```
**********************************************************
   36500 Fri Jan 15 13:15:06 2016
new/usr/src/uts/common/io/cpudrv.c
XXXX cpudrv attach error handling is leaky
**********************************************************
______unchanged_portion_omitted_

 235 /*
 236  * Driver attach(9e) entry point.
 237  */
 238 static int
 239 cpudrv_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
 240 {
 241         int                     instance;
 242         cpudrv_devstate_t       *cpudsp;

 244         instance = ddi_get_instance(dip);

 246         switch (cmd) {
 247         case DDI_ATTACH:
 248                 DPRINTF(D_ATTACH, ("cpudrv_attach: instance %d: "
 249                     "DDI_ATTACH called\n", instance));
 250                 if (!cpudrv_is_enabled(NULL))
 251                         return (DDI_FAILURE);
 252                 if (ddi_soft_state_zalloc(cpudrv_state, instance) !=
 253                     DDI_SUCCESS) {
 254                         cmn_err(CE_WARN, "cpudrv_attach: instance %d: "
 255                             "can't allocate state", instance);
 256                         cpudrv_enabled = B_FALSE;
 257                         return (DDI_FAILURE);
 258                 }
 259                 if ((cpudsp = ddi_get_soft_state(cpudrv_state, instance)) ==
 260                     NULL) {
 261                         cmn_err(CE_WARN, "cpudrv_attach: instance %d: "
 262                             "can't get state", instance);
 263                         ddi_soft_state_free(cpudrv_state, instance);
 264                         cpudrv_enabled = B_FALSE;
 265                         return (DDI_FAILURE);
 266                 }
 267                 cpudsp->dip = dip;

 269                 /*
 270                  * Find CPU number for this dev_info node.
 271                  */
 272                 if (!cpudrv_get_cpu_id(dip, &(cpudsp->cpu_id))) {
 273                         cmn_err(CE_WARN, "cpudrv_attach: instance %d: "
 274                             "can't convert dip to cpu_id", instance);
 275                         ddi_soft_state_free(cpudrv_state, instance);
 276                         cpudrv_enabled = B_FALSE;
 277                         return (DDI_FAILURE);
 278                 }

 280                 if (!cpudrv_is_enabled(cpudsp)) {
 281                         cmn_err(CE_WARN, "cpudrv_attach: instance %d: "
 282                             "not supported or it got disabled on us",
 283                             instance);
 284                         cpudrv_enabled = B_FALSE;
 285                         ddi_soft_state_free(cpudrv_state, instance);
 286                         return (DDI_FAILURE);
 287                 }

 289                 mutex_init(&cpudsp->lock, NULL, MUTEX_DRIVER, NULL);
 290                 if (cpudrv_init(cpudsp) != DDI_SUCCESS) {
 291                         cpudrv_enabled = B_FALSE;
 292                         cpudrv_free(cpudsp);
 293                         ddi_soft_state_free(cpudrv_state, instance);
```



```
 294                         return (DDI_FAILURE);
 295                 }
 296                 if (cpudrv_comp_create(cpudsp) != DDI_SUCCESS) {
 297                         cpudrv_enabled = B_FALSE;
 298                         cpudrv_free(cpudsp);
 299                         ddi_soft_state_free(cpudrv_state, instance);
 300                         return (DDI_FAILURE);
 301                 }
 302                 if (ddi_prop_update_string(DDI_DEV_T_NONE,
 303                     dip, "pm-class", "CPU") != DDI_PROP_SUCCESS) {
 304                         cpudrv_enabled = B_FALSE;
 305                         cpudrv_free(cpudsp);
 306                         ddi_soft_state_free(cpudrv_state, instance);
 307                         return (DDI_FAILURE);
 308                 }

 310                 /*
 311                  * Taskq is used to dispatch routine to monitor CPU
 312                  * activities.
 313                  */
 314                 cpudsp->cpudrv_pm.tq = ddi_taskq_create(dip,
 315                     "cpudrv_monitor", CPUDRV_TASKQ_THREADS,
 316                     TASKQ_DEFAULTPRI, 0);
 317                 if (cpudsp->cpudrv_pm.tq == NULL) {
 318                         cpudrv_enabled = B_FALSE;
 319                         cpudrv_free(cpudsp);
 320                         ddi_soft_state_free(cpudrv_state, instance);
 321                         return (DDI_FAILURE);
 322                 }
 323 #endif /* ! codereview */

 325                 mutex_init(&cpudsp->cpudrv_pm.timeout_lock, NULL,
 326                     MUTEX_DRIVER, NULL);
 327                 cv_init(&cpudsp->cpudrv_pm.timeout_cv, NULL,
 328                     CV_DEFAULT, NULL);

 330                 /*
 331                  * Driver needs to assume that CPU is running at
 332                  * unknown speed at DDI_ATTACH and switch it to the
 333                  * needed speed. We assume that initial needed speed
 334                  * is full speed for us.
 335                  */
 336                 /*
 337                  * We need to take the lock because cpudrv_monitor()
 338                  * will start running in parallel with attach().
 339                  */
 340                 mutex_enter(&cpudsp->lock);
 341                 cpudsp->cpudrv_pm.cur_spd = NULL;
 342                 cpudsp->cpudrv_pm.pm_started = B_FALSE;
 343                 /*
 344                  * We don't call pm_raise_power() directly from attach
 345                  * because driver attach for a slave CPU node can
 346                  * happen before the CPU is even initialized. We just
 347                  * start the monitoring system which understands
 348                  * unknown speed and moves CPU to top speed when it
 349                  * has been initialized.
 350                  */
 351                 CPUDRV_MONITOR_INIT(cpudsp);
 352                 mutex_exit(&cpudsp->lock);

 354                 if (!cpudrv_mach_init(cpudsp)) {
 355                         cmn_err(CE_WARN, "cpudrv_attach: instance %d: "
 356                             "cpudrv_mach_init failed", instance);
 357                         cpudrv_enabled = B_FALSE;
 358                         ddi_taskq_destroy(cpudsp->cpudrv_pm.tq);
 359 #endif /* ! codereview */
```

```
360                         cpudrv_free(cpudsp);
361                         ddi_soft_state_free(cpudrv_state, instance);
362                         return (DDI_FAILURE);
363                 }

365                 CPUDRV_INSTALL_MAX_CHANGE_HANDLER(cpudsp);

367                 (void) ddi_prop_update_int(DDI_DEV_T_NONE, dip,
368                     DDI_NO_AUTODETACH, 1);
369                 ddi_report_dev(dip);
370                 return (DDI_SUCCESS);

372         case DDI_RESUME:
373                 DPRINTF(D_ATTACH, ("cpudrv_attach: instance %d: "
374                     "DDI_RESUME called\n", instance));

376                 cpudsp = ddi_get_soft_state(cpudrv_state, instance);
377                 ASSERT(cpudsp != NULL);

379                 /*
380                  * Nothing to do for resume, if not doing active PM.
381                  */
382                 if (!cpudrv_is_enabled(cpudsp))
383                         return (DDI_SUCCESS);

385                 mutex_enter(&cpudsp->lock);
386                 /*
387                  * Driver needs to assume that CPU is running at unknown speed
388                  * at DDI_RESUME and switch it to the needed speed. We assume
389                  * that the needed speed is full speed for us.
390                  */
391                 cpudsp->cpudrv_pm.cur_spd = NULL;
392                 CPUDRV_MONITOR_INIT(cpudsp);
393                 mutex_exit(&cpudsp->lock);
394                 CPUDRV_REDEFINE_TOPSPEED(dip);
395                 return (DDI_SUCCESS);

397         default:
398                 return (DDI_FAILURE);
399         }
400 }

402 /*
403  * Driver detach(9e) entry point.
404  */
405 static int
406 cpudrv_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
407 {
408         int                     instance;
409         cpudrv_devstate_t       *cpudsp;
410         cpudrv_pm_t             *cpupm;

412         instance = ddi_get_instance(dip);

414         switch (cmd) {
415         case DDI_DETACH:
416                 DPRINTF(D_DETACH, ("cpudrv_detach: instance %d: "
417                     "DDI_DETACH called\n", instance));

419 #if defined(__x86)
420                 cpudsp = ddi_get_soft_state(cpudrv_state, instance);
421                 ASSERT(cpudsp != NULL);

423                 /*
424                  * Nothing to do for detach, if no doing active PM.
425                  */
```



```
426                 if (!cpudrv_is_enabled(cpudsp))
427                         return (DDI_SUCCESS);

429                 /*
430                  * uninstall PPC/_TPC change notification handler
431                  */
432                 CPUDRV_UNINSTALL_MAX_CHANGE_HANDLER(cpudsp);

434                 /*
435                  * destruct platform specific resource
436                  */
437                 if (!cpudrv_mach_fini(cpudsp))
438                         return (DDI_FAILURE);

440                 mutex_enter(&cpudsp->lock);
441                 CPUDRV_MONITOR_FINI(cpudsp);
442                 cv_destroy(&cpudsp->cpudrv_pm.timeout_cv);
443                 mutex_destroy(&cpudsp->cpudrv_pm.timeout_lock);
444                 ddi_taskq_destroy(cpudsp->cpudrv_pm.tq);
445                 cpudrv_free(cpudsp);
446                 mutex_exit(&cpudsp->lock);
447                 mutex_destroy(&cpudsp->lock);
448                 ddi_soft_state_free(cpudrv_state, instance);
449                 (void) ddi_prop_update_int(DDI_DEV_T_NONE, dip,
450                     DDI_NO_AUTODETACH, 0);
451                 return (DDI_SUCCESS);

453 #else
454                 /*
455                  * If the only thing supported by the driver is power
456                  * management, we can in future enhance the driver and
457                  * framework that loads it to unload the driver when
458                  * user has disabled CPU power management.
459                  */
460                 return (DDI_FAILURE);
461 #endif

463         case DDI_SUSPEND:
464                 DPRINTF(D_DETACH, ("cpudrv_detach: instance %d: "
465                     "DDI_SUSPEND called\n", instance));

467                 cpudsp = ddi_get_soft_state(cpudrv_state, instance);
468                 ASSERT(cpudsp != NULL);

470                 /*
471                  * Nothing to do for suspend, if not doing active PM.
472                  */
473                 if (!cpudrv_is_enabled(cpudsp))
474                         return (DDI_SUCCESS);

476                 /*
477                  * During a checkpoint-resume sequence, framework will
478                  * stop interrupts to quiesce kernel activity. This will
479                  * leave our monitoring system ineffective. Handle this
480                  * by stopping our monitoring system and bringing CPU
481                  * to full speed. In case we are in special direct pm
482                  * mode, we leave the CPU at whatever speed it is. This
483                  * is harmless other than speed.
484                  */
485                 mutex_enter(&cpudsp->lock);
486                 cpupm = &(cpudsp->cpudrv_pm);

488                 DPRINTF(D_DETACH, ("cpudrv_detach: instance %d: DDI_SUSPEND - "
489                     "cur_spd %d, topspeed %d\n", instance,
490                     cpupm->cur_spd->pm_level,
491                     CPUDRV_TOPSPEED(cpupm)->pm_level));
```

```
493                 CPUDRV_MONITOR_FINI(cpudsp);

495                 if (!cpudrv_direct_pm && (cpupm->cur_spd !=
496                     CPUDRV_TOPSPEED(cpupm))) {
497                         if (cpupm->pm_busycnt < 1) {
498                                 if ((pm_busy_component(dip, CPUDRV_COMP_NUM)
499                                     == DDI_SUCCESS)) {
500                                         cpupm->pm_busycnt++;
501                                 } else {
502                                         CPUDRV_MONITOR_INIT(cpudsp);
503                                         mutex_exit(&cpudsp->lock);
504                                         cmn_err(CE_WARN, "cpudrv_detach: "
505                                             "instance %d: can't busy CPU "
506                                             "component", instance);
507                                         return (DDI_FAILURE);
508                                 }
509                         }
510                         mutex_exit(&cpudsp->lock);
511                         if (pm_raise_power(dip, CPUDRV_COMP_NUM,
512                             CPUDRV_TOPSPEED(cpupm)->pm_level) !=
513                             DDI_SUCCESS) {
514                                 mutex_enter(&cpudsp->lock);
515                                 CPUDRV_MONITOR_INIT(cpudsp);
516                                 mutex_exit(&cpudsp->lock);
517                                 cmn_err(CE_WARN, "cpudrv_detach: instance %d: "
518                                     "can't raise CPU power level to %d",
519                                     instance,
520                                     CPUDRV_TOPSPEED(cpupm)->pm_level);
521                                 return (DDI_FAILURE);
522                         } else {
523                                 return (DDI_SUCCESS);
524                         }
525                 } else {
526                         mutex_exit(&cpudsp->lock);
527                         return (DDI_SUCCESS);
528                 }

530         default:
531                 return (DDI_FAILURE);
532         }
533 }

535 /*
536  * Driver power(9e) entry point.
537  *
538  * Driver's notion of current power is set *only* in power(9e) entry point
539  * after actual power change operation has been successfully completed.
540  */
541 /* ARGSUSED */
542 static int
543 cpudrv_power(dev_info_t *dip, int comp, int level)
544 {
545         int                     instance;
546         cpudrv_devstate_t       *cpudsp;
547         cpudrv_pm_t             *cpudrvpm;
548         cpudrv_pm_spd_t         *new_spd;
549         boolean_t               is_ready;
550         int                     ret;

552         instance = ddi_get_instance(dip);

554         DPRINTF(D_POWER, ("cpudrv_power: instance %d: level %d\n",
555             instance, level));

557         if ((cpudsp = ddi_get_soft_state(cpudrv_state, instance)) == NULL) {
```



```
558                 cmn_err(CE_WARN, "cpudrv_power: instance %d: can't "
559                     "get state", instance);
560                 return (DDI_FAILURE);
561         }

563         /*
564          * We're not ready until we can  get a cpu_t
565          */
566         is_ready = (cpudrv_get_cpu(cpudsp) == DDI_SUCCESS);

568         mutex_enter(&cpudsp->lock);
569         cpudrvpm = &(cpudsp->cpudrv_pm);

571         /*
572          * In normal operation, we fail if we are busy and request is
573          * to lower the power level. We let this go through if the driver
574          * is in special direct pm mode. On x86, we also let this through
575          * if the change is due to a request to govern the max speed.
576          */
577         if (!cpudrv_direct_pm && (cpudrvpm->pm_busycnt >= 1) &&
578             !cpudrv_is_governor_thread(cpudrvpm)) {
579                 if ((cpudrvpm->cur_spd != NULL) &&
580                     (level < cpudrvpm->cur_spd->pm_level)) {
581                         mutex_exit(&cpudsp->lock);
582                         return (DDI_FAILURE);
583                 }
584         }

586         for (new_spd = cpudrvpm->head_spd; new_spd; new_spd =
587             new_spd->down_spd) {
588                 if (new_spd->pm_level == level)
589                         break;
590         }
591         if (!new_spd) {
592                 CPUDRV_RESET_GOVERNOR_THREAD(cpudrvpm);
593                 mutex_exit(&cpudsp->lock);
594                 cmn_err(CE_WARN, "cpudrv_power: instance %d: "
595                     "can't locate new CPU speed", instance);
596                 return (DDI_FAILURE);
597         }

599         /*
600          * We currently refuse to power manage if the CPU is not ready to
601          * take cross calls (cross calls fail silently if CPU is not ready
602          * for it).
603          *
604          * Additionally, for x86 platforms we cannot power manage an instance,
605          * until it has been initialized.
606          */
607         if (is_ready) {
608                 is_ready = CPUDRV_XCALL_IS_READY(cpudsp->cpu_id);
609                 if (!is_ready) {
610                         DPRINTF(D_POWER, ("cpudrv_power: instance %d: "
611                             "CPU not ready for x-calls\n", instance));
612                 } else if (!(is_ready = cpudrv_power_ready(cpudsp->cp))) {
613                         DPRINTF(D_POWER, ("cpudrv_power: instance %d: "
614                             "waiting for all CPUs to be power manageable\n",
615                             instance));
616                 }
617         }
618         if (!is_ready) {
619                 CPUDRV_RESET_GOVERNOR_THREAD(cpudrvpm);
620                 mutex_exit(&cpudsp->lock);
621                 return (DDI_FAILURE);
622         }
```

```
624         /*
625          * Execute CPU specific routine on the requested CPU to
626          * change its speed to normal-speed/divisor.
627          */
628         if ((ret = cpudrv_change_speed(cpudsp, new_spd)) != DDI_SUCCESS) {
629                 cmn_err(CE_WARN, "cpudrv_power: "
630                     "cpudrv_change_speed() return = %d", ret);
631                 mutex_exit(&cpudsp->lock);
632                 return (DDI_FAILURE);
633         }

635         /*
636          * Reset idle threshold time for the new power level.
637          */
638         if ((cpudrvpm->cur_spd != NULL) && (level <
639             cpudrvpm->cur_spd->pm_level)) {
640                 if (pm_idle_component(dip, CPUDRV_COMP_NUM) ==
641                     DDI_SUCCESS) {
642                         if (cpudrvpm->pm_busycnt >= 1)
643                                 cpudrvpm->pm_busycnt--;
644                 } else {
645                         cmn_err(CE_WARN, "cpudrv_power: instance %d: "
646                             "can't idle CPU component",
647                             ddi_get_instance(dip));
648                 }
649         }
650         /*
651          * Reset various parameters because we are now running at new speed.
652          */
653         cpudrvpm->lastquan_mstate[CMS_IDLE] = 0;
654         cpudrvpm->lastquan_mstate[CMS_SYSTEM] = 0;
655         cpudrvpm->lastquan_mstate[CMS_USER] = 0;
656         cpudrvpm->lastquan_ticks = 0;
657         cpudrvpm->cur_spd = new_spd;
658         CPUDRV_RESET_GOVERNOR_THREAD(cpudrvpm);
659         mutex_exit(&cpudsp->lock);

661         return (DDI_SUCCESS);
662 }

664 /*
665  * Initialize power management data.
666  */
667 static int
668 cpudrv_init(cpudrv_devstate_t *cpudsp)
669 {
670         cpudrv_pm_t     *cpupm = &(cpudsp->cpudrv_pm);
671         cpudrv_pm_spd_t *cur_spd;
672         cpudrv_pm_spd_t *prev_spd = NULL;
673         int             *speeds;
674         uint_t          nspeeds;
675         int             idle_cnt_percent;
676         int             user_cnt_percent;
677         int             i;

679         CPUDRV_GET_SPEEDS(cpudsp, speeds, nspeeds);
680         if (nspeeds < 2) {
681                 /* Need at least two speeds to power manage */
682                 CPUDRV_FREE_SPEEDS(speeds, nspeeds);
683                 return (DDI_FAILURE);
684         }
685         cpupm->num_spd = nspeeds;

687         /*
688          * Calculate the watermarks and other parameters based on the
689          * supplied speeds.
```



```
690          *
691          * One of the basic assumption is that for X amount of CPU work,
692          * if CPU is slowed down by a factor of N, the time it takes to
693          * do the same work will be N * X.
694          *
695          * The driver declares that a CPU is idle and ready for slowed down,
696          * if amount of idle thread is more than the current speed idle_hwm
697          * without dropping below idle_hwm a number of consecutive sampling
698          * intervals and number of running threads in user mode are below
699          * user_lwm.  We want to set the current user_lwm such that if we
700          * just switched to the next slower speed with no change in real work
701          * load, the amount of user threads at the slower speed will be such
702          * that it falls below the slower speed's user_hwm.  If we didn't do
703          * that then we will just come back to the higher speed as soon as we
704          * go down even with no change in work load.
705          * The user_hwm is a fixed precentage and not calculated dynamically.
706          *
707          * We bring the CPU up if idle thread at current speed is less than
708          * the current speed idle_lwm for a number of consecutive sampling
709          * intervals or user threads are above the user_hwm for the current
710          * speed.
711          */
712         for (i = 0; i < nspeeds; i++) {
713                 cur_spd = kmem_zalloc(sizeof (cpudrv_pm_spd_t), KM_SLEEP);
714                 cur_spd->speed = speeds[i];
715                 if (i == 0) {   /* normal speed */
716                         cpupm->head_spd = cur_spd;
717                         CPUDRV_TOPSPEED(cpupm) = cur_spd;
718                         cur_spd->quant_cnt = CPUDRV_QUANT_CNT_NORMAL;
719                         cur_spd->idle_hwm =
720                             (cpudrv_idle_hwm * cur_spd->quant_cnt) / 100;
721                         /* can't speed anymore */
722                         cur_spd->idle_lwm = 0;
723                         cur_spd->user_hwm = UINT_MAX;
724                 } else {
725                         cur_spd->quant_cnt = CPUDRV_QUANT_CNT_OTHR;
726                         ASSERT(prev_spd != NULL);
727                         prev_spd->down_spd = cur_spd;
728                         cur_spd->up_spd = cpupm->head_spd;

730                         /*
731                          * Let's assume CPU is considered idle at full speed
732                          * when it is spending I% of time in running the idle
733                          * thread.  At full speed, CPU will be busy (100 - I) %
734                          * of times.  This % of busyness increases by factor of
735                          * N as CPU slows down.  CPU that is idle I% of times
736                          * in full speed, it is idle (100 - ((100 - I) * N)) %
737                          * of times in N speed.  The idle_lwm is a fixed
738                          * percentage.  A large value of N may result in
739                          * idle_hwm to go below idle_lwm.  We need to make sure
740                          * that there is at least a buffer zone seperation
741                          * between the idle_lwm and idle_hwm values.
742                          */
743                         idle_cnt_percent = CPUDRV_IDLE_CNT_PERCENT(
744                             cpudrv_idle_hwm, speeds, i);
745                         idle_cnt_percent = max(idle_cnt_percent,
746                             (cpudrv_idle_lwm + cpudrv_idle_buf_zone));
747                         cur_spd->idle_hwm =
748                             (idle_cnt_percent * cur_spd->quant_cnt) / 100;
749                         cur_spd->idle_lwm =
750                             (cpudrv_idle_lwm * cur_spd->quant_cnt) / 100;

752                         /*
753                          * The lwm for user threads are determined such that
754                          * if CPU slows down, the load of work in the
755                          * new speed would still keep the CPU at or below the
```

```
                         * user_hwm in the new speed.  This is to prevent
                         * the quick jump back up to higher speed.
                         */
                        cur_spd->user_hwm = (cpudrv_user_hwm *
                            cur_spd->quant_cnt) / 100;
                        user_cnt_percent = CPUDRV_USER_CNT_PERCENT(
                            cpudrv_user_hwm, speeds, i);
                        prev_spd->user_lwm =
                            (user_cnt_percent * prev_spd->quant_cnt) / 100;
                }
                prev_spd = cur_spd;
        }
        /* Slowest speed. Can't slow down anymore */
        cur_spd->idle_hwm = UINT_MAX;
        cur_spd->user_lwm = -1;
#ifdef  DEBUG
        DPRINTF(D_PM_INIT, ("cpudrv_init: instance %d: head_spd spd %d, "
            "num_spd %d\n", ddi_get_instance(cpudsp->dip),
            cpupm->head_spd->speed, cpupm->num_spd));
        for (cur_spd = cpupm->head_spd; cur_spd; cur_spd = cur_spd->down_spd) {
                DPRINTF(D_PM_INIT, ("cpudrv_init: instance %d: speed %d, "
                    "down_spd spd %d, idle_hwm %d, user_lwm %d, "
                    "up_spd spd %d, idle_lwm %d, user_hwm %d, "
                    "quant_cnt %d\n", ddi_get_instance(cpudsp->dip),
                    cur_spd->speed,
                    (cur_spd->down_spd ? cur_spd->down_spd->speed : 0),
                    cur_spd->idle_hwm, cur_spd->user_lwm,
                    (cur_spd->up_spd ? cur_spd->up_spd->speed : 0),
                    cur_spd->idle_lwm, cur_spd->user_hwm,
                    cur_spd->quant_cnt));
        }
#endif  /* DEBUG */
        CPUDRV_FREE_SPEEDS(speeds, nspeeds);
        return (DDI_SUCCESS);
}

/*
 * Free CPU power management data.
 */
static void
cpudrv_free(cpudrv_devstate_t *cpudsp)
{
        cpudrv_pm_t     *cpupm = &(cpudsp->cpudrv_pm);
        cpudrv_pm_spd_t *cur_spd, *next_spd;

        cur_spd = cpupm->head_spd;
        while (cur_spd) {
                next_spd = cur_spd->down_spd;
                kmem_free(cur_spd, sizeof (cpudrv_pm_spd_t));
                cur_spd = next_spd;
        }
        bzero(cpupm, sizeof (cpudrv_pm_t));
}

/*
 * Create pm-components property.
 */
static int
cpudrv_comp_create(cpudrv_devstate_t *cpudsp)
{
        cpudrv_pm_t     *cpupm = &(cpudsp->cpudrv_pm);
        cpudrv_pm_spd_t *cur_spd;
        char            **pmc;
        int             size;
        char            name[] = "NAME=CPU Speed";
        int             i, j;
```



```
        uint_t          comp_spd;
        int             result = DDI_FAILURE;

        pmc = kmem_zalloc((cpupm->num_spd + 1) * sizeof (char *), KM_SLEEP);
        size = CPUDRV_COMP_SIZE();
        if (cpupm->num_spd > CPUDRV_COMP_MAX_VAL) {
                cmn_err(CE_WARN, "cpudrv_comp_create: instance %d: "
                    "number of speeds exceeded limits",
                    ddi_get_instance(cpudsp->dip));
                kmem_free(pmc, (cpupm->num_spd + 1) * sizeof (char *));
                return (result);
        }

        for (i = cpupm->num_spd, cur_spd = cpupm->head_spd; i > 0;
            i--, cur_spd = cur_spd->down_spd) {
                cur_spd->pm_level = i;
                pmc[i] = kmem_zalloc((size * sizeof (char)), KM_SLEEP);
                comp_spd = CPUDRV_COMP_SPEED(cpupm, cur_spd);
                if (comp_spd > CPUDRV_COMP_MAX_VAL) {
                        cmn_err(CE_WARN, "cpudrv_comp_create: "
                            "instance %d: speed exceeded limits",
                            ddi_get_instance(cpudsp->dip));
                        for (j = cpupm->num_spd; j >= i; j--) {
                                kmem_free(pmc[j], size * sizeof (char));
                        }
                        kmem_free(pmc, (cpupm->num_spd + 1) *
                            sizeof (char *));
                        return (result);
                }
                CPUDRV_COMP_SPRINT(pmc[i], cpupm, cur_spd, comp_spd)
                DPRINTF(D_PM_COMP_CREATE, ("cpudrv_comp_create: "
                    "instance %d: pm-components power level %d string '%s'\n",
                    ddi_get_instance(cpudsp->dip), i, pmc[i]));
        }
        pmc[0] = kmem_zalloc(sizeof (name), KM_SLEEP);
        (void) strcat(pmc[0], name);
        DPRINTF(D_PM_COMP_CREATE, ("cpudrv_comp_create: instance %d: "
            "pm-components component name '%s'\n",
            ddi_get_instance(cpudsp->dip), pmc[0]));

        if (ddi_prop_update_string_array(DDI_DEV_T_NONE, cpudsp->dip,
            "pm-components", pmc, cpupm->num_spd + 1) == DDI_PROP_SUCCESS) {
                result = DDI_SUCCESS;
        } else {
                cmn_err(CE_WARN, "cpudrv_comp_create: instance %d: "
                    "can't create pm-components property",
                    ddi_get_instance(cpudsp->dip));
        }

        for (i = cpupm->num_spd; i > 0; i--) {
                kmem_free(pmc[i], size * sizeof (char));
        }
        kmem_free(pmc[0], sizeof (name));
        kmem_free(pmc, (cpupm->num_spd + 1) * sizeof (char *));
        return (result);
}

/*
 * Mark a component idle.
 */
#define CPUDRV_MONITOR_PM_IDLE_COMP(dip, cpupm) { \
        if ((cpupm)->pm_busycnt >= 1) { \
                if (pm_idle_component((dip), CPUDRV_COMP_NUM) == \
                    DDI_SUCCESS) { \
                        DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: " \
                            "instance %d: pm_idle_component called\n", \
```

```
888                             ddi_get_instance((dip)))); \
889                         (cpupm)->pm_busycnt--; \
890                 } else { \
891                         cmn_err(CE_WARN, "cpudrv_monitor: instance %d: " \
892                             "can't idle CPU component", \
893                             ddi_get_instance((dip))); \
894                 } \
895         } \
896 }

898 /*
899  * Marks a component busy in both PM framework and driver state structure.
900  */
901 #define CPUDRV_MONITOR_PM_BUSY_COMP(dip, cpupm) { \
902         if ((cpupm)->pm_busycnt < 1) { \
903                 if (pm_busy_component((dip), CPUDRV_COMP_NUM) == \
904                     DDI_SUCCESS) { \
905                         DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: " \
906                             "instance %d: pm_busy_component called\n", \
907                             ddi_get_instance((dip)))); \
908                         (cpupm)->pm_busycnt++; \
909                 } else { \
910                         cmn_err(CE_WARN, "cpudrv_monitor: instance %d: " \
911                             "can't busy CPU component", \
912                             ddi_get_instance((dip))); \
913                 } \
914         } \
915 }

917 /*
918  * Marks a component busy and calls pm_raise_power().
919  */
920 #define CPUDRV_MONITOR_PM_BUSY_AND_RAISE(dip, cpudsp, cpupm, new_spd) { \
921         int ret; \
922         /* \
923          * Mark driver and PM framework busy first so framework doesn't try \
924          * to bring CPU to lower speed when we need to be at higher speed. \
925          */ \
926         CPUDRV_MONITOR_PM_BUSY_COMP((dip), (cpupm)); \
927         mutex_exit(&(cpudsp)->lock); \
928         DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: instance %d: " \
929             "pm_raise_power called to %d\n", ddi_get_instance((dip)), \
930                 (new_spd->pm_level))); \
931         ret = pm_raise_power((dip), CPUDRV_COMP_NUM, (new_spd->pm_level)); \
932         if (ret != DDI_SUCCESS) { \
933                 cmn_err(CE_WARN, "cpudrv_monitor: instance %d: can't " \
934                     "raise CPU power level", ddi_get_instance((dip))); \
935         } \
936         mutex_enter(&(cpudsp)->lock); \
937         if (ret == DDI_SUCCESS && cpudsp->cpudrv_pm.cur_spd == NULL) { \
938                 cpudsp->cpudrv_pm.cur_spd = new_spd; \
939         } \
940 }

942 /*
943  * In order to monitor a CPU, we need to hold cpu_lock to access CPU
944  * statistics. Holding cpu_lock is not allowed from a callout routine.
945  * We dispatch a taskq to do that job.
946  */
947 static void
948 cpudrv_monitor_disp(void *arg)
949 {
950         cpudrv_devstate_t       *cpudsp = (cpudrv_devstate_t *)arg;

952         /*
953          * We are here because the last task has scheduled a timeout.
```



```
954          * The queue should be empty at this time.
955          */
956         mutex_enter(&cpudsp->cpudrv_pm.timeout_lock);
957         if ((ddi_taskq_dispatch(cpudsp->cpudrv_pm.tq, cpudrv_monitor, arg,
958             DDI_NOSLEEP)) != DDI_SUCCESS) {
959                 mutex_exit(&cpudsp->cpudrv_pm.timeout_lock);
960                 DPRINTF(D_PM_MONITOR, ("cpudrv_monitor_disp: failed to "
961                     "dispatch the cpudrv_monitor taskq\n"));
962                 mutex_enter(&cpudsp->lock);
963                 CPUDRV_MONITOR_INIT(cpudsp);
964                 mutex_exit(&cpudsp->lock);
965                 return;
966         }
967         cpudsp->cpudrv_pm.timeout_count++;
968         mutex_exit(&cpudsp->cpudrv_pm.timeout_lock);
969 }

971 /*
972  * Monitors each CPU for the amount of time idle thread was running in the
973  * last quantum and arranges for the CPU to go to the lower or higher speed.
974  * Called at the time interval appropriate for the current speed. The
975  * time interval for normal speed is CPUDRV_QUANT_CNT_NORMAL. The time
976  * interval for other speeds (including unknown speed) is
977  * CPUDRV_QUANT_CNT_OTHR.
978  */
979 static void
980 cpudrv_monitor(void *arg)
981 {
982         cpudrv_devstate_t       *cpudsp = (cpudrv_devstate_t *)arg;
983         cpudrv_pm_t             *cpupm;
984         cpudrv_pm_spd_t         *cur_spd, *new_spd;
985         dev_info_t              *dip;
986         uint_t                  idle_cnt, user_cnt, system_cnt;
987         clock_t                 ticks;
988         uint_t                  tick_cnt;
989         hrtime_t                msnsecs[NCMSTATES];
990         boolean_t               is_ready;

992 #define GET_CPU_MSTATE_CNT(state, cnt) \
993         msnsecs[state] = NSEC_TO_TICK(msnsecs[state]); \
994         if (cpupm->lastquan_mstate[state] > msnsecs[state]) \
995                 msnsecs[state] = cpupm->lastquan_mstate[state]; \
996         cnt = msnsecs[state] - cpupm->lastquan_mstate[state]; \
997         cpupm->lastquan_mstate[state] = msnsecs[state]

999         /*
1000         * We're not ready until we can  get a cpu_t
1001         */
1002        is_ready = (cpudrv_get_cpu(cpudsp) == DDI_SUCCESS);

1004        mutex_enter(&cpudsp->lock);
1005        cpupm = &(cpudsp->cpudrv_pm);
1006        if (cpupm->timeout_id == 0) {
1007                mutex_exit(&cpudsp->lock);
1008                goto do_return;
1009        }
1010        cur_spd = cpupm->cur_spd;
1011        dip = cpudsp->dip;

1013        /*
1014         * We assume that a CPU is initialized and has a valid cpu_t
1015         * structure, if it is ready for cross calls. If this changes,
1016         * additional checks might be needed.
1017         *
1018         * Additionally, for x86 platforms we cannot power manage an
1019         * instance, until it has been initialized.
```

```
1020          */
1021         if (is_ready) {
1022                 is_ready = CPUDRV_XCALL_IS_READY(cpudsp->cpu_id);
1023                 if (!is_ready) {
1024                         DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: instance %d: "
1025                             "CPU not ready for x-calls\n",
1026                             ddi_get_instance(dip)));
1027                 } else if (!(is_ready = cpudrv_power_ready(cpudsp->cp))) {
1028                         DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: instance %d: "
1029                             "waiting for all CPUs to be power manageable\n",
1030                             ddi_get_instance(dip)));
1031                 }
1032         }
1033         if (!is_ready) {
1034                 /*
1035                  * Make sure that we are busy so that framework doesn't
1036                  * try to bring us down in this situation.
1037                  */
1038                 CPUDRV_MONITOR_PM_BUSY_COMP(dip, cpupm);
1039                 CPUDRV_MONITOR_INIT(cpudsp);
1040                 mutex_exit(&cpudsp->lock);
1041                 goto do_return;
1042         }
1043
1044         /*
1045          * Make sure that we are still not at unknown power level.
1046          */
1047         if (cur_spd == NULL) {
1048                 DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: instance %d: "
1049                     "cur_spd is unknown\n", ddi_get_instance(dip)));
1050                 CPUDRV_MONITOR_PM_BUSY_AND_RAISE(dip, cpudsp, cpupm,
1051                     CPUDRV_TOPSPEED(cpupm));
1052                 /*
1053                  * We just changed the speed. Wait till at least next
1054                  * call to this routine before proceeding ahead.
1055                  */
1056                 CPUDRV_MONITOR_INIT(cpudsp);
1057                 mutex_exit(&cpudsp->lock);
1058                 goto do_return;
1059         }
1060
1061         if (!cpupm->pm_started) {
1062                 cpupm->pm_started = B_TRUE;
1063                 cpudrv_set_supp_freqs(cpudsp);
1064         }
1065
1066         get_cpu_mstate(cpudsp->cp, msnsecs);
1067         GET_CPU_MSTATE_CNT(CMS_IDLE, idle_cnt);
1068         GET_CPU_MSTATE_CNT(CMS_USER, user_cnt);
1069         GET_CPU_MSTATE_CNT(CMS_SYSTEM, system_cnt);
1070
1071         /*
1072          * We can't do anything when we have just switched to a state
1073          * because there is no valid timestamp.
1074          */
1075         if (cpupm->lastquan_ticks == 0) {
1076                 cpupm->lastquan_ticks = NSEC_TO_TICK(gethrtime());
1077                 CPUDRV_MONITOR_INIT(cpudsp);
1078                 mutex_exit(&cpudsp->lock);
1079                 goto do_return;
1080         }
1081
1082         /*
1083          * Various watermarks are based on this routine being called back
1084          * exactly at the requested period. This is not guaranteed
1085          * because this routine is called from a taskq that is dispatched
```



```
1086          * from a timeout routine.  Handle this by finding out how many
1087          * ticks have elapsed since the last call and adjusting
1088          * the idle_cnt based on the delay added to the requested period
1089          * by timeout and taskq.
1090          */
1091         ticks = NSEC_TO_TICK(gethrtime());
1092         tick_cnt = ticks - cpupm->lastquan_ticks;
1093         ASSERT(tick_cnt != 0);
1094         cpupm->lastquan_ticks = ticks;
1095
1096         /*
1097          * Time taken between recording the current counts and
1098          * arranging the next call of this routine is an error in our
1099          * calculation. We minimize the error by calling
1100          * CPUDRV_MONITOR_INIT() here instead of end of this routine.
1101          */
1102         CPUDRV_MONITOR_INIT(cpudsp);
1103         DPRINTF(D_PM_MONITOR_VERBOSE, ("cpudrv_monitor: instance %d: "
1104             "idle count %d, user count %d, system count %d, pm_level %d, "
1105             "pm_busycnt %d\n", ddi_get_instance(dip), idle_cnt, user_cnt,
1106             system_cnt, cur_spd->pm_level, cpupm->pm_busycnt));
1107
1108 #ifdef  DEBUG
1109         /*
1110          * Notify that timeout and taskq has caused delays and we need to
1111          * scale our parameters accordingly.
1112          *
1113          * To get accurate result, don't turn on other DPRINTFs with
1114          * the following DPRINTF. PROM calls generated by other
1115          * DPRINTFs changes the timing.
1116          */
1117         if (tick_cnt > cur_spd->quant_cnt) {
1118                 DPRINTF(D_PM_MONITOR_DELAY, ("cpudrv_monitor: instance %d: "
1119                     "tick count %d > quantum_count %u\n",
1120                     ddi_get_instance(dip), tick_cnt, cur_spd->quant_cnt));
1121         }
1122 #endif  /* DEBUG */
1123
1124         /*
1125          * Adjust counts based on the delay added by timeout and taskq.
1126          */
1127         idle_cnt = (idle_cnt * cur_spd->quant_cnt) / tick_cnt;
1128         user_cnt = (user_cnt * cur_spd->quant_cnt) / tick_cnt;
1129
1130         if ((user_cnt > cur_spd->user_hwm) || (idle_cnt < cur_spd->idle_lwm &&
1131             cur_spd->idle_blwm_cnt >= cpudrv_idle_blwm_cnt_max)) {
1132                 cur_spd->idle_blwm_cnt = 0;
1133                 cur_spd->idle_bhwm_cnt = 0;
1134                 /*
1135                  * In normal situation, arrange to go to next higher speed.
1136                  * If we are running in special direct pm mode, we just stay
1137                  * at the current speed.
1138                  */
1139                 if (cur_spd == cur_spd->up_spd || cpudrv_direct_pm) {
1140                         CPUDRV_MONITOR_PM_BUSY_COMP(dip, cpupm);
1141                 } else {
1142                         new_spd = cur_spd->up_spd;
1143                         CPUDRV_MONITOR_PM_BUSY_AND_RAISE(dip, cpudsp, cpupm,
1144                             new_spd);
1145                 }
1146         } else if ((user_cnt <= cur_spd->user_lwm) &&
1147             (idle_cnt >= cur_spd->idle_hwm) || !CPU_ACTIVE(cpudsp->cp)) {
1148                 cur_spd->idle_blwm_cnt = 0;
1149                 cur_spd->idle_bhwm_cnt = 0;
1150                 /*
1151                  * Arrange to go to next lower speed by informing our idle
```

```
1152                  * status to the power management framework.
1153                  */
1154                 CPUDRV_MONITOR_PM_IDLE_COMP(dip, cpupm);
1155         } else {
1156                 /*
1157                  * If we are between the idle water marks and have not
1158                  * been here enough consecutive times to be considered
1159                  * busy, just increment the count and return.
1160                  */
1161                 if ((idle_cnt < cur_spd->idle_hwm) &&
1162                     (idle_cnt >= cur_spd->idle_lwm) &&
1163                     (cur_spd->idle_bhwm_cnt < cpudrv_idle_bhwm_cnt_max)) {
1164                         cur_spd->idle_blwm_cnt = 0;
1165                         cur_spd->idle_bhwm_cnt++;
1166                         mutex_exit(&cpudsp->lock);
1167                         goto do_return;
1168                 }
1169                 if (idle_cnt < cur_spd->idle_lwm) {
1170                         cur_spd->idle_blwm_cnt++;
1171                         cur_spd->idle_bhwm_cnt = 0;
1172                 }
1173                 /*
1174                  * Arranges to stay at the current speed.
1175                  */
1176                 CPUDRV_MONITOR_PM_BUSY_COMP(dip, cpupm);
1177         }
1178         mutex_exit(&cpudsp->lock);
1179 do_return:
1180         mutex_enter(&cpupm->timeout_lock);
1181         ASSERT(cpupm->timeout_count > 0);
1182         cpupm->timeout_count--;
1183         cv_signal(&cpupm->timeout_cv);
1184         mutex_exit(&cpupm->timeout_lock);
1185 }

1187 /*
1188  * get cpu_t structure for cpudrv_devstate_t
1189  */
1190 int
1191 cpudrv_get_cpu(cpudrv_devstate_t *cpudsp)
1192 {
1193         ASSERT(cpudsp != NULL);

1195         /*
1196          * return DDI_SUCCESS if cpudrv_devstate_t
1197          * already contains cpu_t structure
1198          */
1199         if (cpudsp->cp != NULL)
1200                 return (DDI_SUCCESS);

1202         if (MUTEX_HELD(&cpu_lock)) {
1203                 cpudsp->cp = cpu_get(cpudsp->cpu_id);
1204         } else {
1205                 mutex_enter(&cpu_lock);
1206                 cpudsp->cp = cpu_get(cpudsp->cpu_id);
1207                 mutex_exit(&cpu_lock);
1208         }

1210         if (cpudsp->cp == NULL)
1211                 return (DDI_FAILURE);

1213         return (DDI_SUCCESS);
1214 }
```