM;
2769
2770                 switch (cpi->cpi_vendor) {
2771                 case X86_VENDOR_AMD:
2772                         if (*edx & CPUID_AMD_EDX_TSCP)
2773                                 hwcap_flags |= AV_386_TSCP;
2774                         if (*ecx & CPUID_AMD_ECX_AHF64)
2775                                 hwcap_flags |= AV_386_AHF;
2776                         if (*ecx & CPUID_AMD_ECX_SSE4A)
2777                                 hwcap_flags |= AV_386_AMD_SSE4A;
2778                         if (*ecx & CPUID_AMD_ECX_LZCNT)
2779                                 hwcap_flags |= AV_386_AMD_LZCNT;
2780                         break;
2781
2782                 case X86_VENDOR_Intel:
2783                         if (*edx & CPUID_AMD_EDX_TSCP)
2784                                 hwcap_flags |= AV_386_TSCP;
2785                         /*
2786                          * Aarrgh.
2787                          * Intel uses a different bit in the same word.
2788                          */
2789                         if (*ecx & CPUID_INTC_ECX_AHF64)
2790                                 hwcap_flags |= AV_386_AHF;
2791                         break;
2792
2793                 default:
2794                         break;
2795                 }
2796                 break;
2797
2798         case X86_VENDOR_TM:
2799                 cp.cp_eax = 0x80860001;
2800                 (void) __cpuid_insn(&cp);
2801                 cpi->cpi_support[TM_EDX_FEATURES] = cp.cp_edx;
2802                 break;
2803
2804         default:
2805                 break;
2806         }
2807
2808 pass4_done:
2809         cpi->cpi_pass = 4;
2810         if (hwcap_out != NULL) {
2811                 hwcap_out[0] = hwcap_flags;
2812                 hwcap_out[1] = hwcap_flags_2;
2813         }
2814 }
2815
2816
2817 /*
2818  * Simulate the cpuid instruction using the data we previously
2819  * captured about this CPU.  We try our best to return the truth
2820  * about the hardware, independently of kernel support.
2821  */
2822 uint32_t
2823 cpuid_insn(cpu_t *cpu, struct cpuid_regs *cp)
2824 {
2825         struct cpuid_info *cpi;
2826         struct cpuid_regs *xcp;
2827
2828         if (cpu == NULL)
2829                 cpu = CPU;
2830         cpi = cpu->cpu_m.mcpu_cpi;
```

```
2832         ASSERT(cpuid_checkpass(cpu, 3));

2834         /*
2835          * CPUID data is cached in two separate places: cpi_std for standard
2836          * CPUID functions, and cpi_extd for extended CPUID functions.
2837          */
2838         if (cp->cp_eax <= cpi->cpi_maxeax && cp->cp_eax < NMAX_CPI_STD)
2839                 xcp = &cpi->cpi_std[cp->cp_eax];
2840         else if (cp->cp_eax >= 0x80000000 && cp->cp_eax <= cpi->cpi_xmaxeax &&
2841             cp->cp_eax < 0x80000000 + NMAX_CPI_EXTD)
2842                 xcp = &cpi->cpi_extd[cp->cp_eax - 0x80000000];
2843         else
2844                 /*
2845                  * The caller is asking for data from an input parameter which
2846                  * the kernel has not cached.  In this case we go fetch from
2847                  * the hardware and return the data directly to the user.
2848                  */
2849                 return (__cpuid_insn(cp));

2851         cp->cp_eax = xcp->cp_eax;
2852         cp->cp_ebx = xcp->cp_ebx;
2853         cp->cp_ecx = xcp->cp_ecx;
2854         cp->cp_edx = xcp->cp_edx;
2855         return (cp->cp_eax);
2856 }

2858 int
2859 cpuid_checkpass(cpu_t *cpu, int pass)
2860 {
2861         return (cpu != NULL && cpu->cpu_m.mcpu_cpi != NULL &&
2862             cpu->cpu_m.mcpu_cpi->cpi_pass >= pass);
2863 }

2865 int
2866 cpuid_getbrandstr(cpu_t *cpu, char *s, size_t n)
2867 {
2868         ASSERT(cpuid_checkpass(cpu, 3));

2870         return (snprintf(s, n, "%s", cpu->cpu_m.mcpu_cpi->cpi_brandstr));
2871 }

2873 int
2874 cpuid_is_cmt(cpu_t *cpu)
2875 {
2876         if (cpu == NULL)
2877                 cpu = CPU;

2879         ASSERT(cpuid_checkpass(cpu, 1));

2881         return (cpu->cpu_m.mcpu_cpi->cpi_chipid >= 0);
2882 }

2884 /*
2885  * AMD and Intel both implement the 64-bit variant of the syscall
2886  * instruction (syscallq), so if there's -any- support for syscall,
2887  * cpuid currently says "yes, we support this".
2888  *
2889  * However, Intel decided to -not- implement the 32-bit variant of the
2890  * syscall instruction, so we provide a predicate to allow our caller
2891  * to test that subtlety here.
2892  *
2893  * XXPV Currently, 32-bit syscall instructions don't work via the hypervisor,
2894  *      even in the case where the hardware would in fact support it.
2895  */
2896 /*ARGSUSED*/
```



```
2897 int
2898 cpuid_syscall32_insn(cpu_t *cpu)
2899 {
2900         ASSERT(cpuid_checkpass((cpu == NULL ? CPU : cpu), 1));

2902 #if !defined(__xpv)
2903         if (cpu == NULL)
2904                 cpu = CPU;

2906         /*CSTYLED*/
2907         {
2908                 struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;

2910                 if (cpi->cpi_vendor == X86_VENDOR_AMD &&
2911                     cpi->cpi_xmaxeax >= 0x80000001 &&
2912                     (CPI_FEATURES_XTD_EDX(cpi) & CPUID_AMD_EDX_SYSC))
2913                         return (1);
2914         }
2915 #endif
2916         return (0);
2917 }

2919 int
2920 cpuid_getidstr(cpu_t *cpu, char *s, size_t n)
2921 {
2922         struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;

2924         static const char fmt[] =
2925             "x86 (%s %X family %d model %d step %d clock %d MHz)";
2926         static const char fmt_ht[] =
2927             "x86 (chipid 0x%x %s %X family %d model %d step %d clock %d MHz)";

2929         ASSERT(cpuid_checkpass(cpu, 1));

2931         if (cpuid_is_cmt(cpu))
2932                 return (snprintf(s, n, fmt_ht, cpi->cpi_chipid,
2933                     cpi->cpi_vendorstr, cpi->cpi_std[1].cp_eax,
2934                     cpi->cpi_family, cpi->cpi_model,
2935                     cpi->cpi_step, cpu->cpu_type_info.pi_clock));
2936         return (snprintf(s, n, fmt,
2937             cpi->cpi_vendorstr, cpi->cpi_std[1].cp_eax,
2938             cpi->cpi_family, cpi->cpi_model,
2939             cpi->cpi_step, cpu->cpu_type_info.pi_clock));
2940 }

2942 const char *
2943 cpuid_getvendorstr(cpu_t *cpu)
2944 {
2945         ASSERT(cpuid_checkpass(cpu, 1));
2946         return ((const char *)cpu->cpu_m.mcpu_cpi->cpi_vendorstr);
2947 }

2949 uint_t
2950 cpuid_getvendor(cpu_t *cpu)
2951 {
2952         ASSERT(cpuid_checkpass(cpu, 1));
2953         return (cpu->cpu_m.mcpu_cpi->cpi_vendor);
2954 }

2956 uint_t
2957 cpuid_getfamily(cpu_t *cpu)
2958 {
2959         ASSERT(cpuid_checkpass(cpu, 1));
2960         return (cpu->cpu_m.mcpu_cpi->cpi_family);
2961 }
```

```
2963 uint_t
2964 cpuid_getmodel(cpu_t *cpu)
2965 {
2966         ASSERT(cpuid_checkpass(cpu, 1));
2967         return (cpu->cpu_m.mcpu_cpi->cpi_model);
2968 }

2970 uint_t
2971 cpuid_get_ncpu_per_chip(cpu_t *cpu)
2972 {
2973         ASSERT(cpuid_checkpass(cpu, 1));
2974         return (cpu->cpu_m.mcpu_cpi->cpi_ncpu_per_chip);
2975 }

2977 uint_t
2978 cpuid_get_ncore_per_chip(cpu_t *cpu)
2979 {
2980         ASSERT(cpuid_checkpass(cpu, 1));
2981         return (cpu->cpu_m.mcpu_cpi->cpi_ncore_per_chip);
2982 }

2984 uint_t
2985 cpuid_get_ncpu_sharing_last_cache(cpu_t *cpu)
2986 {
2987         ASSERT(cpuid_checkpass(cpu, 2));
2988         return (cpu->cpu_m.mcpu_cpi->cpi_ncpu_shr_last_cache);
2989 }

2991 id_t
2992 cpuid_get_last_lvl_cacheid(cpu_t *cpu)
2993 {
2994         ASSERT(cpuid_checkpass(cpu, 2));
2995         return (cpu->cpu_m.mcpu_cpi->cpi_last_lvl_cacheid);
2996 }

2998 uint_t
2999 cpuid_getstep(cpu_t *cpu)
3000 {
3001         ASSERT(cpuid_checkpass(cpu, 1));
3002         return (cpu->cpu_m.mcpu_cpi->cpi_step);
3003 }

3005 uint_t
3006 cpuid_getsig(struct cpu *cpu)
3007 {
3008         ASSERT(cpuid_checkpass(cpu, 1));
3009         return (cpu->cpu_m.mcpu_cpi->cpi_std[1].cp_eax);
3010 }

3012 uint32_t
3013 cpuid_getchiprev(struct cpu *cpu)
3014 {
3015         ASSERT(cpuid_checkpass(cpu, 1));
3016         return (cpu->cpu_m.mcpu_cpi->cpi_chiprev);
3017 }

3019 const char *
3020 cpuid_getchiprevstr(struct cpu *cpu)
3021 {
3022         ASSERT(cpuid_checkpass(cpu, 1));
3023         return (cpu->cpu_m.mcpu_cpi->cpi_chiprevstr);
3024 }

3026 uint32_t
3027 cpuid_getsockettype(struct cpu *cpu)
3028 {
```



```
3029         ASSERT(cpuid_checkpass(cpu, 1));
3030         return (cpu->cpu_m.mcpu_cpi->cpi_socket);
3031 }

3033 const char *
3034 cpuid_getsocketstr(cpu_t *cpu)
3035 {
3036         static const char *socketstr = NULL;
3037         struct cpuid_info *cpi;

3039         ASSERT(cpuid_checkpass(cpu, 1));
3040         cpi = cpu->cpu_m.mcpu_cpi;

3042         /* Assume that socket types are the same across the system */
3043         if (socketstr == NULL)
3044                 socketstr = _cpuid_sktstr(cpi->cpi_vendor, cpi->cpi_family,
3045                     cpi->cpi_model, cpi->cpi_step);


3048         return (socketstr);
3049 }

3051 int
3052 cpuid_get_chipid(cpu_t *cpu)
3053 {
3054         ASSERT(cpuid_checkpass(cpu, 1));

3056         if (cpuid_is_cmt(cpu))
3057                 return (cpu->cpu_m.mcpu_cpi->cpi_chipid);
3058         return (cpu->cpu_id);
3059 }

3061 id_t
3062 cpuid_get_coreid(cpu_t *cpu)
3063 {
3064         ASSERT(cpuid_checkpass(cpu, 1));
3065         return (cpu->cpu_m.mcpu_cpi->cpi_coreid);
3066 }

3068 int
3069 cpuid_get_pkgcoreid(cpu_t *cpu)
3070 {
3071         ASSERT(cpuid_checkpass(cpu, 1));
3072         return (cpu->cpu_m.mcpu_cpi->cpi_pkgcoreid);
3073 }

3075 int
3076 cpuid_get_clogid(cpu_t *cpu)
3077 {
3078         ASSERT(cpuid_checkpass(cpu, 1));
3079         return (cpu->cpu_m.mcpu_cpi->cpi_clogid);
3080 }

3082 int
3083 cpuid_get_cacheid(cpu_t *cpu)
3084 {
3085         ASSERT(cpuid_checkpass(cpu, 1));
3086         return (cpu->cpu_m.mcpu_cpi->cpi_last_lvl_cacheid);
3087 }

3089 uint_t
3090 cpuid_get_procnodeid(cpu_t *cpu)
3091 {
3092         ASSERT(cpuid_checkpass(cpu, 1));
3093         return (cpu->cpu_m.mcpu_cpi->cpi_procnodeid);
3094 }
```

```
3096 uint_t
3097 cpuid_get_procnodes_per_pkg(cpu_t *cpu)
3098 {
3099         ASSERT(cpuid_checkpass(cpu, 1));
3100         return (cpu->cpu_m.mcpu_cpi->cpi_procnodes_per_pkg);
3101 }

3103 uint_t
3104 cpuid_get_compunitid(cpu_t *cpu)
3105 {
3106         ASSERT(cpuid_checkpass(cpu, 1));
3107         return (cpu->cpu_m.mcpu_cpi->cpi_compunitid);
3108 }

3110 uint_t
3111 cpuid_get_cores_per_compunit(cpu_t *cpu)
3112 {
3113         ASSERT(cpuid_checkpass(cpu, 1));
3114         return (cpu->cpu_m.mcpu_cpi->cpi_cores_per_compunit);
3115 }

3117 /*ARGSUSED*/
3118 int
3119 cpuid_have_cr8access(cpu_t *cpu)
3120 {
3121 #if defined(__amd64)
3122         return (1);
3123 #else
3124         struct cpuid_info *cpi;

3126         ASSERT(cpu != NULL);
3127         cpi = cpu->cpu_m.mcpu_cpi;
3128         if (cpi->cpi_vendor == X86_VENDOR_AMD && cpi->cpi_maxeax >= 1 &&
3129             (CPI_FEATURES_XTD_ECX(cpi) & CPUID_AMD_ECX_CR8D) != 0)
3130                 return (1);
3131         return (0);
3132 #endif
3133 }

3135 uint32_t
3136 cpuid_get_apicid(cpu_t *cpu)
3137 {
3138         ASSERT(cpuid_checkpass(cpu, 1));
3139         if (cpu->cpu_m.mcpu_cpi->cpi_maxeax < 1) {
3140                 return (UINT32_MAX);
3141         } else {
3142                 return (cpu->cpu_m.mcpu_cpi->cpi_apicid);
3143         }
3144 }

3146 void
3147 cpuid_get_addrsize(cpu_t *cpu, uint_t *pabits, uint_t *vabits)
3148 {
3149         struct cpuid_info *cpi;

3151         if (cpu == NULL)
3152                 cpu = CPU;
3153         cpi = cpu->cpu_m.mcpu_cpi;

3155         ASSERT(cpuid_checkpass(cpu, 1));

3157         if (pabits)
3158                 *pabits = cpi->cpi_pabits;
3159         if (vabits)
3160                 *vabits = cpi->cpi_vabits;
```



```
3161 }

3163 /*
3164  * Returns the number of data TLB entries for a corresponding
3165  * pagesize.  If it can't be computed, or isn't known, the
3166  * routine returns zero.  If you ask about an architecturally
3167  * impossible pagesize, the routine will panic (so that the
3168  * hat implementor knows that things are inconsistent.)
3169  */
3170 uint_t
3171 cpuid_get_dtlb_nent(cpu_t *cpu, size_t pagesize)
3172 {
3173         struct cpuid_info *cpi;
3174         uint_t dtlb_nent = 0;

3176         if (cpu == NULL)
3177                 cpu = CPU;
3178         cpi = cpu->cpu_m.mcpu_cpi;

3180         ASSERT(cpuid_checkpass(cpu, 1));

3182         /*
3183          * Check the L2 TLB info
3184          */
3185         if (cpi->cpi_xmaxeax >= 0x80000006) {
3186                 struct cpuid_regs *cp = &cpi->cpi_extd[6];

3188                 switch (pagesize) {

3190                 case 4 * 1024:
3191                         /*
3192                          * All zero in the top 16 bits of the register
3193                          * indicates a unified TLB. Size is in low 16 bits.
3194                          */
3195                         if ((cp->cp_ebx & 0xffff0000) == 0)
3196                                 dtlb_nent = cp->cp_ebx & 0x0000ffff;
3197                         else
3198                                 dtlb_nent = BITX(cp->cp_ebx, 27, 16);
3199                         break;

3201                 case 2 * 1024 * 1024:
3202                         if ((cp->cp_eax & 0xffff0000) == 0)
3203                                 dtlb_nent = cp->cp_eax & 0x0000ffff;
3204                         else
3205                                 dtlb_nent = BITX(cp->cp_eax, 27, 16);
3206                         break;

3208                 default:
3209                         panic("unknown L2 pagesize");
3210                         /*NOTREACHED*/
3211                 }
3212         }

3214         if (dtlb_nent != 0)
3215                 return (dtlb_nent);

3217         /*
3218          * No L2 TLB support for this size, try L1.
3219          */
3220         if (cpi->cpi_xmaxeax >= 0x80000005) {
3221                 struct cpuid_regs *cp = &cpi->cpi_extd[5];

3223                 switch (pagesize) {
3224                 case 4 * 1024:
3225                         dtlb_nent = BITX(cp->cp_ebx, 23, 16);
3226                         break;
```

```
3227                 case 2 * 1024 * 1024:
3228                         dtlb_nent = BITX(cp->cp_eax, 23, 16);
3229                         break;
3230                 default:
3231                         panic("unknown L1 d-TLB pagesize");
3232                         /*NOTREACHED*/
3233                 }
3234         }

3236         return (dtlb_nent);
3237 }

3239 /*
3240  * Return 0 if the erratum is not present or not applicable, positive
3241  * if it is, and negative if the status of the erratum is unknown.
3242  *
3243  * See "Revision Guide for AMD Athlon(tm) 64 and AMD Opteron(tm)
3244  * Processors" #25759, Rev 3.57, August 2005
3245  */
3246 int
3247 cpuid_opteron_erratum(cpu_t *cpu, uint_t erratum)
3248 {
3249         struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;
3250         uint_t eax;

3252         /*
3253          * Bail out if this CPU isn't an AMD CPU, or if it's
3254          * a legacy (32-bit) AMD CPU.
3255          */
3256         if (cpi->cpi_vendor != X86_VENDOR_AMD ||
3257             cpi->cpi_family == 4 || cpi->cpi_family == 5 ||
3258             cpi->cpi_family == 6)

3260                 return (0);

3262         eax = cpi->cpi_std[1].cp_eax;

3264 #define SH_B0(eax)      (eax == 0xf40 || eax == 0xf50)
3265 #define SH_B3(eax)      (eax == 0xf51)
3266 #define B(eax)          (SH_B0(eax) || SH_B3(eax))

3268 #define SH_C0(eax)      (eax == 0xf48 || eax == 0xf58)

3270 #define SH_CG(eax)      (eax == 0xf4a || eax == 0xf5a || eax == 0xf7a)
3271 #define DH_CG(eax)      (eax == 0xfc0 || eax == 0xfe0 || eax == 0xff0)
3272 #define CH_CG(eax)      (eax == 0xf82 || eax == 0xfb2)
3273 #define CG(eax)         (SH_CG(eax) || DH_CG(eax) || CH_CG(eax))

3275 #define SH_D0(eax)      (eax == 0x10f40 || eax == 0x10f50 || eax == 0x10f70)
3276 #define DH_D0(eax)      (eax == 0x10fc0 || eax == 0x10ff0)
3277 #define CH_D0(eax)      (eax == 0x10f80 || eax == 0x10fb0)
3278 #define D0(eax)         (SH_D0(eax) || DH_D0(eax) || CH_D0(eax))

3280 #define SH_E0(eax)      (eax == 0x20f50 || eax == 0x20f40 || eax == 0x20f70)
3281 #define JH_E1(eax)      (eax == 0x20f10)        /* JH8_E0 had 0x20f30 */
3282 #define DH_E3(eax)      (eax == 0x20fc0 || eax == 0x20ff0)
3283 #define SH_E4(eax)      (eax == 0x20f51 || eax == 0x20f71)
3284 #define BH_E4(eax)      (eax == 0x20fb1)
3285 #define SH_E5(eax)      (eax == 0x20f42)
3286 #define DH_E6(eax)      (eax == 0x20ff2 || eax == 0x20fc2)
3287 #define JH_E6(eax)      (eax == 0x20f12 || eax == 0x20f32)
3288 #define EX(eax)         (SH_E0(eax) || JH_E1(eax) || DH_E3(eax) || \
3289                             SH_E4(eax) || BH_E4(eax) || SH_E5(eax) || \
3290                             DH_E6(eax) || JH_E6(eax))

3292 #define DR_AX(eax)      (eax == 0x100f00 || eax == 0x100f01 || eax == 0x100f02)
```



```
3293 #define DR_B0(eax)      (eax == 0x100f20)
3294 #define DR_B1(eax)      (eax == 0x100f21)
3295 #define DR_BA(eax)      (eax == 0x100f2a)
3296 #define DR_B2(eax)      (eax == 0x100f22)
3297 #define DR_B3(eax)      (eax == 0x100f23)
3298 #define RB_C0(eax)      (eax == 0x100f40)

3300         switch (erratum) {
3301         case 1:
3302                 return (cpi->cpi_family < 0x10);
3303         case 51:        /* what does the asterisk mean? */
3304                 return (B(eax) || SH_C0(eax) || CG(eax));
3305         case 52:
3306                 return (B(eax));
3307         case 57:
3308                 return (cpi->cpi_family <= 0x11);
3309         case 58:
3310                 return (B(eax));
3311         case 60:
3312                 return (cpi->cpi_family <= 0x11);
3313         case 61:
3314         case 62:
3315         case 63:
3316         case 64:
3317         case 65:
3318         case 66:
3319         case 68:
3320         case 69:
3321         case 70:
3322         case 71:
3323                 return (B(eax));
3324         case 72:
3325                 return (SH_B0(eax));
3326         case 74:
3327                 return (B(eax));
3328         case 75:
3329                 return (cpi->cpi_family < 0x10);
3330         case 76:
3331                 return (B(eax));
3332         case 77:
3333                 return (cpi->cpi_family <= 0x11);
3334         case 78:
3335                 return (B(eax) || SH_C0(eax));
3336         case 79:
3337                 return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax) || EX(eax));
3338         case 80:
3339         case 81:
3340         case 82:
3341                 return (B(eax));
3342         case 83:
3343                 return (B(eax) || SH_C0(eax) || CG(eax));
3344         case 85:
3345                 return (cpi->cpi_family < 0x10);
3346         case 86:
3347                 return (SH_C0(eax) || CG(eax));
3348         case 88:
3349 #if !defined(__amd64)
3350                 return (0);
3351 #else
3352                 return (B(eax) || SH_C0(eax));
3353 #endif
3354         case 89:
3355                 return (cpi->cpi_family < 0x10);
3356         case 90:
3357                 return (B(eax) || SH_C0(eax) || CG(eax));
3358         case 91:
```

```
3359         case 92:
3360                 return (B(eax) || SH_C0(eax));
3361         case 93:
3362                 return (SH_C0(eax));
3363         case 94:
3364                 return (B(eax) || SH_C0(eax) || CG(eax));
3365         case 95:
3366 #if !defined(__amd64)
3367                 return (0);
3368 #else
3369                 return (B(eax) || SH_C0(eax));
3370 #endif
3371         case 96:
3372                 return (B(eax) || SH_C0(eax) || CG(eax));
3373         case 97:
3374         case 98:
3375                 return (SH_C0(eax) || CG(eax));
3376         case 99:
3377                 return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax));
3378         case 100:
3379                 return (B(eax) || SH_C0(eax));
3380         case 101:
3381         case 103:
3382                 return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax));
3383         case 104:
3384                 return (SH_C0(eax) || CG(eax) || D0(eax));
3385         case 105:
3386         case 106:
3387         case 107:
3388                 return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax));
3389         case 108:
3390                 return (DH_CG(eax));
3391         case 109:
3392                 return (SH_C0(eax) || CG(eax) || D0(eax));
3393         case 110:
3394                 return (D0(eax) || EX(eax));
3395         case 111:
3396                 return (CG(eax));
3397         case 112:
3398                 return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax) || EX(eax));
3399         case 113:
3400                 return (eax == 0x20fc0);
3401         case 114:
3402                 return (SH_E0(eax) || JH_E1(eax) || DH_E3(eax));
3403         case 115:
3404                 return (SH_E0(eax) || JH_E1(eax));
3405         case 116:
3406                 return (SH_E0(eax) || JH_E1(eax) || DH_E3(eax));
3407         case 117:
3408                 return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax));
3409         case 118:
3410                 return (SH_E0(eax) || JH_E1(eax) || SH_E4(eax) || BH_E4(eax) ||
3411                     JH_E6(eax));
3412         case 121:
3413                 return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax) || EX(eax));
3414         case 122:
3415                 return (cpi->cpi_family < 0x10 || cpi->cpi_family == 0x11);
3416         case 123:
3417                 return (JH_E1(eax) || BH_E4(eax) || JH_E6(eax));
3418         case 131:
3419                 return (cpi->cpi_family < 0x10);
3420         case 6336786:
3421                 /*
3422                  * Test for AdvPowerMgmtInfo.TscPStateInvariant
3423                  * if this is a K8 family or newer processor
3424                  */
```



```
3425                 if (CPI_FAMILY(cpi) == 0xf) {
3426                         struct cpuid_regs regs;
3427                         regs.cp_eax = 0x80000007;
3428                         (void) __cpuid_insn(&regs);
3429                         return (!(regs.cp_edx & 0x100));
3430                 }
3431                 return (0);
3432         case 6323525:
3433                 return (((((eax >> 12) & 0xff00) + (eax & 0xf00)) |
3434                     (((eax >> 4) & 0xf) | ((eax >> 12) & 0xf0))) < 0xf40);
3435
3436         case 6671130:
3437                 /*
3438                  * check for processors (pre-Shanghai) that do not provide
3439                  * optimal management of 1gb ptes in its tlb.
3440                  */
3441                 return (cpi->cpi_family == 0x10 && cpi->cpi_model < 4);
3442
3443         case 298:
3444                 return (DR_AX(eax) || DR_B0(eax) || DR_B1(eax) || DR_BA(eax) ||
3445                     DR_B2(eax) || RB_C0(eax));
3446
3447         case 721:
3448 #if defined(__amd64)
3449                 return (cpi->cpi_family == 0x10 || cpi->cpi_family == 0x12);
3450 #else
3451                 return (0);
3452 #endif
3453
3454         default:
3455                 return (-1);
3456
3457         }
3458 }
3459
3460 /*
3461  * Determine if specified erratum is present via OSVW (OS Visible Workaround).
3462  * Return 1 if erratum is present, 0 if not present and -1 if indeterminate.
3463  */
3464 int
3465 osvw_opteron_erratum(cpu_t *cpu, uint_t erratum)
3466 {
3467         struct cpuid_info       *cpi;
3468         uint_t                  osvwid;
3469         static int              osvwfeature = -1;
3470         uint64_t                osvwlength;
3471
3472
3473         cpi = cpu->cpu_m.mcpu_cpi;
3474
3475         /* confirm OSVW supported */
3476         if (osvwfeature == -1) {
3477                 osvwfeature = cpi->cpi_extd[1].cp_ecx & CPUID_AMD_ECX_OSVW;
3478         } else {
3479                 /* assert that osvw feature setting is consistent on all cpus */
3480                 ASSERT(osvwfeature ==
3481                     (cpi->cpi_extd[1].cp_ecx & CPUID_AMD_ECX_OSVW));
3482         }
3483         if (!osvwfeature)
3484                 return (-1);
3485
3486         osvwlength = rdmsr(MSR_AMD_OSVW_ID_LEN) & OSVW_ID_LEN_MASK;
3487
3488         switch (erratum) {
3489         case 298:       /* osvwid is 0 */
3490                 osvwid = 0;
```

```
3491                 if (osvwlength <= (uint64_t)osvwid) {
3492                         /* osvwid 0 is unknown */
3493                         return (-1);
3494                 }
3495 
3496                 /*
3497                  * Check the OSVW STATUS MSR to determine the state
3498                  * of the erratum where:
3499                  *   0 - fixed by HW
3500                  *   1 - BIOS has applied the workaround when BIOS
3501                  *   workaround is available. (Or for other errata,
3502                  *   OS workaround is required.)
3503                  * For a value of 1, caller will confirm that the
3504                  * erratum 298 workaround has indeed been applied by BIOS.
3505                  *
3506                  * A 1 may be set in cpus that have a HW fix
3507                  * in a mixed cpu system. Regarding erratum 298:
3508                  *   In a multiprocessor platform, the workaround above
3509                  *   should be applied to all processors regardless of
3510                  *   silicon revision when an affected processor is
3511                  *   present.
3512                  */
3513 
3514                 return (rdmsr(MSR_AMD_OSVW_STATUS +
3515                     (osvwid / OSVW_ID_CNT_PER_MSR)) &
3516                     (1ULL << (osvwid % OSVW_ID_CNT_PER_MSR)));
3517 
3518         default:
3519                 return (-1);
3520         }
3521 }
3522 
3523 static const char assoc_str[] = "associativity";
3524 static const char line_str[] = "line-size";
3525 static const char size_str[] = "size";
3526 
3527 static void
3528 add_cache_prop(dev_info_t *devi, const char *label, const char *type,
3529     uint32_t val)
3530 {
3531         char buf[128];
3532 
3533         /*
3534          * ndi_prop_update_int() is used because it is desirable for
3535          * DDI_PROP_HW_DEF and DDI_PROP_DONTSLEEP to be set.
3536          */
3537         if (snprintf(buf, sizeof (buf), "%s-%s", label, type) < sizeof (buf))
3538                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, devi, buf, val);
3539 }
3540 
3541 /*
3542  * Intel-style cache/tlb description
3543  *
3544  * Standard cpuid level 2 gives a randomly ordered
3545  * selection of tags that index into a table that describes
3546  * cache and tlb properties.
3547  */
3548 
3549 static const char l1_icache_str[] = "l1-icache";
3550 static const char l1_dcache_str[] = "l1-dcache";
3551 static const char l2_cache_str[] = "l2-cache";
3552 static const char l3_cache_str[] = "l3-cache";
3553 static const char itlb4k_str[] = "itlb-4K";
3554 static const char dtlb4k_str[] = "dtlb-4K";
3555 static const char itlb2M_str[] = "itlb-2M";
3556 static const char itlb4M_str[] = "itlb-4M";
```



```
3557 static const char dtlb4M_str[] = "dtlb-4M";
3558 static const char dtlb24_str[] = "dtlb0-2M-4M";
3559 static const char itlb424_str[] = "itlb-4K-2M-4M";
3560 static const char itlb24_str[] = "itlb-2M-4M";
3561 static const char dtlb44_str[] = "dtlb-4K-4M";
3562 static const char sl1_dcache_str[] = "sectored-l1-dcache";
3563 static const char sl2_cache_str[] = "sectored-l2-cache";
3564 static const char itrace_str[] = "itrace-cache";
3565 static const char sl3_cache_str[] = "sectored-l3-cache";
3566 static const char sh_l2_tlb4k_str[] = "shared-l2-tlb-4k";
3567 
3568 static const struct cachetab {
3569         uint8_t         ct_code;
3570         uint8_t         ct_assoc;
3571         uint16_t        ct_line_size;
3572         size_t          ct_size;
3573         const char      *ct_label;
3574 } intel_ctab[] = {
3575         /*
3576          * maintain descending order!
3577          *
3578          * Codes ignored - Reason
3579          * ----------------------
3580          * 40H - intel_cpuid_4_cache_info() disambiguates l2/l3 cache
3581          * f0H/f1H - Currently we do not interpret prefetch size by design
3582          */
3583         { 0xe4, 16, 64, 8*1024*1024, l3_cache_str},
3584         { 0xe3, 16, 64, 4*1024*1024, l3_cache_str},
3585         { 0xe2, 16, 64, 2*1024*1024, l3_cache_str},
3586         { 0xde, 12, 64, 6*1024*1024, l3_cache_str},
3587         { 0xdd, 12, 64, 3*1024*1024, l3_cache_str},
3588         { 0xdc, 12, 64, ((1*1024*1024)+(512*1024)), l3_cache_str},
3589         { 0xd8, 8, 64, 4*1024*1024, l3_cache_str},
3590         { 0xd7, 8, 64, 2*1024*1024, l3_cache_str},
3591         { 0xd6, 8, 64, 1*1024*1024, l3_cache_str},
3592         { 0xd2, 4, 64, 2*1024*1024, l3_cache_str},
3593         { 0xd1, 4, 64, 1*1024*1024, l3_cache_str},
3594         { 0xd0, 4, 64, 512*1024, l3_cache_str},
3595         { 0xca, 4, 0, 512, sh_l2_tlb4k_str},
3596         { 0xc0, 4, 0, 8, dtlb44_str },
3597         { 0xba, 4, 0, 64, dtlb4k_str },
3598         { 0xb4, 4, 0, 256, dtlb4k_str },
3599         { 0xb3, 4, 0, 128, dtlb4k_str },
3600         { 0xb2, 4, 0, 64, itlb4k_str },
3601         { 0xb0, 4, 0, 128, itlb4k_str },
3602         { 0x87, 8, 64, 1024*1024, l2_cache_str},
3603         { 0x86, 4, 64, 512*1024, l2_cache_str},
3604         { 0x85, 8, 32, 2*1024*1024, l2_cache_str},
3605         { 0x84, 8, 32, 1024*1024, l2_cache_str},
3606         { 0x83, 8, 32, 512*1024, l2_cache_str},
3607         { 0x82, 8, 32, 256*1024, l2_cache_str},
3608         { 0x80, 8, 64, 512*1024, l2_cache_str},
3609         { 0x7f, 2, 64, 512*1024, l2_cache_str},
3610         { 0x7d, 8, 64, 2*1024*1024, sl2_cache_str},
3611         { 0x7c, 8, 64, 1024*1024, sl2_cache_str},
3612         { 0x7b, 8, 64, 512*1024, sl2_cache_str},
3613         { 0x7a, 8, 64, 256*1024, sl2_cache_str},
3614         { 0x79, 8, 64, 128*1024, sl2_cache_str},
3615         { 0x78, 8, 64, 1024*1024, l2_cache_str},
3616         { 0x73, 8, 0, 64*1024, itrace_str},
3617         { 0x72, 8, 0, 32*1024, itrace_str},
3618         { 0x71, 8, 0, 16*1024, itrace_str},
3619         { 0x70, 8, 0, 12*1024, itrace_str},
3620         { 0x68, 4, 64, 32*1024, sl1_dcache_str},
3621         { 0x67, 4, 64, 16*1024, sl1_dcache_str},
3622         { 0x66, 4, 64, 8*1024, sl1_dcache_str},
```

```
3623         { 0x60, 8, 64, 16*1024, sl1_dcache_str},
3624         { 0x5d, 0, 0, 256, dtlb44_str},
3625         { 0x5c, 0, 0, 128, dtlb44_str},
3626         { 0x5b, 0, 0, 64, dtlb44_str},
3627         { 0x5a, 4, 0, 32, dtlb24_str},
3628         { 0x59, 0, 0, 16, dtlb4k_str},
3629         { 0x57, 4, 0, 16, dtlb4k_str},
3630         { 0x56, 4, 0, 16, dtlb4M_str},
3631         { 0x55, 0, 0, 7, itlb24_str},
3632         { 0x52, 0, 0, 256, itlb424_str},
3633         { 0x51, 0, 0, 128, itlb424_str},
3634         { 0x50, 0, 0, 64, itlb424_str},
3635         { 0x4f, 0, 0, 32, itlb4k_str},
3636         { 0x4e, 24, 64, 6*1024*1024, l2_cache_str},
3637         { 0x4d, 16, 64, 16*1024*1024, l3_cache_str},
3638         { 0x4c, 12, 64, 12*1024*1024, l3_cache_str},
3639         { 0x4b, 16, 64, 8*1024*1024, l3_cache_str},
3640         { 0x4a, 12, 64, 6*1024*1024, l3_cache_str},
3641         { 0x49, 16, 64, 4*1024*1024, l3_cache_str},
3642         { 0x48, 12, 64, 3*1024*1024, l2_cache_str},
3643         { 0x47, 8, 64, 8*1024*1024, l3_cache_str},
3644         { 0x46, 4, 64, 4*1024*1024, l3_cache_str},
3645         { 0x45, 4, 32, 2*1024*1024, l2_cache_str},
3646         { 0x44, 4, 32, 1024*1024, l2_cache_str},
3647         { 0x43, 4, 32, 512*1024, l2_cache_str},
3648         { 0x42, 4, 32, 256*1024, l2_cache_str},
3649         { 0x41, 4, 32, 128*1024, l2_cache_str},
3650         { 0x3e, 4, 64, 512*1024, sl2_cache_str},
3651         { 0x3d, 6, 64, 384*1024, sl2_cache_str},
3652         { 0x3c, 4, 64, 256*1024, sl2_cache_str},
3653         { 0x3b, 2, 64, 128*1024, sl2_cache_str},
3654         { 0x3a, 6, 64, 192*1024, sl2_cache_str},
3655         { 0x39, 4, 64, 128*1024, sl2_cache_str},
3656         { 0x30, 8, 64, 32*1024, l1_icache_str},
3657         { 0x2c, 8, 64, 32*1024, l1_dcache_str},
3658         { 0x29, 8, 64, 4096*1024, sl3_cache_str},
3659         { 0x25, 8, 64, 2048*1024, sl3_cache_str},
3660         { 0x23, 8, 64, 1024*1024, sl3_cache_str},
3661         { 0x22, 4, 64, 512*1024, sl3_cache_str},
3662         { 0x0e, 6, 64, 24*1024, l1_dcache_str},
3663         { 0x0d, 4, 32, 16*1024, l1_dcache_str},
3664         { 0x0c, 4, 32, 16*1024, l1_dcache_str},
3665         { 0x0b, 4, 0, 4, itlb4M_str},
3666         { 0x0a, 2, 32, 8*1024, l1_dcache_str},
3667         { 0x08, 4, 32, 16*1024, l1_icache_str},
3668         { 0x06, 4, 32, 8*1024, l1_icache_str},
3669         { 0x05, 4, 0, 32, dtlb4M_str},
3670         { 0x04, 4, 0, 8, dtlb4M_str},
3671         { 0x03, 4, 0, 64, dtlb4k_str},
3672         { 0x02, 4, 0, 2, itlb4M_str},
3673         { 0x01, 4, 0, 32, itlb4k_str},
3674         { 0 }
3675 };

3677 static const struct cachetab cyrix_ctab[] = {
3678         { 0x70, 4, 0, 32, "tlb-4K" },
3679         { 0x80, 4, 16, 16*1024, "l1-cache" },
3680         { 0 }
3681 };

3683 /*
3684  * Search a cache table for a matching entry
3685  */
3686 static const struct cachetab *
3687 find_cacheent(const struct cachetab *ct, uint_t code)
3688 {
```



```
3689         if (code != 0) {
3690                 for (; ct->ct_code != 0; ct++)
3691                         if (ct->ct_code <= code)
3692                                 break;
3693                 if (ct->ct_code == code)
3694                         return (ct);
3695         }
3696         return (NULL);
3697 }

3699 /*
3700  * Populate cachetab entry with L2 or L3 cache-information using
3701  * cpuid function 4. This function is called from intel_walk_cacheinfo()
3702  * when descriptor 0x49 is encountered. It returns 0 if no such cache
3703  * information is found.
3704  */
3705 static int
3706 intel_cpuid_4_cache_info(struct cachetab *ct, struct cpuid_info *cpi)
3707 {
3708         uint32_t level, i;
3709         int ret = 0;

3711         for (i = 0; i < cpi->cpi_std_4_size; i++) {
3712                 level = CPI_CACHE_LVL(cpi->cpi_std_4[i]);

3714                 if (level == 2 || level == 3) {
3715                         ct->ct_assoc = CPI_CACHE_WAYS(cpi->cpi_std_4[i]) + 1;
3716                         ct->ct_line_size =
3717                             CPI_CACHE_COH_LN_SZ(cpi->cpi_std_4[i]) + 1;
3718                         ct->ct_size = ct->ct_assoc *
3719                             (CPI_CACHE_PARTS(cpi->cpi_std_4[i]) + 1) *
3720                             ct->ct_line_size *
3721                             (cpi->cpi_std_4[i]->cp_ecx + 1);

3723                         if (level == 2) {
3724                                 ct->ct_label = l2_cache_str;
3725                         } else if (level == 3) {
3726                                 ct->ct_label = l3_cache_str;
3727                         }
3728                         ret = 1;
3729                 }
3730         }

3732         return (ret);
3733 }

3735 /*
3736  * Walk the cacheinfo descriptor, applying 'func' to every valid element
3737  * The walk is terminated if the walker returns non-zero.
3738  */
3739 static void
3740 intel_walk_cacheinfo(struct cpuid_info *cpi,
3741     void *arg, int (*func)(void *, const struct cachetab *))
3742 {
3743         const struct cachetab *ct;
3744         struct cachetab des_49_ct, des_b1_ct;
3745         uint8_t *dp;
3746         int i;

3748         if ((dp = cpi->cpi_cacheinfo) == NULL)
3749                 return;
3750         for (i = 0; i < cpi->cpi_ncache; i++, dp++) {
3751                 /*
3752                  * For overloaded descriptor 0x49 we use cpuid function 4
3753                  * if supported by the current processor, to create
3754                  * cache information.
```

```
3755                  * For overloaded descriptor 0xb1 we use X86_PAE flag
3756                  * to disambiguate the cache information.
3757                  */
3758                 if (*dp == 0x49 && cpi->cpi_maxeax >= 0x4 &&
3759                     intel_cpuid_4_cache_info(&des_49_ct, cpi) == 1) {
3760                         ct = &des_49_ct;
3761                 } else if (*dp == 0xb1) {
3762                         des_b1_ct.ct_code = 0xb1;
3763                         des_b1_ct.ct_assoc = 4;
3764                         des_b1_ct.ct_line_size = 0;
3765                         if (is_x86_feature(x86_featureset, X86FSET_PAE)) {
3766                                 des_b1_ct.ct_size = 8;
3767                                 des_b1_ct.ct_label = itlb2M_str;
3768                         } else {
3769                                 des_b1_ct.ct_size = 4;
3770                                 des_b1_ct.ct_label = itlb4M_str;
3771                         }
3772                         ct = &des_b1_ct;
3773                 } else {
3774                         if ((ct = find_cacheent(intel_ctab, *dp)) == NULL) {
3775                                 continue;
3776                         }
3777                 }

3779                 if (func(arg, ct) != 0) {
3780                         break;
3781                 }
3782         }
3783 }

3785 /*
3786  * (Like the Intel one, except for Cyrix CPUs)
3787  */
3788 static void
3789 cyrix_walk_cacheinfo(struct cpuid_info *cpi,
3790     void *arg, int (*func)(void *, const struct cachetab *))
3791 {
3792         const struct cachetab *ct;
3793         uint8_t *dp;
3794         int i;

3796         if ((dp = cpi->cpi_cacheinfo) == NULL)
3797                 return;
3798         for (i = 0; i < cpi->cpi_ncache; i++, dp++) {
3799                 /*
3800                  * Search Cyrix-specific descriptor table first ..
3801                  */
3802                 if ((ct = find_cacheent(cyrix_ctab, *dp)) != NULL) {
3803                         if (func(arg, ct) != 0)
3804                                 break;
3805                         continue;
3806                 }
3807                 /*
3808                  * .. else fall back to the Intel one
3809                  */
3810                 if ((ct = find_cacheent(intel_ctab, *dp)) != NULL) {
3811                         if (func(arg, ct) != 0)
3812                                 break;
3813                         continue;
3814                 }
3815         }
3816 }

3818 /*
3819  * A cacheinfo walker that adds associativity, line-size, and size properties
3820  * to the devinfo node it is passed as an argument.
```



```
3821  */
3822 static int
3823 add_cacheent_props(void *arg, const struct cachetab *ct)
3824 {
3825         dev_info_t *devi = arg;

3827         add_cache_prop(devi, ct->ct_label, assoc_str, ct->ct_assoc);
3828         if (ct->ct_line_size != 0)
3829                 add_cache_prop(devi, ct->ct_label, line_str,
3830                     ct->ct_line_size);
3831         add_cache_prop(devi, ct->ct_label, size_str, ct->ct_size);
3832         return (0);
3833 }


3836 static const char fully_assoc[] = "fully-associative?";

3838 /*
3839  * AMD style cache/tlb description
3840  *
3841  * Extended functions 5 and 6 directly describe properties of
3842  * tlbs and various cache levels.
3843  */
3844 static void
3845 add_amd_assoc(dev_info_t *devi, const char *label, uint_t assoc)
3846 {
3847         switch (assoc) {
3848         case 0: /* reserved; ignore */
3849                 break;
3850         default:
3851                 add_cache_prop(devi, label, assoc_str, assoc);
3852                 break;
3853         case 0xff:
3854                 add_cache_prop(devi, label, fully_assoc, 1);
3855                 break;
3856         }
3857 }

3859 static void
3860 add_amd_tlb(dev_info_t *devi, const char *label, uint_t assoc, uint_t size)
3861 {
3862         if (size == 0)
3863                 return;
3864         add_cache_prop(devi, label, size_str, size);
3865         add_amd_assoc(devi, label, assoc);
3866 }

3868 static void
3869 add_amd_cache(dev_info_t *devi, const char *label,
3870     uint_t size, uint_t assoc, uint_t lines_per_tag, uint_t line_size)
3871 {
3872         if (size == 0 || line_size == 0)
3873                 return;
3874         add_amd_assoc(devi, label, assoc);
3875         /*
3876          * Most AMD parts have a sectored cache. Multiple cache lines are
3877          * associated with each tag. A sector consists of all cache lines
3878          * associated with a tag. For example, the AMD K6-III has a sector
3879          * size of 2 cache lines per tag.
3880          */
3881         if (lines_per_tag != 0)
3882                 add_cache_prop(devi, label, "lines-per-tag", lines_per_tag);
3883         add_cache_prop(devi, label, line_str, line_size);
3884         add_cache_prop(devi, label, size_str, size * 1024);
3885 }
```

```
3887 static void
3888 add_amd_l2_assoc(dev_info_t *devi, const char *label, uint_t assoc)
3889 {
3890         switch (assoc) {
3891         case 0: /* off */
3892                 break;
3893         case 1:
3894         case 2:
3895         case 4:
3896                 add_cache_prop(devi, label, assoc_str, assoc);
3897                 break;
3898         case 6:
3899                 add_cache_prop(devi, label, assoc_str, 8);
3900                 break;
3901         case 8:
3902                 add_cache_prop(devi, label, assoc_str, 16);
3903                 break;
3904         case 0xf:
3905                 add_cache_prop(devi, label, fully_assoc, 1);
3906                 break;
3907         default: /* reserved; ignore */
3908                 break;
3909         }
3910 }

3912 static void
3913 add_amd_l2_tlb(dev_info_t *devi, const char *label, uint_t assoc, uint_t size)
3914 {
3915         if (size == 0 || assoc == 0)
3916                 return;
3917         add_amd_l2_assoc(devi, label, assoc);
3918         add_cache_prop(devi, label, size_str, size);
3919 }

3921 static void
3922 add_amd_l2_cache(dev_info_t *devi, const char *label,
3923     uint_t size, uint_t assoc, uint_t lines_per_tag, uint_t line_size)
3924 {
3925         if (size == 0 || assoc == 0 || line_size == 0)
3926                 return;
3927         add_amd_l2_assoc(devi, label, assoc);
3928         if (lines_per_tag != 0)
3929                 add_cache_prop(devi, label, "lines-per-tag", lines_per_tag);
3930         add_cache_prop(devi, label, line_str, line_size);
3931         add_cache_prop(devi, label, size_str, size * 1024);
3932 }

3934 static void
3935 amd_cache_info(struct cpuid_info *cpi, dev_info_t *devi)
3936 {
3937         struct cpuid_regs *cp;

3939         if (cpi->cpi_xmaxeax < 0x80000005)
3940                 return;
3941         cp = &cpi->cpi_extd[5];

3943         /*
3944          * 4M/2M L1 TLB configuration
3945          *
3946          * We report the size for 2M pages because AMD uses two
3947          * TLB entries for one 4M page.
3948          */
3949         add_amd_tlb(devi, "dtlb-2M",
3950             BITX(cp->cp_eax, 31, 24), BITX(cp->cp_eax, 23, 16));
3951         add_amd_tlb(devi, "itlb-2M",
3952             BITX(cp->cp_eax, 15, 8), BITX(cp->cp_eax, 7, 0));
```



```
3954         /*
3955          * 4K L1 TLB configuration
3956          */

3958         switch (cpi->cpi_vendor) {
3959                 uint_t nentries;
3960         case X86_VENDOR_TM:
3961                 if (cpi->cpi_family >= 5) {
3962                         /*
3963                          * Crusoe processors have 256 TLB entries, but
3964                          * cpuid data format constrains them to only
3965                          * reporting 255 of them.
3966                          */
3967                         if ((nentries = BITX(cp->cp_ebx, 23, 16)) == 255)
3968                                 nentries = 256;
3969                         /*
3970                          * Crusoe processors also have a unified TLB
3971                          */
3972                         add_amd_tlb(devi, "tlb-4K", BITX(cp->cp_ebx, 31, 24),
3973                             nentries);
3974                         break;
3975                 }
3976                 /*FALLTHROUGH*/
3977         default:
3978                 add_amd_tlb(devi, itlb4k_str,
3979                     BITX(cp->cp_ebx, 31, 24), BITX(cp->cp_ebx, 23, 16));
3980                 add_amd_tlb(devi, dtlb4k_str,
3981                     BITX(cp->cp_ebx, 15, 8), BITX(cp->cp_ebx, 7, 0));
3982                 break;
3983         }

3985         /*
3986          * data L1 cache configuration
3987          */

3989         add_amd_cache(devi, l1_dcache_str,
3990             BITX(cp->cp_ecx, 31, 24), BITX(cp->cp_ecx, 23, 16),
3991             BITX(cp->cp_ecx, 15, 8), BITX(cp->cp_ecx, 7, 0));

3993         /*
3994          * code L1 cache configuration
3995          */

3997         add_amd_cache(devi, l1_icache_str,
3998             BITX(cp->cp_edx, 31, 24), BITX(cp->cp_edx, 23, 16),
3999             BITX(cp->cp_edx, 15, 8), BITX(cp->cp_edx, 7, 0));

4001         if (cpi->cpi_xmaxeax < 0x80000006)
4002                 return;
4003         cp = &cpi->cpi_extd[6];

4005         /* Check for a unified L2 TLB for large pages */

4007         if (BITX(cp->cp_eax, 31, 16) == 0)
4008                 add_amd_l2_tlb(devi, "l2-tlb-2M",
4009                     BITX(cp->cp_eax, 15, 12), BITX(cp->cp_eax, 11, 0));
4010         else {
4011                 add_amd_l2_tlb(devi, "l2-dtlb-2M",
4012                     BITX(cp->cp_eax, 31, 28), BITX(cp->cp_eax, 27, 16));
4013                 add_amd_l2_tlb(devi, "l2-itlb-2M",
4014                     BITX(cp->cp_eax, 15, 12), BITX(cp->cp_eax, 11, 0));
4015         }

4017         /* Check for a unified L2 TLB for 4K pages */
```

```
4019         if (BITX(cp->cp_ebx, 31, 16) == 0) {
4020                 add_amd_l2_tlb(devi, "l2-tlb-4K",
4021                     BITX(cp->cp_eax, 15, 12), BITX(cp->cp_eax, 11, 0));
4022         } else {
4023                 add_amd_l2_tlb(devi, "l2-dtlb-4K",
4024                     BITX(cp->cp_eax, 31, 28), BITX(cp->cp_eax, 27, 16));
4025                 add_amd_l2_tlb(devi, "l2-itlb-4K",
4026                     BITX(cp->cp_eax, 15, 12), BITX(cp->cp_eax, 11, 0));
4027         }

4029         add_amd_l2_cache(devi, l2_cache_str,
4030             BITX(cp->cp_ecx, 31, 16), BITX(cp->cp_ecx, 15, 12),
4031             BITX(cp->cp_ecx, 11, 8), BITX(cp->cp_ecx, 7, 0));
4032 }

4034 /*
4035  * There are two basic ways that the x86 world describes it cache
4036  * and tlb architecture - Intel's way and AMD's way.
4037  *
4038  * Return which flavor of cache architecture we should use
4039  */
4040 static int
4041 x86_which_cacheinfo(struct cpuid_info *cpi)
4042 {
4043         switch (cpi->cpi_vendor) {
4044         case X86_VENDOR_Intel:
4045                 if (cpi->cpi_maxeax >= 2)
4046                         return (X86_VENDOR_Intel);
4047                 break;
4048         case X86_VENDOR_AMD:
4049                 /*
4050                  * The K5 model 1 was the first part from AMD that reported
4051                  * cache sizes via extended cpuid functions.
4052                  */
4053                 if (cpi->cpi_family > 5 ||
4054                     (cpi->cpi_family == 5 && cpi->cpi_model >= 1))
4055                         return (X86_VENDOR_AMD);
4056                 break;
4057         case X86_VENDOR_TM:
4058                 if (cpi->cpi_family >= 5)
4059                         return (X86_VENDOR_AMD);
4060                 /*FALLTHROUGH*/
4061         default:
4062                 /*
4063                  * If they have extended CPU data for 0x80000005
4064                  * then we assume they have AMD-format cache
4065                  * information.
4066                  *
4067                  * If not, and the vendor happens to be Cyrix,
4068                  * then try our-Cyrix specific handler.
4069                  *
4070                  * If we're not Cyrix, then assume we're using Intel's
4071                  * table-driven format instead.
4072                  */
4073                 if (cpi->cpi_xmaxeax >= 0x80000005)
4074                         return (X86_VENDOR_AMD);
4075                 else if (cpi->cpi_vendor == X86_VENDOR_Cyrix)
4076                         return (X86_VENDOR_Cyrix);
4077                 else if (cpi->cpi_maxeax >= 2)
4078                         return (X86_VENDOR_Intel);
4079                 break;
4080         }
4081         return (-1);
4082 }

4084 void
```



```
4085 cpuid_set_cpu_properties(void *dip, processorid_t cpu_id,
4086     struct cpuid_info *cpi)
4087 {
4088         dev_info_t *cpu_devi;
4089         int create;

4091         cpu_devi = (dev_info_t *)dip;

4093         /* device_type */
4094         (void) ndi_prop_update_string(DDI_DEV_T_NONE, cpu_devi,
4095             "device_type", "cpu");

4097         /* reg */
4098         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4099             "reg", cpu_id);

4101         /* cpu-mhz, and clock-frequency */
4102         if (cpu_freq > 0) {
4103                 long long mul;

4105                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4106                     "cpu-mhz", cpu_freq);
4107                 if ((mul = cpu_freq * 1000000LL) <= INT_MAX)
4108                         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4109                             "clock-frequency", (int)mul);
4110         }

4112         if (!is_x86_feature(x86_featureset, X86FSET_CPUID)) {
4113                 return;
4114         }

4116         /* vendor-id */
4117         (void) ndi_prop_update_string(DDI_DEV_T_NONE, cpu_devi,
4118             "vendor-id", cpi->cpi_vendorstr);

4120         if (cpi->cpi_maxeax == 0) {
4121                 return;
4122         }

4124         /*
4125          * family, model, and step
4126          */
4127         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4128             "family", CPI_FAMILY(cpi));
4129         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4130             "cpu-model", CPI_MODEL(cpi));
4131         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4132             "stepping-id", CPI_STEP(cpi));

4134         /* type */
4135         switch (cpi->cpi_vendor) {
4136         case X86_VENDOR_Intel:
4137                 create = 1;
4138                 break;
4139         default:
4140                 create = 0;
4141                 break;
4142         }
4143         if (create)
4144                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4145                     "type", CPI_TYPE(cpi));

4147         /* ext-family */
4148         switch (cpi->cpi_vendor) {
4149         case X86_VENDOR_Intel:
4150         case X86_VENDOR_AMD:
```

```
4151                create = cpi->cpi_family >= 0xf;
4152                break;
4153        default:
4154                create = 0;
4155                break;
4156        }
4157        if (create)
4158                (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4159                    "ext-family", CPI_FAMILY_XTD(cpi));

4161        /* ext-model */
4162        switch (cpi->cpi_vendor) {
4163        case X86_VENDOR_Intel:
4164                create = IS_EXTENDED_MODEL_INTEL(cpi);
4165                break;
4166        case X86_VENDOR_AMD:
4167                create = CPI_FAMILY(cpi) == 0xf;
4168                break;
4169        default:
4170                create = 0;
4171                break;
4172        }
4173        if (create)
4174                (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4175                    "ext-model", CPI_MODEL_XTD(cpi));

4177        /* generation */
4178        switch (cpi->cpi_vendor) {
4179        case X86_VENDOR_AMD:
4180                /*
4181                 * AMD K5 model 1 was the first part to support this
4182                 */
4183                create = cpi->cpi_xmaxeax >= 0x80000001;
4184                break;
4185        default:
4186                create = 0;
4187                break;
4188        }
4189        if (create)
4190                (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4191                    "generation", BITX((cpi)->cpi_extd[1].cp_eax, 11, 8));

4193        /* brand-id */
4194        switch (cpi->cpi_vendor) {
4195        case X86_VENDOR_Intel:
4196                /*
4197                 * brand id first appeared on Pentium III Xeon model 8,
4198                 * and Celeron model 8 processors and Opteron
4199                 */
4200                create = cpi->cpi_family > 6 ||
4201                    (cpi->cpi_family == 6 && cpi->cpi_model >= 8);
4202                break;
4203        case X86_VENDOR_AMD:
4204                create = cpi->cpi_family >= 0xf;
4205                break;
4206        default:
4207                create = 0;
4208                break;
4209        }
4210        if (create && cpi->cpi_brandid != 0) {
4211                (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4212                    "brand-id", cpi->cpi_brandid);
4213        }

4215        /* chunks, and apic-id */
4216        switch (cpi->cpi_vendor) {
```



```
4217                /*
4218                 * first available on Pentium IV and Opteron (K8)
4219                 */
4220        case X86_VENDOR_Intel:
4221                create = IS_NEW_F6(cpi) || cpi->cpi_family >= 0xf;
4222                break;
4223        case X86_VENDOR_AMD:
4224                create = cpi->cpi_family >= 0xf;
4225                break;
4226        default:
4227                create = 0;
4228                break;
4229        }
4230        if (create) {
4231                (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4232                    "chunks", CPI_CHUNKS(cpi));
4233                (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4234                    "apic-id", cpi->cpi_apicid);
4235                if (cpi->cpi_chipid >= 0) {
4236                        (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4237                            "chip#", cpi->cpi_chipid);
4238                        (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4239                            "clog#", cpi->cpi_clogid);
4240                }
4241        }

4243        /* cpuid-features */
4244        (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4245            "cpuid-features", CPI_FEATURES_EDX(cpi));


4248        /* cpuid-features-ecx */
4249        switch (cpi->cpi_vendor) {
4250        case X86_VENDOR_Intel:
4251                create = IS_NEW_F6(cpi) || cpi->cpi_family >= 0xf;
4252                break;
4253        case X86_VENDOR_AMD:
4254                create = cpi->cpi_family >= 0xf;
4255                break;
4256        default:
4257                create = 0;
4258                break;
4259        }
4260        if (create)
4261                (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4262                    "cpuid-features-ecx", CPI_FEATURES_ECX(cpi));

4264        /* ext-cpuid-features */
4265        switch (cpi->cpi_vendor) {
4266        case X86_VENDOR_Intel:
4267        case X86_VENDOR_AMD:
4268        case X86_VENDOR_Cyrix:
4269        case X86_VENDOR_TM:
4270        case X86_VENDOR_Centaur:
4271                create = cpi->cpi_xmaxeax >= 0x80000001;
4272                break;
4273        default:
4274                create = 0;
4275                break;
4276        }
4277        if (create) {
4278                (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4279                    "ext-cpuid-features", CPI_FEATURES_XTD_EDX(cpi));
4280                (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4281                    "ext-cpuid-features-ecx", CPI_FEATURES_XTD_ECX(cpi));
4282        }
```

```
4284         /*
4285          * Brand String first appeared in Intel Pentium IV, AMD K5
4286          * model 1, and Cyrix GXm.  On earlier models we try and
4287          * simulate something similar .. so this string should always
4288          * same -something- about the processor, however lame.
4289          */
4290         (void) ndi_prop_update_string(DDI_DEV_T_NONE, cpu_devi,
4291             "brand-string", cpi->cpi_brandstr);

4293         /*
4294          * Finally, cache and tlb information
4295          */
4296         switch (x86_which_cacheinfo(cpi)) {
4297         case X86_VENDOR_Intel:
4298                 intel_walk_cacheinfo(cpi, cpu_devi, add_cacheent_props);
4299                 break;
4300         case X86_VENDOR_Cyrix:
4301                 cyrix_walk_cacheinfo(cpi, cpu_devi, add_cacheent_props);
4302                 break;
4303         case X86_VENDOR_AMD:
4304                 amd_cache_info(cpi, cpu_devi);
4305                 break;
4306         default:
4307                 break;
4308         }
4309 }

4311 struct l2info {
4312         int *l2i_csz;
4313         int *l2i_lsz;
4314         int *l2i_assoc;
4315         int l2i_ret;
4316 };

4318 /*
4319  * A cacheinfo walker that fetches the size, line-size and associativity
4320  * of the L2 cache
4321  */
4322 static int
4323 intel_l2cinfo(void *arg, const struct cachetab *ct)
4324 {
4325         struct l2info *l2i = arg;
4326         int *ip;

4328         if (ct->ct_label != l2_cache_str &&
4329             ct->ct_label != sl2_cache_str)
4330                 return (0);     /* not an L2 -- keep walking */

4332         if ((ip = l2i->l2i_csz) != NULL)
4333                 *ip = ct->ct_size;
4334         if ((ip = l2i->l2i_lsz) != NULL)
4335                 *ip = ct->ct_line_size;
4336         if ((ip = l2i->l2i_assoc) != NULL)
4337                 *ip = ct->ct_assoc;
4338         l2i->l2i_ret = ct->ct_size;
4339         return (1);             /* was an L2 -- terminate walk */
4340 }

4342 /*
4343  * AMD L2/L3 Cache and TLB Associativity Field Definition:
4344  *
4345  *      Unlike the associativity for the L1 cache and tlb where the 8 bit
4346  *      value is the associativity, the associativity for the L2 cache and
4347  *      tlb is encoded in the following table. The 4 bit L2 value serves as
4348  *      an index into the amd_afd[] array to determine the associativity.
```



```
4349  *      -1 is undefined. 0 is fully associative.
4350  */

4352 static int amd_afd[] =
4353         {-1, 1, 2, -1, 4, -1, 8, -1, 16, -1, 32, 48, 64, 96, 128, 0};

4355 static void
4356 amd_l2cacheinfo(struct cpuid_info *cpi, struct l2info *l2i)
4357 {
4358         struct cpuid_regs *cp;
4359         uint_t size, assoc;
4360         int i;
4361         int *ip;

4363         if (cpi->cpi_xmaxeax < 0x80000006)
4364                 return;
4365         cp = &cpi->cpi_extd[6];

4367         if ((i = BITX(cp->cp_ecx, 15, 12)) != 0 &&
4368             (size = BITX(cp->cp_ecx, 31, 16)) != 0) {
4369                 uint_t cachesz = size * 1024;
4370                 assoc = amd_afd[i];

4372                 ASSERT(assoc != -1);

4374                 if ((ip = l2i->l2i_csz) != NULL)
4375                         *ip = cachesz;
4376                 if ((ip = l2i->l2i_lsz) != NULL)
4377                         *ip = BITX(cp->cp_ecx, 7, 0);
4378                 if ((ip = l2i->l2i_assoc) != NULL)
4379                         *ip = assoc;
4380                 l2i->l2i_ret = cachesz;
4381         }
4382 }

4384 int
4385 getl2cacheinfo(cpu_t *cpu, int *csz, int *lsz, int *assoc)
4386 {
4387         struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;
4388         struct l2info __l2info, *l2i = &__l2info;

4390         l2i->l2i_csz = csz;
4391         l2i->l2i_lsz = lsz;
4392         l2i->l2i_assoc = assoc;
4393         l2i->l2i_ret = -1;

4395         switch (x86_which_cacheinfo(cpi)) {
4396         case X86_VENDOR_Intel:
4397                 intel_walk_cacheinfo(cpi, l2i, intel_l2cinfo);
4398                 break;
4399         case X86_VENDOR_Cyrix:
4400                 cyrix_walk_cacheinfo(cpi, l2i, intel_l2cinfo);
4401                 break;
4402         case X86_VENDOR_AMD:
4403                 amd_l2cacheinfo(cpi, l2i);
4404                 break;
4405         default:
4406                 break;
4407         }
4408         return (l2i->l2i_ret);
4409 }

4411 #if !defined(__xpv)

4413 uint32_t *
4414 cpuid_mwait_alloc(cpu_t *cpu)
```

```
{
        uint32_t        *ret;
        size_t          mwait_size;

        ASSERT(cpuid_checkpass(CPU, 2));

        mwait_size = CPU->cpu_m.mcpu_cpi->cpi_mwait.mon_max;
        if (mwait_size == 0)
                return (NULL);

        /*
         * kmem_alloc() returns cache line size aligned data for mwait_size
         * allocations.  mwait_size is currently cache line sized.  Neither
         * of these implementation details are guarantied to be true in the
         * future.
         *
         * First try allocating mwait_size as kmem_alloc() currently returns
         * correctly aligned memory.  If kmem_alloc() does not return
         * mwait_size aligned memory, then use mwait_size ROUNDUP.
         *
         * Set cpi_mwait.buf_actual and cpi_mwait.size_actual in case we
         * decide to free this memory.
         */
        ret = kmem_zalloc(mwait_size, KM_SLEEP);
        if (ret == (uint32_t *)P2ROUNDUP((uintptr_t)ret, mwait_size)) {
                cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual = ret;
                cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual = mwait_size;
                *ret = MWAIT_RUNNING;
                return (ret);
        } else {
                kmem_free(ret, mwait_size);
                ret = kmem_zalloc(mwait_size * 2, KM_SLEEP);
                cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual = ret;
                cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual = mwait_size * 2;
                ret = (uint32_t *)P2ROUNDUP((uintptr_t)ret, mwait_size);
                *ret = MWAIT_RUNNING;
                return (ret);
        }
}

void
cpuid_mwait_free(cpu_t *cpu)
{
        if (cpu->cpu_m.mcpu_cpi == NULL) {
                return;
        }

        if (cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual != NULL &&
            cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual > 0) {
                kmem_free(cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual,
                    cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual);
        }

        cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual = NULL;
        cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual = 0;
}

void
patch_tsc_read(int flag)
{
        size_t cnt;

        switch (flag) {
        case X86_NO_TSC:
                cnt = &_no_rdtsc_end - &_no_rdtsc_start;
                (void) memcpy((void *)tsc_read, (void *)&_no_rdtsc_start, cnt);
```



```
                break;
        case X86_HAVE_TSCP:
                cnt = &_tscp_end - &_tscp_start;
                (void) memcpy((void *)tsc_read, (void *)&_tscp_start, cnt);
                break;
        case X86_TSC_MFENCE:
                cnt = &_tsc_mfence_end - &_tsc_mfence_start;
                (void) memcpy((void *)tsc_read,
                    (void *)&_tsc_mfence_start, cnt);
                break;
        case X86_TSC_LFENCE:
                cnt = &_tsc_lfence_end - &_tsc_lfence_start;
                (void) memcpy((void *)tsc_read,
                    (void *)&_tsc_lfence_start, cnt);
                break;
        default:
                break;
        }
}

int
cpuid_deep_cstates_supported(void)
{
        struct cpuid_info *cpi;
        struct cpuid_regs regs;

        ASSERT(cpuid_checkpass(CPU, 1));

        cpi = CPU->cpu_m.mcpu_cpi;

        if (!is_x86_feature(x86_featureset, X86FSET_CPUID))
                return (0);

        switch (cpi->cpi_vendor) {
        case X86_VENDOR_Intel:
                if (cpi->cpi_xmaxeax < 0x80000007)
                        return (0);

                /*
                 * TSC run at a constant rate in all ACPI C-states?
                 */
                regs.cp_eax = 0x80000007;
                (void) __cpuid_insn(&regs);
                return (regs.cp_edx & CPUID_TSC_CSTATE_INVARIANCE);

        default:
                return (0);
        }
}

#endif  /* !__xpv */

void
post_startup_cpu_fixups(void)
{
#ifndef __xpv
        /*
         * Some AMD processors support C1E state. Entering this state will
         * cause the local APIC timer to stop, which we can't deal with at
         * this time.
         */
        if (cpuid_getvendor(CPU) == X86_VENDOR_AMD) {
                on_trap_data_t otd;
                uint64_t reg;

                if (!on_trap(&otd, OT_DATA_ACCESS)) {
```

```
4547                        reg = rdmsr(MSR_AMD_INT_PENDING_CMP_HALT);
4548                        /* Disable C1E state if it is enabled by BIOS */
4549                        if ((reg >> AMD_ACTONCMPHALT_SHIFT) &
4550                            AMD_ACTONCMPHALT_MASK) {
4551                                reg &= ~(AMD_ACTONCMPHALT_MASK <<
4552                                    AMD_ACTONCMPHALT_SHIFT);
4553                                wrmsr(MSR_AMD_INT_PENDING_CMP_HALT, reg);
4554                        }
4555                }
4556                no_trap();
4557        }
4558 #endif  /* !__xpv */
4559 }

4561 /*
4562  * Setup necessary registers to enable XSAVE feature on this processor.
4563  * This function needs to be called early enough, so that no xsave/xrstor
4564  * ops will execute on the processor before the MSRs are properly set up.
4565  *
4566  * Current implementation has the following assumption:
4567  * - cpuid_pass1() is done, so that X86 features are known.
4568  * - fpu_probe() is done, so that fp_save_mech is chosen.
4569  */
4570 void
4571 xsave_setup_msr(cpu_t *cpu)
4572 {
4573        ASSERT(fp_save_mech == FP_XSAVE);
4574        ASSERT(is_x86_feature(x86_featureset, X86FSET_XSAVE));

4576        /* Enable OSXSAVE in CR4. */
4577        setcr4(getcr4() | CR4_OSXSAVE);
4578        /*
4579         * Update SW copy of ECX, so that /dev/cpu/self/cpuid will report
4580         * correct value.
4581         */
4582        cpu->cpu_m.mcpu_cpi->cpi_std[1].cp_ecx |= CPUID_INTC_ECX_OSXSAVE;
4583        setup_xfem();
4584 }

4586 /*
4587  * Starting with the Westmere processor the local
4588  * APIC timer will continue running in all C-states,
4589  * including the deepest C-states.
4590  */
4591 int
4592 cpuid_arat_supported(void)
4593 {
4594        struct cpuid_info *cpi;
4595        struct cpuid_regs regs;

4597        ASSERT(cpuid_checkpass(CPU, 1));
4598        ASSERT(is_x86_feature(x86_featureset, X86FSET_CPUID));

4600        cpi = CPU->cpu_m.mcpu_cpi;

4602        switch (cpi->cpi_vendor) {
4603        case X86_VENDOR_Intel:
4604                /*
4605                 * Always-running Local APIC Timer is
4606                 * indicated by CPUID.6.EAX[2].
4607                 */
4608                if (cpi->cpi_maxeax >= 6) {
4609                        regs.cp_eax = 6;
4610                        (void) cpuid_insn(NULL, &regs);
4611                        return (regs.cp_eax & CPUID_CSTATE_ARAT);
4612                } else {
```



```
4613                        return (0);
4614                }
4615        default:
4616                return (0);
4617        }
4618 }

4620 /*
4621  * Check support for Intel ENERGY_PERF_BIAS feature
4622  */
4623 int
4624 cpuid_iepb_supported(struct cpu *cp)
4625 {
4626        struct cpuid_info *cpi = cp->cpu_m.mcpu_cpi;
4627        struct cpuid_regs regs;

4629        ASSERT(cpuid_checkpass(cp, 1));

4631        if (!(is_x86_feature(x86_featureset, X86FSET_CPUID)) ||
4632            !(is_x86_feature(x86_featureset, X86FSET_MSR))) {
4633                return (0);
4634        }

4636        /*
4637         * Intel ENERGY_PERF_BIAS MSR is indicated by
4638         * capability bit CPUID.6.ECX.3
4639         */
4640        if ((cpi->cpi_vendor != X86_VENDOR_Intel) || (cpi->cpi_maxeax < 6))
4641                return (0);

4643        regs.cp_eax = 0x6;
4644        (void) cpuid_insn(NULL, &regs);
4645        return (regs.cp_ecx & CPUID_EPB_SUPPORT);
4646 }

4648 /*
4649  * Check support for TSC deadline timer
4650  *
4651  * TSC deadline timer provides a superior software programming
4652  * model over local APIC timer that eliminates "time drifts".
4653  * Instead of specifying a relative time, software specifies an
4654  * absolute time as the target at which the processor should
4655  * generate a timer event.
4656  */
4657 int
4658 cpuid_deadline_tsc_supported(void)
4659 {
4660        struct cpuid_info *cpi = CPU->cpu_m.mcpu_cpi;
4661        struct cpuid_regs regs;

4663        ASSERT(cpuid_checkpass(CPU, 1));
4664        ASSERT(is_x86_feature(x86_featureset, X86FSET_CPUID));

4666        switch (cpi->cpi_vendor) {
4667        case X86_VENDOR_Intel:
4668                if (cpi->cpi_maxeax >= 1) {
4669                        regs.cp_eax = 1;
4670                        (void) cpuid_insn(NULL, &regs);
4671                        return (regs.cp_ecx & CPUID_DEADLINE_TSC);
4672                } else {
4673                        return (0);
4674                }
4675        default:
4676                return (0);
4677        }
4678 }
```

```
#if defined(__amd64) && !defined(__xpv)
/*
 * Patch in versions of bcopy for high performance Intel Nhm processors
 * and later...
 */
void
patch_memops(uint_t vendor)
{
        size_t cnt, i;
        caddr_t to, from;

        if ((vendor == X86_VENDOR_Intel) &&
            is_x86_feature(x86_featureset, X86FSET_SSE4_2)) {
                cnt = &bcopy_patch_end - &bcopy_patch_start;
                to = &bcopy_ck_size;
                from = &bcopy_patch_start;
                for (i = 0; i < cnt; i++) {
                        *to++ = *from++;
                }
        }
}
#endif  /* __amd64 && !__xpv */

/*
 * This function finds the number of bits to represent the number of cores per
 * chip and the number of strands per core for the Intel platforms.
 * It re-uses the x2APIC cpuid code of the cpuid_pass2().
 */
void
cpuid_get_ext_topo(uint_t vendor, uint_t *core_nbits, uint_t *strand_nbits)
{
        struct cpuid_regs regs;
        struct cpuid_regs *cp = &regs;

        if (vendor != X86_VENDOR_Intel) {
                return;
        }

        /* if the cpuid level is 0xB, extended topo is available. */
        cp->cp_eax = 0;
        if (__cpuid_insn(cp) >= 0xB) {

                cp->cp_eax = 0xB;
                cp->cp_edx = cp->cp_ebx = cp->cp_ecx = 0;
                (void) __cpuid_insn(cp);

                /*
                 * Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero, which
                 * indicates that the extended topology enumeration leaf is
                 * available.
                 */
                if (cp->cp_ebx) {
                        uint_t coreid_shift = 0;
                        uint_t chipid_shift = 0;
                        uint_t i;
                        uint_t level;

                        for (i = 0; i < CPI_FNB_ECX_MAX; i++) {
                                cp->cp_eax = 0xB;
                                cp->cp_ecx = i;

                                (void) __cpuid_insn(cp);
                                level = CPI_CPU_LEVEL_TYPE(cp);

                                if (level == 1) {
```



```
                                        /*
                                         * Thread level processor topology
                                         * Number of bits shift right APIC ID
                                         * to get the coreid.
                                         */
                                        coreid_shift = BITX(cp->cp_eax, 4, 0);
                                } else if (level == 2) {
                                        /*
                                         * Core level processor topology
                                         * Number of bits shift right APIC ID
                                         * to get the chipid.
                                         */
                                        chipid_shift = BITX(cp->cp_eax, 4, 0);
                                }
                        }

                        if (coreid_shift > 0 && chipid_shift > coreid_shift) {
                                *strand_nbits = coreid_shift;
                                *core_nbits = chipid_shift - coreid_shift;
                        }
                }
        }
}
```

```
**********************************************************
   34387 Fri Jan  3 22:11:56 2014
new/usr/src/uts/i86pc/os/machdep.c
patch cpu-pause-func-deglobalize
**********************************************************
```

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

/*
 * Copyright (c) 1992, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright (c) 2010, Intel Corporation.
 * All rights reserved.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/segments.h>
#include <sys/sysmacros.h>
#include <sys/signal.h>
#include <sys/systm.h>
#include <sys/user.h>
#include <sys/mman.h>
#include <sys/vm.h>

#include <sys/disp.h>
#include <sys/class.h>

#include <sys/proc.h>
#include <sys/buf.h>
#include <sys/kmem.h>

#include <sys/reboot.h>
#include <sys/uadmin.h>
#include <sys/callb.h>

#include <sys/cred.h>
#include <sys/vnode.h>
#include <sys/file.h>

#include <sys/procfs.h>
#include <sys/acct.h>

#include <sys/vfs.h>
#include <sys/dnlc.h>
#include <sys/var.h>
```



```
#include <sys/cmn_err.h>
#include <sys/utsname.h>
#include <sys/debug.h>

#include <sys/dumphdr.h>
#include <sys/bootconf.h>
#include <sys/varargs.h>
#include <sys/promif.h>
#include <sys/modctl.h>

#include <sys/consdev.h>
#include <sys/frame.h>

#include <sys/sunddi.h>
#include <sys/ddidmareq.h>
#include <sys/psw.h>
#include <sys/regset.h>
#include <sys/privregs.h>
#include <sys/clock.h>
#include <sys/tss.h>
#include <sys/cpu.h>
#include <sys/stack.h>
#include <sys/trap.h>
#include <sys/pic.h>
#include <vm/hat.h>
#include <vm/anon.h>
#include <vm/as.h>
#include <vm/page.h>
#include <vm/seg.h>
#include <vm/seg_kmem.h>
#include <vm/seg_map.h>
#include <vm/seg_vn.h>
#include <vm/seg_kp.h>
#include <vm/hat_i86.h>
#include <sys/swap.h>
#include <sys/thread.h>
#include <sys/sysconf.h>
#include <sys/vm_machparam.h>
#include <sys/archsystm.h>
#include <sys/machsystm.h>
#include <sys/machlock.h>
#include <sys/x_call.h>
#include <sys/instance.h>

#include <sys/time.h>
#include <sys/smp_impldefs.h>
#include <sys/psm_types.h>
#include <sys/atomic.h>
#include <sys/panic.h>
#include <sys/cpuvar.h>
#include <sys/dtrace.h>
#include <sys/bl.h>
#include <sys/nvpair.h>
#include <sys/x86_archext.h>
#include <sys/pool_pset.h>
#include <sys/autoconf.h>
#include <sys/mem.h>
#include <sys/dumphdr.h>
#include <sys/compress.h>
#include <sys/cpu_module.h>
#if defined(__xpv)
#include <sys/hypervisor.h>
#include <sys/xpv_panic.h>
#endif

#include <sys/fastboot.h>
```

```
#include <sys/machelf.h>
#include <sys/kobj.h>
#include <sys/multiboot.h>

#ifdef  TRAPTRACE
#include <sys/traptrace.h>
#endif  /* TRAPTRACE */

#include <c2/audit.h>
#include <sys/clock_impl.h>

extern void audit_enterprom(int);
extern void audit_exitprom(int);

/*
 * Tunable to enable apix PSM; if set to 0, pcplusmp PSM will be used.
 */
int     apix_enable = 1;

int     apic_nvidia_io_max = 0; /* no. of NVIDIA i/o apics */

/*
 * Occassionally the kernel knows better whether to power-off or reboot.
 */
int force_shutdown_method = AD_UNKNOWN;

/*
 * The panicbuf array is used to record messages and state:
 */
char panicbuf[PANICBUFSIZE];

/*
 * Flags to control Dynamic Reconfiguration features.
 */
uint64_t plat_dr_options;

/*
 * Maximum physical address for memory DR operations.
 */
uint64_t plat_dr_physmax;

/*
 * maxphys - used during physio
 * klustsize - used for klustering by swapfs and specfs
 */
int maxphys = 56 * 1024;    /* XXX See vm_subr.c - max b_count in physio */
int klustsize = 56 * 1024;

caddr_t p0_va;          /* Virtual address for accessing physical page 0 */

/*
 * defined here, though unused on x86,
 * to make kstat_fr.c happy.
 */
int vac;

void debug_enter(char *);

extern void pm_cfb_check_and_powerup(void);
extern void pm_cfb_rele(void);

extern fastboot_info_t newkernel;

/*
 * Machine dependent code to reboot.
 * "mdep" is interpreted as a character pointer; if non-null, it is a pointer
```



```
 * to a string to be used as the argument string when rebooting.
 *
 * "invoke_cb" is a boolean. It is set to true when mdboot() can safely
 * invoke CB_CL_MDBOOT callbacks before shutting the system down, i.e. when
 * we are in a normal shutdown sequence (interrupts are not blocked, the
 * system is not panic'ing or being suspended).
 */
/*ARGSUSED*/
void
mdboot(int cmd, int fcn, char *mdep, boolean_t invoke_cb)
{
        processorid_t bootcpuid = 0;
        static int is_first_quiesce = 1;
        static int is_first_reset = 1;
        int reset_status = 0;
        static char fallback_str[] = "Falling back to regular reboot.\n";

        if (fcn == AD_FASTREBOOT && !newkernel.fi_valid)
                fcn = AD_BOOT;

        if (!panicstr) {
                kpreempt_disable();
                if (fcn == AD_FASTREBOOT) {
                        mutex_enter(&cpu_lock);
                        if (CPU_ACTIVE(cpu_get(bootcpuid))) {
                                affinity_set(bootcpuid);
                        }
                        mutex_exit(&cpu_lock);
                } else {
                        affinity_set(CPU_CURRENT);
                }
        }

        if (force_shutdown_method != AD_UNKNOWN)
                fcn = force_shutdown_method;

        /*
         * XXX - rconsvp is set to NULL to ensure that output messages
         * are sent to the underlying "hardware" device using the
         * monitor's printf routine since we are in the process of
         * either rebooting or halting the machine.
         */
        rconsvp = NULL;

        /*
         * Print the reboot message now, before pausing other cpus.
         * There is a race condition in the printing support that
         * can deadlock multiprocessor machines.
         */
        if (!(fcn == AD_HALT || fcn == AD_POWEROFF))
                prom_printf("rebooting...\n");

        if (IN_XPV_PANIC())
                reset();

        /*
         * We can't bring up the console from above lock level, so do it now
         */
        pm_cfb_check_and_powerup();

        /* make sure there are no more changes to the device tree */
        devtree_freeze();

        if (invoke_cb)
                (void) callb_execute_class(CB_CL_MDBOOT, NULL);
```

```
260         /*
261          * Clear any unresolved UEs from memory.
262          */
263         page_retire_mdboot();

265 #if defined(__xpv)
266         /*
267          * XXPV Should probably think some more about how we deal
268          *      with panicing before it's really safe to panic.
269          *      On hypervisors, we reboot very quickly..  Perhaps panic
270          *      should only attempt to recover by rebooting if,
271          *      say, we were able to mount the root filesystem,
272          *      or if we successfully launched init(1m).
273          */
274         if (panicstr && proc_init == NULL)
275                 (void) HYPERVISOR_shutdown(SHUTDOWN_poweroff);
276 #endif
277         /*
278          * stop other cpus and raise our priority.  since there is only
279          * one active cpu after this, and our priority will be too high
280          * for us to be preempted, we're essentially single threaded
281          * from here on out.
282          */
283         (void) spl6();
284         if (!panicstr) {
285                 mutex_enter(&cpu_lock);
286                 pause_cpus(NULL, NULL);
286                 pause_cpus(NULL);
287                 mutex_exit(&cpu_lock);
288         }

290         /*
291          * If the system is panicking, the preloaded kernel is valid, and
292          * fastreboot_onpanic has been set, and the system has been up for
293          * longer than fastreboot_onpanic_uptime (default to 10 minutes),
294          * choose Fast Reboot.
295          */
296         if (fcn == AD_BOOT && panicstr && newkernel.fi_valid &&
297             fastreboot_onpanic &&
298             (panic_lbolt - lbolt_at_boot) > fastreboot_onpanic_uptime) {
299                 fcn = AD_FASTREBOOT;
300         }

302         /*
303          * Try to quiesce devices.
304          */
305         if (is_first_quiesce) {
306                 /*
307                  * Clear is_first_quiesce before calling quiesce_devices()
308                  * so that if quiesce_devices() causes panics, it will not
309                  * be invoked again.
310                  */
311                 is_first_quiesce = 0;

313                 quiesce_active = 1;
314                 quiesce_devices(ddi_root_node(), &reset_status);
315                 if (reset_status == -1) {
316                         if (fcn == AD_FASTREBOOT && !force_fastreboot) {
317                                 prom_printf("Driver(s) not capable of fast "
318                                     "reboot.\n");
319                                 prom_printf(fallback_str);
320                                 fastreboot_capable = 0;
321                                 fcn = AD_BOOT;
322                         } else if (fcn != AD_FASTREBOOT)
323                                 fastreboot_capable = 0;
324                 }
```



```
325                 quiesce_active = 0;
326         }

328         /*
329          * Try to reset devices. reset_leaves() should only be called
330          * a) when there are no other threads that could be accessing devices,
331          *    and
332          * b) on a system that's not capable of fast reboot (fastreboot_capable
333          *    being 0), or on a system where quiesce_devices() failed to
334          *    complete (quiesce_active being 1).
335          */
336         if (is_first_reset && (!fastreboot_capable || quiesce_active)) {
337                 /*
338                  * Clear is_first_reset before calling reset_devices()
339                  * so that if reset_devices() causes panics, it will not
340                  * be invoked again.
341                  */
342                 is_first_reset = 0;
343                 reset_leaves();
344         }

346         /* Verify newkernel checksum */
347         if (fastreboot_capable && fcn == AD_FASTREBOOT &&
348             fastboot_cksum_verify(&newkernel) != 0) {
349                 fastreboot_capable = 0;
350                 prom_printf("Fast reboot: checksum failed for the new "
351                     "kernel.\n");
352                 prom_printf(fallback_str);
353         }

355         (void) spl8();

357         if (fastreboot_capable && fcn == AD_FASTREBOOT) {
358                 /*
359                  * psm_shutdown is called within fast_reboot()
360                  */
361                 fast_reboot();
362         } else {
363                 (*psm_shutdownf)(cmd, fcn);

365                 if (fcn == AD_HALT || fcn == AD_POWEROFF)
366                         halt((char *)NULL);
367                 else
368                         prom_reboot("");
369         }
370         /*NOTREACHED*/
371 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   17010 Fri Jan  3 22:11:56 2014
new/usr/src/uts/i86pc/os/mp_pc.c
patch cpu-pause-func-deglobalize
**********************************************************
```

```
   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 (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
  23  */
  24 /*
  25  * Copyright (c) 2010, Intel Corporation.
  26  * All rights reserved.
  27  */
  28 /*
  29  * Copyright 2011 Joyent, Inc. All rights reserved.
  30  */

  32 /*
  33  * Welcome to the world of the "real mode platter".
  34  * See also startup.c, mpcore.s and apic.c for related routines.
  35  */

  37 #include <sys/types.h>
  38 #include <sys/systm.h>
  39 #include <sys/cpuvar.h>
  40 #include <sys/cpu_module.h>
  41 #include <sys/kmem.h>
  42 #include <sys/archsystm.h>
  43 #include <sys/machsystm.h>
  44 #include <sys/controlregs.h>
  45 #include <sys/x86_archext.h>
  46 #include <sys/smp_impldefs.h>
  47 #include <sys/sysmacros.h>
  48 #include <sys/mach_mmu.h>
  49 #include <sys/promif.h>
  50 #include <sys/cpu.h>
  51 #include <sys/cpu_event.h>
  52 #include <sys/sunndi.h>
  53 #include <sys/fs/dv_node.h>
  54 #include <vm/hat_i86.h>
  55 #include <vm/as.h>

  57 extern cpuset_t cpu_ready_set;

  59 extern int  mp_start_cpu_common(cpu_t *cp, boolean_t boot);
  60 extern void real_mode_start_cpu(void);
  61 extern void real_mode_start_cpu_end(void);
```



```
  62 extern void real_mode_stop_cpu_stage1(void);
  63 extern void real_mode_stop_cpu_stage1_end(void);
  64 extern void real_mode_stop_cpu_stage2(void);
  65 extern void real_mode_stop_cpu_stage2_end(void);
  66 extern void *(*cpu_pause_func)(void *);

  67 void rmp_gdt_init(rm_platter_t *);

  69 /*
  70  * Fill up the real mode platter to make it easy for real mode code to
  71  * kick it off. This area should really be one passed by boot to kernel
  72  * and guaranteed to be below 1MB and aligned to 16 bytes. Should also
  73  * have identical physical and virtual address in paged mode.
  74  */
  75 static ushort_t *warm_reset_vector = NULL;

  77 int
  78 mach_cpucontext_init(void)
  79 {
  80         ushort_t *vec;
  81         ulong_t addr;
  82         struct rm_platter *rm = (struct rm_platter *)rm_platter_va;

  84         if (!(vec = (ushort_t *)psm_map_phys(WARM_RESET_VECTOR,
  85             sizeof (vec), PROT_READ | PROT_WRITE)))
  86                 return (-1);

  88         /*
  89          * setup secondary cpu bios boot up vector
  90          * Write page offset to 0x467 and page frame number to 0x469.
  91          */
  92         addr = (ulong_t)((caddr_t)rm->rm_code - (caddr_t)rm) + rm_platter_pa;
  93         vec[0] = (ushort_t)(addr & PAGEOFFSET);
  94         vec[1] = (ushort_t)((addr & (0xffffff & PAGEMASK)) >> 4);
  95         warm_reset_vector = vec;

  97         /* Map real mode platter into kas so kernel can access it. */
  98         hat_devload(kas.a_hat,
  99             (caddr_t)(uintptr_t)rm_platter_pa, MMU_PAGESIZE,
 100             btop(rm_platter_pa), PROT_READ | PROT_WRITE | PROT_EXEC,
 101             HAT_LOAD_NOCONSIST);

 103         /* Copy CPU startup code to rm_platter if it's still during boot. */
 104         if (!plat_dr_enabled()) {
 105                 ASSERT((size_t)real_mode_start_cpu_end -
 106                     (size_t)real_mode_start_cpu <= RM_PLATTER_CODE_SIZE);
 107                 bcopy((caddr_t)real_mode_start_cpu, (caddr_t)rm->rm_code,
 108                     (size_t)real_mode_start_cpu_end -
 109                     (size_t)real_mode_start_cpu);
 110         }

 112         return (0);
 113 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   18803 Fri Jan  3 22:11:56 2014
new/usr/src/uts/i86pc/os/x_call.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 266 #define XC_FLUSH_MAX_WAITS              1000

 268 /* Flush inflight message buffers. */
 269 int
 270 xc_flush_cpu(struct cpu *cpup)
 271 {
 272         int i;

 274         ASSERT((cpup->cpu_flags & CPU_READY) == 0);

 276         /*
 277          * Pause all working CPUs, which ensures that there's no CPU in
 278          * function xc_common().
 279          * This is used to work around a race condition window in xc_common()
 280          * between checking CPU_READY flag and increasing working item count.
 281          */
 282         pause_cpus(cpup, NULL);
 282         pause_cpus(cpup);
 283         start_cpus();

 285         for (i = 0; i < XC_FLUSH_MAX_WAITS; i++) {
 286                 if (cpup->cpu_m.xc_work_cnt == 0) {
 287                         break;
 288                 }
 289                 DELAY(1);
 290         }
 291         for (; i < XC_FLUSH_MAX_WAITS; i++) {
 292                 if (!BT_TEST(xc_priority_set, cpup->cpu_id)) {
 293                         break;
 294                 }
 295                 DELAY(1);
 296         }

 298         return (i >= XC_FLUSH_MAX_WAITS ? ETIME : 0);
 299 }
______unchanged_portion_omitted_
```

****
25091 Fri Jan 3 22:11:56 2014
new/usr/src/uts/i86xpv/os/mp_xen.c
patch cpu-pause-func-deglobalize
****
_____***unchanged_portion_omitted_***

```
 579 void
 580 mp_enter_barrier(void)
 581 {
 582         hrtime_t last_poke_time = 0;
 583         int poke_allowed = 0;
 584         int done = 0;
 585         int i;

 587         ASSERT(MUTEX_HELD(&cpu_lock));

 589         pause_cpus(NULL, NULL);
 589         pause_cpus(NULL);

 591         while (!done) {
 592                 done = 1;
 593                 poke_allowed = 0;

 595                 if (xpv_gethrtime() - last_poke_time > POKE_TIMEOUT) {
 596                         last_poke_time = xpv_gethrtime();
 597                         poke_allowed = 1;
 598                 }

 600                 for (i = 0; i < NCPU; i++) {
 601                         cpu_t *cp = cpu_get(i);

 603                         if (cp == NULL || cp == CPU)
 604                                 continue;

 606                         switch (cpu_phase[i]) {
 607                         case CPU_PHASE_NONE:
 608                                 cpu_phase[i] = CPU_PHASE_WAIT_SAFE;
 609                                 poke_cpu(i);
 610                                 done = 0;
 611                                 break;

 613                         case CPU_PHASE_WAIT_SAFE:
 614                                 if (poke_allowed)
 615                                         poke_cpu(i);
 616                                 done = 0;
 617                                 break;

 619                         case CPU_PHASE_SAFE:
 620                         case CPU_PHASE_POWERED_OFF:
 621                                 break;
 622                         }
 623                 }

 625                 SMT_PAUSE();
 626         }
 627 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   28993 Fri Jan  3 22:11:56 2014
new/usr/src/uts/intel/sys/x86_archext.h
patch x2apic-x86fset
patch remove-unused-vars
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1995, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011 by Delphix. All rights reserved.
 * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
 */
/*
 * Copyright (c) 2010, Intel Corporation.
 * All rights reserved.
 */
/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright 2012 Jens Elkner <jel+illumos@cs.uni-magdeburg.de>
 * Copyright 2012 Hans Rosenfeld <rosenfeld@grumpf.hope-2000.org>
 */

#ifndef _SYS_X86_ARCHEXT_H
#define _SYS_X86_ARCHEXT_H

#if !defined(_ASM)
#include <sys/regset.h>
#include <sys/processor.h>
#include <vm/seg_enum.h>
#include <vm/page.h>
#endif  /* _ASM */

#ifdef __cplusplus
extern "C" {
#endif

/*
 * cpuid instruction feature flags in %edx (standard function 1)
 */

#define CPUID_INTC_EDX_FPU      0x00000001      /* x87 fpu present */
#define CPUID_INTC_EDX_VME      0x00000002      /* virtual-8086 extension */
#define CPUID_INTC_EDX_DE       0x00000004      /* debugging extensions */
#define CPUID_INTC_EDX_PSE      0x00000008      /* page size extension */
#define CPUID_INTC_EDX_TSC      0x00000010      /* time stamp counter */
#define CPUID_INTC_EDX_MSR      0x00000020      /* rdmsr and wrmsr */
#define CPUID_INTC_EDX_PAE      0x00000040      /* physical addr extension */
```



```
#define CPUID_INTC_EDX_MCE      0x00000080      /* machine check exception */
#define CPUID_INTC_EDX_CX8      0x00000100      /* cmpxchg8b instruction */
#define CPUID_INTC_EDX_APIC     0x00000200      /* local APIC */
                                                /* 0x400 - reserved */
#define CPUID_INTC_EDX_SEP      0x00000800      /* sysenter and sysexit */
#define CPUID_INTC_EDX_MTRR     0x00001000      /* memory type range reg */
#define CPUID_INTC_EDX_PGE      0x00002000      /* page global enable */
#define CPUID_INTC_EDX_MCA      0x00004000      /* machine check arch */
#define CPUID_INTC_EDX_CMOV     0x00008000      /* conditional move insns */
#define CPUID_INTC_EDX_PAT      0x00010000      /* page attribute table */
#define CPUID_INTC_EDX_PSE36    0x00020000      /* 36-bit pagesize extension */
#define CPUID_INTC_EDX_PSN      0x00040000      /* processor serial number */
#define CPUID_INTC_EDX_CLFSH    0x00080000      /* clflush instruction */
                                                /* 0x100000 - reserved */
#define CPUID_INTC_EDX_DS       0x00200000      /* debug store exists */
#define CPUID_INTC_EDX_ACPI     0x00400000      /* monitoring + clock ctrl */
#define CPUID_INTC_EDX_MMX      0x00800000      /* MMX instructions */
#define CPUID_INTC_EDX_FXSR     0x01000000      /* fxsave and fxrstor */
#define CPUID_INTC_EDX_SSE      0x02000000      /* streaming SIMD extensions */
#define CPUID_INTC_EDX_SSE2     0x04000000      /* SSE extensions */
#define CPUID_INTC_EDX_SS       0x08000000      /* self-snoop */
#define CPUID_INTC_EDX_HTT      0x10000000      /* Hyper Thread Technology */
#define CPUID_INTC_EDX_TM       0x20000000      /* thermal monitoring */
#define CPUID_INTC_EDX_IA64     0x40000000      /* Itanium emulating IA32 */
#define CPUID_INTC_EDX_PBE      0x80000000      /* Pending Break Enable */

#define FMT_CPUID_INTC_EDX                                              \
        "\20"                                                           \
        "\40pbe\37ia64\36tm\35htt\34ss\33sse2\32sse\31fxsr"     \
        "\30mmx\27acpi\26ds\24clfsh\23psn\22pse36\21pat"        \
        "\20cmov\17mca\16pge\15mtrr\14sep\12apic\11cx8"         \
        "\10mce\7pae\6msr\5tsc\4pse\3de\2vme\1fpu"

/*
 * cpuid instruction feature flags in %ecx (standard function 1)
 */

#define CPUID_INTC_ECX_SSE3     0x00000001      /* Yet more SSE extensions */
#define CPUID_INTC_ECX_PCLMULQDQ 0x00000002     /* PCLMULQDQ insn */
                                                /* 0x00000004 - reserved */
#define CPUID_INTC_ECX_MON      0x00000008      /* MONITOR/MWAIT */
#define CPUID_INTC_ECX_DSCPL    0x00000010      /* CPL-qualified debug store */
#define CPUID_INTC_ECX_VMX      0x00000020      /* Hardware VM extensions */
#define CPUID_INTC_ECX_SMX      0x00000040      /* Secure mode extensions */
#define CPUID_INTC_ECX_EST      0x00000080      /* enhanced SpeedStep */
#define CPUID_INTC_ECX_TM2      0x00000100      /* thermal monitoring */
#define CPUID_INTC_ECX_SSSE3    0x00000200      /* Supplemental SSE3 insns */
#define CPUID_INTC_ECX_CID      0x00000400      /* L1 context ID */
                                                /* 0x00000800 - reserved */
                                                /* 0x00001000 - reserved */
#define CPUID_INTC_ECX_CX16     0x00002000      /* cmpxchg16 */
#define CPUID_INTC_ECX_ETPRD    0x00004000      /* extended task pri messages */
                                                /* 0x00008000 - reserved */
                                                /* 0x00010000 - reserved */
                                                /* 0x00020000 - reserved */
#define CPUID_INTC_ECX_DCA      0x00040000      /* direct cache access */
#define CPUID_INTC_ECX_SSE4_1   0x00080000      /* SSE4.1 insns */
#define CPUID_INTC_ECX_SSE4_2   0x00100000      /* SSE4.2 insns */
#define CPUID_INTC_ECX_X2APIC   0x00200000      /* x2APIC */
#endif /* ! codereview */
#define CPUID_INTC_ECX_MOVBE    0x00400000      /* MOVBE insn */
#define CPUID_INTC_ECX_POPCNT   0x00800000      /* POPCNT insn */
#define CPUID_INTC_ECX_AES      0x02000000      /* AES insns */
#define CPUID_INTC_ECX_XSAVE    0x04000000      /* XSAVE/XRESTOR insns */
#define CPUID_INTC_ECX_OSXSAVE  0x08000000      /* OS supports XSAVE insns */
#define CPUID_INTC_ECX_AVX      0x10000000      /* AVX supported */
```

```
#define CPUID_INTC_ECX_F16C     0x20000000      /* F16C supported */
#define CPUID_INTC_ECX_RDRAND   0x40000000      /* RDRAND supported */
#define CPUID_INTC_ECX_HV       0x80000000      /* Hypervisor */

#define FMT_CPUID_INTC_ECX                                          \
        "\20"                                                       \
        "\37rdrand\36f16c\35avx\34osxsav\33xsave"                   \
        "\32aes"                                                    \
        "\30popcnt\27movbe\26x2apic\25sse4.2\24sse4.1\23dca"        \
        "\30popcnt\27movbe\25sse4.2\24sse4.1\23dca"                 \
        "\20\17etprd\16cx16\13cid\12ssse3\11tm2"                    \
        "\10est\7smx\6vmx\5dscpl\4mon\2pclmulqdq\1sse3"

/*
 * cpuid instruction feature flags in %edx (extended function 0x80000001)
 */

#define CPUID_AMD_EDX_FPU       0x00000001      /* x87 fpu present */
#define CPUID_AMD_EDX_VME       0x00000002      /* virtual-8086 extension */
#define CPUID_AMD_EDX_DE        0x00000004      /* debugging extensions */
#define CPUID_AMD_EDX_PSE       0x00000008      /* page size extensions */
#define CPUID_AMD_EDX_TSC       0x00000010      /* time stamp counter */
#define CPUID_AMD_EDX_MSR       0x00000020      /* rdmsr and wrmsr */
#define CPUID_AMD_EDX_PAE       0x00000040      /* physical addr extension */
#define CPUID_AMD_EDX_MCE       0x00000080      /* machine check exception */
#define CPUID_AMD_EDX_CX8       0x00000100      /* cmpxchg8b instruction */
#define CPUID_AMD_EDX_APIC      0x00000200      /* local APIC */
                                                /* 0x00000400 - sysc on K6m6 */
#define CPUID_AMD_EDX_SYSC      0x00000800      /* AMD: syscall and sysret */
#define CPUID_AMD_EDX_MTRR      0x00001000      /* memory type and range reg */
#define CPUID_AMD_EDX_PGE       0x00002000      /* page global enable */
#define CPUID_AMD_EDX_MCA       0x00004000      /* machine check arch */
#define CPUID_AMD_EDX_CMOV      0x00008000      /* conditional move insns */
#define CPUID_AMD_EDX_PAT       0x00010000      /* K7: page attribute table */
#define CPUID_AMD_EDX_FCMOV     0x00010000      /* FCMOVcc etc. */
#define CPUID_AMD_EDX_PSE36     0x00020000      /* 36-bit pagesize extension */
                                /* 0x00040000 - reserved */
                                /* 0x00080000 - reserved */
#define CPUID_AMD_EDX_NX        0x00100000      /* AMD: no-execute page prot */
                                /* 0x00200000 - reserved */
#define CPUID_AMD_EDX_MMXamd    0x00400000      /* AMD: MMX extensions */
#define CPUID_AMD_EDX_MMX       0x00800000      /* MMX instructions */
#define CPUID_AMD_EDX_FXSR      0x01000000      /* fxsave and fxrstor */
#define CPUID_AMD_EDX_FFXSR     0x02000000      /* fast fxsave/fxrstor */
#define CPUID_AMD_EDX_1GPG      0x04000000      /* 1GB page */
#define CPUID_AMD_EDX_TSCP      0x08000000      /* rdtscp instruction */
                                /* 0x10000000 - reserved */
#define CPUID_AMD_EDX_LM        0x20000000      /* AMD: long mode */
#define CPUID_AMD_EDX_3DNowx    0x40000000      /* AMD: extensions to 3DNow! */
#define CPUID_AMD_EDX_3DNow     0x80000000      /* AMD: 3DNow! instructions */

#define FMT_CPUID_AMD_EDX                                           \
        "\20"                                                       \
        "\40a3d\37a3d+\36lm\34tscp\32ffxsr\31fxsr"                  \
        "\30mmx\27mmxext\25nx\22pse\21pat"                          \
        "\20cmov\17mca\16pge\15mtrr\14syscall\12apic\11cx8"         \
        "\10mce\7pae\6msr\5tsc\4pse\3de\2vme\1fpu"

#define CPUID_AMD_ECX_AHF64     0x00000001      /* LAHF and SAHF in long mode */
#define CPUID_AMD_ECX_CMP_LGCY  0x00000002      /* AMD: multicore chip */
#define CPUID_AMD_ECX_SVM       0x00000004      /* AMD: secure VM */
#define CPUID_AMD_ECX_EAS       0x00000008      /* extended apic space */
#define CPUID_AMD_ECX_CR8D      0x00000010      /* AMD: 32-bit mov %cr8 */
#define CPUID_AMD_ECX_LZCNT     0x00000020      /* AMD: LZCNT insn */
#define CPUID_AMD_ECX_SSE4A     0x00000040      /* AMD: SSE4A insns */
#define CPUID_AMD_ECX_MAS       0x00000080      /* AMD: MisAlignSse mnode */
```



```
#define CPUID_AMD_ECX_3DNP      0x00000100      /* AMD: 3DNowPrefectch */
#define CPUID_AMD_ECX_OSVW      0x00000200      /* AMD: OSVW */
#define CPUID_AMD_ECX_IBS       0x00000400      /* AMD: IBS */
#define CPUID_AMD_ECX_SSE5      0x00000800      /* AMD: SSE5 */
#define CPUID_AMD_ECX_SKINIT    0x00001000      /* AMD: SKINIT */
#define CPUID_AMD_ECX_WDT       0x00002000      /* AMD: WDT */
#define CPUID_AMD_ECX_TOPOEXT   0x00400000      /* AMD: Topology Extensions */

#define FMT_CPUID_AMD_ECX                                           \
        "\20"                                                       \
        "\22topoext"                                                \
        "\14wdt\13skinit\12sse5\11ibs\10osvw\93dnp\8mas"            \
        "\7sse4a\6lzcnt\5cr8d\3svm\2lcmplgcy\1ahf64"

/*
 * Intel now seems to have claimed part of the "extended" function
 * space that we previously for non-Intel implementors to use.
 * More excitingly still, they've claimed bit 20 to mean LAHF/SAHF
 * is available in long mode i.e. what AMD indicate using bit 0.
 * On the other hand, everything else is labelled as reserved.
 */
#define CPUID_INTC_ECX_AHF64    0x00100000      /* LAHF and SAHF in long mode */


#define P5_MCHADDR      0x0
#define P5_CESR         0x11
#define P5_CTR0         0x12
#define P5_CTR1         0x13

#define K5_MCHADDR      0x0
#define K5_MCHTYPE      0x01
#define K5_TSC          0x10
#define K5_TR12         0x12

#define REG_PAT         0x277

#define REG_MC0_CTL             0x400
#define REG_MC5_MISC            0x417
#define REG_PERFCTR0            0xc1
#define REG_PERFCTR1            0xc2

#define REG_PERFEVNT0           0x186
#define REG_PERFEVNT1           0x187

#define REG_TSC                 0x10    /* timestamp counter */
#define REG_APIC_BASE_MSR       0x1b
#define REG_X2APIC_BASE_MSR     0x800   /* The MSR address offset of x2APIC */

#if !defined(__xpv)
/*
 * AMD C1E
 */
#define MSR_AMD_INT_PENDING_CMP_HALT    0xC0010055
#define AMD_ACTONCMPHALT_SHIFT  27
#define AMD_ACTONCMPHALT_MASK   3
#endif

#define MSR_DEBUGCTL            0x1d9

#define DEBUGCTL_LBR            0x01
#define DEBUGCTL_BTF            0x02

/* Intel P6, AMD */
#define MSR_LBR_FROM            0x1db
#define MSR_LBR_TO              0x1dc
#define MSR_LEX_FROM            0x1dd
```

```
258 #define MSR_LEX_TO              0x1de

260 /* Intel P4 (pre-Prescott, non P4 M) */
261 #define MSR_P4_LBSTK_TOS        0x1da
262 #define MSR_P4_LBSTK_0          0x1db
263 #define MSR_P4_LBSTK_1          0x1dc
264 #define MSR_P4_LBSTK_2          0x1dd
265 #define MSR_P4_LBSTK_3          0x1de

267 /* Intel Pentium M */
268 #define MSR_P6M_LBSTK_TOS       0x1c9
269 #define MSR_P6M_LBSTK_0         0x040
270 #define MSR_P6M_LBSTK_1         0x041
271 #define MSR_P6M_LBSTK_2         0x042
272 #define MSR_P6M_LBSTK_3         0x043
273 #define MSR_P6M_LBSTK_4         0x044
274 #define MSR_P6M_LBSTK_5         0x045
275 #define MSR_P6M_LBSTK_6         0x046
276 #define MSR_P6M_LBSTK_7         0x047

278 /* Intel P4 (Prescott) */
279 #define MSR_PRP4_LBSTK_TOS      0x1da
280 #define MSR_PRP4_LBSTK_FROM_0   0x680
281 #define MSR_PRP4_LBSTK_FROM_1   0x681
282 #define MSR_PRP4_LBSTK_FROM_2   0x682
283 #define MSR_PRP4_LBSTK_FROM_3   0x683
284 #define MSR_PRP4_LBSTK_FROM_4   0x684
285 #define MSR_PRP4_LBSTK_FROM_5   0x685
286 #define MSR_PRP4_LBSTK_FROM_6   0x686
287 #define MSR_PRP4_LBSTK_FROM_7   0x687
288 #define MSR_PRP4_LBSTK_FROM_8   0x688
289 #define MSR_PRP4_LBSTK_FROM_9   0x689
290 #define MSR_PRP4_LBSTK_FROM_10  0x68a
291 #define MSR_PRP4_LBSTK_FROM_11  0x68b
292 #define MSR_PRP4_LBSTK_FROM_12  0x68c
293 #define MSR_PRP4_LBSTK_FROM_13  0x68d
294 #define MSR_PRP4_LBSTK_FROM_14  0x68e
295 #define MSR_PRP4_LBSTK_FROM_15  0x68f
296 #define MSR_PRP4_LBSTK_TO_0     0x6c0
297 #define MSR_PRP4_LBSTK_TO_1     0x6c1
298 #define MSR_PRP4_LBSTK_TO_2     0x6c2
299 #define MSR_PRP4_LBSTK_TO_3     0x6c3
300 #define MSR_PRP4_LBSTK_TO_4     0x6c4
301 #define MSR_PRP4_LBSTK_TO_5     0x6c5
302 #define MSR_PRP4_LBSTK_TO_6     0x6c6
303 #define MSR_PRP4_LBSTK_TO_7     0x6c7
304 #define MSR_PRP4_LBSTK_TO_8     0x6c8
305 #define MSR_PRP4_LBSTK_TO_9     0x6c9
306 #define MSR_PRP4_LBSTK_TO_10    0x6ca
307 #define MSR_PRP4_LBSTK_TO_11    0x6cb
308 #define MSR_PRP4_LBSTK_TO_12    0x6cc
309 #define MSR_PRP4_LBSTK_TO_13    0x6cd
310 #define MSR_PRP4_LBSTK_TO_14    0x6ce
311 #define MSR_PRP4_LBSTK_TO_15    0x6cf

313 #define MCI_CTL_VALUE           0xffffffff

315 #define MTRR_TYPE_UC            0
316 #define MTRR_TYPE_WC            1
317 #define MTRR_TYPE_WT            4
318 #define MTRR_TYPE_WP            5
319 #define MTRR_TYPE_WB            6
320 #define MTRR_TYPE_UC_           7

322 /*
323  * For Solaris we set up the page attritubute table in the following way:
```



```
324  * PAT0 Write-Back
325  * PAT1 Write-Through
326  * PAT2 Unchacheable-
327  * PAT3 Uncacheable
328  * PAT4 Write-Back
329  * PAT5 Write-Through
330  * PAT6 Write-Combine
331  * PAT7 Uncacheable
332  * The only difference from h/w default is entry 6.
333  */
334 #define PAT_DEFAULT_ATTRIBUTE                   \
335         ((uint64_t)MTRR_TYPE_WB |               \
336         ((uint64_t)MTRR_TYPE_WT << 8) |         \
337         ((uint64_t)MTRR_TYPE_UC_ << 16) |       \
338         ((uint64_t)MTRR_TYPE_UC << 24) |        \
339         ((uint64_t)MTRR_TYPE_WB << 32) |        \
340         ((uint64_t)MTRR_TYPE_WT << 40) |        \
341         ((uint64_t)MTRR_TYPE_WC << 48) |        \
342         ((uint64_t)MTRR_TYPE_UC << 56))

344 #define X86FSET_LARGEPAGE       0
345 #define X86FSET_TSC             1
346 #define X86FSET_MSR             2
347 #define X86FSET_MTRR            3
348 #define X86FSET_PGE             4
349 #define X86FSET_DE              5
350 #define X86FSET_CMOV            6
351 #define X86FSET_MMX             7
352 #define X86FSET_MCA             8
353 #define X86FSET_PAE             9
354 #define X86FSET_CX8             10
355 #define X86FSET_PAT             11
356 #define X86FSET_SEP             12
357 #define X86FSET_SSE             13
358 #define X86FSET_SSE2            14
359 #define X86FSET_HTT             15
360 #define X86FSET_ASYSC           16
361 #define X86FSET_NX              17
362 #define X86FSET_SSE3            18
363 #define X86FSET_CX16            19
364 #define X86FSET_CMP             20
365 #define X86FSET_TSCP            21
366 #define X86FSET_MWAIT           22
367 #define X86FSET_SSE4A           23
368 #define X86FSET_CPUID           24
369 #define X86FSET_SSSE3           25
370 #define X86FSET_SSE4_1          26
371 #define X86FSET_SSE4_2          27
372 #define X86FSET_1GPG            28
373 #define X86FSET_CLFSH           29
374 #define X86FSET_64              30
375 #define X86FSET_AES             31
376 #define X86FSET_PCLMULQDQ       32
377 #define X86FSET_XSAVE           33
378 #define X86FSET_AVX             34
379 #define X86FSET_VMX             35
380 #define X86FSET_SVM             36
381 #define X86FSET_TOPOEXT         37
382 #define X86FSET_F16C            38
383 #define X86FSET_RDRAND          39
384 #define X86FSET_X2APIC          40
385 #endif /* ! codereview */

387 /*
388  * flags to patch tsc_read routine.
389  */
```

```
390 #define X86_NO_TSC              0x0
391 #define X86_HAVE_TSCP           0x1
392 #define X86_TSC_MFENCE          0x2
393 #define X86_TSC_LFENCE          0x4

395 /*
396  * Intel Deep C-State invariant TSC in leaf 0x80000007.
397  */
398 #define CPUID_TSC_CSTATE_INVARIANCE     (0x100)

400 /*
401  * Intel Deep C-state always-running local APIC timer
402  */
403 #define CPUID_CSTATE_ARAT       (0x4)

405 /*
406  * Intel ENERGY_PERF_BIAS MSR indicated by feature bit CPUID.6.ECX[3].
407  */
408 #define CPUID_EPB_SUPPORT       (1 << 3)

410 /*
411  * Intel TSC deadline timer
412  */
413 #define CPUID_DEADLINE_TSC      (1 << 24)

415 /*
416  * x86_type is a legacy concept; this is supplanted
417  * for most purposes by x86_featureset; modern CPUs
418  * should be X86_TYPE_OTHER
419  */
420 #define X86_TYPE_OTHER          0
421 #define X86_TYPE_486            1
422 #define X86_TYPE_P5             2
423 #define X86_TYPE_P6             3
424 #define X86_TYPE_CYRIX_486      4
425 #define X86_TYPE_CYRIX_6x86L    5
426 #define X86_TYPE_CYRIX_6x86     6
427 #define X86_TYPE_CYRIX_GXm      7
428 #define X86_TYPE_CYRIX_6x86MX   8
429 #define X86_TYPE_CYRIX_MediaGX  9
430 #define X86_TYPE_CYRIX_MII      10
431 #define X86_TYPE_VIA_CYRIX_III  11
432 #define X86_TYPE_P4             12

434 /*
435  * x86_vendor allows us to select between
436  * implementation features and helps guide
437  * the interpretation of the cpuid instruction.
438  */
439 #define X86_VENDOR_Intel        0
440 #define X86_VENDORSTR_Intel     "GenuineIntel"

442 #define X86_VENDOR_IntelClone   1

444 #define X86_VENDOR_AMD          2
445 #define X86_VENDORSTR_AMD       "AuthenticAMD"

447 #define X86_VENDOR_Cyrix        3
448 #define X86_VENDORSTR_CYRIX     "CyrixInstead"

450 #define X86_VENDOR_UMC          4
451 #define X86_VENDORSTR_UMC       "UMC UMC UMC "

453 #define X86_VENDOR_NexGen       5
454 #define X86_VENDORSTR_NexGen    "NexGenDriven"
```



```
456 #define X86_VENDOR_Centaur      6
457 #define X86_VENDORSTR_Centaur   "CentaurHauls"

459 #define X86_VENDOR_Rise         7
460 #define X86_VENDORSTR_Rise      "RiseRiseRise"

462 #define X86_VENDOR_SiS          8
463 #define X86_VENDORSTR_SiS       "SiS SiS SiS "

465 #define X86_VENDOR_TM           9
466 #define X86_VENDORSTR_TM        "GenuineTMx86"

468 #define X86_VENDOR_NSC          10
469 #define X86_VENDORSTR_NSC       "Geode by NSC"

471 /*
472  * Vendor string max len + \0
473  */
474 #define X86_VENDOR_STRLEN       13

476 /*
477  * Some vendor/family/model/stepping ranges are commonly grouped under
478  * a single identifying banner by the vendor.  The following encode
479  * that "revision" in a uint32_t with the 8 most significant bits
480  * identifying the vendor with X86_VENDOR_*, the next 8 identifying the
481  * family, and the remaining 16 typically forming a bitmask of revisions
482  * within that family with more significant bits indicating "later" revisions.
483  */

485 #define _X86_CHIPREV_VENDOR_MASK        0xff000000u
486 #define _X86_CHIPREV_VENDOR_SHIFT       24
487 #define _X86_CHIPREV_FAMILY_MASK        0x00ff0000u
488 #define _X86_CHIPREV_FAMILY_SHIFT       16
489 #define _X86_CHIPREV_REV_MASK           0x0000ffffu

491 #define _X86_CHIPREV_VENDOR(x) \
492         (((x) & _X86_CHIPREV_VENDOR_MASK) >> _X86_CHIPREV_VENDOR_SHIFT)
493 #define _X86_CHIPREV_FAMILY(x) \
494         (((x) & _X86_CHIPREV_FAMILY_MASK) >> _X86_CHIPREV_FAMILY_SHIFT)
495 #define _X86_CHIPREV_REV(x) \
496         ((x) & _X86_CHIPREV_REV_MASK)

498 /* True if x matches in vendor and family and if x matches the given rev mask */
499 #define X86_CHIPREV_MATCH(x, mask) \
500         (_X86_CHIPREV_VENDOR(x) == _X86_CHIPREV_VENDOR(mask) && \
501         _X86_CHIPREV_FAMILY(x) == _X86_CHIPREV_FAMILY(mask) && \
502         ((_X86_CHIPREV_REV(x) & _X86_CHIPREV_REV(mask)) != 0))

504 /* True if x matches in vendor and family, and rev is at least minx */
505 #define X86_CHIPREV_ATLEAST(x, minx) \
506         (_X86_CHIPREV_VENDOR(x) == _X86_CHIPREV_VENDOR(minx) && \
507         _X86_CHIPREV_FAMILY(x) == _X86_CHIPREV_FAMILY(minx) && \
508         _X86_CHIPREV_REV(x) >= _X86_CHIPREV_REV(minx))

510 #define _X86_CHIPREV_MKREV(vendor, family, rev) \
511         ((uint32_t)(vendor) << _X86_CHIPREV_VENDOR_SHIFT | \
512         (family) << _X86_CHIPREV_FAMILY_SHIFT | (rev))

514 /* True if x matches in vendor, and family is at least minx */
515 #define X86_CHIPFAM_ATLEAST(x, minx) \
516         (_X86_CHIPREV_VENDOR(x) == _X86_CHIPREV_VENDOR(minx) && \
517         _X86_CHIPREV_FAMILY(x) >= _X86_CHIPREV_FAMILY(minx))

519 /* Revision default */
520 #define X86_CHIPREV_UNKNOWN     0x0
```

```
522 /*
523  * Definitions for AMD Family 0xf. Minor revisions C0 and CG are
524  * sufficiently different that we will distinguish them; in all other
525  * case we will identify the major revision.
526  */
527 #define X86_CHIPREV_AMD_F_REV_B _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0001)
528 #define X86_CHIPREV_AMD_F_REV_C0 _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0002)
529 #define X86_CHIPREV_AMD_F_REV_CG _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0004)
530 #define X86_CHIPREV_AMD_F_REV_D _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0008)
531 #define X86_CHIPREV_AMD_F_REV_E _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0010)
532 #define X86_CHIPREV_AMD_F_REV_F _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0020)
533 #define X86_CHIPREV_AMD_F_REV_G _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0040)

535 /*
536  * Definitions for AMD Family 0x10.  Rev A was Engineering Samples only.
537  */
538 #define X86_CHIPREV_AMD_10_REV_A \
539         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0001)
540 #define X86_CHIPREV_AMD_10_REV_B \
541         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0002)
542 #define X86_CHIPREV_AMD_10_REV_C2 \
543         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0004)
544 #define X86_CHIPREV_AMD_10_REV_C3 \
545         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0008)
546 #define X86_CHIPREV_AMD_10_REV_D0 \
547         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0010)
548 #define X86_CHIPREV_AMD_10_REV_D1 \
549         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0020)
550 #define X86_CHIPREV_AMD_10_REV_E \
551         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0040)

553 /*
554  * Definitions for AMD Family 0x11.
555  */
556 #define X86_CHIPREV_AMD_11_REV_B \
557         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x11, 0x0002)

559 /*
560  * Definitions for AMD Family 0x12.
561  */
562 #define X86_CHIPREV_AMD_12_REV_B \
563         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x12, 0x0002)

565 /*
566  * Definitions for AMD Family 0x14.
567  */
568 #define X86_CHIPREV_AMD_14_REV_B \
569         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x14, 0x0002)
570 #define X86_CHIPREV_AMD_14_REV_C \
571         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x14, 0x0004)

573 /*
574  * Definitions for AMD Family 0x15
575  */
576 #define X86_CHIPREV_AMD_15OR_REV_B2 \
577         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x15, 0x0001)

579 #define X86_CHIPREV_AMD_15TN_REV_A1 \
580         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x15, 0x0002)

582 /*
583  * Various socket/package types, extended as the need to distinguish
584  * a new type arises.  The top 8 byte identfies the vendor and the
585  * remaining 24 bits describe 24 socket types.
586  */
```



```
588 #define _X86_SOCKET_VENDOR_SHIFT        24
589 #define _X86_SOCKET_VENDOR(x)   ((x) >> _X86_SOCKET_VENDOR_SHIFT)
590 #define _X86_SOCKET_TYPE_MASK   0x00ffffff
591 #define _X86_SOCKET_TYPE(x)             ((x) & _X86_SOCKET_TYPE_MASK)

593 #define _X86_SOCKET_MKVAL(vendor, bitval) \
594         ((uint32_t)(vendor) << _X86_SOCKET_VENDOR_SHIFT | (bitval))

596 #define X86_SOCKET_MATCH(s, mask) \
597         (_X86_SOCKET_VENDOR(s) == _X86_SOCKET_VENDOR(mask) && \
598         (_X86_SOCKET_TYPE(s) & _X86_SOCKET_TYPE(mask)) != 0)

600 #define X86_SOCKET_UNKNOWN 0x0
601         /*
602          * AMD socket types
603          */
604 #define X86_SOCKET_754          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000001)
605 #define X86_SOCKET_939          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000002)
606 #define X86_SOCKET_940          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000004)
607 #define X86_SOCKET_S1g1         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000008)
608 #define X86_SOCKET_AM2          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000010)
609 #define X86_SOCKET_F1207        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000020)
610 #define X86_SOCKET_S1g2         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000040)
611 #define X86_SOCKET_S1g3         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000080)
612 #define X86_SOCKET_AM           _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000100)
613 #define X86_SOCKET_AM2R2        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000200)
614 #define X86_SOCKET_AM3          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000400)
615 #define X86_SOCKET_G34          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000800)
616 #define X86_SOCKET_ASB2         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x001000)
617 #define X86_SOCKET_C32          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x002000)
618 #define X86_SOCKET_S1g4         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x004000)
619 #define X86_SOCKET_FT1          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x008000)
620 #define X86_SOCKET_FM1          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x010000)
621 #define X86_SOCKET_FS1          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x020000)
622 #define X86_SOCKET_AM3R2        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x040000)
623 #define X86_SOCKET_FP2          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x080000)
624 #define X86_SOCKET_FS1R2        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x100000)
625 #define X86_SOCKET_FM2          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x200000)

627 /*
628  * xgetbv/xsetbv support
629  */

631 #define XFEATURE_ENABLED_MASK   0x0
632 /*
633  * XFEATURE_ENABLED_MASK values (eax)
634  */
635 #define XFEATURE_LEGACY_FP      0x1
636 #define XFEATURE_SSE            0x2
637 #define XFEATURE_AVX            0x4
638 #define XFEATURE_MAX            XFEATURE_AVX
639 #define XFEATURE_FP_ALL \
640         (XFEATURE_LEGACY_FP|XFEATURE_SSE|XFEATURE_AVX)

642 #if !defined(_ASM)

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

646 #define NUM_X86_FEATURES        41
368 #define NUM_X86_FEATURES        40
647 extern uchar_t x86_featureset[];

649 extern void free_x86_featureset(void *featureset);
650 extern boolean_t is_x86_feature(void *featureset, uint_t feature);
651 extern void add_x86_feature(void *featureset, uint_t feature);
652 extern void remove_x86_feature(void *featureset, uint_t feature);
```

```
653 extern boolean_t compare_x86_featureset(void *setA, void *setB);
654 extern void print_x86_featureset(void *featureset);


657 extern uint_t x86_type;
658 extern uint_t x86_vendor;
659 extern uint_t x86_clflush_size;

661 extern uint_t pentiumpro_bug4046376;
384 extern uint_t pentiumpro_bug4064495;

386 extern uint_t enable486;

663 extern const char CyrixInstead[];

665 #endif

667 #if defined(_KERNEL)

669 /*
670  * This structure is used to pass arguments and get return values back
671  * from the CPUID instruction in __cpuid_insn() routine.
672  */
673 struct cpuid_regs {
674         uint32_t        cp_eax;
675         uint32_t        cp_ebx;
676         uint32_t        cp_ecx;
677         uint32_t        cp_edx;
678 };
```

_____unchanged_portion_omitted_

```
**********************************************************
    6667 Fri Jan  3 22:11:57 2014
new/usr/src/uts/sun4/os/mp_states.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 187 /*
 188  * Stop all other cpu's before halting or rebooting. We pause the cpu's
 189  * instead of sending a cross call.
 190  */
 191 void
 192 stop_other_cpus(void)
 193 {
 194         mutex_enter(&cpu_lock);
 195         if (cpu_are_paused) {
 196                 mutex_exit(&cpu_lock);
 197                 return;
 198         }

 200         if (ncpus > 1)
 201                 intr_redist_all_cpus_shutdown();

 203         pause_cpus(NULL, NULL);
 203         pause_cpus(NULL);
 204         cpu_are_paused = 1;

 206         mutex_exit(&cpu_lock);
 207 }
______unchanged_portion_omitted_
```

```
**********************************************************
   16840 Fri Jan  3 22:11:57 2014
new/usr/src/uts/sun4/os/prom_subr.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 406 /*
 407  * This routine is a special form of pause_cpus().  It ensures that
 408  * prom functions are callable while the cpus are paused.
 409  */
 410 void
 411 promsafe_pause_cpus(void)
 412 {
 413         pause_cpus(NULL, NULL);
 413         pause_cpus(NULL);

 415         /* If some other cpu is entering or is in the prom, spin */
 416         while (prom_cpu || mutex_owner(&prom_mutex)) {

 418                 start_cpus();
 419                 mutex_enter(&prom_mutex);

 421                 /* Wait for other cpu to exit prom */
 422                 while (prom_cpu)
 423                         cv_wait(&prom_cv, &prom_mutex);

 425                 mutex_exit(&prom_mutex);
 426                 pause_cpus(NULL, NULL);
 426                 pause_cpus(NULL);
 427         }

 429         /* At this point all cpus are paused and none are in the prom */
 430 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   24533 Fri Jan  3 22:11:57 2014
new/usr/src/uts/sun4u/io/mem_cache.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 553 static int
 554 mem_cache_ioctl_ops(int cmd, int mode, cache_info_t *cache_info)
 555 {
 556         int     ret_val = 0;
 557         uint64_t afar, tag_addr;
 558         ch_cpu_logout_t clop;
 559         uint64_t Lxcache_tag_data[PN_CACHE_NWAYS];
 560         int     i, retire_retry_count;
 561         cpu_t   *cpu;
 562         uint64_t tag_data;
 563         uint8_t state;

 565         if (cache_info->way >= PN_CACHE_NWAYS)
 566                 return (EINVAL);
 567         switch (cache_info->cache) {
 568                 case L2_CACHE_TAG:
 569                 case L2_CACHE_DATA:
 570                         if (cache_info->index >=
 571                             (PN_L2_SET_SIZE/PN_L2_LINESIZE))
 572                                 return (EINVAL);
 573                         break;
 574                 case L3_CACHE_TAG:
 575                 case L3_CACHE_DATA:
 576                         if (cache_info->index >=
 577                             (PN_L3_SET_SIZE/PN_L3_LINESIZE))
 578                                 return (EINVAL);
 579                         break;
 580                 default:
 581                         return (ENOTSUP);
 582         }
 583         /*
 584          * Check if we have a valid cpu ID and that
 585          * CPU is ONLINE.
 586          */
 587         mutex_enter(&cpu_lock);
 588         cpu = cpu_get(cache_info->cpu_id);
 589         if ((cpu == NULL) || (!cpu_is_online(cpu))) {
 590                 mutex_exit(&cpu_lock);
 591                 return (EINVAL);
 592         }
 593         mutex_exit(&cpu_lock);
 594         pattern = 0;    /* default value of TAG PA when cacheline is retired. */
 595         switch (cmd) {
 596                 case MEM_CACHE_RETIRE:
 597                         tag_addr = get_tag_addr(cache_info);
 598                         pattern |= PN_ECSTATE_NA;
 599                         retire_retry_count = 0;
 600                         affinity_set(cache_info->cpu_id);
 601                         switch (cache_info->cache) {
 602                                 case L2_CACHE_DATA:
 603                                 case L2_CACHE_TAG:
 604                                         if ((cache_info->bit & MSB_BIT_MASK) ==
 605                                             MSB_BIT_MASK)
 606                                                 pattern |= PN_L2TAG_PA_MASK;
 607 retry_l2_retire:
 608                                         if (tag_addr_collides(tag_addr,
 609                                             cache_info->cache,
 610                                             retire_l2_start, retire_l2_end))
 611                                                 ret_val =
```



```
 612                                                     retire_l2_alternate(
 613                                                     tag_addr, pattern);
 614                                         else
 615                                                 ret_val = retire_l2(tag_addr,
 616                                                     pattern);
 617                                         if (ret_val == 1) {
 618                                                 /*
 619                                                  * cacheline was in retired
 620                                                  * STATE already.
 621                                                  * so return success.
 622                                                  */
 623                                                 ret_val = 0;
 624                                         }
 625                                         if (ret_val < 0) {
 626                                                 cmn_err(CE_WARN,
 627                 "retire_l2() failed. index = 0x%x way %d. Retrying...\n",
 628                                                     cache_info->index,
 629                                                     cache_info->way);
 630                                                 if (retire_retry_count >= 2) {
 631                                                         retire_failures++;
 632                                                         affinity_clear();
 633                                                         return (EIO);
 634                                                 }
 635                                                 retire_retry_count++;
 636                                                 goto retry_l2_retire;
 637                                         }
 638                                         if (ret_val == 2)
 639                                                 l2_flush_retries_done++;
 640                                 /*
 641                                  * We bind ourself to a CPU and send cross trap to
 642                                  * ourself. On return from xt_one we can rely on the
 643                                  * data in tag_data being filled in. Normally one would
 644                                  * do a xt_sync to make sure that the CPU has completed
 645                                  * the cross trap call xt_one.
 646                                  */
 647                                         xt_one(cache_info->cpu_id,
 648                                             (xcfunc_t *)(get_l2_tag_tl1),
 649                                             tag_addr, (uint64_t)(&tag_data));
 650                                         state = tag_data & CH_ECSTATE_MASK;
 651                                         if (state != PN_ECSTATE_NA) {
 652                                                 retire_failures++;
 653                                                 print_l2_tag(tag_addr,
 654                                                     tag_data);
 655                                                 cmn_err(CE_WARN,
 656                 "L2 RETIRE:failed for index 0x%x way %d. Retrying...\n",
 657                                                     cache_info->index,
 658                                                     cache_info->way);
 659                                                 if (retire_retry_count >= 2) {
 660                                                         retire_failures++;
 661                                                         affinity_clear();
 662                                                         return (EIO);
 663                                                 }
 664                                                 retire_retry_count++;
 665                                                 goto retry_l2_retire;
 666                                         }
 667                                         break;
 668                                 case L3_CACHE_TAG:
 669                                 case L3_CACHE_DATA:
 670                                         if ((cache_info->bit & MSB_BIT_MASK) ==
 671                                             MSB_BIT_MASK)
 672                                                 pattern |= PN_L3TAG_PA_MASK;
 673                                         if (tag_addr_collides(tag_addr,
 674                                             cache_info->cache,
 675                                             retire_l3_start, retire_l3_end))
 676                                                 ret_val =
 677                                                     retire_l3_alternate(
```

```
678                                             tag_addr, pattern);
679                                     else
680                                             ret_val = retire_l3(tag_addr,
681                                                 pattern);
682                                     if (ret_val == 1) {
683                                             /*
684                                              * cacheline was in retired
685                                              * STATE already.
686                                              * so return success.
687                                              */
688                                             ret_val = 0;
689                                     }
690                                     if (ret_val < 0) {
691                                             cmn_err(CE_WARN,
692                             "retire_l3() failed. ret_val = %d index = 0x%x\n",
693                                                 ret_val,
694                                                 cache_info->index);
695                                             retire_failures++;
696                                             affinity_clear();
697                                             return (EIO);
698                                     }
699                             /*
700                              * We bind ourself to a CPU and send cross trap to
701                              * ourself. On return from xt_one we can rely on the
702                              * data in tag_data being filled in. Normally one would
703                              * do a xt_sync to make sure that the CPU has completed
704                              * the cross trap call xt_one.
705                              */
706                                     xt_one(cache_info->cpu_id,
707                                         (xcfunc_t *)(get_l3_tag_tl1),
708                                         tag_addr, (uint64_t)(&tag_data));
709                                     state = tag_data & CH_ECSTATE_MASK;
710                                     if (state != PN_ECSTATE_NA) {
711                                             cmn_err(CE_WARN,
712                                     "L3 RETIRE failed for index 0x%x\n",
713                                                 cache_info->index);
714                                             retire_failures++;
715                                             affinity_clear();
716                                             return (EIO);
717                                     }

719                                     break;
720                             }
721                             affinity_clear();
722                             break;
723                     case MEM_CACHE_UNRETIRE:
724                             tag_addr = get_tag_addr(cache_info);
725                             pattern = PN_ECSTATE_INV;
726                             affinity_set(cache_info->cpu_id);
727                             switch (cache_info->cache) {
728                                     case L2_CACHE_DATA:
729                                     case L2_CACHE_TAG:
730                             /*
731                              * We bind ourself to a CPU and send cross trap to
732                              * ourself. On return from xt_one we can rely on the
733                              * data in tag_data being filled in. Normally one would
734                              * do a xt_sync to make sure that the CPU has completed
735                              * the cross trap call xt_one.
736                              */
737                                     xt_one(cache_info->cpu_id,
738                                         (xcfunc_t *)(get_l2_tag_tl1),
739                                         tag_addr, (uint64_t)(&tag_data));
740                                     state = tag_data & CH_ECSTATE_MASK;
741                                     if (state != PN_ECSTATE_NA) {
742                                             affinity_clear();
743                                             return (EINVAL);
```



```
744                                     }
745                                     if (tag_addr_collides(tag_addr,
746                                         cache_info->cache,
747                                         unretire_l2_start, unretire_l2_end))
748                                             ret_val =
749                                                 unretire_l2_alternate(
750                                                 tag_addr, pattern);
751                                     else
752                                             ret_val =
753                                                 unretire_l2(tag_addr,
754                                                 pattern);
755                                     if (ret_val != 0) {
756                                             cmn_err(CE_WARN,
757                             "unretire_l2() failed. ret_val = %d index = 0x%x\n",
758                                                 ret_val,
759                                                 cache_info->index);
760                                             retire_failures++;
761                                             affinity_clear();
762                                             return (EIO);
763                                     }
764                                     break;
765                             case L3_CACHE_TAG:
766                             case L3_CACHE_DATA:
767                             /*
768                              * We bind ourself to a CPU and send cross trap to
769                              * ourself. On return from xt_one we can rely on the
770                              * data in tag_data being filled in. Normally one would
771                              * do a xt_sync to make sure that the CPU has completed
772                              * the cross trap call xt_one.
773                              */
774                                     xt_one(cache_info->cpu_id,
775                                         (xcfunc_t *)(get_l3_tag_tl1),
776                                         tag_addr, (uint64_t)(&tag_data));
777                                     state = tag_data & CH_ECSTATE_MASK;
778                                     if (state != PN_ECSTATE_NA) {
779                                             affinity_clear();
780                                             return (EINVAL);
781                                     }
782                                     if (tag_addr_collides(tag_addr,
783                                         cache_info->cache,
784                                         unretire_l3_start, unretire_l3_end))
785                                             ret_val =
786                                                 unretire_l3_alternate(
787                                                 tag_addr, pattern);
788                                     else
789                                             ret_val =
790                                                 unretire_l3(tag_addr,
791                                                 pattern);
792                                     if (ret_val != 0) {
793                                             cmn_err(CE_WARN,
794                             "unretire_l3() failed. ret_val = %d index = 0x%x\n",
795                                                 ret_val,
796                                                 cache_info->index);
797                                             affinity_clear();
798                                             return (EIO);
799                                     }
800                                     break;
801                             }
802                             affinity_clear();
803                             break;
804                     case MEM_CACHE_ISRETIRED:
805                     case MEM_CACHE_STATE:
806                             return (ENOTSUP);
807                     case MEM_CACHE_READ_TAGS:
808 #ifdef DEBUG
809                     case MEM_CACHE_READ_ERROR_INJECTED_TAGS:
```

```
#endif
                        /*
                         * Read tag and data for all the ways at a given afar
                         */
                        afar = (uint64_t)(cache_info->index
                            << PN_CACHE_LINE_SHIFT);
                        mutex_enter(&cpu_lock);
                        affinity_set(cache_info->cpu_id);
                        (void) pause_cpus(NULL, NULL);
                        (void) pause_cpus(NULL);
                        mutex_exit(&cpu_lock);
                        /*
                         * We bind ourself to a CPU and send cross trap to
                         * ourself. On return from xt_one we can rely on the
                         * data in clop being filled in. Normally one would
                         * do a xt_sync to make sure that the CPU has completed
                         * the cross trap call xt_one.
                         */
                        xt_one(cache_info->cpu_id,
                            (xcfunc_t *)(get_ecache_dtags_tl1),
                            afar, (uint64_t)(&clop));
                        mutex_enter(&cpu_lock);
                        (void) start_cpus();
                        mutex_exit(&cpu_lock);
                        affinity_clear();
                        switch (cache_info->cache) {
                                case L2_CACHE_TAG:
                                        for (i = 0; i < PN_CACHE_NWAYS; i++) {
                                                Lxcache_tag_data[i] =
                                                    clop.clo_data.chd_l2_data
                                                    [i].ec_tag;
                                        }
#ifdef DEBUG
                                        last_error_injected_bit =
                                            last_l2tag_error_injected_bit;
                                        last_error_injected_way =
                                            last_l2tag_error_injected_way;
#endif
                                        break;
                                case L3_CACHE_TAG:
                                        for (i = 0; i < PN_CACHE_NWAYS; i++) {
                                                Lxcache_tag_data[i] =
                                                    clop.clo_data.chd_ec_data
                                                    [i].ec_tag;
                                        }
#ifdef DEBUG
                                        last_error_injected_bit =
                                            last_l3tag_error_injected_bit;
                                        last_error_injected_way =
                                            last_l3tag_error_injected_way;
#endif
                                        break;
                                default:
                                        return (ENOTSUP);
                        }       /* end if switch(cache) */
#ifdef DEBUG
                        if ((cmd == MEM_CACHE_READ_ERROR_INJECTED_TAGS) &&
                            (inject_anonymous_tag_error == 0) &&
                            (last_error_injected_way >= 0) &&
                            (last_error_injected_way <= 3)) {
                                pattern = ((uint64_t)1 <<
                                    last_error_injected_bit);
                                /*
                                 * If error bit is ECC we need to make sure
                                 * ECC on all all WAYS are corrupted.
                                 */
```



```
                                if ((last_error_injected_bit >= 6) &&
                                    (last_error_injected_bit <= 14)) {
                                        for (i = 0; i < PN_CACHE_NWAYS; i++)
                                                Lxcache_tag_data[i] ^=
                                                    pattern;
                                } else
                                        Lxcache_tag_data
                                            [last_error_injected_way] ^=
                                            pattern;
                        }
#endif
                        if (ddi_copyout((caddr_t)Lxcache_tag_data,
                            (caddr_t)cache_info->datap,
                            sizeof (Lxcache_tag_data), mode)
                            != DDI_SUCCESS) {
                                return (EFAULT);
                        }
                        break;  /* end of READ_TAGS */
                default:
                        return (ENOTSUP);
        }       /* end if switch(cmd) */
        return (ret_val);
}
```

_____unchanged_portion_omitted_

```
**********************************************************
   25005 Fri Jan  3 22:11:57 2014
new/usr/src/uts/sun4u/ngdr/io/dr_quiesce.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 822 int
 823 dr_suspend(dr_sr_handle_t *srh)
 824 {
 825         dr_handle_t     *handle;
 826         int             force;
 827         int             dev_errs_idx;
 828         uint64_t        dev_errs[DR_MAX_ERR_INT];
 829         int             rc = DDI_SUCCESS;

 831         handle = srh->sr_dr_handlep;

 833         force = dr_cmd_flags(handle) & SBD_FLAG_FORCE;

 835         /*
 836          * update the signature block
 837          */
 838         CPU_SIGNATURE(OS_SIG, SIGST_QUIESCE_INPROGRESS, SIGSUBST_NULL,
 839             CPU->cpu_id);

 841         prom_printf("\nDR: suspending user threads...\n");
 842         srh->sr_suspend_state = DR_SRSTATE_USER;
 843         if (((rc = dr_stop_user_threads(srh)) != DDI_SUCCESS) &&
 844             dr_check_user_stop_result) {
 845                 dr_resume(srh);
 846                 return (rc);
 847         }

 849         if (!force) {
 850                 struct dr_ref drc = {0};

 852                 prom_printf("\nDR: checking devices...\n");
 853                 dev_errs_idx = 0;

 855                 drc.arr = dev_errs;
 856                 drc.idx = &dev_errs_idx;
 857                 drc.len = DR_MAX_ERR_INT;

 859                 /*
 860                  * Since the root node can never go away, it
 861                  * doesn't have to be held.
 862                  */
 863                 ddi_walk_devs(ddi_root_node(), dr_check_unsafe_major, &drc);
 864                 if (dev_errs_idx) {
 865                         handle->h_err = drerr_int(ESBD_UNSAFE, dev_errs,
 866                             dev_errs_idx, 1);
 867                         dr_resume(srh);
 868                         return (DDI_FAILURE);
 869                 }
 870                 PR_QR("done\n");
 871         } else {
 872                 prom_printf("\nDR: dr_suspend invoked with force flag\n");
 873         }

 875 #ifndef SKIP_SYNC
 876         /*
 877          * This sync swap out all user pages
 878          */
 879         vfs_sync(SYNC_ALL);
 880 #endif
```



```
 882         /*
 883          * special treatment for lock manager
 884          */
 885         lm_cprsuspend();

 887 #ifndef SKIP_SYNC
 888         /*
 889          * sync the file system in case we never make it back
 890          */
 891         sync();
 892 #endif

 894         /*
 895          * now suspend drivers
 896          */
 897         prom_printf("DR: suspending drivers...\n");
 898         srh->sr_suspend_state = DR_SRSTATE_DRIVER;
 899         srh->sr_err_idx = 0;
 900         /* No parent to hold busy */
 901         if ((rc = dr_suspend_devices(ddi_root_node(), srh)) != DDI_SUCCESS) {
 902                 if (srh->sr_err_idx && srh->sr_dr_handlep) {
 903                         (srh->sr_dr_handlep)->h_err = drerr_int(ESBD_SUSPEND,
 904                             srh->sr_err_ints, srh->sr_err_idx, 1);
 905                 }
 906                 dr_resume(srh);
 907                 return (rc);
 908         }

 910         drmach_suspend_last();

 912         /*
 913          * finally, grab all cpus
 914          */
 915         srh->sr_suspend_state = DR_SRSTATE_FULL;

 917         /*
 918          * if watchdog was activated, disable it
 919          */
 920         if (watchdog_activated) {
 921                 mutex_enter(&tod_lock);
 922                 tod_ops.tod_clear_watchdog_timer();
 923                 mutex_exit(&tod_lock);
 924                 srh->sr_flags |= SR_FLAG_WATCHDOG;
 925         } else {
 926                 srh->sr_flags &= ~(SR_FLAG_WATCHDOG);
 927         }

 929         /*
 930          * Update the signature block.
 931          * This must be done before cpus are paused, since on Starcat the
 932          * cpu signature update aquires an adaptive mutex in the iosram driver.
 933          * Blocking with cpus paused can lead to deadlock.
 934          */
 935         CPU_SIGNATURE(OS_SIG, SIGST_QUIESCED, SIGSUBST_NULL, CPU->cpu_id);

 937         mutex_enter(&cpu_lock);
 938         pause_cpus(NULL, NULL);
 938         pause_cpus(NULL);
 939         dr_stop_intr();

 941         return (rc);
 942 }
______unchanged_portion_omitted_
```

```
**********************************************************
   50490 Fri Jan  3 22:11:57 2014
new/usr/src/uts/sun4u/os/cpr_impl.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_


/*
 * launch slave cpus into kernel text, pause them,
 * and restore the original prom pages
 */
void
i_cpr_mp_setup(void)
{
        extern void restart_other_cpu(int);
        cpu_t *cp;

        uint64_t kctx = kcontextreg;

        /*
         * Do not allow setting page size codes in MMU primary context
         * register while using cif wrapper. This is needed to work
         * around OBP incorrect handling of this MMU register.
         */
        kcontextreg = 0;

        /*
         * reset cpu_ready_set so x_calls work properly
         */
        CPUSET_ZERO(cpu_ready_set);
        CPUSET_ADD(cpu_ready_set, getprocessorid());

        /*
         * setup cif to use the cookie from the new/tmp prom
         * and setup tmp handling for calling prom services.
         */
        i_cpr_cif_setup(CIF_SPLICE);

        /*
         * at this point, only the nucleus and a few cpr pages are
         * mapped in.  once we switch to the kernel trap table,
         * we can access the rest of kernel space.
         */
        prom_set_traptable(&trap_table);

        if (ncpus > 1) {
                sfmmu_init_tsbs();

                mutex_enter(&cpu_lock);
                /*
                 * All of the slave cpus are not ready at this time,
                 * yet the cpu structures have various cpu_flags set;
                 * clear cpu_flags and mutex_ready.
                 * Since we are coming up from a CPU suspend, the slave cpus
                 * are frozen.
                 */
                for (cp = CPU->cpu_next; cp != CPU; cp = cp->cpu_next) {
                        cp->cpu_flags = CPU_FROZEN;
                        cp->cpu_m.mutex_ready = 0;
                }

                for (cp = CPU->cpu_next; cp != CPU; cp = cp->cpu_next)
                        restart_other_cpu(cp->cpu_id);

                pause_cpus(NULL, NULL);
```



```
                pause_cpus(NULL);
                mutex_exit(&cpu_lock);

                i_cpr_xcall(i_cpr_clear_entries);
        } else
                i_cpr_clear_entries(0, 0);

        /*
         * now unlink the cif wrapper;  WARNING: do not call any
         * prom_xxx() routines until after prom pages are restored.
         */
        i_cpr_cif_setup(CIF_UNLINK);

        (void) i_cpr_prom_pages(CPR_PROM_RESTORE);

        /* allow setting page size codes in MMU primary context register */
        kcontextreg = kctx;
}
______unchanged_portion_omitted_
```

```
**********************************************************
   19584 Fri Jan  3 22:11:57 2014
new/usr/src/uts/sun4u/serengeti/io/sbdp_quiesce.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 764 int
 765 sbdp_suspend(sbdp_sr_handle_t *srh)
 766 {
 767         int force;
 768         int rc = DDI_SUCCESS;

 770         force = (srh && (srh->sr_flags & SBDP_IOCTL_FLAG_FORCE));

 772         /*
 773          * if no force flag, check for unsafe drivers
 774          */
 775         if (force) {
 776                 SBDP_DBG_QR("\nsbdp_suspend invoked with force flag");
 777         }

 779         /*
 780          * update the signature block
 781          */
 782         CPU_SIGNATURE(OS_SIG, SIGST_QUIESCE_INPROGRESS, SIGSUBST_NULL,
 783             CPU->cpu_id);

 785         /*
 786          * first, stop all user threads
 787          */
 788         SBDP_DBG_QR("SBDP: suspending user threads...\n");
 789         SR_SET_STATE(srh, SBDP_SRSTATE_USER);
 790         if (((rc = sbdp_stop_user_threads(srh)) != DDI_SUCCESS) &&
 791             sbdp_check_user_stop_result) {
 792                 sbdp_resume(srh);
 793                 return (rc);
 794         }

 796 #ifndef SKIP_SYNC
 797         /*
 798          * This sync swap out all user pages
 799          */
 800         vfs_sync(SYNC_ALL);
 801 #endif

 803         /*
 804          * special treatment for lock manager
 805          */
 806         lm_cprsuspend();

 808 #ifndef SKIP_SYNC
 809         /*
 810          * sync the file system in case we never make it back
 811          */
 812         sync();

 814 #endif
 815         /*
 816          * now suspend drivers
 817          */
 818         SBDP_DBG_QR("SBDP: suspending drivers...\n");
 819         SR_SET_STATE(srh, SBDP_SRSTATE_DRIVER);

 821         /*
 822          * Root node doesn't have to be held in any way.
```



```
 823          */
 824         if ((rc = sbdp_suspend_devices(ddi_root_node(), srh)) != DDI_SUCCESS) {
 825                 sbdp_resume(srh);
 826                 return (rc);
 827         }

 829         /*
 830          * finally, grab all cpus
 831          */
 832         SR_SET_STATE(srh, SBDP_SRSTATE_FULL);

 834         /*
 835          * if watchdog was activated, disable it
 836          */
 837         if (watchdog_activated) {
 838                 mutex_enter(&tod_lock);
 839                 saved_watchdog_seconds = tod_ops.tod_clear_watchdog_timer();
 840                 mutex_exit(&tod_lock);
 841                 SR_SET_FLAG(srh, SR_FLAG_WATCHDOG);
 842         } else {
 843                 SR_CLEAR_FLAG(srh, SR_FLAG_WATCHDOG);
 844         }

 846         mutex_enter(&cpu_lock);
 847         pause_cpus(NULL, NULL);
 847         pause_cpus(NULL);
 848         sbdp_stop_intr();

 850         /*
 851          * update the signature block
 852          */
 853         CPU_SIGNATURE(OS_SIG, SIGST_QUIESCED, SIGSUBST_NULL, CPU->cpu_id);

 855         return (rc);
 856 }
______unchanged_portion_omitted_
```

```
**********************************************************
   54782 Fri Jan  3 22:11:58 2014
new/usr/src/uts/sun4v/os/mpo.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 210 /*
 211  * The MPO locks are to protect the MPO metadata while that
 212  * information is updated as a result of a memory DR operation.
 213  * The read lock must be acquired to read the metadata and the
 214  * write locks must be acquired to update it.
 215  */
 216 #define mpo_rd_lock     kpreempt_disable
 217 #define mpo_rd_unlock   kpreempt_enable

 219 static void
 220 mpo_wr_lock()
 221 {
 222         mutex_enter(&cpu_lock);
 223         pause_cpus(NULL, NULL);
 223         pause_cpus(NULL);
 224         mutex_exit(&cpu_lock);
 225 }
______unchanged_portion_omitted_
```

```
**********************************************************
   21873 Fri Jan  3 22:11:58 2014
new/usr/src/uts/sun4v/os/suspend.c
patch cpu-pause-func-deglobalize
**********************************************************
______unchanged_portion_omitted_

 348 /*
 349  * Obtain an updated MD from the hypervisor and update cpunodes, CPU HW
 350  * sharing data structures, and processor groups.
 351  */
 352 static void
 353 update_cpu_mappings(void)
 354 {
 355         md_t            *mdp;
 356         processorid_t   id;
 357         cpu_t           *cp;
 358         cpu_pg_t        *pgps[NCPU];

 360         if ((mdp = md_get_handle()) == NULL) {
 361                 DBG("suspend: md_get_handle failed");
 362                 return;
 363         }

 365         DBG("suspend: updating CPU mappings");

 367         mutex_enter(&cpu_lock);

 369         setup_chip_mappings(mdp);
 370         setup_exec_unit_mappings(mdp);
 371         for (id = 0; id < NCPU; id++) {
 372                 if ((cp = cpu_get(id)) == NULL)
 373                         continue;
 374                 cpu_map_exec_units(cp);
 375         }

 377         /*
 378          * Re-calculate processor groups.
 379          *
 380          * First tear down all PG information before adding any new PG
 381          * information derived from the MD we just downloaded. We must
 382          * call pg_cpu_inactive and pg_cpu_active with CPUs paused and
 383          * we want to minimize the number of times pause_cpus is called.
 384          * Inactivating all CPUs would leave PGs without any active CPUs,
 385          * so while CPUs are paused, call pg_cpu_inactive and swap in the
 386          * bootstrap PG structure saving the original PG structure to be
 387          * fini'd afterwards. This prevents the dispatcher from encountering
 388          * PGs in which all CPUs are inactive. Offline CPUs are already
 389          * inactive in their PGs and shouldn't be reactivated, so we must
 390          * not call pg_cpu_inactive or pg_cpu_active for those CPUs.
 391          */
 392         pause_cpus(NULL, NULL);
 392         pause_cpus(NULL);
 393         for (id = 0; id < NCPU; id++) {
 394                 if ((cp = cpu_get(id)) == NULL)
 395                         continue;
 396                 if ((cp->cpu_flags & CPU_OFFLINE) == 0)
 397                         pg_cpu_inactive(cp);
 398                 pgps[id] = cp->cpu_pg;
 399                 pg_cpu_bootstrap(cp);
 400         }
 401         start_cpus();

 403         /*
 404          * pg_cpu_fini* and pg_cpu_init* must be called while CPUs are
 405          * not paused. Use two separate loops here so that we do not
```



```
 406          * initialize PG data for CPUs until all the old PG data structures
 407          * are torn down.
 408          */
 409         for (id = 0; id < NCPU; id++) {
 410                 if ((cp = cpu_get(id)) == NULL)
 411                         continue;
 412                 pg_cpu_fini(cp, pgps[id]);
 413                 mpo_cpu_remove(id);
 414         }

 416         /*
 417          * Initialize PG data for each CPU, but leave the bootstrapped
 418          * PG structure in place to avoid running with any PGs containing
 419          * nothing but inactive CPUs.
 420          */
 421         for (id = 0; id < NCPU; id++) {
 422                 if ((cp = cpu_get(id)) == NULL)
 423                         continue;
 424                 mpo_cpu_add(mdp, id);
 425                 pgps[id] = pg_cpu_init(cp, B_TRUE);
 426         }

 428         /*
 429          * Now that PG data has been initialized for all CPUs in the
 430          * system, replace the bootstrapped PG structure with the
 431          * initialized PG structure and call pg_cpu_active for each CPU.
 432          */
 433         pause_cpus(NULL, NULL);
 433         pause_cpus(NULL);
 434         for (id = 0; id < NCPU; id++) {
 435                 if ((cp = cpu_get(id)) == NULL)
 436                         continue;
 437                 cp->cpu_pg = pgps[id];
 438                 if ((cp->cpu_flags & CPU_OFFLINE) == 0)
 439                         pg_cpu_active(cp);
 440         }
 441         start_cpus();

 443         mutex_exit(&cpu_lock);

 445         (void) md_fini_handle(mdp);
 446 }
______unchanged_portion_omitted_

 585 /*
 586  * Suspends the OS by pausing CPUs and calling into the HV to initiate
 587  * the suspend. When the HV routine hv_guest_suspend returns, the system
 588  * will be resumed. Must be called after a successful call to suspend_pre.
 589  * suspend_post must be called after suspend_start, whether or not
 590  * suspend_start returns an error.
 591  */
 592 /*ARGSUSED*/
 593 int
 594 suspend_start(char *error_reason, size_t max_reason_len)
 595 {
 596         uint64_t        source_tick;
 597         uint64_t        source_stick;
 598         uint64_t        rv;
 599         timestruc_t     source_tod;
 600         int             spl;

 602         ASSERT(suspend_supported());
 603         DBG("suspend: %s", __func__);

 605         sfmmu_ctxdoms_lock();
```

```
607         mutex_enter(&cpu_lock);

609         /* Suspend the watchdog */
610         watchdog_suspend();

612         /* Record the TOD */
613         mutex_enter(&tod_lock);
614         source_tod = tod_get();
615         mutex_exit(&tod_lock);

617         /* Pause all other CPUs */
618         pause_cpus(NULL, NULL);
618         pause_cpus(NULL);
619         DBG_PROM("suspend: CPUs paused\n");

621         /* Suspend cyclics */
622         cyclic_suspend();
623         DBG_PROM("suspend: cyclics suspended\n");

625         /* Disable interrupts */
626         spl = spl8();
627         DBG_PROM("suspend: spl8()\n");

629         source_tick = gettick_counter();
630         source_stick = gettick();
631         DBG_PROM("suspend: source_tick: 0x%lx\n", source_tick);
632         DBG_PROM("suspend: source_stick: 0x%lx\n", source_stick);

634         /*
635          * Call into the HV to initiate the suspend. hv_guest_suspend()
636          * returns after the guest has been resumed or if the suspend
637          * operation failed or was cancelled. After a successful suspend,
638          * the %tick and %stick registers may have changed by an amount
639          * that is not proportional to the amount of time that has passed.
640          * They may have jumped forwards or backwards. Some variation is
641          * allowed and accounted for using suspend_tick_stick_max_delta,
642          * but otherwise this jump must be uniform across all CPUs and we
643          * operate under the assumption that it is (maintaining two global
644          * offset variables--one for %tick and one for %stick.)
645          */
646         DBG_PROM("suspend: suspending... \n");
647         rv = hv_guest_suspend();
648         if (rv != 0) {
649                 splx(spl);
650                 cyclic_resume();
651                 start_cpus();
652                 watchdog_resume();
653                 mutex_exit(&cpu_lock);
654                 sfmmu_ctxdoms_unlock();
655                 DBG("suspend: failed, rv: %ld\n", rv);
656                 return (rv);
657         }

659         suspend_count++;

661         /* Update the global tick and stick offsets and the preserved TOD */
662         set_tick_offsets(source_tick, source_stick, &source_tod);

664         /* Ensure new offsets are globally visible before resuming CPUs */
665         membar_sync();

667         /* Enable interrupts */
668         splx(spl);

670         /* Set the {%tick,%stick}.NPT bits on all CPUs */
671         if (enable_user_tick_stick_emulation) {
```



```
672                 xc_all((xcfunc_t *)enable_tick_stick_npt, NULL, NULL);
673                 xt_sync(cpu_ready_set);
674                 ASSERT(gettick_npt() != 0);
675                 ASSERT(getstick_npt() != 0);
676         }

678         /* If emulation is enabled, but not currently active, enable it */
679         if (enable_user_tick_stick_emulation && !tick_stick_emulation_active) {
680                 tick_stick_emulation_active = B_TRUE;
681         }

683         sfmmu_ctxdoms_remove();

685         /* Resume cyclics, unpause CPUs */
686         cyclic_resume();
687         start_cpus();

689         /* Set the TOD */
690         mutex_enter(&tod_lock);
691         tod_set(source_tod);
692         mutex_exit(&tod_lock);

694         /* Re-enable the watchdog */
695         watchdog_resume();

697         mutex_exit(&cpu_lock);

699         /* Download the latest MD */
700         if ((rv = mach_descrip_update()) != 0)
701                 cmn_err(CE_PANIC, "suspend: mach_descrip_update failed: %ld",
702                     rv);

704         sfmmu_ctxdoms_update();
705         sfmmu_ctxdoms_unlock();

707         /* Get new MD, update CPU mappings/relationships */
708         if (suspend_update_cpu_mappings)
709                 update_cpu_mappings();

711         DBG("suspend: target tick: 0x%lx", gettick_counter());
712         DBG("suspend: target stick: 0x%llx", gettick());
713         DBG("suspend: user %%tick/%%stick emulation is %d",
714             tick_stick_emulation_active);
715         DBG("suspend: finished");

717         return (0);
718 }
```

_____***unchanged_portion_omitted_***