99

```
/*
 *  arch/i386/kernel/mxt.c
 *
 *  Memory eXpansion Technology (MXT) support module.
 *
 *  MXT is a hardware for doubling the effective size
 *  of the Linux memory: 256MB becomes 512MB, 1GB becomes 2GB, etc...
 *  MXT is transparent to all hardware and software including the
 *  Linux kernel, drivers, apps, peripherals etc etc.
 *
 *  MXT is implemented in ServerWorks Inc. Pinnacle memory controller
 *  chipset.  This modules manages the MXT memory.
 *  For detailed documentation see
 *  http://oss.software.ibm.com/developerworks/opensource/mxt
 *
 *  Copyright (C) 1999,2000,2001 IBM <mxt@us.ibm.com>
 *
 *
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  version 2, or later. See /usr/src/linux/COPYING for more detail.
 *  This program is distributed WITHOUT any warranty, merchantability
 *  or fitness for a particular purpose. See /usr/src/linux/COPYING
 *  for licensing details
 *
 *  Developers are welcome to comment and contribute to this project.
 *  Please send email to <mxt@us.ibm.com> or visit the project web site
 *  http://oss.software.ibm.com/developerworks/opensource/mxt
 *
 *  Authors:
 *  Bulent Abali <abali@us.ibm.com>
 *  Hubertus Franke <frankeh@us.ibm.com>
 */

/*
 *  generic compression routines are usually prefixed with cmp_
 *  MXT hardware specific routines are usually prefixed with mxt_
 *
 *  Contents:
 *  ---------
 *  Section 1.  Performance counter support
 *  Section 2.  /proc/sys/mxt support
 *  Section 3.  Compression hardware specific routines
 *  Section 4.  MXT fast page operations
 *  Section 5.  Interrupt handling
 *  Section 6.  Generic memory compression support routines
 *  Section 7.  CPU grabbers help reduce compression pressure
 *  Section 8.  Timer routine calls compression manager periodically
 *  Section 9.  Page clearing threads reduce compression pressure
 *  Section 10. Scheduling priority of big memory tasks reduced
 *  Section 11. Kernel pages backed in the compressed memory to cover
 *              the worst case compressibility.
 *  Section 100. Module load/unload
 *
 */

#include <linux/config.h>
#include <linux/version.h>
#include <linux/module.h>
#include <linux/init.h>

#include <asm/uaccess.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/sysctl.h>
#include <linux/proc_fs.h>
#include <linux/swap.h>
#include <asm/mxt.h>
#include <asm/system.h>
#include <linux/list.h>
#include <linux/spinlock.h>
#include <linux/threads.h>
#include <linux/interrupt.h>

#define _ABS(x)   (((x)>0)?(x):-(x))
#define _MIN(x,y) (((x)>(y))?(y):(x))
#define _MAX(x,y) (((x)>(y))?(x):(y))

#define CONFIG_MXT_HIDDEN_PAGES

extern int nr_inactive_pages;  /* mm/page_alloc.c */

static int pages_min;  /*  pages_min sum in all zones; page_alloc.c */
static int pages_low;
static int pages_high;

/* extern unsigned long totalram_pages; in linux/arch/i386/mm/init.c */
static unsigned long totalram_pages;
/* extern unsigned long totalhigh_pages; in linux/arch/i386/mm/init.c */
static unsigned long totalhigh_pages;
```

```
91  extern void si_meminfo(struct sysinfo *val);
92
93  /*
94   * release < acquire < danger < int < panic is necessary
95   * Resetting max periodically to zero will not hurt.
96   * It tells you the highest utilization you have reached since
97   * boot time or last writing of max with 0.
98   */
99  static struct _thresholds {
100         unsigned long max;       /* maximum we have ever reached */
101         unsigned long release;   /* start releasing pages */
102         unsigned long acquire;   /* recall pages left in system */
103         unsigned long danger;    /* do not return any pages */
104         unsigned long intr;       /* interrupt notification occurs */
105         unsigned long panic;     /* panic() */
106 } mc_th;
107
108 unsigned long max_lockedpages;
109 long swap_reserve=0;
110 long kernel_reserve=0;
111 unsigned long mxt_printk=0; /* != 0 prints debug information */
112
113 /* __alloc_clear uses if(1) mxt_clear_page() else
114    if(0) clear_highpage() */
115 static int fclr = 0;
116
117 /* number of set-aside pages needed  */
118 volatile long nr_rsrv_pages = 0;
119 volatile long nr_pages_per_thread = 0;
120
121 /* and currently held by eater threads  */
122 static atomic_t total_pages_held  __cacheline_aligned = { 0 };
123 DECLARE_WAIT_QUEUE_HEAD(cmp_eatmem_wait);
124
125 /* pages hidden to back kernel pages in physical memory */
126 atomic_t nr_hide_pages __cacheline_aligned = { 0 };
127 atomic_t nr_hidden_pages __cacheline_aligned = { 0 };
128 DECLARE_WAIT_QUEUE_HEAD(cmp_hide_pages_wait);
129
130 static inline unsigned long get_memutil(void);
131
132 /* cmp_idle threads */
133 static int  cmp_idle(void *);
134 static void cmp_grab_cpus(void);
135 static void cmp_release_cpus(void);
136 DECLARE_WAIT_QUEUE_HEAD(cmp_idle_wait);
137
138 /* main routine which determines whether grab pages or not */
139 int memcompress_check(void);
140
141 /* reclaim and zero thread */
142 static void wake_up_cmp_eatmem(void);
143 static inline long mc_decayed_release(long pages);
144
145 /* statistics */
146 static struct _mxt_stats {
147         volatile unsigned long interrupt;
148         volatile unsigned long state;
149         volatile unsigned long bumped;
150         volatile unsigned long held;
151         volatile unsigned long esr;
152         volatile unsigned long esr_sticky;
153 } mxt_stats;
154
155 /* tuning knobs */
156 static struct _mxt_tuning {
157         long period;       /* Poll period in jiffies; min=1 max=HZ;
158                               Do not set it outside the range.
159                               1 slowest polling, HZ fastest polling */
160         long high_decay;   /* highest memory release rate(KB per period)*/
161         long low_decay;    /* lowest memory release rate (KB per period */
162         long delay;        /* wait in seconds before releasing pages */
163         long bump_rate;    /* arm the bumper if physmem util is
164                               increasing faster than this per period */
165         long spin_period;  /* spin period in jiffies */
166 } tuning = {
167         10,
168         1000,                /* e.g. 1000 KB per 100 ms */
169         100,                 /* e.g. 100 KB per 100 ms   */
170         10,                  /* 10 seconds */
171         30000,               /* e.g. 20,000 KB per 100 ms */
172         5                    /* e.g. 5 jiffies or 50 milliseconds */
173 };
174
175 /* ---------------------------------------------------------------- */
176
177 /*
178  *  Real and Physical memory usage exposed by /proc/sys/mxt/cmpmemi
179  *  See Documentation/mxt.txt for how to use
180  */
```

```
struct cmpmem_info_t cmpmem_info;

/* ---------------------- Section 1 ------------------------------ */

/*
 *  Pinnacle chip performance counter support
 *  Performance counters exposed from /proc/sys/mxt/cmpperf
 *  See Documentation/mxt.txt for how to use
 */
struct cmpperf_info_t perf_counters;
unsigned long perf_counters_request;  /* ICR mode */
int perf_counters_state;              /* started or stopped */

static unsigned long prev_ira;
static unsigned long prev_irb;

/*
 *  Reads counters from hardware and copies to soft registers
 */
inline void update_from_perf_counters(void)
{
        unsigned long n;

        n = READ_CTRL(IRA);
        perf_counters.ira += n;

        n = READ_CTRL(IRB);
        perf_counters.irb += n;

        /* following creates two 64 bit soft registers */
        perf_counters.irax += (perf_counters.ira < prev_ira ) ? 1 : 0;
        prev_ira = perf_counters.ira;
        perf_counters.irbx += (perf_counters.irb < prev_irb ) ? 1 : 0;
        prev_irb = perf_counters.irb;

        /* IRB consists of two 16−bit halves for some ICR modes. */
        /* The following may be meaningless in those modes       */
        /* but we will do and display the calculation in         */
        /* the file /proc/sys/mxt/cmpperf anyway                 */

        perf_counters.irbl += (0xffff & n);
        perf_counters.irbh += (0xffff & (n>>16));
}

/*
 *  This is called periodically to read hardware counters and copy
 *  them to 64−bit soft registers.  It also processes
 *  user commands entered from /proc/sys/mxt/icr (start|stop counting)
 */
inline void perf_counters_check(void)
{
        unsigned long n;

        if ( perf_counters_request ) { /* start request */

                if ( perf_counters_state == 0 ) { /* stopped state */

                        /* reset the counters first */
                        WRITE_CTRL( ICR, 0 );
                        /* Reading while ICR=0 should clear the reg */
                        n = READ_CTRL( IRA );
                        n = READ_CTRL( IRB );
                        WRITE_CTRL( IRA, 0 );
                        WRITE_CTRL( IRB, 0 );

                        /* clear the soft registers */
                        perf_counters.mode=perf_counters_request;
                        perf_counters.ira=0;
                        perf_counters.irb=0;
                        perf_counters.irbl=0;
                        perf_counters.irbh=0;
                        perf_counters.irax =0;
                        perf_counters.irbx = 0;
                        prev_ira=0;
                        prev_irb=0;

                        /* put hardware in counting mode */
                        WRITE_CTRL( ICR, perf_counters_request );
                        perf_counters_state = 1;

                } else { /* in started state */
                        update_from_perf_counters();
                }

        } else {  /* stop request */

                if (  perf_counters_state  ) { /* started state */

                        /* reset the counters first */
                        WRITE_CTRL( ICR, 0 );
```

```
271                         /* Reading while ICR=0 should clear the reg */
272                         n = READ_CTRL( IRA );
273                         n = READ_CTRL( IRB );
274                         WRITE_CTRL( IRA, 0 );
275                         WRITE_CTRL( IRB, 0 );
276
277                         /* switch to stopped state */
278                         perf_counters_state = 0;
279
280                         /* clear the soft registers */
281                         perf_counters.mode=0;
282                         perf_counters.ira=0;
283                         perf_counters.irb=0;
284                         perf_counters.irbl=0;
285                         perf_counters.irbh=0;
286                         perf_counters.irax=0;
287                         perf_counters.irbx=0;
288                         prev_ira=0;
289                         prev_irb=0;
290                 }
291         }
292 }
293
294 /* ---------------------- Section 2 ------------------------------ */
295
296 /*
297  *   /proc/sys/mxt support
298  */
299 enum {
300         MXT_ICR=1,
301         MXT_SWAP_RSRV=7,     /* amount of phys mem reserved in swap sp */
302         MXT_KERN_RSRV=8,     /* amount of phys mem reserved in kern sp */
303         MXT_DEBUG=9,         /* turn on/off repeating printk() */
304         MXT_CMPPERF=10,
305         MXT_CMPMEMI=11,
306         MXT_STATS=12,
307         MXT_THRESHOLDS=13,
308         MXT_TUNING=14,
309         MXT_FCLR=15
310 };
311
312 void get_cmpmem_info(struct cmpmem_info_t *si)
313 {
314         si->memfree  = nr_free_pages();
315         si->usedmem  = get_used_pages();
316         si->physused = katsina_phys_used();
317         si->util     = get_memutil();
318         si->cmphold  = nr_rsrv_pages;
319         si->cmpheld  = atomic_read(&total_pages_held);
320         si->cmphide  = atomic_read(&nr_hide_pages);
321         si->cmphidden  = atomic_read(&nr_hidden_pages);
322 }
323
324 /*
325  * I need a wrapper because the requested data requires making
326  * few function calls as shown in get_cmpmem_info()
327  */
328 static int wrap1_proc_dointvec(ctl_table *table,
329                                int write, struct file *filp,
330                                void *buffer, size_t *lenp)
331 {
332         if(table->data==&cmpmem_info && table->maxlen==sizeof(cmpmem_info))
333                 get_cmpmem_info(&cmpmem_info);
334
335         return proc_dointvec(table,write,filp,buffer,lenp);
336 }
337
338 /*
339  * A user writing to /proc/sys/mxt/thresholds
340  * must cause a write to the actual hardware register SUTLR.
341  * This wrapper performs the task needed.
342  */
343 static int wrap2_proc_dointvec(ctl_table *table,
344                                int write, struct file *filp,
345                                void *buffer, size_t *lenp)
346 {
347         int status = proc_dointvec(table,write,filp,buffer,lenp);
348
349         /* at this point memory copy of mc_th.intr
350            has been updated by user */
351
352         if ( !write )
353                 return status;
354
355         if ( mc_th.intr > 1000 ) {
356                 panic("<MC>: illegal value in /proc/sys/mxt/thresholds");
357                 return status;
358         }
359
360         if ( !status && table->data==&mc_th ) {
```

```
361                     /* this will write that value to the
362                        hardware register */
363                     unsigned long th = memutil_to_sectors( mc_th.intr );
364                     WRITE_CTRL(SUTLR, th);
365                     printk("<MC>: Set SUTLR=%08lx\n", th);
366             }
367             else {
368                     printk("<MC>: error: /proc/sys/mxt/thresholds\n");
369             }
370
371             return status;
372     }
373
374     /*
375      * I need a wrapper because the requested data requires making
376      * few function calls
377      *
378      * CAVEAT: you must periodically read /proc/sys/mxt/cmpperf
379      * so that the hardware counters on the memory controller are copied
380      * to the 64−bit soft registers.  Otherwise 32−bit hardware counters may
381      * roll over and you will get incorrect readings.
382      * Likewise if you set /proc/sys/mxt/icr you must read this file once so
383      * that the value you wrote in icr is copied to the actual hardware
384      * register
385      */
386     static int wrap3_proc_dointvec(ctl_table *table,
387                                    int write, struct file *filp,
388                                    void *buffer, size_t *lenp)
389     {
390             if(table−>data==&perf_counters && !write &&
391                table−>maxlen==sizeof(perf_counters))
392                     perf_counters_check();
393
394             return proc_dointvec(table,write,filp,buffer,lenp);
395     }
396
397
398     static struct ctl_table_header *mxt_table_header;
399
400     static ctl_table mxt_table[] = {
401             { MXT_ICR, "icr",
402               &perf_counters_request, sizeof(perf_counters_request),
403               0644, NULL, &proc_dointvec },
404             { MXT_THRESHOLDS, "thresholds",
405               &mc_th, sizeof(mc_th),
406               0644, NULL, &wrap2_proc_dointvec },
407             { MXT_SWAP_RSRV, "swap_rsrv",
408               &swap_reserve, sizeof(swap_reserve),
409               0644, NULL, &proc_dointvec },
410             { MXT_KERN_RSRV, "kernel_rsrv",
411               &kernel_reserve, sizeof(kernel_reserve),
412               0644, NULL, &proc_dointvec },
413             { MXT_DEBUG, "debug",
414               &mxt_printk, sizeof(mxt_printk),
415               0644, NULL, &proc_dointvec },
416             { MXT_CMPPERF, "cmpperf",
417               &perf_counters, sizeof(perf_counters),
418               0444,NULL, &wrap3_proc_dointvec },
419             { MXT_CMPMEMI, "cmpmemi",
420               &cmpmem_info, sizeof(cmpmem_info),
421               0444,NULL, &wrap1_proc_dointvec },
422             { MXT_STATS, "stats",
423               &mxt_stats, sizeof(mxt_stats),
424               0644,NULL, &proc_dointvec },
425             { MXT_TUNING, "tuning",
426               &tuning, sizeof(tuning),
427               0644,NULL, &proc_dointvec },
428             { MXT_FCLR, "fclr",
429               &fclr, sizeof(fclr),
430               0644,NULL, &proc_dointvec },
431             {0}
432     };
433
434     static ctl_table mxt_root_table[] = {
435             {254, "mxt", NULL, 0, 0555, mxt_table},
436             {0}
437     };
438
439     void mxt_register_sysctl(void)
440     {
441             mxt_table_header = register_sysctl_table(mxt_root_table, 0);
442     }
443
444     void mxt_unregister_sysctl(void)
445     {
446             unregister_sysctl_table(mxt_table_header);
447     }
448
449     /* −−−−−−−−−−−−−−−−−−−− Section 3 −−−−−−−−−−−−−−−−−−−−−−−−−− */
450
```

```
/*
 *   Compression hardware specific stuff
 */

#include <linux/mm.h>
#include <linux/version.h>
#include <linux/highmem.h>
#include <linux/swap.h>
#include <linux/swapctl.h>
#include <linux/smp_lock.h>
#include <linux/tqueue.h>
#include <asm/page.h>
#include <asm/fixmap.h>
#include <asm/mxt.h>
#include <asm/uaccess.h>
// #include <asm/spinlock.h>
#include <asm/processor.h>
#include <asm/io.h>
#include <linux/pci.h>
#include <linux/init.h>

int mxt_bios_found;
int mxt_memory_expanded;
int mxt_compression_enabled;
int mxt_device_found;
int mxt_irq;

/*
 * compression ratio becomes 1:1 we may need to swap pages out.
 * Thus, swap_reserve is the amount reserved in the
 * swap space for compression purposes. Swap_reserve must be equal to the
 * committed but missing amount of memory, which is the real minus physical
 * amount of memory.  For example
 * we told kernel that it has 1 GB of memory but infact it has only 512MB
 * in the box. Since real=2*physical, the swap_reserve amount must be equal
 * to the physical amount of memory (that is amount of installed memory in
 * the box)  This value may be overridden from /proc/sys/mxt.
 *
 * Your total vm size is roughly ram size + swap size − swap_reserve
 * One side effect of swap_reserve is that if you turn off swap or do not
 * have large enough swap processes cannot allocate more memory than
 * ram size − swap_reserve.
 *
 * When swap_reserve is 0, and if the process pages becomes incompressible
 * and there is not swap space left, then the process will be killed.
 */

/*  num_physpages is a Linux kernel variable.
 *  It is equal to the amount of "real memory" in MXT terminology
 *  e.g. If you have 512MB worth of DIMMs then real memory will be 1GB.
 *  cmp_num_physpages is the hardware "physical memory"
 *  available in the system.  Real is roughly two times physical due to
 *  compression.
 */
unsigned long cmp_num_physpages=0;

static unsigned long L3_size=0;
static spinlock_t fast_page_lock;
void * katsina_vaddr;
extern long mxt_bios_table[16];

/* address offsets of various fields in the mxt_bios_table
   copied from EBDA; sizes are in units of 4Kbytes */
enum {
        _magic_=0,
        _version_=0x4,
        _nr_real_,
        _nr_inhibit_,
        _nr_interrupt_,
        _mmio_base_,
        _L3_size_=0xc,
        _mxt_enabled_=0xc,
        _physical_size_=0x10,
        _real_start_=0x14
};

struct _mxtbios_addr {
        unsigned long address;
        unsigned long len;
};

#define address_region(nr,base) \
((struct _mxtbios_addr *)((nr)*sizeof(struct _mxtbios_addr)+(char *)(base)))

/*
 *  arch/i386/kernel/setup.c::setup_mxt_memory() function had
 *  copied the MXT bios data from EBDA to mxt_bios_table.
 *  Now we scan this table to determine hardware characteristics.
 */
static int mxt_scan_bios(void)
```

```
{
        unsigned long *bp = mxt_bios_table;
        unsigned char *p;
        int nr_real;
        int nr_inhibit;
        int i;

        printk("mxt_scan_bios: enter\n");

        if( ! mxt_bios_table[0] )
                return −ENODEV;

        mxt_bios_found = 1;

        p = (unsigned char *)bp;

        /* PIC mode interrupt nr */
        mxt_irq = (unsigned int) p[_nr_interrupt_];

        /* real and inhibit regions of the memory */
        nr_real = p[_nr_real_];
        nr_inhibit = p[_nr_inhibit_];

        if ( *((unsigned long *)(p+_L3_size_)) & 0x80000000 ) {
                mxt_compression_enabled = 1;
                printk("<MC> Compression is enabled\n");
        }
        else {
                mxt_compression_enabled = 0;
                printk("<MC> Compression is disabled\n");
        }

        if (nr_real) {
                /* E820 reported physical mem size (1X) but
                 * additional memory was added to address
                 * space (in linux/arch/i386/kernel/setup.c)
                 */
                mxt_memory_expanded = 1;
        }
        else {
                /* No real mem regions were reported */
                mxt_memory_expanded = 0;
        }

        cmp_num_physpages = *((unsigned long *) (p+_physical_size_));
        /* take out the compression inhibited region(s),
           typically the lower 1MB
        */
        for(i=0; i<nr_inhibit; i++)
                cmp_num_physpages −=
                        address_region(0+nr_real, p+_real_start_)->len;
        printk("<MC> cmp_num_physpages: 0x%08lx\n", cmp_num_physpages);

        /* This is for the testing guy who want to locks his
           pages mm/mlock.c */
        if ( cmp_num_physpages )
                max_lockedpages = cmp_num_physpages/2;

        /* Parse the L3 cache size */
        L3_size = *((unsigned long *)(p+_L3_size_)) & 0x1fffff;
        printk("<MC> L3 pages: 0x%08lx\n", L3_size);

        return 0;
}

/*
 * − make the memory controller registers addressable.
 * − calculate swap_reserve amount.
 * − calculate compression management thresholds.
 */
static int mxt_setup_vm(void)
{
        unsigned long L3_fraction;
        struct sysinfo val;

        /* map the memory mapped registers of the chip
         * to kernel virtual space
         */
        katsina_vaddr = ioremap_nocache( KATSINA_PHYS, KATSINA_LENGTH );
        if( ! katsina_vaddr )
                panic("<MC> couldn't virtual map MXT chip!");

        printk("<MC>: MXT chip virtual address: 0x%08lx\n",
               (unsigned long) katsina_vaddr);

        si_meminfo(&val);
        totalram_pages = val.totalram;
        totalhigh_pages = val.totalhigh;

        /* Reserve swap space for MXT purposes.
```

```
631             * swap_reserve and kernel_reserve will be in units of pages
632             * (note: this was in units of kilobytes in 2.2 kernel support
633             * patch.) Due to uncompressed
634             * 1MB region I am reserving more than necessary, but 1MB of disk
635             * space is small enough on that I will keep it like this.
636             */
637            if ( mxt_compression_enabled &&
638                 totalram_pages > cmp_num_physpages ) {
639    #ifdef SWAP_RESERVATION
640                    swap_reserve = totalram_pages − cmp_num_physpages;
641    #else
642                    swap_reserve = 0;
643    #endif
644                    if ( swap_reserve < 0 )
645                            swap_reserve = 0;
646
647    #ifdef MXT_KERNEL_RESERVATION
648                    /* Undef'ing this section makes kernel_reserve=0 the default.
649                       In the Beta 2 kernel, changes in the alloc_page() API
650                       busted the add_hidden_pages() call which I fixed.
651                       To double the safety, I force kernel_reserve=0 here.
652                    */
653
654                    /* this is the overcommitted memory, i.e. missing amount */
655                    kernel_reserve = totalram_pages − cmp_num_physpages;
656
657                    /* relax the restrictions to account for the memory
658                       already allocated or reserved by kernel at boot time
659                    */
660                    kernel_reserve −= (val.totalram − val.totalhigh) −
661                            (val.freeram − val.freehigh);
662                    kernel_reserve = _MIN(kernel_reserve,
663                                          val.freeram − val.freehigh );
664                    if ( kernel_reserve < 0 )
665                            kernel_reserve = 0;
666    #endif
667
668                    printk("<MC>: %ld pages reserved in swap space\n",
669                           swap_reserve);
670                    printk("<MC>: %ld pages may be hidden for kernel pages\n",
671                           kernel_reserve);
672                    printk("<MC>: %ld compressed physical pages\n",
673                           cmp_num_physpages);
674                    printk("<MC>: %ld totalram_pages\n",
675                           totalram_pages);
676                    printk("<MC>: %ld totalhigh_pages\n",
677                           totalhigh_pages);
678            }
679
680            /* +1 for truncating to the highest integer */
681            L3_fraction = ( (1000 * L3_size)/cmp_num_physpages )+1;
682
683            /* Setup physical memory pressure thresholds.
684             * This is mostly guess work but conservative.
685             * We know that we do not want physical to exceed
686             * (1000−L3_fraction)/1000 utilization.
687             * We add some fat to cover worst possible expansion.
688             */
689            if ( mxt_compression_enabled  ) {
690                    unsigned long th;
691                    /* th is about 4MB physical; +1 is needed
692                     * for very large memory so that th!=0 */
693                    th = ((1000 * 1024) / cmp_num_physpages) + 1;
694
695                    /* dont use less than 1 percent */
696                    if (th < 10) th = 10;
697
698                    mc_th.intr    = 1000 − L3_fraction − th;
699                    mc_th.danger  = mc_th.intr − 2*th;
700                    mc_th.acquire = mc_th.danger − th;
701                    // mc_th.release = mc_th.acquire − th;
702                    // this is a temp workaround for kswapd burning too much cpu
703                    mc_th.release = mc_th.acquire − 1;
704                    mc_th.panic   = MC_TH_PANIC;
705            }
706            else {
707                    mc_th.panic = mc_th.intr = mc_th.danger =
708                            mc_th.acquire = mc_th.release = 1000;
709            }
710            return 0;
711    }
712
713    static void mxt_cleanup_vm(void)
714    {
715            iounmap(katsina_vaddr);
716    }
717
718    /*
719     *  Reads hardware registers and returns physical memory utilization
720     *  in fraction of 1000
```

```
 */
unsigned long katsina_memutil(void)
{
        unsigned long sectors;
        unsigned long numpages;
        unsigned long util;

        /* return some dummy number so that people's
           divisions don't fail */
        if ( ! mxt_compression_enabled )
                return 1000;

        sectors  = READ_CTRL(SUR);
        numpages = sectors / (PAGE_SIZE/SECTOR_SIZE);

        /* the MXT box has 64GB max real address space; doing
           this arithmetic with 32-bit integers is tricky:
           There are max 16 million pages. Dividing by 8
           ensures that (1000 * num) < 2**32
        */
        util = ( 1000 *  (numpages/8) ) / (cmp_num_physpages/8);

        if ( !util )
                util = 1;

        return ( (unsigned long) util );
}

/*
 *   util is between 0 and 1000
 */
unsigned long memutil_to_sectors(unsigned long util)
{
        return (  ( util * (cmp_num_physpages/1000) )
                  * (PAGE_SIZE/SECTOR_SIZE) );
}

static unsigned long memutil_to_pages(unsigned long util)
{
        return ( util * (cmp_num_physpages/1000) );
}

/*
 *  Returns physical memory utilization in units of page
 */
unsigned long katsina_phys_used(void)
{
        long sectors;
        long numpages;

        /* return some dummy number so that divisions don't fail */
        if ( ! mxt_compression_enabled )
                return cmp_num_physpages;

        sectors  = READ_CTRL(SUR);
        numpages = sectors / (PAGE_SIZE/SECTOR_SIZE);

        return ((unsigned long)numpages);
}

/* ---------------------- Section 4 ------------------------------ */

/*
 * Fast page operation is a method for manipulating pages using the
 * Katsina chip "Fast Page Operation".  These merely update pointers
 * in the chip instead of doing 4KB bulk transfers.
 * These clear a page about 10 times faster than processor can.
 * See CNB 3.0 spec RMPR, RMPCR register pairs.
 *
 * FIXME: if compression is disabled or this is not an MXT box
 * this function will silently return.  It will not do what it is
 * expected to do.  For this reason, we will not export it for
 * others to use this nifty function.  We should put an emulator
 * in so that it performs its functions regardless if
 * this is MXT hardware or not.
 */

/* there is a coherency bug for zero page op in pass-3 chips
 * we will use "move page and clear source" operation instead */
#define FAST_ZERO_PAGE_BUG

inline
void fast_page_op( unsigned long cmd,
                   unsigned long src_page, /* page number in RMPR reg */
                   unsigned long dst_page, /* page number in RMPCR reg */
                   unsigned long class_code,
                   int wait)
{
        unsigned long flags;
        unsigned int tmp;
```

```
811
812         if (! mxt_compression_enabled )
813                 return;
814
815         spin_lock_irqsave(&fast_page_lock,flags);
816
817         class_code &=  RMPR_CLASSCODE_BITMASK;
818
819         switch (cmd) {
820         case RMPR_CMD_NOOP:
821         case RMPR_CMD_ZEROPAGE:
822         case RMPR_CMD_FLUSHINVALIDATE:
823         case RMPR_CMD_TRANSFERCC_CTT:
824         case RMPR_CMD_MOVECTT2LAST:
825         case RMPR_CMD_INVALIDATE_XFERCC:
826                 WRITE_CTRL( RMPR, RMPR_BUILD_REQUEST(cmd, class_code,
827                                                      src_page) );
828                 break;
829         case RMPR_CMD_MOVEZEROSRC_CC:
830         case RMPR_CMD_MOVEZEROSRC:
831         case RMPR_CMD_SWAPPAGE_CC:
832         case RMPR_CMD_SWAPPAGE:
833                 WRITE_CTRL( RMPCR, RMPCR_LOCK_BIT | dst_page);
834                 WRITE_CTRL( RMPR, RMPR_BUILD_REQUEST(cmd, class_code,
835                                                      src_page) );
836                 break;
837         default:
838                 printk("<MC>: error: fast page op undefined\n");
839         }
840
841         if (wait) {
842                 volatile unsigned long tmp;
843                 do {
844                         tmp = READ_CTRL( RMPR );
845                 } while ( (tmp & RMPR_CMD_MASK) );
846         }
847
848         /* Serialize instruction execution */
849         cpuid(0,&tmp,&tmp,&tmp,&tmp);
850
851         spin_unlock_irqrestore(&fast_page_lock, flags);
852 }
853
854 #ifdef FAST_ZERO_PAGE_BUG
855 /* don't touch this memory at all; helps fix the hardware bug */
856 static char scratch[PAGE_SIZE] __attribute__ ((aligned(PAGE_SIZE)));
857 #endif
858
859 #ifndef FAST_ZERO_PAGE_BUG
860 /*
861  * Zero page operations is a method for clearing a page using a Katsina
862  * chip "Fast Page Operation".  Clearing a page is merely updating a
863  * pointer instead of a 4KB bulk transfer.  It also invalidates the
864  * corresponding lines in L3 cache. See CNB 3.0 spec, RMPR register.
865  */
866 inline
867 static void cmpmem_zero_page(unsigned long page_no, int wait)
868 {
869         unsigned long flags;
870         unsigned int tmp;
871
872         spin_lock_irqsave(&fast_page_lock,flags);
873
874         /* zero a single physical page with page frame number <page_no>*/
875         WRITE_CTRL( RMPR, RMPR_BUILD_REQUEST(RMPR_CMD_ZEROPAGE,
876                                              0, page_no) );
877
878         if (wait) {
879                 volatile unsigned long tmp;
880                 do {
881                         tmp = READ_CTRL( RMPR );
882                 } while ( (tmp & RMPR_CMD_MASK) );
883         }
884
885         /* Serialize instruction execution */
886         cpuid(0,&tmp,&tmp,&tmp,&tmp);
887
888         spin_unlock_irqrestore(&fast_page_lock, flags);
889 }
890 #else
891 inline
892 static void cmpmem_zero_page(unsigned long page_no, int wait)
893 {
894         unsigned long va = PAGE_ALIGN((unsigned long)scratch);
895         fast_page_op( RMPR_CMD_MOVEZEROSRC_CC, page_no,
896                       __pa(va)>>PAGE_SHIFT, 0, 1);
897 }
898 #endif  /* FAST_ZERO_PAGE_BUG */
899
900
```

```
901 #include <asm/page.h>
902 #include <linux/mm.h>
903 #include <linux/highmem.h>
904
905 void *mxt_clear_virtpage(unsigned long vaddr)
906 {
907         /* First 1 Meg does not compress on Katsina */
908
909         unsigned long paddr = __pa(vaddr);
910
911         if ( paddr < MIN_ADDR_FASTCLEAR || !mxt_compression_enabled ) {
912                 memset((void*)vaddr, 0, PAGE_SIZE);
913         }
914         else {
915                 cmpmem_zero_page(paddr>>PAGE_SHIFT, 1);
916         }
917         return ( (void *)vaddr );
918 }
919
920 void mxt_clear_page(struct page *page)
921 {
922         unsigned long page_nr = (unsigned long)(page − mem_map);
923
924         if ( !VALID_PAGE(page) )
925                 BUG();
926
927         if ( mxt_compression_enabled &&
928              page_nr > (MIN_ADDR_FASTCLEAR >> PAGE_SHIFT) ) {
929                 cmpmem_zero_page(page_nr, 1);
930                 return;
931         }
932
933         if ( ! PageHighMem(page) )
934                 clear_page( page_address(page) );
935         else
936                 clear_highpage(page);
937 }
938
939 void mxt_clear_highpage(struct page *page)
940 {
941         mxt_clear_page(page);
942 }
943
944 /* arch/i386/kernel/mxt.c */
945 extern void (*mxt_fast_clear)(struct page *);
946
947 /* make the function available for general use */
948 static void mxt_hook_clear_page(void)
949 {
950         unsigned long flags;
951         spin_lock_irqsave(&fast_page_lock,flags);
952         mxt_fast_clear = mxt_clear_page;
953         spin_unlock_irqrestore(&fast_page_lock, flags);
954 }
955
956 static void mxt_unhook_clear_page(void)
957 {
958         unsigned long flags;
959         MCPRINTK("<MC> mxt_unhook_clear_page: enter\n");
960         spin_lock_irqsave(&fast_page_lock,flags);
961         mxt_fast_clear = NULL;
962         spin_unlock_irqrestore(&fast_page_lock, flags);
963 }
964
965 /* ---------------------- Section 5 ------------------------------ */
966
967 /*
968  * Interrupt support: if SUR register exceeds SUTLR register controller
969  * will fire an interrupt which is an indication of physical memory
970  * pressure.
971  * We're already tracking this by polling; but interrupts will catch the
972  * pressure in between two polls if pressure is increasing fast.
973  */
974
975 static struct tq_struct sutlr_task;
976
977 static void execute_sutlr_task(void * n)
978 {
979         if ( katsina_memutil() > mc_th.panic )
980                 panic("<MC> cannot reduce physical memory pressure!\n");
981         cmp_grab_cpus();
982         memcompress_check();
983         return;
984 }
985
986 static void cmpintr_isr(int irq, void *dev_instance, struct pt_regs *regs)
987 {
988         volatile unsigned long status;
989
990         MCPRINTK("cmpintr_isr: enter\n");
```

```
991
992             status = READ_CTRL(ESR);
993
994             mxt_stats.esr = status;
995             mxt_stats.esr_sticky |= status;
996
997             if ( status & ESR_SUTLR_MASK ) {
998                     /* there is physical memory pressure */
999                     mxt_disable_sutlr_interrupt();
1000                    mxt_clear_sutlr_interrupt();
1001                    queue_task(&sutlr_task, &tq_immediate);
1002                    mark_bh(IMMEDIATE_BH);
1003            }
1004            else {
1005                    mxt_clear_all_interrupts();
1006            }
1007    }
1008
1009    static struct pci_dev *pdev;
1010
1011    static int  mxt_setup_interrupts(void)
1012    {
1013            unsigned char pci_bus, pci_device_fn, pci_intr_pin, pci_intr_line;
1014            volatile unsigned long status;
1015
1016            if ( ! mxt_bios_found ) {
1017                    printk("<MC>: No MXT bios found\n");
1018                    return 1;
1019            }
1020
1021            if ( ! mxt_compression_enabled ) {
1022                    printk("<MC>: Compression disabled\n");
1023                    return 1;
1024            }
1025
1026            if ( ! pcibios_present() ) {
1027                    printk("<MC>: No pci bios, no interrupts! Polling only\n");
1028                    return 1;
1029            }
1030
1031            if(pcibios_find_device( IBM_PCI_VENDORID_COMPMEM,
1032                                    IBM_PCI_DEVICEID_COMPMEM,
1033                                    0,
1034                                    &pci_bus,
1035                                    &pci_device_fn) ) {
1036                    printk("<MC>: no PCI; no interrupts!\n");
1037                    return 1;
1038            }
1039
1040            mxt_device_found = 1;
1041
1042            pdev = pci_find_slot(pci_bus, pci_device_fn);
1043            pcibios_read_config_byte(pci_bus, pci_device_fn ,
1044                                     PCI_INTERRUPT_PIN, &pci_intr_pin);
1045            pcibios_read_config_byte(pci_bus, pci_device_fn,
1046                                     PCI_INTERRUPT_LINE, &pci_intr_line);
1047
1048            printk("<MC>: bus=%x, devfn=%x, vid=%04x, did=%04x\n"
1049                   "    class=%x, irq=%d, pin=%x, line=%x\n",
1050                   pci_bus,
1051                   pci_device_fn,
1052                   pdev->vendor,
1053                   pdev->device,
1054                   pdev->class,
1055                   pdev->irq,
1056                   pci_intr_pin,
1057                   pci_intr_line);
1058
1059            /*
1060             * Initialize for interrupts
1061             */
1062            // WRITE_CTRL(ESMR0, 0); /* disable interrupts connected to NMI */
1063
1064            /* physical memory fatal UE; Brett says it works. Other
1065               ServerWorks bits occasionally will light up */
1066            WRITE_CTRL(ESMR0, 1<<5);
1067
1068            /* Driver is supposed to handle SUTLR interrupts
1069             * this will turn off SUTLR interrupts from reaching NMI */
1070            WRITE_CTRL(ESMR0, (READ_CTRL(ESMR0) &
1071                               ~((unsigned long)ESMR1_SUTLR_INT)));
1072            WRITE_CTRL(ESMR1, 0); /* disable interrupts connected to INTx */
1073            mxt_clear_all_interrupts(); /* clear existing ones */
1074            WRITE_CTRL(SUTLR, 0xffffffff);  /* set SUTLR to a high value */
1075            WRITE_CTRL(SUTHR, 0xffffffff);  /* set SUTHR to a high value */
1076
1077            /*
1078             * Initialize the immediate tasks that ISR will dispatch
1079             */
1080            /* sutlr_task.next = NULL; */
```

```
1081          sutlr_task.sync = 0;
1082          sutlr_task.routine = execute_sutlr_task;
1083          sutlr_task.data = 0;
1084
1085          /*
1086           * Try connecting to the interrupt
1087           */
1088
1089          if ( pdev->irq ) {
1090                  /* Note: in uniprocessor build of Linux
1091                   * pdev->irq comes back as zero. No clue why.
1092                   * In SMP build with APICs on pdev->irq comes back as a
1093                   * valid number. Must have to do with PIC vs APIC modes.
1094                   * Thus, in uni build I pick the IRQ from $MXT table in
1095                   * EBDA.  In smp build I use pdev->irq */
1096                  mxt_irq = pdev->irq;
1097          }
1098
1099          if (request_irq(mxt_irq, &cmpintr_isr,
1100                          0, "MXT Memory Controller", pdev)) {
1101                  printk("<MC>: request_irq() failed\n");
1102                  return 1;
1103          }
1104
1105          printk("<MC>: MXT Memory Controller interrupt connected IRQ=%d\n",
1106                 mxt_irq);
1107
1108          /*
1109           * set the interrupt threshold which when exceeded
1110           * memory controller will send an interrupt
1111           */
1112          status = memutil_to_sectors(mc_th.intr);
1113          WRITE_CTRL(SUTLR, status);
1114          printk("<MC>: Set SUTLR=%08lx\n", status);
1115
1116          mxt_enable_sutlr_interrupt();
1117          status = READ_CTRL(ESMR1);
1118          printk("<MC>: Set ESMR1=%08lx\n", status);
1119
1120          return 0;
1121  }
1122
1123  static int mxt_disconnect_interrupts(void)
1124  {
1125          mxt_disable_sutlr_interrupt();
1126          mxt_clear_sutlr_interrupt();
1127          free_irq(mxt_irq, pdev);
1128          return 0;
1129  }
1130
1131  /* ---------------------- Section 6 ------------------------------ */
1132
1133  /*
1134   *  Generic Memory Compression Support
1135   */
1136
1137  #include <linux/slab.h>
1138  #include <linux/kernel_stat.h>
1139  #include <linux/swap.h>
1140  #include <linux/swapctl.h>
1141  #include <linux/smp_lock.h>
1142  #include <linux/pagemap.h>
1143  #include <linux/init.h>
1144  #include <linux/version.h>
1145  #include <linux/highmem.h>
1146  #include <linux/cache.h>
1147
1148  #include <asm/pgtable.h>
1149  #include <asm/mxt.h>
1150  // #include <asm/spinlock.h>
1151
1152  /* cmp_idle thread related */
1153  #include <linux/delay.h>
1154  #include <linux/sched.h>
1155
1156  /* cmptimer related */
1157  #include <linux/tqueue.h>
1158
1159  /* cmp_idle thread related */
1160  #define CMP_NMAX_TASKS  32
1161  #define CMP_IDLE_RUN    1
1162  #define CMP_IDLE_SLEEP  0
1163
1164  static struct task_struct *cmp_idle_task[CMP_NMAX_TASKS];
1165  static volatile int cmp_idle_run=CMP_IDLE_SLEEP;
1166
1167  /* static struct task_struct *cmp_zerod_task[CMP_NMAX_TASKS]; */
1168
1169  static struct task_struct *cmp_eatmem_task[CMP_NMAX_TASKS];
1170
```

```
1171 /* cmptimer related */
1172 static void start_periodic_task(void);
1173 static void stop_periodic_task(void);
1174
1175 /* for stalling processes */
1176 static void zap_bad_processes(void);
1177
1178 static spinlock_t mc_lock __cacheline_aligned;
1179
1180 /*
1181  * We basically check for 5 different ranges of memory utilization
1182  * between 0 and 100% of memory utilization.
1183  * Consider the 100 to be some max number ..
1184  *
1185  *    [  1   |   2   |   3   |   4   |  5  ]
1186  *    0   th_rel  th_acq   th_dan  th_int 100
1187  *
1188  * we always try to keep the system operating in 1−3
1189  * in phase 1 there is nothing to do...
1190  * in phase 2 we are willing to release a few pages if necessary/possible
1191  * in phase 3 we are trying to acquire pages if possible
1192  * in phase 4 we are bringing the utilization back to 3 threshold
1193  * in phase 5 we are in high danger mode, we should basically block CPUs
1194  */
1195
1196 extern unsigned long totalreserved;
1197
1198 static inline unsigned long get_memutil(void)
1199 {
1200         /* memutil is normalized to 1000 */
1201         unsigned long memutil = katsina_memutil();
1202         if (memutil > mc_th.max) {
1203                 mc_th.max = memutil;
1204                 if (mc_th.max > mc_th.intr)
1205                         MCPRINTK("mc_th.max = %ld\n", mc_th.max);
1206                 if (mc_th.max > mc_th.panic )
1207                         panic("<MC> memory pressure!\n");
1208         }
1209         return memutil;
1210 }
1211
1212 unsigned long get_used_pages(void)
1213 {
1214         return (totalram_pages − nr_free_pages());
1215 }
1216
1217 static inline int memutil_2_state(unsigned long memutil)
1218 {
1219         /* classify the current memory utilization into a
1220            particular state */
1221         int state;
1222
1223         if      (memutil < mc_th.release)
1224                 state = 1;
1225
1226         else if (memutil < mc_th.acquire)
1227                 state = 2;
1228
1229         else if (memutil < mc_th.danger)
1230                 state = 3;
1231
1232         else if (memutil < mc_th.intr)
1233                 state = 4;
1234
1235         else
1236                 state = 5;
1237
1238         return state;
1239 }
1240
1241 /*
1242  * mu:     current physical memory utilization
1243  * maxmu:  maximum desired physical utilization
1244  * if mu exceeds maxmu we will create a free page pool deficieny
1245  * so that kswapd and friends can bring some pages back to the pool.
1246  *
1247  * algorithm:
1248  *    assume compression rate piecewise linear and stable in time
1249  *    then
1250  *            maxmu       maxpages
1251  *            −−−−−   =   −−−−−−−−
1252  *            mu          usedpages
1253  *
1254  * this equation drives how many pages we have to take out of the system.
1255  */
1256
1257
1258 #define HIGH_DECAY  ((tuning.high_decay*1024)/PAGE_SIZE)
1259 #define LOW_DECAY   ((tuning.low_decay*1024)/PAGE_SIZE)
1260 /*
```

```
1261  * Slowly release held pages at a rate of one percent
1262  * per period (period is 100ms) with limits
1263  */
1264 static inline long mc_decayed_release(long pages)
1265 {
1266         long decay;
1267
1268         if ( pages <= 0 )
1269                 return 0;
1270
1271         decay = pages/500;  /* 2 percent decay per second */
1272
1273         if ( decay > HIGH_DECAY )
1274                 decay = HIGH_DECAY;
1275         else if ( decay < LOW_DECAY )
1276                 decay = LOW_DECAY;
1277
1278         pages = _MAX (pages − decay, 0);
1279
1280         return pages;
1281 }
1282
1283 /*
1284  * mu is current physical memory utilization
1285  * maxmu is target physical memory utilization
1286  * this function will calculate how many pages we
1287  * should take away from the system to reduce
1288  * physical memory pressure.
1289  */
1290 static inline long mc_adjust_check(unsigned long mu, unsigned long maxmu)
1291 {
1292         long adj;
1293         long usedpages;
1294         long maxpages;
1295         long nrfree;
1296         long tmp;
1297         long actual_reserved;
1298         long ceil;
1299         long hidden;
1300
1301         /* in steady state nr_rsrv_pages == actual_reserved.
1302          * in transient state nr_rsrv_pages might be much larger
1303          * than actual_reserved because eatmem threads take
1304          * time to allocated and clear memory */
1305
1306         nrfree = nr_free_pages();
1307         usedpages = totalram_pages − nrfree;
1308
1309         /*
1310          * Maxpages is max number of used pages we should allow for.
1311          * We are not going to count the pages we reserved
1312          * because they are zeroed and therefore they do
1313          * not contribute to the physical utilization mu
1314          */
1315         hidden = atomic_read(&nr_hidden_pages);
1316         hidden = (hidden/NR_CPUS)*NR_CPUS;
1317
1318         actual_reserved = atomic_read(&total_pages_held) + hidden;
1319
1320         maxpages = (long)maxmu * ((usedpages−actual_reserved) / (long)mu);
1321
1322         /* pages we should take out from the system
1323          *  to meet (mu <= maxmu) target
1324          */
1325         tmp  = totalram_pages − maxpages;
1326         tmp  = ( (tmp+NR_CPUS−1)/NR_CPUS )*NR_CPUS;
1327
1328         /*
1329          * sanity check: no need to reserve more than the missing
1330          * amount of physical memory.  But the reason for this equation
1331          * (as opposed to (usedpages+nrfree)/2) is because,
1332          * a) the user might have limited memory size to less than
1333          *    the real memory size.  For example, 512MB physical
1334          *    and 1024MB real is in the system.  But user specifies
1335          *    mem=700M in /etc/lilo.conf.
1336          * b) we need the pressure the utilization down below mc_th.danger
1337          * c) we add 10% to cover any slop we might have forgotten
1338          *    such as SetPageReserved() pages.
1339          * d) L3 contents need to be flushed out to the physical memory;
1340          *    (although fast page clear does that.)
1341          */
1342         ceil = (long)totalram_pages
1343                 − ( (long)mc_th.release * ((long)cmp_num_physpages / 1000))
1344                 + (long)cmp_num_physpages / 10L + (long)L3_size;
1345         if ( tmp > ceil )
1346                 tmp = ceil;
1347
1348         /* sanity check: cannot reserve negative count of pages  */
1349         if ( tmp < 0)
1350                 tmp = 0;
```

```
1351
1352            adj = tmp − actual_reserved;
1353
1354            /* positive means grab more pages than previously;
1355             * negative means release some pages */
1356
1357            /* let the decay routine release the pages; we are not going
1358             * to do it here (adj < 0 case).  And if the positive
1359             * adjustment is too small don't bother (adj > 0 case) */
1360            if ( adj < (L3_size/32) )
1361                    return 0;
1362
1363            /* nr_rsrv_pages is what memeater threads should grab
1364             * hidden is what page hiding threads are hiding
1365             */
1366            nr_rsrv_pages = tmp − hidden;
1367            nr_pages_per_thread = nr_rsrv_pages/smp_num_cpus;
1368
1369            return adj;
1370    }
1371
1372
1373    /* If memory utilization is running up too fast we will try
1374     * to bump it by quickly allocating pages even when we are in the
1375     * safe state 1.
1376     */
1377    static long bump_fast(long curr_util, long prev_util, long curr_held)
1378    {
1379            long tmp = memutil_to_pages (curr_util − prev_util);
1380            tmp = _MIN( curr_held + tmp, totalram_pages − cmp_num_physpages );
1381            return ( _MAX( tmp, 0 ) );
1382    }
1383
1384    static int release_delay    __cacheline_aligned;
1385    static volatile int cur_state __cacheline_aligned;
1386    static atomic_t prev_state  __cacheline_aligned = { 0 };
1387    static atomic_t prev_util   __cacheline_aligned = { 0 };
1388
1389    int memcompress_check(void)
1390    {
1391            long memutil;
1392            unsigned long flags;
1393            long th_bump;
1394            int bumped=0;
1395            int state;
1396
1397            memutil = get_memutil();
1398            cur_state = state = memutil_2_state(memutil);
1399
1400            if ( mxt_printk )
1401                    if ( state != atomic_read(&prev_state) ) {
1402                            /* eliminate excessive printing */
1403                            MCPRINTK("memcompress_check mu=%ld state=%d\n",
1404                                     memutil, state );
1405                            atomic_set (&prev_state, state);
1406                    }
1407
1408            /* bump the bad ass processes trying to run up the
1409             * compressed mem util
1410             */
1411            th_bump = tuning.bump_rate/(PAGE_SIZE/1024); /*pages per period*/
1412            th_bump = (1000 * th_bump) / (long) cmp_num_physpages;
1413
1414            if (((long)memutil − (long)atomic_read(&prev_util) ) > th_bump) {
1415                    spin_lock_irqsave(&mc_lock,flags);
1416                    nr_rsrv_pages = bump_fast(memutil, atomic_read(&prev_util),
1417                                              atomic_read(&total_pages_held) );
1418                    nr_pages_per_thread = nr_rsrv_pages/smp_num_cpus;
1419                    bumped=1;
1420                    spin_unlock_irqrestore(&mc_lock, flags);
1421            }
1422            atomic_set (&prev_util, memutil);
1423
1424            /*
1425             * This is main decision point to determine if
1426             * there is compressed memory pressure (or not),
1427             * and then allocate/clear pages (or free pages)
1428             */
1429            switch (state) {
1430
1431            case 1: /* safe state */
1432
1433                    if ( bumped )
1434                            wake_up_cmp_eatmem();
1435
1436                    /* release pages back slowly  */
1437                    if ( !bumped && (atomic_read(&total_pages_held) > 0) ) {
1438                            spin_lock_irqsave(&mc_lock,flags);
1439
1440                            if ( −−release_delay < 0 ) {
```

```
1441                                    release_delay = 0;
1442                                    nr_rsrv_pages = mc_decayed_release(
1443                                            atomic_read(&total_pages_held) );
1444                                    nr_pages_per_thread =
1445                                            nr_rsrv_pages/smp_num_cpus;
1446                                    wake_up_cmp_eatmem();
1447                            }
1448                            spin_unlock_irqrestore(&mc_lock, flags);
1449                    }
1450                    break;
1451
1452            case 2: /* mc_th.release */
1453
1454                    if ( bumped )
1455                            wake_up_cmp_eatmem();
1456
1457                    break;
1458
1459            case 5: /* mc_th.intr: we are near edge; start blocking CPUs */
1460
1461                    spin_lock_irqsave(&mc_lock,flags);
1462
1463                    cmp_grab_cpus();
1464
1465                    zap_bad_processes();
1466
1467                    /* bring it down to acquire threshold */
1468                    mc_adjust_check (memutil, mc_th.acquire);
1469
1470                    wake_up_cmp_eatmem();
1471
1472                    release_delay = (tuning.delay * HZ)/tuning.period;
1473
1474                    spin_unlock_irqrestore(&mc_lock, flags);
1475
1476                    break;
1477
1478            case 4: /* mc_th.danger */
1479
1480                    spin_lock_irqsave(&mc_lock,flags);
1481
1482                    zap_bad_processes();
1483
1484                    if ( bumped )
1485                            wake_up_cmp_eatmem();
1486
1487                    /* bring it down to acquire threshold */
1488                    if ( mc_adjust_check (memutil, mc_th.acquire) > 0 ) {
1489                            wake_up_cmp_eatmem();
1490                    }
1491
1492                    release_delay = (tuning.delay * HZ)/tuning.period;
1493
1494                    spin_unlock_irqrestore(&mc_lock, flags);
1495
1496                    break;
1497
1498            case 3: /* mc_th.acquire */
1499
1500                    spin_lock_irqsave(&mc_lock,flags);
1501
1502                    if ( bumped )
1503                            wake_up_cmp_eatmem();
1504
1505                    /* bring it down to acquire threshold */
1506                    if ( mc_adjust_check (memutil, mc_th.acquire) > 0 ) {
1507                            wake_up_cmp_eatmem();
1508                    }
1509
1510                    release_delay = (tuning.delay * HZ)/tuning.period;
1511
1512                    spin_unlock_irqrestore(&mc_lock, flags);
1513
1514                    break;
1515
1516            }
1517
1518            return state;
1519    }
1520
1521    /* ---------------------- Section 7 ------------------------------ */
1522
1523    /*
1524     * cmp_idle thread routines prevent compressed memory pressure
1525     * by grabbing cpus. These routines
1526     * 1) start cmp_idle threads one per cpu,
1527     * 2) increase or decrease priority of big tasks, kswapd,
1528     *    and cmp_idle threads as needed
1529     * 3) grab cpus at higher than big task priority levels
1530     * 4) grab cpus at higher than default priority level
```

```
 */

static int cmp_idle_init(void)
{
        int t;

        if ( !mxt_compression_enabled ) return 0;

        printk("<MC> Starting cpu blocker threads: %d\n", smp_num_cpus);
        for (t=0; t < smp_num_cpus; t++) {
                kernel_thread(cmp_idle, (void*)t,
                        CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
        }
        return 0;
}

/*
 * Idle tasks exist one or more per CPU.  When there is extreme physical
 * memory pressure due to low compressibility, they do nothing but eat
 * CPU cycles to prevent other tasks from running and therefore
 * further degrading compressibility until kswapd cleans up the mess.
 *
 * We could have changed the scheduler.c for this purpose, but this code
 * results in less changes in the original kernel.  These threads
 * are only for compression emergency anyway and not supposed to be running
 * if we have enough memory in the system.
 */
int cmp_idle(void *cpu_num)
{
        struct task_struct *tsk;
        int thread_id = (int) cpu_num;
        char banner[] = "mxtspn0";

        daemonize();

        lock_kernel();
        tsk = current;
        tsk->session = 1;
        tsk->pgrp = 1;
        sigfillset(&tsk->blocked);
        tsk->tty = NULL;    /* get rid of controlling tty */
        tsk->policy = SCHED_OTHER;
        set_user_nice(tsk, -18);
        banner[ sizeof("mxtspn0") - 2 ] += thread_id;
        strcpy(tsk->comm, banner );
        cmp_idle_task[thread_id] = tsk;
        unlock_kernel();

        if ( !thread_id )
                start_periodic_task(); /* start one periodic task only */

looping:
        if ( cmp_idle_run == CMP_IDLE_SLEEP ) { /* must go to sleep */
                interruptible_sleep_on(&cmp_idle_wait);
        }
        else {
                /* we're told to wake up and begin wasting cpu cycles */
                MCPRINTK("%s up\n", banner);
                while(1) {
                        int i;
                        /* waste cpu cycles */
                        for(i=0; i<tuning.spin_period; i++)
                                udelay(1000000/HZ);

                        if ( current->need_resched ) {
                                set_current_state(TASK_RUNNING);
                                schedule();
                        }

                        if ( !thread_id ) {
                                /*
                                 * if the first thread detects that
                                 * utilization drop below the danger
                                 * threshold it will tell other threads
                                 * to quit spinning
                                 */
                                unsigned long memutil = get_memutil();
                                if ( memutil_2_state(memutil) < 4 ) {
                                        /*
                                         * Return to state 4 to add some
                                         * for some hysterisis.
                                         * This should keep running
                                         * until util < 4
                                         */
                                        unsigned long flags;
                                        spin_lock_irqsave(&mc_lock,flags);
                                        cmp_release_cpus();
                                        spin_unlock_irqrestore(&mc_lock,
                                                               flags);
                                        mxt_clear_sutlr_interrupt();
```

```
1621                                        mxt_enable_sutlr_interrupt();
1622                                }
1623                        }
1624                        if ( cmp_idle_run == CMP_IDLE_SLEEP )
1625                                break;
1626                }
1627                MCPRINTK("%s down\n", banner);
1628        }
1629        goto looping;
1630 }
1631
1632 static void cmp_grab_cpus(void)
1633 {
1634        cmp_idle_run = CMP_IDLE_RUN;
1635        if ( waitqueue_active( &cmp_idle_wait ) )
1636                wake_up_interruptible_all(&cmp_idle_wait);
1637 }
1638
1639 static void cmp_release_cpus(void)
1640 {
1641        if ( cmp_idle_run == CMP_IDLE_RUN )
1642                cmp_idle_run = CMP_IDLE_SLEEP;
1643 }
1644
1645
1646 /* ---------------------- Section 8 ------------------------------ */
1647
1648 /*
1649  * Timer routine calls kernel compression service periodically
1650  */
1651
1652 #ifdef MXT_OLD_TIMERS
1653 static struct tq_struct  tq;
1654 static int tq_counter=2000; /* any nonzero initial value will do */
1655
1656 /* This routine is called each time tq_timer queue is run
1657  * The routine will enqueue itself again
1658  */
1659 static inline void periodic_task(void * run)
1660 {
1661
1662        if ( ! run ) {
1663                tq.routine = NULL;
1664                return;
1665        }
1666
1667        if ( --tq_counter <= 0 ) { /* HZ/tuning.period times a second */
1668                memcompress_check();
1669                tq_counter = HZ/tuning.period;
1670        }
1671
1672        /* enqueue the task to the timer queue which runs HZ/second */
1673        queue_task( &tq, &tq_timer );
1674 }
1675
1676 static void start_periodic_task(void)
1677 {
1678        tq.routine = periodic_task; /* function to execute */
1679        tq.data = (void*) 1;        /* argument of the function */
1680        queue_task(&tq, &tq_timer);
1681 }
1682
1683 static void stop_periodic_task(void)
1684 {
1685        unsigned long flags;
1686
1687        /* this tells periodic task to stop during its last execution */
1688        spin_lock_irqsave(&tqueue_lock, flags);
1689        tq.data = (void*) 0;
1690        spin_unlock_irqrestore(&tqueue_lock, flags);
1691
1692        /* wait until the task stops
1693         * tq.routine == NULL is an indication that it stopped
1694         */
1695        while ( tq.routine ) {
1696                set_current_state(TASK_RUNNING);
1697                yield();
1698        }
1699 }
1700 #else  /* MXT_OLD_TIMERS */
1701
1702 #include <linux/timer.h>
1703
1704 #define RUN_AT(x)   (jiffies + (x))
1705 static struct timer_list tq;
1706
1707
1708 /* This routine is called each time tq_timer queue is run
1709  * The routine will enqueue itself again
1710  */
```

```
1711 static inline void periodic_task(unsigned long data)
1712 {
1713         if ( data ) {
1714
1715                 memcompress_check();
1716
1717                 /* repeat so many ticks a second */
1718                 tq.expires = RUN_AT( HZ/tuning.period );
1719                 add_timer(&tq);
1720         }
1721 }
1722
1723 static void start_periodic_task(void)
1724 {
1725         init_timer(&tq);
1726         tq.function = &periodic_task; /* function to execute */
1727         tq.data = 1;                  /* argument of the function */
1728         tq.expires = RUN_AT(10*HZ);   /* wait 10 seconds initially */
1729         add_timer(&tq);
1730 }
1731
1732 static void stop_periodic_task(void)
1733 {
1734         tq.data = 0;
1735         del_timer_sync( &tq );
1736 }
1737 #endif  /* MXT_OLD_TIMERS */
1738
1739
1740 /* ---------------------- Section 9 ------------------------------ */
1741 /*
1742  *  Page clearing threads reduce compression pressure by allocating
1743  *  pages from the system and clearing them with zeros.
1744  *  A 4KB page filled with zeros occupies only 64 bytes in physical
1745  *  memory.
1746  */
1747
1748 /* #define EATSIZE  ((long)freepages.high) */
1749 #define EATSIZE  256
1750
1751 static long __alloc_clear(long nr_pages,
1752                           unsigned int gfp_mask, struct list_head * head)
1753 {
1754         long count=0;
1755         long cleared=0;
1756         struct page * page;
1757
1758         while ( cleared < nr_pages ) {
1759
1760                 page = alloc_pages(gfp_mask,0);
1761                 if ( ! page ) {
1762                         MCPRINTK("failed allocation %s %d %8x\n",
1763                                  __FILE__, __LINE__, gfp_mask);
1764                         if (waitqueue_active(&kswapd_wait))
1765                                 wake_up_interruptible(&kswapd_wait);
1766                         wakeup_bdflush();
1767                         run_task_queue(&tq_disk);
1768                         break;
1769                 }
1770
1771                 if ( fclr || cur_state == 5)
1772                         mxt_clear_page(page);
1773                 else
1774                         clear_highpage(page);
1775                 ++cleared;
1776                 list_add( &page->list, head );
1777
1778                 if ( current->need_resched ) {
1779                         set_current_state(TASK_RUNNING);
1780                         schedule();
1781                 }
1782
1783                 /* check once in a while if there is shortage */
1784                 if ( count++ < pages_min/4  )
1785                         continue;
1786
1787                 count=0;
1788
1789                 /* if we're down to last few pages ???? */
1790                 if ( nr_free_pages() < pages_high ) {
1791
1792                         if (waitqueue_active(&kswapd_wait))
1793                                 wake_up_interruptible(&kswapd_wait);
1794
1795                         wakeup_bdflush();
1796                         run_task_queue(&tq_disk);
1797
1798                         set_current_state(TASK_RUNNING);
1799                         yield();
1800                 }
```

```
1801
1802            }
1803            return cleared;
1804    }
1805
1806    /*
1807     * Retrieve the pages_min and pages_low watermarks
1808     */
1809    void low_mem_watermarks(void)
1810    {
1811            zone_t *zone;
1812            pg_data_t *pgdat = pgdat_list;
1813
1814            pages_min = 0;
1815            pages_low = 0;
1816            pages_high = 0;
1817
1818            while (pgdat) {
1819                    /*
1820                     * Always allocate from the last zone with GFP_HIGHUSER so
1821                     * let's see the watermarks from such point of view
1822                     */
1823                    int point_of_view = pgdat->nr_zones-1;
1824
1825                    for (zone = pgdat->node_zones; zone < pgdat->node_zones + pgdat->nr_zones; zone++) {
1826                            pages_min += zone->watermarks[point_of_view].min;
1827                            pages_low += zone->watermarks[point_of_view].low;
1828                            pages_high += zone->watermarks[point_of_view].high;
1829                    }
1830
1831                    pgdat = pgdat->node_next;
1832            }
1833    }
1834
1835    /*
1836     * Allocate, clear and add pages to the list.
1837     * Use hardware fast clear to clear & invalidate L3 cache.
1838     * This helps reduce compression pressure.
1839     * Returns number of pages in processed.
1840     *
1841     */
1842    static long alloc_clear(long nr_pages, struct list_head * held_list)
1843    {
1844            int total;
1845            int ask;
1846            int remainder=nr_pages;
1847            long cleared;
1848            volatile long initial = nr_pages_per_thread;
1849
1850            while (1) {
1851
1852                    total = nr_free_pages() + nr_inactive_pages;
1853
1854                    /* this will try pushing usage down to pages_min */
1855                    ask = _MAX( total - pages_min, 0L ) / smp_num_cpus;
1856
1857                    /* don't alloc more than requested or more than
1858                     * EATSIZE per iteration */
1859                    ask = _MIN( ask, remainder );
1860                    ask = _MIN( ask, EATSIZE );
1861
1862                    /* ZONE_HIGHMEM inclusive of ZONE_NORMAL and ZONE_DMA
1863                     * see mm/page_alloc.c::build_zonelists() */
1864                    cleared = __alloc_clear (ask, GFP_HIGHUSER, held_list);
1865
1866                    atomic_add (cleared, &total_pages_held);
1867                    remainder -= cleared;
1868
1869                    if ( remainder <= 0 )
1870                            return (nr_pages-remainder);
1871
1872                    /* nr pages to be held has been reduced; quit now */
1873                    if ( nr_pages_per_thread < initial ) {
1874                            return (nr_pages-remainder);
1875                    }
1876
1877                    /* yield in anticipation of kswapd and friends making
1878                       new free pages */
1879                    if ( !cleared ) {
1880                            set_current_state(TASK_RUNNING);
1881                            schedule();
1882                    }
1883
1884            }
1885    }
1886
1887
1888    /*
1889     * all=1 means release everything
1890     */
```

```
1891 static long release_cleared (long nr_pages,
1892                               struct list_head * held_list,
1893                               int all)
1894 {
1895         struct page * page;
1896         long released=0;
1897         long count=0;
1898         volatile long initial = nr_pages_per_thread;
1899
1900         /*
1901          * Note that we don't want to create a flurry of atomic
1902          * updates on the bus.
1903          * So we will subtract once in a while in bigger quantities
1904          */
1905
1906         while( !list_empty(held_list) && ((released < nr_pages) || all) ){
1907
1908                 page = list_entry(held_list->next, struct page, list);
1909                 list_del_init( &page->list );
1910                 __free_page(page);
1911                 ++released;
1912                 ++count;
1913
1914                 /* if someone wants more pages to be held
1915                  * must quit freeing and return immediately
1916                  */
1917                 if ( nr_pages_per_thread > initial ) {
1918                         atomic_sub (count, &total_pages_held);
1919                         return released;
1920                 }
1921
1922                 if ( current->need_resched && !all ) {
1923                         atomic_sub (count, &total_pages_held);
1924                         count = 0;
1925                         schedule();
1926                 }
1927         }
1928         atomic_sub (count, &total_pages_held);
1929
1930         return released;
1931 }
1932
1933 /*
1934  * use before module unload
1935  */
1936 static void release_held_pages(void)
1937 {
1938         printk("<MC> release_held_pages: enter\n");
1939
1940         nr_rsrv_pages = 0;
1941         nr_pages_per_thread = 0;
1942         wake_up_cmp_eatmem();
1943         set_current_state(TASK_RUNNING);
1944         yield();
1945 }
1946
1947
1948 static int cmp_eatmem(void *cpu_num)
1949 {
1950         struct task_struct *tsk = current;
1951         int myid = (int) cpu_num;
1952         char banner[] = "mxtclr0";
1953         LIST_HEAD(held_list);
1954         long held=0;
1955         long count=0;
1956
1957         INIT_LIST_HEAD(&held_list);
1958
1959         daemonize();
1960
1961         lock_kernel();
1962         tsk->session = 1;
1963         tsk->pgrp = 1;
1964         sigfillset(&tsk->blocked);
1965         tsk->tty = NULL;    /* get rid of controlling tty */
1966         tsk->policy = SCHED_OTHER;
1967         set_user_nice(tsk, -19);
1968         banner[ sizeof("mxtclr0") - 2 ] += myid;
1969         strcpy(tsk->comm, banner );
1970         cmp_eatmem_task[myid] = tsk;
1971         /* bind to a processor */
1972         tsk->cpus_allowed &= cpu_logical_map(1<<myid);
1973         unlock_kernel();
1974
1975         while(1) {
1976                 long nr;
1977
1978                 interruptible_sleep_on(&cmp_eatmem_wait);
1979
1980                 if ( held < 0 )
```

```
1981                            MCPRINTK("<MC> sanity check: held = %ld\n", held);
1982
1983            more_pages_to_alloc:
1984
1985                    nr = nr_pages_per_thread;
1986
1987                    if ( ! nr ) {
1988                            /* free all of them */
1989                            count = release_cleared(0, &held_list, 1);
1990                            held −= count;
1991                    } else if ( nr > held ) {
1992                            /* allocate and clear pages */
1993                            count=alloc_clear(nr−held, &held_list);
1994                            held += count;
1995                    }
1996                    else {
1997                            /* free some of them */
1998                            count = release_cleared(held−nr, &held_list, 0);
1999                            held −= count;
2000                    }
2001
2002                    /* nr_pages_per_thread changed while we were releasing
2003                       or allocating.  or we didn't get enough
2004                       go at it again */
2005                    if (nr != nr_pages_per_thread || held < nr_pages_per_thread)
2006                            goto more_pages_to_alloc;
2007            }
2008    }
2009
2010    static int cmp_eatmem_init(void)
2011    {
2012            int t;
2013
2014            if ( !mxt_compression_enabled )
2015                    return 0;
2016
2017            low_mem_watermarks();
2018            printk("<MC> Memory watermarks min,low,high=%d,%d,%d\n",
2019                   pages_min,
2020                   pages_low,
2021                   pages_high);
2022
2023            printk("<MC> Starting page eater threads: %d\n", smp_num_cpus);
2024            for (t=0; t < smp_num_cpus; t++) {
2025                    kernel_thread(cmp_eatmem, (void*)t,
2026                                  CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
2027            }
2028            return 0;
2029    }
2030
2031
2032    static void wake_up_cmp_eatmem(void)
2033    {
2034            /* this should wake up all threads waiting
2035             * on this queue */
2036            if ( waitqueue_active( &cmp_eatmem_wait ) )
2037                    wake_up_interruptible_all( &cmp_eatmem_wait );
2038    }
2039
2040
2041    /* ---------------------- Section 10 ------------------------------ */
2042    /*
2043     * Predict processes causing compression pressure.
2044     * Based on oom_kill.c
2045     */
2046
2047    static inline int is_heavy_process(struct task_struct *p)
2048    {
2049            unsigned long th;
2050
2051            /* kernel processes should not be stalled */
2052            if ( ! p->mm )
2053                    return 0;
2054
2055            /* L3 contents is not accounted for in the
2056             * compressed memory utilization;
2057             * It means a 32 MB spike in the worst case L3 flush
2058             */
2059            th = L3_size;
2060
2061            if ( p->mm->total_vm > th )
2062                    return 1;
2063
2064            return 0;
2065    }
2066
2067    /* Make sure that this task is not considered
2068     * in the next round of scheduling.
2069     */
2070    static inline void force_reschedule(struct task_struct *tsk)
```

```
{
        if ( tsk->time_slice )
                tsk->time_slice = 0;
}


/*
 * Steal some scheduling ticks from bad processes. We need to call this
 * repeatedly in every poll. It seems like a waste but we're in the danger
 * zone and we're intentionally wasting CPU cycles with cmp_idle threads
 * anyway. I can also suspend processes, but then I need to remember
 * which processes I suspended later and then what happens if the process
 * died. This method is stateless.
 */
static void zap_bad_processes(void)
{
        struct task_struct *p = NULL;

        read_lock(&tasklist_lock);
        for_each_task(p) {

                if ( ! p->pid )
                        continue;

                if ( ! is_heavy_process(p) )
                        continue;

                force_reschedule(p);
        }
        read_unlock(&tasklist_lock);
}

/* ---------------------- Section 11 ------------------------------ */

/* These routines are for covering the worst case condition of
 * incompressible kernel pages.  Discussion per Alan Cox on 5/22/2001.
 * For every non-user page (!= GFP_HIGHUSER or GFP_USER) we will allocate
 * and hide away one page.  The hidden page will be cleared.
 * This ensures that there is a matching cleared page for every
 * incompressible page.  Therefore maintaining 2 to 1 compression ratio.
 */
#ifdef CONFIG_MXT_HIDDEN_PAGES
static int cmp_hide_pages(void *cpu_num)
{
        struct task_struct *tsk = current;
        struct page *page;
        char banner[] = "mxtrsrv";
        LIST_HEAD(held_list);

        INIT_LIST_HEAD(&held_list);

        daemonize();

        lock_kernel();
        tsk->session = 1;
        tsk->pgrp = 1;
        sigfillset(&tsk->blocked);
        tsk->tty = NULL;    /* get rid of controlling tty */
        tsk->policy = SCHED_OTHER;
        set_user_nice(tsk, 0);
        strcpy(tsk->comm, banner );
        unlock_kernel();

        set_current_state(TASK_UNINTERRUPTIBLE);
        schedule_timeout( HZ * 60 );

        while(1) {
                int m,n;

                interruptible_sleep_on(&cmp_hide_pages_wait);

                m = atomic_read(&nr_hide_pages);
                n = atomic_read(&nr_hidden_pages);

                m = _MIN(m, kernel_reserve);

                if ( m > n ) {
                        int i;
                        for(i=0; i<(m-n); i++) {
                                page = alloc_pages(GFP_HIGHUSER,0);
                                list_add( &page->list, &held_list );
                                atomic_inc(&nr_hidden_pages);

                                if ( fclr )
                                        mxt_clear_page(page);
                                else
                                        clear_highpage(page);

                                if ( current->need_resched ) {
                                        set_current_state(TASK_RUNNING);
```

```
2161                                            yield();
2162                                    }
2163                            }
2164                    }
2165                    else {
2166                            int i;
2167                            for(i=0; i<(n−m); i++) {
2168
2169                                    if ( list_empty(&held_list) )
2170                                            break;
2171
2172                                    page = list_entry(held_list.next,
2173                                                      struct page, list);
2174                                    list_del_init( &page−>list );
2175                                    __free_page(page);
2176                                    atomic_dec(&nr_hidden_pages);
2177
2178                                    if ( current−>need_resched ) {
2179                                            set_current_state(TASK_RUNNING);
2180                                            yield();
2181                                    }
2182                            }
2183                    }
2184            }
2185    }
2186
2187    static int cmp_hide_pages_init(void)
2188    {
2189            if ( ! mxt_compression_enabled )
2190                    return 0;
2191
2192            printk("<MC> Starting kernel page reserving thread\n");
2193
2194            kernel_thread(cmp_hide_pages, 0,
2195                          CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
2196            return 0;
2197    }
2198
2199    /*
2200     * use before module unload
2201     */
2202    static void release_hidden_pages(void)
2203    {
2204            printk("<MC> release_hidden_pages: enter\n");
2205
2206            atomic_set(&nr_hide_pages, 0);
2207            wake_up_hide_pages();
2208            set_current_state(TASK_RUNNING);
2209            yield();
2210    }
2211    #else
2212    static void release_hidden_pages(void) {;}
2213    static int cmp_hide_pages_init(void) {;}
2214    #endif
2215
2216    void wake_up_hide_pages(void)
2217    {
2218    #ifdef CONFIG_MXT_HIDDEN_PAGES
2219            if ( waitqueue_active(  &cmp_hide_pages_wait ) )
2220                    wake_up_interruptible_all( &cmp_hide_pages_wait );
2221    #endif
2222    }
2223
2224
2225    /* ---------------------- Section 100 ------------------------------ */
2226
2227    /*
2228     *   Module initialization
2229     */
2230    static const char copyrite[] = "MXT Driver, Copyright (C) 1999−2001, IBM";
2231
2232    static int __init mxt_init(void)
2233    {
2234            int status;
2235
2236            spin_lock_init(&fast_page_lock);
2237            spin_lock_init(&mc_lock);
2238
2239            status = mxt_scan_bios();
2240            if ( status )
2241                    return status;
2242
2243            printk("%s\n", copyrite);
2244            mxt_setup_vm();
2245            mxt_register_sysctl();
2246            mxt_setup_interrupts();
2247            mxt_hook_clear_page();
2248            cmp_idle_init();
2249            cmp_eatmem_init();
2250            cmp_hide_pages_init();
```

```
2251
2252            return 0;
2253    }
2254
2255    static void __exit mxt_exit(void)
2256    {
2257            printk("<MC> mxt_exit: enter\n");
2258            if ( ! mxt_bios_found )
2259                    return;
2260            mxt_unregister_sysctl();
2261            mxt_unhook_clear_page();
2262            stop_periodic_task();
2263            mxt_disconnect_interrupts();
2264            release_held_pages();
2265            release_hidden_pages();
2266            cmp_release_cpus();
2267            mxt_cleanup_vm();
2268            printk("<MC> mxt_exit: exit\n");
2269    }
2270
2271    module_init(mxt_init);
2272    module_exit(mxt_exit);
2273
2274    MODULE_DESCRIPTION("MXT Compressed Memory Manager");
2275    MODULE_AUTHOR("Bulent Abali <abali@us.ibm.com>");
2276    MODULE_PARM(mxt_printk, "i");
2277    MODULE_PARM_DESC(mxt_printk, "Enable debug mode");
```