```
**********************************************************
   36500 Fri Jan 15 13:15:33 2016
new/usr/src/uts/common/io/cpudrv.c
XXXX cpudrv attach error handling is leaky
XXXX cpudrv attach is racy
**********************************************************
______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 #endif /* ! codereview */
 290                 mutex_init(&cpudsp->lock, NULL, MUTEX_DRIVER, NULL);
 280                 if (cpudrv_is_enabled(cpudsp)) {
 291                 if (cpudrv_init(cpudsp) != DDI_SUCCESS) {
```



```
 292                         cpudrv_enabled = B_FALSE;
 293                         cpudrv_free(cpudsp);
 294                         ddi_soft_state_free(cpudrv_state, instance);
 295                         return (DDI_FAILURE);
 296                 }
 297                 if (cpudrv_comp_create(cpudsp) != DDI_SUCCESS) {
 298                         cpudrv_enabled = B_FALSE;
 299                         cpudrv_free(cpudsp);
 300                         ddi_soft_state_free(cpudrv_state, instance);
 301                         return (DDI_FAILURE);
 302                 }
 303                 if (ddi_prop_update_string(DDI_DEV_T_NONE,
 304                     dip, "pm-class", "CPU") != DDI_PROP_SUCCESS) {
 305                         cpudrv_enabled = B_FALSE;
 306                         cpudrv_free(cpudsp);
 307                         ddi_soft_state_free(cpudrv_state, instance);
 308                         return (DDI_FAILURE);
 309                 }

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

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

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

 308                 }

 355                 if (!cpudrv_mach_init(cpudsp)) {
```

```
356                         cmn_err(CE_WARN, "cpudrv_attach: instance %d: "
357                             "cpudrv_mach_init failed", instance);
358                         cpudrv_enabled = B_FALSE;
359                         ddi_taskq_destroy(cpudsp->cpudrv_pm.tq);
360 #endif /* ! codereview */
361                         cpudrv_free(cpudsp);
362                         ddi_soft_state_free(cpudrv_state, instance);
363                         return (DDI_FAILURE);
364                 }

366                 CPUDRV_INSTALL_MAX_CHANGE_HANDLER(cpudsp);

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

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

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

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

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

398         default:
399                 return (DDI_FAILURE);
400         }
401 }

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

413         instance = ddi_get_instance(dip);

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

420 #if defined(__x86)
421                 cpudsp = ddi_get_soft_state(cpudrv_state, instance);
```



```
422                 ASSERT(cpudsp != NULL);

424                 /*
425                  * Nothing to do for detach, if no doing active PM.
426                  */
427                 if (!cpudrv_is_enabled(cpudsp))
428                         return (DDI_SUCCESS);

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

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

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

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

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

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

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

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

```
489                 DPRINTF(D_DETACH, ("cpudrv_detach: instance %d: DDI_SUSPEND - "
490                     "cur_spd %d, topspeed %d\n", instance,
491                     cpupm->cur_spd->pm_level,
492                     CPUDRV_TOPSPEED(cpupm)->pm_level));
493
494                 CPUDRV_MONITOR_FINI(cpudsp);
495
496                 if (!cpudrv_direct_pm && (cpupm->cur_spd !=
497                     CPUDRV_TOPSPEED(cpupm))) {
498                         if (cpupm->pm_busycnt < 1) {
499                                 if ((pm_busy_component(dip, CPUDRV_COMP_NUM)
500                                     == DDI_SUCCESS)) {
501                                         cpupm->pm_busycnt++;
502                                 } else {
503                                         CPUDRV_MONITOR_INIT(cpudsp);
504                                         mutex_exit(&cpudsp->lock);
505                                         cmn_err(CE_WARN, "cpudrv_detach: "
506                                             "instance %d: can't busy CPU "
507                                             "component", instance);
508                                         return (DDI_FAILURE);
509                                 }
510                         }
511                         mutex_exit(&cpudsp->lock);
512                         if (pm_raise_power(dip, CPUDRV_COMP_NUM,
513                             CPUDRV_TOPSPEED(cpupm)->pm_level) !=
514                             DDI_SUCCESS) {
515                                 mutex_enter(&cpudsp->lock);
516                                 CPUDRV_MONITOR_INIT(cpudsp);
517                                 mutex_exit(&cpudsp->lock);
518                                 cmn_err(CE_WARN, "cpudrv_detach: instance %d: "
519                                     "can't raise CPU power level to %d",
520                                     instance,
521                                     CPUDRV_TOPSPEED(cpupm)->pm_level);
522                                 return (DDI_FAILURE);
523                         } else {
524                                 return (DDI_SUCCESS);
525                         }
526                 } else {
527                         mutex_exit(&cpudsp->lock);
528                         return (DDI_SUCCESS);
529                 }
530
531         default:
532                 return (DDI_FAILURE);
533         }
534 }
535
536 /*
537  * Driver power(9e) entry point.
538  *
539  * Driver's notion of current power is set *only* in power(9e) entry point
540  * after actual power change operation has been successfully completed.
541  */
542 /* ARGSUSED */
543 static int
544 cpudrv_power(dev_info_t *dip, int comp, int level)
545 {
546         int                     instance;
547         cpudrv_devstate_t       *cpudsp;
548         cpudrv_pm_t             *cpudrvpm;
549         cpudrv_pm_spd_t         *new_spd;
550         boolean_t               is_ready;
551         int                     ret;
552
553         instance = ddi_get_instance(dip);
```



```
554
555         DPRINTF(D_POWER, ("cpudrv_power: instance %d: level %d\n",
556             instance, level));
557
558         if ((cpudsp = ddi_get_soft_state(cpudrv_state, instance)) == NULL) {
559                 cmn_err(CE_WARN, "cpudrv_power: instance %d: can't "
560                     "get state", instance);
561                 return (DDI_FAILURE);
562         }
563
564         /*
565          * We're not ready until we can  get a cpu_t
566          */
567         is_ready = (cpudrv_get_cpu(cpudsp) == DDI_SUCCESS);
568
569         mutex_enter(&cpudsp->lock);
570         cpudrvpm = &(cpudsp->cpudrv_pm);
571
572         /*
573          * In normal operation, we fail if we are busy and request is
574          * to lower the power level. We let this go through if the driver
575          * is in special direct pm mode. On x86, we also let this through
576          * if the change is due to a request to govern the max speed.
577          */
578         if (!cpudrv_direct_pm && (cpudrvpm->pm_busycnt >= 1) &&
579             !cpudrv_is_governor_thread(cpudrvpm)) {
580                 if ((cpudrvpm->cur_spd != NULL) &&
581                     (level < cpudrvpm->cur_spd->pm_level)) {
582                         mutex_exit(&cpudsp->lock);
583                         return (DDI_FAILURE);
584                 }
585         }
586
587         for (new_spd = cpudrvpm->head_spd; new_spd; new_spd =
588             new_spd->down_spd) {
589                 if (new_spd->pm_level == level)
590                         break;
591         }
592         if (!new_spd) {
593                 CPUDRV_RESET_GOVERNOR_THREAD(cpudrvpm);
594                 mutex_exit(&cpudsp->lock);
595                 cmn_err(CE_WARN, "cpudrv_power: instance %d: "
596                     "can't locate new CPU speed", instance);
597                 return (DDI_FAILURE);
598         }
599
600         /*
601          * We currently refuse to power manage if the CPU is not ready to
602          * take cross calls (cross calls fail silently if CPU is not ready
603          * for it).
604          *
605          * Additionally, for x86 platforms we cannot power manage an instance,
606          * until it has been initialized.
607          */
608         if (is_ready) {
609                 is_ready = CPUDRV_XCALL_IS_READY(cpudsp->cpu_id);
610                 if (!is_ready) {
611                         DPRINTF(D_POWER, ("cpudrv_power: instance %d: "
612                             "CPU not ready for x-calls\n", instance));
613                 } else if (!(is_ready = cpudrv_power_ready(cpudsp->cp))) {
614                         DPRINTF(D_POWER, ("cpudrv_power: instance %d: "
615                             "waiting for all CPUs to be power manageable\n",
616                             instance));
617                 }
618         }
619         if (!is_ready) {
```

```
620                 CPUDRV_RESET_GOVERNOR_THREAD(cpudrvpm);
621                 mutex_exit(&cpudsp->lock);
622                 return (DDI_FAILURE);
623         }

625         /*
626          * Execute CPU specific routine on the requested CPU to
627          * change its speed to normal-speed/divisor.
628          */
629         if ((ret = cpudrv_change_speed(cpudsp, new_spd)) != DDI_SUCCESS) {
630                 cmn_err(CE_WARN, "cpudrv_power: "
631                     "cpudrv_change_speed() return = %d", ret);
632                 mutex_exit(&cpudsp->lock);
633                 return (DDI_FAILURE);
634         }

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

662         return (DDI_SUCCESS);
663 }

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

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



```
686         cpupm->num_spd = nspeeds;

688         /*
689          * Calculate the watermarks and other parameters based on the
690          * supplied speeds.
691          *
692          * One of the basic assumption is that for X amount of CPU work,
693          * if CPU is slowed down by a factor of N, the time it takes to
694          * do the same work will be N * X.
695          *
696          * The driver declares that a CPU is idle and ready for slowed down,
697          * if amount of idle thread is more than the current speed idle_hwm
698          * without dropping below idle_hwm a number of consecutive sampling
699          * intervals and number of running threads in user mode are below
700          * user_lwm.  We want to set the current user_lwm such that if we
701          * just switched to the next slower speed with no change in real work
702          * load, the amount of user threads at the slower speed will be such
703          * that it falls below the slower speed's user_hwm.  If we didn't do
704          * that then we will just come back to the higher speed as soon as we
705          * go down even with no change in work load.
706          * The user_hwm is a fixed precentage and not calculated dynamically.
707          *
708          * We bring the CPU up if idle thread at current speed is less than
709          * the current speed idle_lwm for a number of consecutive sampling
710          * intervals or user threads are above the user_hwm for the current
711          * speed.
712          */
713         for (i = 0; i < nspeeds; i++) {
714                 cur_spd = kmem_zalloc(sizeof (cpudrv_pm_spd_t), KM_SLEEP);
715                 cur_spd->speed = speeds[i];
716                 if (i == 0) {   /* normal speed */
717                         cpupm->head_spd = cur_spd;
718                         CPUDRV_TOPSPEED(cpupm) = cur_spd;
719                         cur_spd->quant_cnt = CPUDRV_QUANT_CNT_NORMAL;
720                         cur_spd->idle_hwm =
721                             (cpudrv_idle_hwm * cur_spd->quant_cnt) / 100;
722                         /* can't speed anymore */
723                         cur_spd->idle_lwm = 0;
724                         cur_spd->user_hwm = UINT_MAX;
725                 } else {
726                         cur_spd->quant_cnt = CPUDRV_QUANT_CNT_OTHR;
727                         ASSERT(prev_spd != NULL);
728                         prev_spd->down_spd = cur_spd;
729                         cur_spd->up_spd = cpupm->head_spd;

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

```
753                         /*
754                          * The lwm for user threads are determined such that
755                          * if CPU slows down, the load of work in the
756                          * new speed would still keep the CPU at or below the
757                          * user_hwm in the new speed.  This is to prevent
758                          * the quick jump back up to higher speed.
759                          */
760                         cur_spd->user_hwm = (cpudrv_user_hwm *
761                             cur_spd->quant_cnt) / 100;
762                         user_cnt_percent = CPUDRV_USER_CNT_PERCENT(
763                             cpudrv_user_hwm, speeds, i);
764                         prev_spd->user_lwm =
765                             (user_cnt_percent * prev_spd->quant_cnt) / 100;
766                 }
767                 prev_spd = cur_spd;
768         }
769         /* Slowest speed. Can't slow down anymore */
770         cur_spd->idle_hwm = UINT_MAX;
771         cur_spd->user_lwm = -1;
772 #ifdef  DEBUG
773         DPRINTF(D_PM_INIT, ("cpudrv_init: instance %d: head_spd spd %d, "
774             "num_spd %d\n", ddi_get_instance(cpudsp->dip),
775             cpupm->head_spd->speed, cpupm->num_spd));
776         for (cur_spd = cpupm->head_spd; cur_spd; cur_spd = cur_spd->down_spd) {
777                 DPRINTF(D_PM_INIT, ("cpudrv_init: instance %d: speed %d, "
778                     "down_spd spd %d, idle_hwm %d, user_lwm %d, "
779                     "up_spd spd %d, idle_lwm %d, user_hwm %d, "
780                     "quant_cnt %d\n", ddi_get_instance(cpudsp->dip),
781                     cur_spd->speed,
782                     (cur_spd->down_spd ? cur_spd->down_spd->speed : 0),
783                     cur_spd->idle_hwm, cur_spd->user_lwm,
784                     (cur_spd->up_spd ? cur_spd->up_spd->speed : 0),
785                     cur_spd->idle_lwm, cur_spd->user_hwm,
786                     cur_spd->quant_cnt));
787         }
788 #endif  /* DEBUG */
789         CPUDRV_FREE_SPEEDS(speeds, nspeeds);
790         return (DDI_SUCCESS);
791 }

793 /*
794  * Free CPU power management data.
795  */
796 static void
797 cpudrv_free(cpudrv_devstate_t *cpudsp)
798 {
799         cpudrv_pm_t     *cpupm = &(cpudsp->cpudrv_pm);
800         cpudrv_pm_spd_t *cur_spd, *next_spd;

802         cur_spd = cpupm->head_spd;
803         while (cur_spd) {
804                 next_spd = cur_spd->down_spd;
805                 kmem_free(cur_spd, sizeof (cpudrv_pm_spd_t));
806                 cur_spd = next_spd;
807         }
808         bzero(cpupm, sizeof (cpudrv_pm_t));
809 }

811 /*
812  * Create pm-components property.
813  */
814 static int
815 cpudrv_comp_create(cpudrv_devstate_t *cpudsp)
816 {
817         cpudrv_pm_t     *cpupm = &(cpudsp->cpudrv_pm);
```



```
818         cpudrv_pm_spd_t *cur_spd;
819         char            **pmc;
820         int             size;
821         char            name[] = "NAME=CPU Speed";
822         int             i, j;
823         uint_t          comp_spd;
824         int             result = DDI_FAILURE;

826         pmc = kmem_zalloc((cpupm->num_spd + 1) * sizeof (char *), KM_SLEEP);
827         size = CPUDRV_COMP_SIZE();
828         if (cpupm->num_spd > CPUDRV_COMP_MAX_VAL) {
829                 cmn_err(CE_WARN, "cpudrv_comp_create: instance %d: "
830                     "number of speeds exceeded limits",
831                     ddi_get_instance(cpudsp->dip));
832                 kmem_free(pmc, (cpupm->num_spd + 1) * sizeof (char *));
833                 return (result);
834         }

836         for (i = cpupm->num_spd, cur_spd = cpupm->head_spd; i > 0;
837             i--, cur_spd = cur_spd->down_spd) {
838                 cur_spd->pm_level = i;
839                 pmc[i] = kmem_zalloc((size * sizeof (char)), KM_SLEEP);
840                 comp_spd = CPUDRV_COMP_SPEED(cpupm, cur_spd);
841                 if (comp_spd > CPUDRV_COMP_MAX_VAL) {
842                         cmn_err(CE_WARN, "cpudrv_comp_create: "
843                             "instance %d: speed exceeded limits",
844                             ddi_get_instance(cpudsp->dip));
845                         for (j = cpupm->num_spd; j >= i; j--) {
846                                 kmem_free(pmc[j], size * sizeof (char));
847                         }
848                         kmem_free(pmc, (cpupm->num_spd + 1) *
849                             sizeof (char *));
850                         return (result);
851                 }
852                 CPUDRV_COMP_SPRINT(pmc[i], cpupm, cur_spd, comp_spd)
853                 DPRINTF(D_PM_COMP_CREATE, ("cpudrv_comp_create: "
854                     "instance %d: pm-components power level %d string '%s'\n",
855                     ddi_get_instance(cpudsp->dip), i, pmc[i]));
856         }
857         pmc[0] = kmem_zalloc(sizeof (name), KM_SLEEP);
858         (void) strcat(pmc[0], name);
859         DPRINTF(D_PM_COMP_CREATE, ("cpudrv_comp_create: instance %d: "
860             "pm-components component name '%s'\n",
861             ddi_get_instance(cpudsp->dip), pmc[0]));

863         if (ddi_prop_update_string_array(DDI_DEV_T_NONE, cpudsp->dip,
864             "pm-components", pmc, cpupm->num_spd + 1) == DDI_PROP_SUCCESS) {
865                 result = DDI_SUCCESS;
866         } else {
867                 cmn_err(CE_WARN, "cpudrv_comp_create: instance %d: "
868                     "can't create pm-components property",
869                     ddi_get_instance(cpudsp->dip));
870         }

872         for (i = cpupm->num_spd; i > 0; i--) {
873                 kmem_free(pmc[i], size * sizeof (char));
874         }
875         kmem_free(pmc[0], sizeof (name));
876         kmem_free(pmc, (cpupm->num_spd + 1) * sizeof (char *));
877         return (result);
878 }

880 /*
881  * Mark a component idle.
882  */
883 #define CPUDRV_MONITOR_PM_IDLE_COMP(dip, cpupm) { \
```

```
884         if ((cpupm)->pm_busycnt >= 1) { \
885                 if (pm_idle_component((dip), CPUDRV_COMP_NUM) == \
886                     DDI_SUCCESS) { \
887                         DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: " \
888                             "instance %d: pm_idle_component called\n", \
889                             ddi_get_instance((dip)))); \
890                         (cpupm)->pm_busycnt--; \
891                 } else { \
892                         cmn_err(CE_WARN, "cpudrv_monitor: instance %d: " \
893                             "can't idle CPU component", \
894                             ddi_get_instance((dip))); \
895                 } \
896         } \
897 }

899 /*
900  * Marks a component busy in both PM framework and driver state structure.
901  */
902 #define CPUDRV_MONITOR_PM_BUSY_COMP(dip, cpupm) { \
903         if ((cpupm)->pm_busycnt < 1) { \
904                 if (pm_busy_component((dip), CPUDRV_COMP_NUM) == \
905                     DDI_SUCCESS) { \
906                         DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: " \
907                             "instance %d: pm_busy_component called\n", \
908                             ddi_get_instance((dip)))); \
909                         (cpupm)->pm_busycnt++; \
910                 } else { \
911                         cmn_err(CE_WARN, "cpudrv_monitor: instance %d: " \
912                             "can't busy CPU component", \
913                             ddi_get_instance((dip))); \
914                 } \
915         } \
916 }

918 /*
919  * Marks a component busy and calls pm_raise_power().
920  */
921 #define CPUDRV_MONITOR_PM_BUSY_AND_RAISE(dip, cpudsp, cpupm, new_spd) { \
922         int ret; \
923         /* \
924          * Mark driver and PM framework busy first so framework doesn't try \
925          * to bring CPU to lower speed when we need to be at higher speed. \
926          */ \
927         CPUDRV_MONITOR_PM_BUSY_COMP((dip), (cpupm)); \
928         mutex_exit(&(cpudsp)->lock); \
929         DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: instance %d: " \
930             "pm_raise_power called to %d\n", ddi_get_instance((dip)), \
931                 (new_spd->pm_level))); \
932         ret = pm_raise_power((dip), CPUDRV_COMP_NUM, (new_spd->pm_level)); \
933         if (ret != DDI_SUCCESS) { \
934                 cmn_err(CE_WARN, "cpudrv_monitor: instance %d: can't " \
935                     "raise CPU power level", ddi_get_instance((dip))); \
936         } \
937         mutex_enter(&(cpudsp)->lock); \
938         if (ret == DDI_SUCCESS && cpudsp->cpudrv_pm.cur_spd == NULL) { \
939                 cpudsp->cpudrv_pm.cur_spd = new_spd; \
940         } \
941 }

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



```
950 {
951         cpudrv_devstate_t       *cpudsp = (cpudrv_devstate_t *)arg;

953         /*
954          * We are here because the last task has scheduled a timeout.
955          * The queue should be empty at this time.
956          */
957         mutex_enter(&cpudsp->cpudrv_pm.timeout_lock);
958         if ((ddi_taskq_dispatch(cpudsp->cpudrv_pm.tq, cpudrv_monitor, arg,
959             DDI_NOSLEEP)) != DDI_SUCCESS) {
960                 mutex_exit(&cpudsp->cpudrv_pm.timeout_lock);
961                 DPRINTF(D_PM_MONITOR, ("cpudrv_monitor_disp: failed to "
962                     "dispatch the cpudrv_monitor taskq\n"));
963                 mutex_enter(&cpudsp->lock);
964                 CPUDRV_MONITOR_INIT(cpudsp);
965                 mutex_exit(&cpudsp->lock);
966                 return;
967         }
968         cpudsp->cpudrv_pm.timeout_count++;
969         mutex_exit(&cpudsp->cpudrv_pm.timeout_lock);
970 }

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

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

1000        /*
1001         * We're not ready until we can  get a cpu_t
1002         */
1003        is_ready = (cpudrv_get_cpu(cpudsp) == DDI_SUCCESS);

1005        mutex_enter(&cpudsp->lock);
1006        cpupm = &(cpudsp->cpudrv_pm);
1007        if (cpupm->timeout_id == 0) {
1008                mutex_exit(&cpudsp->lock);
1009                goto do_return;
1010        }
1011        cur_spd = cpupm->cur_spd;
1012        dip = cpudsp->dip;

1014        /*
1015         * We assume that a CPU is initialized and has a valid cpu_t
```

```
1016          * structure, if it is ready for cross calls. If this changes,
1017          * additional checks might be needed.
1018          *
1019          * Additionally, for x86 platforms we cannot power manage an
1020          * instance, until it has been initialized.
1021          */
1022         if (is_ready) {
1023                 is_ready = CPUDRV_XCALL_IS_READY(cpudsp->cpu_id);
1024                 if (!is_ready) {
1025                         DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: instance %d: "
1026                             "CPU not ready for x-calls\n",
1027                             ddi_get_instance(dip)));
1028                 } else if (!(is_ready = cpudrv_power_ready(cpudsp->cp))) {
1029                         DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: instance %d: "
1030                             "waiting for all CPUs to be power manageable\n",
1031                             ddi_get_instance(dip)));
1032                 }
1033         }
1034         if (!is_ready) {
1035                 /*
1036                  * Make sure that we are busy so that framework doesn't
1037                  * try to bring us down in this situation.
1038                  */
1039                 CPUDRV_MONITOR_PM_BUSY_COMP(dip, cpupm);
1040                 CPUDRV_MONITOR_INIT(cpudsp);
1041                 mutex_exit(&cpudsp->lock);
1042                 goto do_return;
1043         }

1045         /*
1046          * Make sure that we are still not at unknown power level.
1047          */
1048         if (cur_spd == NULL) {
1049                 DPRINTF(D_PM_MONITOR, ("cpudrv_monitor: instance %d: "
1050                     "cur_spd is unknown\n", ddi_get_instance(dip)));
1051                 CPUDRV_MONITOR_PM_BUSY_AND_RAISE(dip, cpudsp, cpupm,
1052                     CPUDRV_TOPSPEED(cpupm));
1053                 /*
1054                  * We just changed the speed. Wait till at least next
1055                  * call to this routine before proceeding ahead.
1056                  */
1057                 CPUDRV_MONITOR_INIT(cpudsp);
1058                 mutex_exit(&cpudsp->lock);
1059                 goto do_return;
1060         }

1062         if (!cpupm->pm_started) {
1063                 cpupm->pm_started = B_TRUE;
1064                 cpudrv_set_supp_freqs(cpudsp);
1065         }

1067         get_cpu_mstate(cpudsp->cp, msnsecs);
1068         GET_CPU_MSTATE_CNT(CMS_IDLE, idle_cnt);
1069         GET_CPU_MSTATE_CNT(CMS_USER, user_cnt);
1070         GET_CPU_MSTATE_CNT(CMS_SYSTEM, system_cnt);

1072         /*
1073          * We can't do anything when we have just switched to a state
1074          * because there is no valid timestamp.
1075          */
1076         if (cpupm->lastquan_ticks == 0) {
1077                 cpupm->lastquan_ticks = NSEC_TO_TICK(gethrtime());
1078                 CPUDRV_MONITOR_INIT(cpudsp);
1079                 mutex_exit(&cpudsp->lock);
1080                 goto do_return;
1081         }
```



```
1083         /*
1084          * Various watermarks are based on this routine being called back
1085          * exactly at the requested period. This is not guaranteed
1086          * because this routine is called from a taskq that is dispatched
1087          * from a timeout routine.  Handle this by finding out how many
1088          * ticks have elapsed since the last call and adjusting
1089          * the idle_cnt based on the delay added to the requested period
1090          * by timeout and taskq.
1091          */
1092         ticks = NSEC_TO_TICK(gethrtime());
1093         tick_cnt = ticks - cpupm->lastquan_ticks;
1094         ASSERT(tick_cnt != 0);
1095         cpupm->lastquan_ticks = ticks;

1097         /*
1098          * Time taken between recording the current counts and
1099          * arranging the next call of this routine is an error in our
1100          * calculation. We minimize the error by calling
1101          * CPUDRV_MONITOR_INIT() here instead of end of this routine.
1102          */
1103         CPUDRV_MONITOR_INIT(cpudsp);
1104         DPRINTF(D_PM_MONITOR_VERBOSE, ("cpudrv_monitor: instance %d: "
1105             "idle count %d, user count %d, system count %d, pm_level %d, "
1106             "pm_busycnt %d\n", ddi_get_instance(dip), idle_cnt, user_cnt,
1107             system_cnt, cur_spd->pm_level, cpupm->pm_busycnt));

1109 #ifdef  DEBUG
1110         /*
1111          * Notify that timeout and taskq has caused delays and we need to
1112          * scale our parameters accordingly.
1113          *
1114          * To get accurate result, don't turn on other DPRINTFs with
1115          * the following DPRINTF. PROM calls generated by other
1116          * DPRINTFs changes the timing.
1117          */
1118         if (tick_cnt > cur_spd->quant_cnt) {
1119                 DPRINTF(D_PM_MONITOR_DELAY, ("cpudrv_monitor: instance %d: "
1120                     "tick count %d > quantum_count %u\n",
1121                     ddi_get_instance(dip), tick_cnt, cur_spd->quant_cnt));
1122         }
1123 #endif  /* DEBUG */

1125         /*
1126          * Adjust counts based on the delay added by timeout and taskq.
1127          */
1128         idle_cnt = (idle_cnt * cur_spd->quant_cnt) / tick_cnt;
1129         user_cnt = (user_cnt * cur_spd->quant_cnt) / tick_cnt;

1131         if ((user_cnt > cur_spd->user_hwm) || (idle_cnt < cur_spd->idle_lwm &&
1132             cur_spd->idle_blwm_cnt >= cpudrv_idle_blwm_cnt_max)) {
1133                 cur_spd->idle_blwm_cnt = 0;
1134                 cur_spd->idle_bhwm_cnt = 0;
1135                 /*
1136                  * In normal situation, arrange to go to next higher speed.
1137                  * If we are running in special direct pm mode, we just stay
1138                  * at the current speed.
1139                  */
1140                 if (cur_spd == cur_spd->up_spd || cpudrv_direct_pm) {
1141                         CPUDRV_MONITOR_PM_BUSY_COMP(dip, cpupm);
1142                 } else {
1143                         new_spd = cur_spd->up_spd;
1144                         CPUDRV_MONITOR_PM_BUSY_AND_RAISE(dip, cpudsp, cpupm,
1145                             new_spd);
1146                 }
1147         } else if ((user_cnt <= cur_spd->user_lwm) &&
```

```
1148             (idle_cnt >= cur_spd->idle_hwm) || !CPU_ACTIVE(cpudsp->cp)) {
1149                 cur_spd->idle_blwm_cnt = 0;
1150                 cur_spd->idle_bhwm_cnt = 0;
1151                 /*
1152                  * Arrange to go to next lower speed by informing our idle
1153                  * status to the power management framework.
1154                  */
1155                 CPUDRV_MONITOR_PM_IDLE_COMP(dip, cpupm);
1156         } else {
1157                 /*
1158                  * If we are between the idle water marks and have not
1159                  * been here enough consecutive times to be considered
1160                  * busy, just increment the count and return.
1161                  */
1162                 if ((idle_cnt < cur_spd->idle_hwm) &&
1163                     (idle_cnt >= cur_spd->idle_lwm) &&
1164                     (cur_spd->idle_bhwm_cnt < cpudrv_idle_bhwm_cnt_max)) {
1165                         cur_spd->idle_blwm_cnt = 0;
1166                         cur_spd->idle_bhwm_cnt++;
1167                         mutex_exit(&cpudsp->lock);
1168                         goto do_return;
1169                 }
1170                 if (idle_cnt < cur_spd->idle_lwm) {
1171                         cur_spd->idle_blwm_cnt++;
1172                         cur_spd->idle_bhwm_cnt = 0;
1173                 }
1174                 /*
1175                  * Arranges to stay at the current speed.
1176                  */
1177                 CPUDRV_MONITOR_PM_BUSY_COMP(dip, cpupm);
1178         }
1179         mutex_exit(&cpudsp->lock);
1180 do_return:
1181         mutex_enter(&cpupm->timeout_lock);
1182         ASSERT(cpupm->timeout_count > 0);
1183         cpupm->timeout_count--;
1184         cv_signal(&cpupm->timeout_cv);
1185         mutex_exit(&cpupm->timeout_lock);
1186 }

1188 /*
1189  * get cpu_t structure for cpudrv_devstate_t
1190  */
1191 int
1192 cpudrv_get_cpu(cpudrv_devstate_t *cpudsp)
1193 {
1194         ASSERT(cpudsp != NULL);

1196         /*
1197          * return DDI_SUCCESS if cpudrv_devstate_t
1198          * already contains cpu_t structure
1199          */
1200         if (cpudsp->cp != NULL)
1201                 return (DDI_SUCCESS);

1203         if (MUTEX_HELD(&cpu_lock)) {
1204                 cpudsp->cp = cpu_get(cpudsp->cpu_id);
1205         } else {
1206                 mutex_enter(&cpu_lock);
1207                 cpudsp->cp = cpu_get(cpudsp->cpu_id);
1208                 mutex_exit(&cpu_lock);
1209         }

1211         if (cpudsp->cp == NULL)
1212                 return (DDI_FAILURE);
```



```
1214         return (DDI_SUCCESS);
1215 }
```