

[PATCH 6/11] 2.6.17-rc5 perfmon2 patch for review: new i386 files

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-05/msg08497.html>

- *From:* Stephane Eranian <eranian@xxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Wed, 31 May 2006 06:52:30 -0700
-

This files contains the new files for i386.

```
--- linux-2.6.17-rc5.orig/arch/i386/perfmon/Kconfig 1969-12-31 16:00:00.000000000 -0800
+++ linux-2.6.17-rc5/arch/i386/perfmon/Kconfig 2006-05-30 02:48:12.000000000 -0700
@@ -0,0 +1,36 @@
+config PERFMON_P6
+ tristate "Support for P6/Pentium M processor hardware performance counters"
+ depends on PERFMON
+ default m
+ help
+ Enables support for the P6-style hardware performance counters.
+ To be used for P6 processors (Pentium III, PentiumPro) and also
+ for Pentium M.
+ If unsure, say M.
+
+config PERFMON_P4
+ tristate "Support for 32-bit P4/Xeon hardware performance counters"
+ depends on PERFMON
+ default m
+ help
+ Enables support for the 32-bit P4/Xeon style hardware performance
+ counters.
+ If unsure, say M.
+
+config PERFMON_GEN_IA32
+ tristate "Support for the architected IA-32 PMU"
+ depends on PERFMON
+ default m
+ help
+ Enables support for the architected IA-32 hardware performance counters.
+ You need a Core Duo/Solo processor or newer for this work.
+ If unsure, say M.
+
+config PERFMON_P4_PEBS
+ tristate "Support for Intel P4 PEBS sampling format"
```

```
+ depends on PERFMON_P4
+ default m
+ help
+ Enables support for Precise Event-Based Sampling (PEBS) on the Intel P4
+ processors which support it. Does not work with P6 processors.
+ If unsure, say m.
--- linux-2.6.17-rc5.orig/arch/i386/perfmon/Makefile 1969-12-31 16:00:00.000000000 -0800
+++ linux-2.6.17-rc5/arch/i386/perfmon/Makefile 2006-05-30 02:48:12.000000000 -0700
@@ -0,0 +1,9 @@
+#
+# Copyright (c) 2005-2006 Hewlett-Packard Development Company, L.P.
+# Contributed by Stephane Eranian <eranian@xxxxxxxxxxx>
+#
+obj-$(CONFIG_PERFMON) += perfmon.o
+obj-$(CONFIG_PERFMON_P6) += perfmon_p6.o
+obj-$(CONFIG_PERFMON_P4) += perfmon_p4.o
+obj-$(CONFIG_PERFMON_GEN_IA32) += perfmon_gen_ia32.o
+obj-$(CONFIG_PERFMON_P4_PEBS) += perfmon_p4_pebs_smpl.o
--- linux-2.6.17-rc5.orig/arch/i386/perfmon/perfmon.c 1969-12-31 16:00:00.000000000 -0800
+++ linux-2.6.17-rc5/arch/i386/perfmon/perfmon.c 2006-05-30 02:48:12.000000000 -0700
@@ -0,0 +1,1148 @@
+/*
+ * This file implements the IA-32/X86-64/EM64T specific
+ * support for the perfmon2 interface
+ *
+ * Copyright (c) 2005-2006 Hewlett-Packard Development Company, L.P.
+ * Contributed by Stephane Eranian <eranian@xxxxxxxxxxx>
+ */
+#include <linux/interrupt.h>
+#include <linux/perfmon.h>
+
+#define MSR_IA32_PEBS_ENABLE 0x3f1 /* unique per-thread */
+#define MSR_IA32_DS_AREA 0x600 /* unique per-thread */
+
+DEFINE_PER_CPU(int, apic_state);
+DEFINE_PER_CPU(u64 *, nmi_pmds);
+DEFINE_PER_CPU(u64 *, nmi_pmcs);
+
+#ifdef __i386__
+#define __pfm_wrmsrl(a, b) wrmsrl((a), (b), 0)
+#else
+#define __pfm_wrmsrl(a, b) wrmsrl((a), (b))
+#endif
+
+/*
+ * Debug Store (DS) management area for P4/Xeon/EM64T PEBS
+ */
+struct pfm_ds_area {
+ unsigned long bts_buf_base;
+ unsigned long bts_index;
+ unsigned long bts_abs_max;
```

```

+ unsigned long bts_intr_thres;
+ unsigned long pebs_buf_base;
+ unsigned long pebs_index;
+ unsigned long pebs_abs_max;
+ unsigned long pebs_intr_thres;
+ u64 pebs_cnt_reset;
+};
+
+static void (*pfm_stop_active)(struct task_struct *task,
+ struct pfm_context *ctx,
+ struct pfm_event_set *set);
+
+static inline void pfm_set_pce(void)
+{
+ write_cr4(read_cr4() | X86_CR4_PCE);
+}
+
+static inline void pfm_clear_pce(void)
+{
+ write_cr4(read_cr4() & ~X86_CR4_PCE);
+}
+
+static u8 get_smt_id_mask(void)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ u8 mask = 0;
+ u8 cnt;
+
+ cnt = arch_info->lps_per_core;
+
+ /* TODO : replace this with the algorithm from 7.10.3 as
+ * soon as I figure out this bizarre GCC/AT&T
+ * inline assembler
+ */
+ if (likely(cnt <= 2))
+ mask = 0x01;
+ else if (unlikely(cnt<=4))
+ mask = 0x03;
+ else if (unlikely(cnt<=8))
+ mask = 0x07;
+ else if (unlikely(cnt<=16))
+ mask = 0x0F;
+
+ return mask;
+}
+
+/*
+ * returns the logical processor id (0 or 1) if running on an HT system
+ */
+static u8 get_smt_id(void)
+{

```

```

+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ u8 apic_id ;
+
+ /*
+ * not HT enabled
+ */
+ if (arch_info->lps_per_core <= 1)
+ return 0;
+
+ /*
+ * TODO : add per_cpu caching here to avoid calling cpuid repeatedly
+ */
+ apic_id = (cpuid_ebx(1)&0xff000000) >> 24;
+ return apic_id & get_smt_id_mask();
+}
+
+void __pfm_write_reg(const struct pfm_arch_ext_reg *xreg, u64 val)
+{
+ u64 pmi;
+ int smt_id = get_smt_id();
+
+ if (unlikely(smt_id > MAX_SMT_ID)) {
+ PFM_ERR("%s: smt_id (%d) exceeds maximum number "
+ "of SMTs (%d) supported by perfmon",
+ __FUNCTION__, smt_id, MAX_SMT_ID);
+ return;
+ }
+
+ /*
+ * LBR TOS is read only
+ * TODO : need to save the TOS value somewhere and move the LBRs
+ * according to where ever it is currently pointing
+ */
+ if (xreg->reg_type & PFM_REGT_LBRTOS)
+ return;
+
+ /*
+ * Adjust for T1 if necessary:
+ *
+ * - move the T0_OS/T0_USR bits into T1 slots
+ * - move the OVF_PMI_T0 bits into T1 slot
+ *
+ * The P4/EM64T T1 is cleared by description table.
+ * User only works with T0.
+ */
+ if (smt_id > 0) {
+ if (xreg->reg_type & PFM_REGT_ESCR) {
+
+ /* copy T0_USR & T0_OS to T1 */
+ val |= ((val & 0xc) >> 2);
+
+

```

```

+ /* clear bits T0_USR & T0_OS */
+ val &= ~0xc;
+
+ } else if (xreg->reg_type & PFM_REGT_CCCR) {
+ pmi = (val >> 26) & 0x1;
+ if (pmi) {
+ val &= ~(1UL<<26);
+ val |= 1UL<<27;
+ }
+ }
+ }
+
+ if (xreg->addr[smt_id])
+ wrmsrl(xreg->addr[smt_id], val);
+ }
+
+ void __pfm_read_reg(const struct pfm_arch_ext_reg *xreg, u64 *val)
+ {
+ int smt_id = get_smt_id();
+
+ *val = 0;
+
+ if (unlikely(smt_id > MAX_SMT_ID)) {
+ PFM_ERR("%s: smt_id (%d) exceeds maximum number "
+ "of SMTs (%d) supported by perfmon",
+ __FUNCTION__, smt_id, MAX_SMT_ID);
+ return;
+ }
+
+ if (xreg->addr[smt_id] != 0) {
+ rdmsrl(xreg->addr[smt_id], *val);
+ /*
+ * move the Tx_OS and Tx_USR bits into
+ * T0 slots setting the T1 slots to zero
+ */
+ if (xreg->reg_type & PFM_REGT_ESCR) {
+
+ if (smt_id > 0)
+ *val |= (((*val) & 0x3) << 2);
+
+ /*
+ * zero out bits that are reserved
+ * (including T1_OS and T1_USR)
+ */
+ *val &= PFM_ESCR_RSVD;
+ }
+ }
+ }
+
+ void pfm_arch_init_percpu(void)
+ {

```

```

+ /*
+ * We initialize APIC with LVTPC vector masked.
+ *
+ * this is motivated by the fact that the PMU may be
+ * in a condition where it has already an interrupt pending
+ * as with boot. Given that we cannot touch the PMU registers
+ * at this point, we may not have a way to remove the condition.
+ * As such, we need to keep the interrupt masked until a PMU
+ * description is loaded. At that point, we can enable intr.
+ */
+ if (nmi_watchdog != NMI_LOCAL_APIC) {
+ apic_write(APIC_LVTPC, APIC_LVT_MASKED|LOCAL_PERFMON_VECTOR);
+ PFM_INFO("CPU%d APIC LVTPC vector masked", smp_processor_id());
+ } else {
+ __get_cpu_var(apic_state) = 0;
+ __get_cpu_var(nmi_pmds) = kmalloc(sizeof(u64)*PFM_MAX_PMDS,
+ GFP_KERNEL);
+ __get_cpu_var(nmi_pmcs) = kmalloc(sizeof(u64)*PFM_MAX_PMCS,
+ GFP_KERNEL);
+ if (__get_cpu_var(nmi_pmds) == NULL ||
+ __get_cpu_var(nmi_pmcs) == NULL)
+ PFM_ERR("CPU%d cannot allocate NMI save area",
+ smp_processor_id());
+ }
+ }
+
+ void pfm_arch_context_initialize(struct pfm_context *ctx, u32 ctx_flags)
+ {
+ struct pfm_arch_context *ctx_arch;
+
+ ctx_arch = pfm_ctx_arch(ctx);
+ ctx_arch->flags = ctx_flags & PFM_X86_FL_INSECURE;
+ }
+
+ /*
+ * function called from pfm_load_context_*. Task is not guaranteed to be
+ * current task. If not then other task is guaranteed stopped and off any CPU.
+ * context is locked and interrupts are masked.
+ *
+ * On PFM_LOAD_CONTEXT, the interface guarantees monitoring is stopped.
+ *
+ * For system-wide task is NULL
+ */
+ int pfm_arch_load_context(struct pfm_context *ctx, struct task_struct *task)
+ {
+ struct pfm_arch_context *ctx_arch;
+
+ ctx_arch = pfm_ctx_arch(ctx);
+
+ /*
+ * always authorize user level rdpmc for self-monitoring

```

```

+ */
+ if (task == current || ctx->flags.system) {
+ /*
+ * for system-wide or self-monitoring, we always enable
+ * rdpmc at user level
+ */
+ ctx_arch->flags |= PFM_X86_FL_INSECURE;
+
+ pfm_set_pce();
+ PFM_DBG("setting cr4.pce (rdpmc authorized at user level)");
+ }
+ return 0;
+ }
+
+ /*
+ * function called from pfm_unload_context_*. Context is locked.
+ * interrupts are masked. task is not guaranteed to be current task.
+ * Access to PMU is not guaranteed.
+ *
+ * function must do whatever arch-specific action is required on unload
+ * of a context.
+ *
+ * called for both system-wide and per-thread. task is NULL for ssystem-wide
+ */
+ void pfm_arch_unload_context(struct pfm_context *ctx, struct task_struct *task)
+ {
+ struct pfm_arch_context *ctx_arch;
+
+ ctx_arch = pfm_ctx_arch(ctx);
+
+ if (ctx_arch->flags & PFM_X86_FL_INSECURE) {
+ pfm_clear_pce();
+ PFM_DBG("clearing cr4.pce");
+ }
+ }
+
+ /*
+ * called from __pfm_interrupt_handler(). ctx is not NULL.
+ * ctx is locked. PMU interrupt is masked.
+ *
+ * must stop all monitoring to ensure handler has consistent view.
+ * must collect overflowed PMDs bitmask into povfls_pmds and
+ * npend_ovfls. If no interrupt detected then npend_ovfls
+ * must be set to zero.
+ */
+ void pfm_arch_intr_freeze_pmu(struct pfm_context *ctx)
+ {
+ struct pfm_arch_pmu_info *arch_info;
+ struct pfm_arch_context *ctx_arch;
+ struct pfm_event_set *set;
+ struct pfm_ds_area *ds;

```

```

+
+ arch_info = pfm_pmu_conf->arch_info;
+ ctx_arch = pfm_ctx_arch(ctx);
+
+ set = ctx->active_set;
+
+ /*
+ * stop active monitoring and collect overflow information
+ */
+ pfm_stop_active(current, ctx, set);
+
+ /*
+ * PMU is stopped, thus PEBS is stopped already
+ * on PEBS full interrupt, the IQ_CCCR4 counter does
+ * not have the OVF bit set. Thus we use the pebs index
+ * to detect overflow. This is required because we may
+ * have more than one reason for overflow due to 64-bit
+ * counter virtualization.
+ *
+ * We don't actually keep track of the overflow unless
+ * IQ_CTR4 is actually used.
+ *
+ * With HT enabled, the mappings are such that IQ_CTR4 and IQ_CTR5
+ * are mapped onto the same PMD registers.
+ *
+ * XXX: remove assumption on IQ_CTR4 and IQ_CTR5 being mapped onto
+ * the same PMD.
+ */
+ if (ctx_arch->ds_area) {
+ ds = ctx_arch->ds_area;
+ if (ds->pebs_index >= ds->pebs_intr_thres
+ && pfm_bv_isset(set->used_pmds, arch_info->pebs_ctr_idx)) {
+ pfm_bv_set(set->povfl_pmds, arch_info->pebs_ctr_idx);
+ set->npend_ovfls++;
+ }
+ }
+ }
+
+ /*
+ * unfreeze PMU from pfm_do_interrupt_handler()
+ * ctx may be NULL for spurious
+ */
+void pfm_arch_intr_unfreeze_pmu(struct pfm_context *ctx)
+{
+ struct pfm_arch_context *ctx_arch;
+
+ if (ctx == NULL)
+ return;
+
+ ctx_arch = pfm_ctx_arch(ctx);
+
+

```

```

+ pfm_arch_restore_pmcs(ctx, ctx->active_set);
+
+ /*
+ * reload DS area pointer because it is cleared by
+ * pfm_stop_active()
+ */
+ if (ctx_arch->ds_area) {
+ __pfm_wrmsrl(MSR_IA32_DS_AREA, ctx_arch->ds_area);
+ PFM_DBG("restoring DS");
+ }
+ }
+
+ /*
+ * Called from pfm_ctxswin_*. Task is guaranteed to be current.
+ * set cannot be NULL. Context is locked. Interrupts are masked.
+ * Caller has already restored all PMD and PMC registers.
+ *
+ * must reactivate monitoring
+ */
+void pfm_arch_ctxswin(struct task_struct *task, struct pfm_context *ctx,
+ struct pfm_event_set *set)
+{
+ struct pfm_arch_context *ctx_arch;
+
+ ctx_arch = pfm_ctx_arch(ctx);
+
+ /*
+ * nothing to do for system-wide
+ */
+ if (ctx->flags.system)
+ return;
+
+ if (ctx_arch->flags & PFM_X86_FL_INSECURE) {
+ pfm_set_pce();
+ PFM_DBG("setting cr4.pce");
+ }
+
+ /*
+ * reload DS management area pointer. Pointer
+ * not managed as a PMC thus it is not restored
+ * with the rest of the registers.
+ */
+ if (ctx_arch->ds_area) {
+ __pfm_wrmsrl(MSR_IA32_DS_AREA, ctx_arch->ds_area);
+ PFM_DBG("restored DS");
+ }
+ }
+
+static void __pfm_stop_active_p6(struct task_struct *task,
+ struct pfm_context *ctx,
+ struct pfm_event_set *set)

```

```

+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ struct pfm_arch_ext_reg *xrc, *xrd;
+ unsigned int i, max;
+ u64 val, wmask;
+
+ max = pfm_pmu_conf->max_pmc;
+ xrc = arch_info->pmc_addrs;
+ xrd = arch_info->pmd_addrs;
+ wmask = PFM_ONE_64 << pfm_pmu_conf->counter_width;
+
+ /*
+ * clear enable bits
+ */
+ for (i = 0; i < max; i++) {
+ if (pfm_bv_isset(set->used_pmcs, i))
+ __pfm_wrmsrl(xrc[i].addrs[0], 0);
+ }
+
+ if (set->npend_ovfls)
+ return;
+
+ /*
+ * check for pending overflows
+ */
+ max = pfm_pmu_conf->max_cnt_pmd;
+ for (i = 0; i < max; i++) {
+ if (pfm_bv_isset(set->used_pmds, i)) {
+ rdmsrl(xrd[i].addrs[0], val);
+ if ((val & wmask) == 0) {
+ pfm_bv_set(set->povfl_pmds, i);
+ set->npend_ovfls++;
+ }
+ }
+ }
+ }
+ }
+ }
+
+ /*
+ * stop active set only
+ */
+static void __pfm_stop_active_p4(struct task_struct *task,
+ struct pfm_context *ctx,
+ struct pfm_event_set *set)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ struct pfm_arch_context *ctx_arch;
+ struct pfm_arch_ext_reg *xrc, *xrd;
+ u64 enable_mask[PFM_PMC_BV];
+ unsigned int i, max_pmc;
+ u64 cccr, ctr1, ctr2;

```

```

+
+ ctx_arch = pfm_ctx_arch(ctx);
+ max_pmc = pfm_pmu_conf->max_pmc;
+ xrc = arch_info->pmc_addrs;
+ xrd = arch_info->pmd_addrs;
+
+ bitmap_and(ulp(enable_mask),
+ ulp(set->used_pmcs),
+ ulp(arch_info->enable_mask),
+ max_pmc);
+
+ /*
+ * stop PEBS and clear DS area pointer
+ */
+ if (ctx_arch->ds_area) {
+ __pfm_wrmsrl(MSR_IA32_PEBS_ENABLE, 0);
+ __pfm_wrmsrl(MSR_IA32_DS_AREA, 0);
+ }
+
+ /*
+ * ensures we do not destroy information collected
+ * at ctxswout. This function is called from
+ * pfm_arch_intr_freeze_pmu() as well.
+ */
+ if (set->npend_ovfls) {
+ for (i = 0; i < max_pmc; i++) {
+ if (pfm_bv_isset(enable_mask, i)) {
+ __pfm_write_reg(xrc+i, 0);
+ }
+ }
+ }
+ return;
+ }
+
+ for (i = 0; i < max_pmc; i++) {
+
+ if (pfm_bv_isset(enable_mask, i)) {
+
+ /* read counter controlled by PMC */
+ __pfm_read_reg(xrd+(xrc[i].ctr), &ctr1);
+
+ /* read PMC value */
+ __pfm_read_reg(xrc+i, &cccr);
+
+ /* clear CCCR value (destroy OVF) */
+ __pfm_write_reg(xrc+i, 0);
+
+ /* read counter controlled by CCCR again */
+ __pfm_read_reg(xrd+(xrc[i].ctr), &ctr2);
+
+ /*
+ * there is an overflow if either:

```

```

+ * - CCCR.ovf is set (and we just cleared it)
+ * - ctr2 < ctr1
+ * in that case we set the bit corresponding to the
+ * overflowed PMD in povfl_pmds.
+ */
+ if ((cccr & (PFM_ONE_64<<31)) || (ctr2 < ctr1)) {
+ pfm_bv_set(set->povfl_pmds, xrc[i].ctr);
+ set->npend_ovfls++;
+ }
+ }
+ }
+ }
+
+ /*
+ * Called from pfm_stop() and pfm_ctxswin_*() when idle
+ * task and EXCL_IDLE is on.
+ *
+ * Interrupts are masked. Context is locked. Set is the active set.
+ *
+ * For per-thread:
+ * task is not necessarily current. If not current task, then
+ * task is guaranteed stopped and off any cpu. Access to PMU
+ * is not guaranteed. Interrupts are masked. Context is locked.
+ * Set is the active set.
+ *
+ * For system-wide:
+ * task is current
+ *
+ * must disable active monitoring.
+ */
+ void pfm_arch_stop(struct task_struct *task, struct pfm_context *ctx,
+ struct pfm_event_set *set)
+ {
+ if (task != current)
+ return;
+
+ pfm_stop_active(task, ctx, set);
+ }
+
+ /*
+ * Called from pfm_ctxswout_*(). Task is guaranteed to be current.
+ * Context is locked. Interrupts are masked. Monitoring is active.
+ * PMU access is guaranteed. PMC and PMD registers are live in PMU.
+ *
+ * for per-thread:
+ * must stop monitoring for the task
+ * for system-wide:
+ * must ensure task has monitoring stopped. But monitoring may continue
+ * on the current processor
+ */
+ void pfm_arch_ctxswout(struct task_struct *task, struct pfm_context *ctx,

```

```

+ struct pfm_event_set *set)
+{
+ struct pfm_arch_context *ctx_arch;
+
+ ctx_arch = pfm_ctx_arch(ctx);
+
+ /*
+ * nothing to do in system-wide
+ */
+ if (ctx->flags.system)
+ return;
+
+ if (ctx_arch->flags & PFM_X86_FL_INSECURE) {
+ pfm_clear_pce();
+ PFM_DBG("clearing cr4.pce");
+ }
+
+ /*
+ * disable lazy restore of PMCS on ctxswin because
+ * we modify some of them.
+ */
+ set->priv_flags |= PFM_SETFL_PRIV_MOD_PMCS;
+
+ pfm_stop_active(task, ctx, set);
+}
+
+ /*
+ * called from pfm_start() or pfm_ctxswout_sys() when idle task and
+ * EXCL_IDLE is on.
+ *
+ * Interrupts are masked. Context is locked. Set is the active set.
+ *
+ * For per-thread:
+ * Task is not necessarily current. If not current task, then task
+ * is guaranteed stopped and off any cpu. Access to PMU is not guaranteed.
+ *
+ * For system-wide:
+ * task is always current
+ *
+ * must enable active monitoring.
+ */
+static void __pfm_arch_start(struct task_struct *task, struct pfm_context *ctx,
+ struct pfm_event_set *set)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ struct pfm_arch_context *ctx_arch;
+ struct pfm_arch_ext_reg *xregs;
+ u64 *impl_mask;
+ u16 i, max_pmc;
+
+ if (task != current)

```

```

+ return;
+
+ ctx_arch = pfm_ctx_arch(ctx);
+ max_pmc = pfm_pmu_conf->max_pmc;
+ impl_mask = pfm_pmu_conf->impl_pmcs;
+
+ xregs = arch_info->pmc_addrs;
+
+ /*
+ * we must actually install all implemented pmcs registers because
+ * until started, we do not touch any PMC registers. On P4, touching
+ * only the CCCR (which have the enable field) is not enough. On P6/AMD,
+ * all PMCs have an enable bit, so this is not worse.
+ */
+ for (i = 0; i < max_pmc; i++) {
+ if (pfm_bv_isset(impl_mask, i))
+ __pfm_write_reg(xregs+i, set->pmcs[i]);
+ }
+
+ /*
+ * reload DS area pointer. PEBS_ENABLE is restored with the PMCs
+ * in pfm_restore_pmcs(). PEBS_ENABLE is not considered part of
+ * the set of PMCs with an enable bit. This is reserved for counter
+ * PMC, i.e., CCCR.
+ */
+ if (task == current && ctx_arch->ds_area) {
+ __pfm_wrmsrl(MSR_IA32_DS_AREA, ctx_arch->ds_area);
+ PFM_DBG("restoring DS");
+ }
+ }
+
+ void pfm_arch_start(struct task_struct *task, struct pfm_context *ctx,
+ struct pfm_event_set *set)
+ {
+ /*
+ * mask/unmask uses start/stop mechanism, so we cannot allow
+ * while masked.
+ */
+ if (ctx->state == PFM_CTX_MASKED)
+ return;
+
+ __pfm_arch_start(task, ctx, set);
+ }
+
+ /*
+ * function called from pfm_switch_sets(), pfm_context_load_thread(),
+ * pfm_context_load_sys(), pfm_ctxswin_*(), pfm_switch_sets()
+ * context is locked. Interrupts are masked. set cannot be NULL.
+ * Access to the PMU is guaranteed.
+ *
+ * function must restore all PMD registers from set.

```

```

+ */
+void pfm_arch_restore_pmds(struct pfm_context *ctx, struct pfm_event_set *set)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ struct pfm_arch_ext_reg *xregs;
+ u64 ovfl_mask, val, *pmds;
+ u64 *impl_rw_mask, *cnt_mask;
+ u16 i, max_rw_pmd;
+
+ max_rw_pmd = pfm_pmu_conf->max_rw_pmd;
+ cnt_mask = pfm_pmu_conf->cnt_pmds;
+ ovfl_mask = pfm_pmu_conf->ovfl_mask;
+ impl_rw_mask = pfm_pmu_conf->impl_rw_pmds;
+ pmds = set->view->set_pmds;
+
+ xregs = arch_info->pmd_addrs;
+
+ /*
+ * must restore all pmds to avoid leaking
+ * especially when PFM_X86_FL_INSECURE is set.
+ *
+ * XXX: should check PFM_X86_FL_INSECURE==0 and use used_pmd instead
+ */
+ for (i = 0; i < max_rw_pmd; i++) {
+ if (likely(pfm_bv_isset(impl_rw_mask, i))) {
+ val = pmds[i];
+ if (likely(pfm_bv_isset(cnt_mask, i)))
+ val |= ~ovfl_mask;
+ __pfm_write_reg(xregs+i, val);
+ }
+ }
+ }
+
+ /*
+ * function called from pfm_switch_sets(), pfm_context_load_thread(),
+ * pfm_context_load_sys(), pfm_ctxswin_*.
+ * Context is locked. Interrupts are masked. set cannot be NULL.
+ * Access to the PMU is guaranteed.
+ *
+ * function must restore all PMC registers from set, if needed.
+ */
+void pfm_arch_restore_pmcs(struct pfm_context *ctx, struct pfm_event_set *set)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ struct pfm_arch_ext_reg *xregs;
+ u64 *impl_pmcs;
+ u16 i, max_pmc;
+
+ max_pmc = pfm_pmu_conf->max_pmc;
+ xregs = arch_info->pmc_addrs;
+ impl_pmcs = pfm_pmu_conf->impl_pmcs;

```

```

+
+ /*
+ * - by default, no PMC measures anything
+ * - on ctxswout, all used PMCs are disabled (cccr cleared)
+ *
+ * we need to restore the PMC (incl enable bits) only if
+ * not masked and user issued pfm_start()
+ */
+ if (ctx->state == PFM_CTX_MASKED || ctx->flags.started == 0)
+ return;
+
+ /*
+ * restore all pmcs and not just the ones that are used
+ * to avoid using stale registers
+ */
+ for (i = 0; i < max_pmc; i++)
+ if (pfm_bv_isset(impl_pmcs, i))
+ __pfm_write_reg(xregs+i, set->pmcs[i]);
+ }
+
+ /*
+ * function called from pfm_mask_monitoring(), pfm_switch_sets(),
+ * pfm_ctxswout_thread(), pfm_flush_pmds().
+ * context is locked. interrupts are masked. the set argument cannot
+ * be NULL. Access to PMU is guaranteed.
+ *
+ * function must saved PMD registers into set save area pmds[]
+ */
+ void pfm_arch_save_pmds(struct pfm_context *ctx, struct pfm_event_set *set)
+ {
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ struct pfm_arch_ext_reg *xregs;
+ u64 hw_val, *pmds, ovfl_mask;
+ u64 *used_mask, *cnt_mask;
+ u16 i, max_pmd;
+
+ ovfl_mask = pfm_pmu_conf->ovfl_mask;
+ max_pmd = pfm_pmu_conf->max_pmd;
+ cnt_mask = pfm_pmu_conf->cnt_pmds;
+ used_mask = set->used_pmds;
+ pmds = set->view->set_pmds;
+
+ xregs = arch_info->pmd_addrs;
+
+ /*
+ * save all used pmds
+ */
+ for (i = 0; i < max_pmd; i++) {
+ if (pfm_bv_isset(used_mask, i)) {
+ __pfm_read_reg(xregs+i, &hw_val);
+ if (likely(pfm_bv_isset(cnt_mask, i)))

```

```

+ hw_val = (pmds[i] & ~ovfl_mask) |
+ (hw_val & ovfl_mask);
+ pmds[i] = hw_val;
+ }
+ }
+}
+
+fastcall void smp_pmu_interrupt(struct pt_regs *regs)
+{
+ ack_APIC_irq();
+ irq_enter();
+ pfm_interrupt_handler(LOCAL_PERFMON_VECTOR, NULL, regs);
+ irq_exit();
+ /*
+ * On Intel P6, Pentium M, P4, EM64T:
+ * - it is necessary to clear the MASK field for the LVTPC
+ * vector. Otherwise interrupts remain masked. See
+ * section 8.5.1
+ * AMD X8-64:
+ * - the documentation does not stipulate the behavior.
+ * To be safe, we also rewrite the vector to clear the
+ * mask field
+ *
+ * We only clear the mask field, if there is a PMU description
+ * loaded. Otherwise we would have a problem because without
+ * PMU description we cannot access PMu registers to clear the
+ * overflow condition and may end up in a flood of PMU interrupts.
+ *
+ * The APIC vector is initialized as masked, but we may already
+ * have a pending PMU overflow by the time we get to
+ * pfm_arch_init_percpu(). Such interrupt would generate a call
+ * to this function, which would undo the masking and would
+ * cause a flood.
+ */
+ if (pfm_pmu_conf)
+ apic_write(APIC_LVTPC, LOCAL_PERFMON_VECTOR);
+}
+
+asmlinkage void pmu_interrupt(void);
+void pfm_vector_init(void)
+{
+ set_intr_gate(LOCAL_PERFMON_VECTOR, (void *)pmu_interrupt);
+ PFM_INFO("installed CPU%d gate", smp_processor_id());
+}
+
+static void __pfm_stop_one_pmu(void *dummy)
+{
+ struct pfm_arch_pmu_info *arch_info;
+ struct pfm_arch_ext_reg *xregs;
+ unsigned int i, num_pmcs;
+}

```

```

+ /*
+ * if NMI watchdog is using Local APIC, then
+ * counters are already initialized to a decent
+ * state
+ */
+ if (nmi_watchdog == NMI_LOCAL_APIC)
+ return;
+
+ PFM_DBG("stopping on CPU%d: LVT=0x%x",
+ smp_processor_id(),
+ (unsigned int)apic_read(APIC_LVTPC));
+
+ num_pmcs = pfm_pmu_conf->num_pmcs;
+ arch_info = pfm_pmu_conf->arch_info;
+ xregs = arch_info->pmc_addrs;
+
+ for (i = 0; i < num_pmcs; i++)
+ if (pfm_bv_isset(arch_info->enable_mask, i))
+ __pfm_write_reg(xregs+i, 0);
+
+ /*
+ * now that we have a PMU description we can deal with spurious
+ * interrupts, thus we can safely re-enable the LVTPC vector
+ * by clearing the mask field
+ */
+ apic_write(APIC_LVTPC, LOCAL_PERFMON_VECTOR);
+ }
+
+ /*
+ * called from pfm_register_pmu_config() after the new
+ * config has been validated and installed. The pfm_session_lock
+ * is held.
+ *
+ * Must sanity check the arch-specific config information
+ *
+ * return:
+ * < 0 : if error
+ * 0 : if success
+ */
+int pfm_arch_pmu_config_check(struct pfm_pmu_config *cfg)
+{
+ struct pfm_arch_pmu_info *arch_info = cfg->arch_info;
+
+ /*
+ * adust stop routine based on PMU model
+ *
+ * P6, Pentium M, AMD X86-64 = P6
+ * P4, Xeon, EM64T = P4
+ */
+ switch(arch_info->pmu_style) {
+ case PFM_X86_PMU_P4:

```

```

+ pfm_stop_active = __pfm_stop_active_p4;
+ break;
+ case PFM_X86_PMU_P6:
+ pfm_stop_active = __pfm_stop_active_p6;
+ break;
+ default:
+ PFM_INFO("unknown pmu_style=%d", arch_info->pmu_style);
+ return -EINVAL;
+ }
+ return 0;
+}
+
+/*
+ * called from pfm_register_pmu_config() after the new
+ * config has been validated and installed. No lock
+ * is held. Interrupts are not masked.
+ *
+ * The role of the function is, based on the PMU description, to
+ * put the PMU into a quiet state on each CPU. This function is
+ * not necessary if there is an architected way of doing this
+ * for a processor family.
+ */
+void pfm_arch_pmu_config_init(void)
+{
+ on_each_cpu(__pfm_stop_one_pmu, NULL, 1, 1);
+}
+
+int pfm_arch_initialize(void)
+{
+ return 0;
+}
+
+void pfm_arch_mask_monitoring(struct pfm_context *ctx)
+{
+ /*
+ * on IA-32 masking/unmasking uses start/stop
+ * mechanism
+ */
+ pfm_arch_stop(current, ctx, ctx->active_set);
+}
+
+void pfm_arch_unmask_monitoring(struct pfm_context *ctx)
+{
+ /*
+ * on IA-32 masking/unmasking uses start/stop
+ * mechanism
+ */
+ __pfm_arch_start(current, ctx, ctx->active_set);
+}
+
+static void pfm_save_pmu_nmi(void)

```

```

+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ struct pfm_arch_ext_reg *xregs;
+ u64 *regs;
+ u64 *impl_mask;
+ u16 i, max_reg;
+
+ impl_mask = pfm_pmu_conf->impl_pmcs;
+ regs = __get_cpu_var(nmi_pmcs);
+ max_reg = pfm_pmu_conf->max_pmc;
+ xregs = arch_info->pmc_addrs;
+
+ if (regs == NULL)
+ return;
+
+ /*
+ * save and clear all implemented pmcs
+ */
+ for (i = 0; i < max_reg; i++) {
+ if (pfm_bv_isset(impl_mask, i)) {
+ __pfm_read_reg(xregs+i, regs+i);
+ __pfm_write_reg(xregs+i, 0);
+ }
+ }
+
+ impl_mask = pfm_pmu_conf->impl_pmds;
+ regs = __get_cpu_var(nmi_pmds);
+ max_reg = pfm_pmu_conf->max_pmd;
+ xregs = arch_info->pmd_addrs;
+
+ if (regs == NULL)
+ return;
+
+ /*
+ * save all implemented pmds
+ */
+ for (i = 0; i < max_reg; i++) {
+ if (pfm_bv_isset(impl_mask, i))
+ __pfm_read_reg(xregs+i, regs+i);
+ }
+ }
+
+static void pfm_restore_pmu_nmi(void)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ struct pfm_arch_ext_reg *xregs;
+ u64 *regs;
+ u64 *impl_mask;
+ u16 i, max_reg;
+
+ impl_mask = pfm_pmu_conf->impl_pmds;
+ regs = __get_cpu_var(nmi_pmds);

```

```

+ max_reg = pfm_pmu_conf->max_pmd;
+ xregs = arch_info->pmd_addrs;
+
+ /*
+ * restore all implemented pmds
+ */
+ for (i = 0; i < max_reg; i++) {
+ if (pfm_bv_isset(impl_mask, i)) {
+ __pfm_write_reg(xregs+i, regs[i]);
+ }
+ }
+
+ impl_mask = pfm_pmu_conf->impl_pmcs;
+ regs = __get_cpu_var(nmi_pmcs);
+ max_reg = pfm_pmu_conf->max_pmc;
+ xregs = arch_info->pmc_addrs;
+
+ /*
+ * restore all implemented pmcs
+ */
+ for (i = 0; i < max_reg; i++) {
+ if (pfm_bv_isset(impl_mask, i)) {
+ __pfm_write_reg(xregs+i, regs[i]);
+ }
+ }
+}
+
+static void
+__pfm_reserve_lapic_nmi(void *data)
+{
+ __get_cpu_var(apic_state) = apic_read(APIC_LVTPC);
+ pfm_save_pmu_nmi();
+ apic_write(APIC_LVTPC, LOCAL_PERFMON_VECTOR);
+}
+
+static void
+__pfm_release_lapic_nmi(void *dummy)
+{
+ /*
+ * apic_state assumed identical on all CPUs
+ */
+ apic_write(APIC_LVTPC, __get_cpu_var(apic_state));
+ pfm_restore_pmu_nmi();
+ __get_cpu_var(apic_state) = 0;
+}
+
+static int
+pfm_reserve_lapic_nmi(void)
+{
+ int ret;
+}

```

```
+ /*
+ * The NMI LAPIC watchdog timer is active on every CPU, so we need
+ * to reserve (disable) on each CPU.
+ */
+ ret = reserve_lapic_nmi();
+ if (ret)
+ return ret;
+
+ /*
+ * save current APIC value for LVTPC entry
+ * we assume APIC_LVTPC is identical on all CPUs
+ */
+ __get_cpu_var(apic_state) = apic_read(APIC_LVTPC);
+
+ #ifdef CONFIG_SMP
+ /*
+ * invoke on all CPU but self
+ */
+ smp_call_function(__pfm_reserve_lapic_nmi, NULL, 0, 1);
+ #endif
+ __pfm_reserve_lapic_nmi(NULL);
+
+ return ret;
+ }
+
+ static void
+ pfm_release_lapic_nmi(void)
+ {
+ unsigned long flags;
+
+ /*
+ * The NMI LAPIC watchdog timer is active on every CPU, so we need
+ * to release on each CPU.
+ *
+ * we may be called with interrupts disabled. It is ok to
+ * re-enable given where in perfmon.c this gets called from.
+ *
+ * XXX: if we clean up perfmon.c some more, this can go away
+ */
+ local_save_flags(flags);
+ local_irq_enable();
+
+ __pfm_release_lapic_nmi(NULL);
+
+ #ifdef CONFIG_SMP
+ /*
+ * invoke on all CPU but self
+ */
+ smp_call_function( __pfm_release_lapic_nmi, NULL, 0, 1);
+ #endif
+
+ }
```

```

+ local_irq_disable();
+
+ release_lapic_nmi();
+
+ local_irq_restore(flags);
+}
+
+/*
+ * Called from pfm_release_session() after release is done.
+ * Holding pfs_sessions lock. Interrupts may be masked.
+ */
+void pfm_arch_release_session(struct pfm_sessions *session,
+ struct pfm_context *ctx,
+ u32 cpu)
+{
+ u32 all_sessions;
+
+ all_sessions = session->pfs_task_sessions + session->pfs_sys_sessions;
+
+ /*
+ * release APIC NMI when last session
+ */
+ if (all_sessions == 0 && (__get_cpu_var(apic_state) & APIC_DM_NMI))
+ pfm_release_lapic_nmi();
+}
+
+/*
+ * Called from pfm_reserve_session() before any actual reservation
+ * is made. Holding pfs_sessions lock. Interrupts are not masked.
+ * Return:
+ * < 0 cannot reserve
+ * 0 successful
+ */
+int pfm_arch_reserve_session(struct pfm_sessions *session,
+ struct pfm_context *ctx,
+ u32 cpu)
+{
+ u32 all_sessions;
+
+ all_sessions = session->pfs_task_sessions + session->pfs_sys_sessions;
+ /*
+ * reserve only when first session
+ */
+ if (all_sessions == 0 && nmi_watchdog == NMI_LOCAL_APIC
+ && pfm_reserve_lapic_nmi() < 0) {
+ PFM_WARN("conflict with NMI");
+ return -EBUSY;
+ }
+ return 0;
+}
+

```

```

+static int has_arch_ia32_pmu(void)
+{
+ unsigned int eax, ebx, ecx, edx;
+
+ if (cpu_data->x86_vendor != X86_VENDOR_INTEL)
+ return 0;
+
+ cpuid(0x0, &eax, &ebx, &ecx, &edx);
+ if (eax < 0xa)
+ return 0;
+
+ cpuid(0xa, &eax, &ebx, &ecx, &edx);
+ return (eax & 0xff) < 1 ? 0 : 1;
+}
+
+char *pfm_arch_get_pmu_module_name(void)
+{
+ switch(cpu_data->x86) {
+ case 6:
+ switch(cpu_data->x86_model) {
+ case 7 ... 11:
+ case 13:
+ return "perfmon_p6";
+ default:
+ return NULL;
+ }
+ case 15:
+ /* All Opteron processors */
+ if (cpu_data->x86_vendor == X86_VENDOR_AMD)
+ return "perfmon_amd";
+
+ switch(cpu_data->x86_model) {
+ case 1: /* Willamette */
+ case 2: /* Northwood */
+ case 4:
+ case 5:
+ /* Foster or EM64T */
+ return "perfmon_p4";
+ }
+ /* FALL THROUGH */
+ default:
+ if (has_arch_ia32_pmu())
+ return "perfmon_gen_ia32";
+ return NULL;
+ }
+ return NULL;
+}
--- linux-2.6.17-rc5.orig/arch/i386/perfmon/perfmon_gen_ia32.c 1969-12-31 16:00:00.000000000 -0800
+++ linux-2.6.17-rc5/arch/i386/perfmon/perfmon_gen_ia32.c 2006-05-30 02:48:12.000000000 -0700
@@ -0,0 +1,263 @@
+/*

```

```

+ * This file contains the IA-32 architected PMU registers description tables
+ * and pmc checker used by perfmon.c.
+ *
+ * This module provides architected support for Core Duo/Solo processors
+ * and newer. Core Duo/solo specific support is provided by another description
+ * module.
+ *
+ * Copyright (c) 2006 Hewlett-Packard Development Company, L.P.
+ * Contributed by Stephane Eranian <eranian@xxxxxxxxxxx>
+ */
+#include <linux/module.h>
+#include <linux/perfmon.h>
+#include <asm/msr.h>
+#include <asm/apic.h>
+
+MODULE_AUTHOR("Stephane Eranian <eranian@xxxxxxxxxxx>");
+MODULE_DESCRIPTION("Generic IA-32 PMU description table");
+MODULE_LICENSE("GPL");
+
+/*
+ * - upper 32 bits are reserved
+ * - INT: APIC enable bit is reserved (forced to 1)
+ * - bit 21 is reserved
+ */
+#define PFM_GEN_IA32_PMC_RSVD ~((0xffffffffULL<<32) \
+ | (PFM_ONE_64<<20) \
+ | (PFM_ONE_64<<21))
+
+/*
+ * force Local APIC interrupt on overflow
+ * disable with NO_EMUL64
+ */
+#define PFM_GEN_IA32_PMC_VAL (PFM_ONE_64<<20)
+#define PFM_GEN_IA32_NO64 (PFM_ONE_64<<20)
+
+/*
+ * architecture specifies that:
+ * IA32_PMCx MSR starts at 0xc1 & occupy a contiguous block of MSR addr
+ * IA32_PERFEVTSELx MSR starts at 0x186 & occupy a contiguous block of MSR addr
+ */
+#define MSR_GEN_PERFEVTSEL_BASE MSR_P6_EVNTSEL0
+#define MSR_GEN_PMC_BASE MSR_P6_PERFCTR0
+
+/*
+ * #define PFM_GEN_IA32_SEL(n) { \
+ * .addrs[0] = MSR_GEN_PERFEVTSEL_BASE+(n), \
+ * .addrs[1] = 0, \
+ * .ctr = n, \
+ * .reg_type = PFM_REGT_PERFSEL}
+
+ * #define PFM_GEN_IA32_CTR(n) { \
+ * .addrs[0] = MSR_GEN_PMC_BASE+(n), \

```

```

+ .addrs[1] = 0, \
+ .ctr = n, \
+ .reg_type = PFM_REGT_CTRL}
+
+struct pmu_eax {
+ unsigned int version:8;
+ unsigned int num_cnt:8;
+ unsigned int cnt_width:8;
+ unsigned int ebx_length:8;
+};
+
+/*
+ * physical addresses of MSR controlling the perfvtsel and counter registers
+ */
+struct pfm_arch_pmu_info pfm_gen_ia32_pmu_info={
+ .pmc_addrs = {
+ PFM_GEN_IA32_SEL(0), PFM_GEN_IA32_SEL(1), PFM_GEN_IA32_SEL(2),
+ PFM_GEN_IA32_SEL(3),
+ PFM_GEN_IA32_SEL(4), PFM_GEN_IA32_SEL(5), PFM_GEN_IA32_SEL(6),
+ PFM_GEN_IA32_SEL(7),
+ PFM_GEN_IA32_SEL(8), PFM_GEN_IA32_SEL(9), PFM_GEN_IA32_SEL(10),
+ PFM_GEN_IA32_SEL(11),
+ PFM_GEN_IA32_SEL(12), PFM_GEN_IA32_SEL(13), PFM_GEN_IA32_SEL(14),
+ PFM_GEN_IA32_SEL(15),
+ PFM_GEN_IA32_SEL(16), PFM_GEN_IA32_SEL(17), PFM_GEN_IA32_SEL(18),
+ PFM_GEN_IA32_SEL(19),
+ PFM_GEN_IA32_SEL(20), PFM_GEN_IA32_SEL(21), PFM_GEN_IA32_SEL(22),
+ PFM_GEN_IA32_SEL(23),
+ PFM_GEN_IA32_SEL(24), PFM_GEN_IA32_SEL(25), PFM_GEN_IA32_SEL(26),
+ PFM_GEN_IA32_SEL(27),
+ PFM_GEN_IA32_SEL(28), PFM_GEN_IA32_SEL(29), PFM_GEN_IA32_SEL(30),
+ PFM_GEN_IA32_SEL(31)
+ },
+ .pmd_addrs = {
+ PFM_GEN_IA32_CTRL(0), PFM_GEN_IA32_CTRL(1), PFM_GEN_IA32_CTRL(2),
+ PFM_GEN_IA32_CTRL(3),
+ PFM_GEN_IA32_CTRL(4), PFM_GEN_IA32_CTRL(5), PFM_GEN_IA32_CTRL(6),
+ PFM_GEN_IA32_CTRL(7),
+ PFM_GEN_IA32_CTRL(8), PFM_GEN_IA32_CTRL(9), PFM_GEN_IA32_CTRL(10),
+ PFM_GEN_IA32_CTRL(11),
+ PFM_GEN_IA32_CTRL(12), PFM_GEN_IA32_CTRL(13), PFM_GEN_IA32_CTRL(14),
+ PFM_GEN_IA32_CTRL(15),
+ PFM_GEN_IA32_CTRL(16), PFM_GEN_IA32_CTRL(17), PFM_GEN_IA32_CTRL(18),
+ PFM_GEN_IA32_CTRL(19),
+ PFM_GEN_IA32_CTRL(20), PFM_GEN_IA32_CTRL(21), PFM_GEN_IA32_CTRL(22),
+ PFM_GEN_IA32_CTRL(23),
+ PFM_GEN_IA32_CTRL(24), PFM_GEN_IA32_CTRL(25), PFM_GEN_IA32_CTRL(26),
+ PFM_GEN_IA32_CTRL(27),
+ PFM_GEN_IA32_CTRL(28), PFM_GEN_IA32_CTRL(29), PFM_GEN_IA32_CTRL(30),
+ PFM_GEN_IA32_CTRL(31)
+ },

```

```

+ .pmu_style = PFM_X86_PMU_P6,
+ .lps_per_core = 1
+};
+
+#define PFM_GEN_IA32_C(n) { \
+ .type = PFM_REG_I64, \
+ .desc = "PERFEVTSEL"#n, \
+ .default_value = PFM_GEN_IA32_PMC_VAL, \
+ .reserved_mask = PFM_GEN_IA32_PMC_RSVD, \
+ .no_emul64_mask = PFM_GEN_IA32_NO64 }
+
+#define PFM_GEN_IA32_D(n) { \
+ .type = PFM_REG_C, \
+ .desc = "PMC"#n, \
+ .default_value = 0, \
+ .reserved_mask = -1, \
+ .no_emul64_mask = 0}
+
+static struct pfm_reg_desc pfm_gen_ia32_pmc_desc[]={
+/* pmc0 */ PFM_GEN_IA32_C(0), PFM_GEN_IA32_C(1), PFM_GEN_IA32_C(2),
PFM_GEN_IA32_C(3),
+/* pmc4 */ PFM_GEN_IA32_C(4), PFM_GEN_IA32_C(5), PFM_GEN_IA32_C(6),
PFM_GEN_IA32_C(7),
+/* pmc8 */ PFM_GEN_IA32_C(8), PFM_GEN_IA32_C(9), PFM_GEN_IA32_C(10),
PFM_GEN_IA32_C(11),
+/* pmc12 */ PFM_GEN_IA32_C(12), PFM_GEN_IA32_C(13), PFM_GEN_IA32_C(14),
PFM_GEN_IA32_C(15),
+/* pmc16 */ PFM_GEN_IA32_C(16), PFM_GEN_IA32_C(17), PFM_GEN_IA32_C(18),
PFM_GEN_IA32_C(19),
+/* pmc20 */ PFM_GEN_IA32_C(20), PFM_GEN_IA32_C(21), PFM_GEN_IA32_C(22),
PFM_GEN_IA32_C(23),
+/* pmc24 */ PFM_GEN_IA32_C(24), PFM_GEN_IA32_C(25), PFM_GEN_IA32_C(26),
PFM_GEN_IA32_C(27),
+/* pmc28 */ PFM_GEN_IA32_C(28), PFM_GEN_IA32_C(29), PFM_GEN_IA32_C(30),
PFM_GEN_IA32_C(31)
+};
+
+static struct pfm_reg_desc pfm_gen_ia32_pmd_desc[]={
+/* pmd0 */ PFM_GEN_IA32_D(0), PFM_GEN_IA32_D(1), PFM_GEN_IA32_D(2),
PFM_GEN_IA32_D(3),
+/* pmd4 */ PFM_GEN_IA32_D(4), PFM_GEN_IA32_D(5), PFM_GEN_IA32_D(6),
PFM_GEN_IA32_D(7),
+/* pmd8 */ PFM_GEN_IA32_D(8), PFM_GEN_IA32_D(9), PFM_GEN_IA32_D(10),
PFM_GEN_IA32_D(11),
+/* pmd12 */ PFM_GEN_IA32_D(12), PFM_GEN_IA32_D(13), PFM_GEN_IA32_D(14),
PFM_GEN_IA32_D(15),
+/* pmd16 */ PFM_GEN_IA32_D(16), PFM_GEN_IA32_D(17), PFM_GEN_IA32_D(18),
PFM_GEN_IA32_D(19),
+/* pmd20 */ PFM_GEN_IA32_D(20), PFM_GEN_IA32_D(21), PFM_GEN_IA32_D(22),
PFM_GEN_IA32_D(23),
+/* pmd24 */ PFM_GEN_IA32_D(24), PFM_GEN_IA32_D(25), PFM_GEN_IA32_D(26),

```

```

PFM_GEN_IA32_D(27),
+/* pmd28 */ PFM_GEN_IA32_D(28), PFM_GEN_IA32_D(29), PFM_GEN_IA32_D(30),
PFM_GEN_IA32_D(31)
+};
+#define PFM_GEN_IA32_MAX_PMCS ARRAY_SIZE(pfm_gen_ia32_pmc_desc)
+
+#define MSR_IA32_MISC_ENABLE_PERF_AVAIL (1<<7) /* read-only status bit */
+
+static struct pfm_pmu_config pfm_gen_ia32_pmu_conf;
+
+static int pfm_gen_ia32_probe_pmu(void)
+{
+ union {
+ unsigned int val;
+ struct pmu_eax eax;
+ } eax;
+ unsigned int ebx, ecx, edx;
+ unsigned int num_cnt;
+ int high, low;
+
+ PFM_INFO("family=%d x86_model=%d",
+ cpu_data->x86, cpu_data->x86_model);
+ /*
+ * check for P6 processor family
+ */
+ if (cpu_data->x86 != 6) {
+ PFM_INFO("unsupported family=%d", cpu_data->x86);
+ return -1;
+ }
+
+ /*
+ * only works on Intel processors
+ */
+ if (cpu_data->x86_vendor != X86_VENDOR_INTEL) {
+ PFM_INFO("not running on Intel processor");
+ return -1;
+ }
+
+ /*
+ * check if CPU supports 0xa function of CPUID
+ * 0xa started with Core Duo. Needed to detect if
+ * architected PMU is present
+ */
+ cpuid(0x0, &eax.val, &ebx, &ecx, &edx);
+ if (eax.val < 0xa) {
+ PFM_INFO("CPUID 0xa function not supported\n");
+ return -1;
+ }
+
+ cpuid(0xa, &eax.val, &ebx, &ecx, &edx);
+ if (eax.eax.version < 1) {

```

```

+ PFM_INFO("architected PMU not supported\n");
+ return -1;
+ }
+ num_cnt = eax.eax.num_cnt;
+
+ /*
+ * sanity check number of counters
+ */
+ if (num_cnt == 0 || num_cnt >= PFM_MAX_HW_PMCS) {
+ PFM_INFO("invalid number of counters %u\n", eax.eax.num_cnt);
+ return -1;
+ }
+ /*
+ * instead of dynamically generaint the description table
+ * and MSR addresses, we have a default description with a reasonably
+ * large number of counters (32). We believe this is plenty for quite
+ * some time. Thus allows us to have a much simpler probing and
+ * initialization routine, especially because we have no dynamic
+ * allocation, especially for the counter names
+ */
+ if (num_cnt >= PFM_GEN_IA32_MAX_PMCS) {
+ PFM_INFO("too many counters (max=%d) actual=%u\n",
+ PFM_GEN_IA32_MAX_PMCS, num_cnt);
+ return -1;
+ }
+
+ if (eax.eax.cnt_width > 63) {
+ PFM_INFO("invalid counter width %u\n", eax.eax.cnt_width);
+ return -1;
+ }
+
+ if (!cpu_has_apic) {
+ PFM_INFO("no Local APIC, unsupported");
+ return -1;
+ }
+
+ rdmsr(MSR_IA32_APICBASE, low, high);
+ if ((low & MSR_IA32_APICBASE_ENABLE) == 0) {
+ PFM_INFO("local APIC disabled, you must enable "
+ "with lapic kernel command line option");
+ return -1;
+ }
+ pfm_gen_ia32_pmu_conf.num_pmc_entries = num_cnt;
+ pfm_gen_ia32_pmu_conf.num_pmd_entries = num_cnt;
+
+ return 0;
+ }
+
+ /*
+ * Counters may have model-specific width. Yet the documentation says
+ * that only the lower 32 bits can be written to. bits [w-32]

```

```

+ * are sign extensions of bit 31. As such the effective width of
+ * a counter is 31 bits only.
+ * See IA-32 Intel Architecture Software developer manual Vol 3b:
+ * system programming and section 18.17.2 in particular.
+ */
+static struct pfm_pmu_config pfm_gen_ia32_pmu_conf={
+ .pmu_name = "Generic IA-32",
+ .pmd_desc = pfm_gen_ia32_pmd_desc,
+ .counter_width = 31,
+ .pmc_desc = pfm_gen_ia32_pmc_desc,
+ .probe_pmu = pfm_gen_ia32_probe_pmu,
+ .version = "1.0",
+ .flags = PFM_PMU_BUILTIN_FLAG,
+ .owner = THIS_MODULE,
+ .arch_info = &pfm_gen_ia32_pmu_info
+};
+
+static int __init pfm_gen_ia32_pmu_init_module(void)
+{
+ unsigned int i;
+ /*
+ * XXX: could be hardcoded for this PMU model
+ */
+ bitmap_zero(ulp(pfm_gen_ia32_pmu_info.enable_mask), PFM_MAX_HW_PMCS);
+ for(i=0; i < PFM_MAX_HW_PMCS; i++) {
+ if (pfm_gen_ia32_pmu_info.pmc_addrs[i].reg_type & PFM_REGT_PERFSEL) {
+ pfm_bv_set(pfm_gen_ia32_pmu_info.enable_mask, i);
+ }
+ }
+ return pfm_register_pmu_config(&pfm_gen_ia32_pmu_conf);
+}
+
+static void __exit pfm_gen_ia32_pmu_cleanup_module(void)
+{
+ pfm_unregister_pmu_config(&pfm_gen_ia32_pmu_conf);
+}
+
+module_init(pfm_gen_ia32_pmu_init_module);
+module_exit(pfm_gen_ia32_pmu_cleanup_module);
--- linux-2.6.17-rc5.orig/arch/i386/perfmon/perfmon_p4.c 1969-12-31 16:00:00.000000000 -0800
+++ linux-2.6.17-rc5/arch/i386/perfmon/perfmon_p4.c 2006-05-30 02:48:12.000000000 -0700
@@ -0,0 +1,405 @@
+/*
+ * This file contains the P4/Xeon/EM64T PMU register description tables
+ * and pmc checker used by perfmon.c.
+ *
+ * Copyright (c) 2005 Intel Corporation
+ * Contributed by Bryan Wilkerson <bryan.p.wilkerson@xxxxxxxxxx>
+ */
+#include <linux/module.h>
+#include <linux/perfmon.h>

```

```

+#include <asm/msr.h>
+#include <asm/apic.h>
+
+MODULE_AUTHOR("Bryan Wilkerson <bryan.p.wilkerson@xxxxxxxx>");
+MODULE_DESCRIPTION("P4/Xeon/EM64T PMU description table");
+MODULE_LICENSE("GPL");
+
+/*
+ * CCCR default value:
+ * - OVF_PMI_T0=1 (bit 26)
+ * - OVF_PMI_T1=0 (bit 27) (set if necessary in pfm_write_reg())
+ * - all other bits are zero
+ *
+ * OVF_PMI is force to zero if PFM_REGFL_NO_EMUL64 is set on CCCR
+ */
+#define PFM_CCCR_DFL (PFM_ONE_64<<26)
+
+/*
+ * CCCR reserved fields:
+ * - bits 0-11, 25-29, 31-63
+ * - OVF_PMI (26-27), override with REGFL_NO_EMUL64
+ */
+#define PFM_CCCR_RSVD 0x0000000041fff000
+
+#define PFM_P4_NO64 (3ULL<<26) /* use 3 even in non HT mode */
+
+/* With HyperThreading enabled:
+ *
+ * The ESCRs and CCCRs are divided in half with the top half
+ * belonging to logical processor 0 and the bottom half going to
+ * logical processor 1. Thus only half of the PMU resources are
+ * accessible to applications.
+ *
+ * PEBS is not available due to the fact that:
+ * - MSR_PEBS_MATRIX_VERT is shared between the threads
+ * - IA32_PEBS_ENABLE is shared between the threads
+ *
+ * With HyperThreading disabled:
+ *
+ * The full set of PMU resources is exposed to applications.
+ *
+ * The mapping is chosen such that PMCxx -> MSR is the same
+ * in HT and non HT mode, if register is present in HT mode.
+ */
+#define PFM_REGT_NHTESCR (PFM_REGT_ESCR|PFM_REGT_NOHT)
+#define PFM_REGT_NHTCCCR (PFM_REGT_CCCR|PFM_REGT_NOHT)
+#define PFM_REGT_NHTPEBS (PFM_REGT_PEBS|PFM_REGT_NOHT)
+#define PFM_REGT_NHTCTR (PFM_REGT_CTR|PFM_REGT_NOHT)
+

```

```

+static struct pfm_arch_pmu_info pfm_p4_pmu_info={
+.pmc_addrs = {
+/*pmc 0*/ {{0x3b2, 0x3b3}, 0, PFM_REGT_ESCR}, /* BPU_ESCR0,1 */
+/*pmc 1*/ {{0x3b4, 0x3b5}, 0, PFM_REGT_ESCR}, /* IS_ESCR0,1 */
+/*pmc 2*/ {{0x3aa, 0x3ab}, 0, PFM_REGT_ESCR}, /* MOB_ESCR0,1 */
+/*pmc 3*/ {{0x3b6, 0x3b7}, 0, PFM_REGT_ESCR}, /* ITLB_ESCR0,1 */
+/*pmc 4*/ {{0x3ac, 0x3ad}, 0, PFM_REGT_ESCR}, /* PMH_ESCR0,1 */
+/*pmc 5*/ {{0x3c8, 0x3c9}, 0, PFM_REGT_ESCR}, /* IX_ESCR0,1 */
+/*pmc 6*/ {{0x3a2, 0x3a3}, 0, PFM_REGT_ESCR}, /* FSB_ESCR0,1 */
+/*pmc 7*/ {{0x3a0, 0x3a1}, 0, PFM_REGT_ESCR}, /* BSU_ESCR0,1 */
+/*pmc 8*/ {{0x3c0, 0x3c1}, 0, PFM_REGT_ESCR}, /* MS_ESCR0,1 */
+/*pmc 9*/ {{0x3c4, 0x3c5}, 0, PFM_REGT_ESCR}, /* TC_ESCR0,1 */
+/*pmc 10*/ {{0x3c2, 0x3c3}, 0, PFM_REGT_ESCR}, /* TBPU_ESCR0,1 */
+/*pmc 11*/ {{0x3a6, 0x3a7}, 0, PFM_REGT_ESCR}, /* FLAME_ESCR0,1 */
+/*pmc 12*/ {{0x3a4, 0x3a5}, 0, PFM_REGT_ESCR}, /* FIRM_ESCR0,1 */
+/*pmc 13*/ {{0x3ae, 0x3af}, 0, PFM_REGT_ESCR}, /* SAAT_ESCR0,1 */
+/*pmc 14*/ {{0x3b0, 0x3b1}, 0, PFM_REGT_ESCR}, /* U2L_ESCR0,1 */
+/*pmc 15*/ {{0x3a8, 0x3a9}, 0, PFM_REGT_ESCR}, /* DAC_ESCR0,1 */
+/*pmc 16*/ {{0x3ba, 0x3bb}, 0, PFM_REGT_ESCR}, /* IQ_ESCR0,1 */
+/*pmc 17*/ {{0x3ca, 0x3cb}, 0, PFM_REGT_ESCR}, /* ALF_ESCR0,1 */
+/*pmc 18*/ {{0x3bc, 0x3bd}, 0, PFM_REGT_ESCR}, /* RAT_ESCR0,1 */
+/*pmc 19*/ {{0x3be, 0}, 0, PFM_REGT_ESCR}, /* SSU_ESCR0 */
+/*pmc 20*/ {{0x3b8, 0x3b9}, 0, PFM_REGT_ESCR}, /* CRU_ESCR0,1 */
+/*pmc 21*/ {{0x3cc, 0x3cd}, 0, PFM_REGT_ESCR}, /* CRU_ESCR2,3 */
+/*pmc 22*/ {{0x3e0, 0x3e1}, 0, PFM_REGT_ESCR}, /* CRU_ESCR4,5 */
+
+/*pmc 23*/ {{0x360, 0x362}, 0, PFM_REGT_CCCR}, /* BPU_CCCR0,1 */
+/*pmc 24*/ {{0x361, 0x363}, 1, PFM_REGT_CCCR}, /* BPU_CCCR2,3 */
+/*pmc 25*/ {{0x364, 0x366}, 2, PFM_REGT_CCCR}, /* MS_CCCR0,1 */
+/*pmc 26*/ {{0x365, 0x367}, 3, PFM_REGT_CCCR}, /* MS_CCCR2,3 */
+/*pmc 27*/ {{0x368, 0x36a}, 4, PFM_REGT_CCCR}, /* FLAME_CCCR0,1 */
+/*pmc 28*/ {{0x369, 0x36b}, 5, PFM_REGT_CCCR}, /* FLAME_CCCR2,3 */
+/*pmc 29*/ {{0x36c, 0x36e}, 6, PFM_REGT_CCCR}, /* IQ_CCCR0,1 */
+/*pmc 30*/ {{0x36d, 0x36f}, 7, PFM_REGT_CCCR}, /* IQ_CCCR2,3 */
+/*pmc 31*/ {{0x370, 0x371}, 8, PFM_REGT_CCCR}, /* IQ_CCCR4,5 */
+/* non HT extensions */
+/*pmc 32*/ {{0x3b3, 0}, 0, PFM_REGT_NHTESCR}, /* BPU_ESCR1 */
+/*pmc 33*/ {{0x3b5, 0}, 0, PFM_REGT_NHTESCR}, /* IS_ESCR1 */
+/*pmc 34*/ {{0x3ab, 0}, 0, PFM_REGT_NHTESCR}, /* MOB_ESCR1 */
+/*pmc 35*/ {{0x3b7, 0}, 0, PFM_REGT_NHTESCR}, /* ITLB_ESCR1 */
+/*pmc 36*/ {{0x3ad, 0}, 0, PFM_REGT_NHTESCR}, /* PMH_ESCR1 */
+/*pmc 37*/ {{0x3c9, 0}, 0, PFM_REGT_NHTESCR}, /* IX_ESCR1 */
+/*pmc 38*/ {{0x3a3, 0}, 0, PFM_REGT_NHTESCR}, /* FSB_ESCR1 */
+/*pmc 39*/ {{0x3a1, 0}, 0, PFM_REGT_NHTESCR}, /* BSU_ESCR1 */
+/*pmc 40*/ {{0x3c1, 0}, 0, PFM_REGT_NHTESCR}, /* MS_ESCR1 */
+/*pmc 41*/ {{0x3c5, 0}, 0, PFM_REGT_NHTESCR}, /* TC_ESCR1 */
+/*pmc 42*/ {{0x3c3, 0}, 0, PFM_REGT_NHTESCR}, /* TBPU_ESCR1 */
+/*pmc 43*/ {{0x3a7, 0}, 0, PFM_REGT_NHTESCR}, /* FLAME_ESCR1 */
+/*pmc 44*/ {{0x3a5, 0}, 0, PFM_REGT_NHTESCR}, /* FIRM_ESCR1 */
+/*pmc 45*/ {{0x3af, 0}, 0, PFM_REGT_NHTESCR}, /* SAAT_ESCR1 */
+/*pmc 46*/ {{0x3b1, 0}, 0, PFM_REGT_NHTESCR}, /* U2L_ESCR1 */

```

```

+ /*pmc 47*/ { {0x3a9, 0}, 0, PFM_REGT_NHTESCR}, /* DAC_ESCR1 */
+ /*pmc 48*/ { {0x3bb, 0}, 0, PFM_REGT_NHTESCR}, /* IQ_ESCR1 */
+ /*pmc 49*/ { {0x3cb, 0}, 0, PFM_REGT_NHTESCR}, /* ALF_ESCR1 */
+ /*pmc 50*/ { {0x3bd, 0}, 0, PFM_REGT_NHTESCR}, /* RAT_ESCR1 */
+ /*pmc 51*/ { {0x3b9, 0}, 0, PFM_REGT_NHTESCR}, /* CRU_ESCR1 */
+ /*pmc 52*/ { {0x3cd, 0}, 0, PFM_REGT_NHTESCR}, /* CRU_ESCR3 */
+ /*pmc 53*/ { {0x3e1, 0}, 0, PFM_REGT_NHTESCR}, /* CRU_ESCR5 */
+ /*pmc 54*/ { {0x362, 0}, 9, PFM_REGT_NHTCCCR}, /* BPU_CCCR1 */
+ /*pmc 55*/ { {0x363, 0}, 10, PFM_REGT_NHTCCCR}, /* BPU_CCCR3 */
+ /*pmc 56*/ { {0x366, 0}, 11, PFM_REGT_NHTCCCR}, /* MS_CCCR1 */
+ /*pmc 57*/ { {0x367, 0}, 12, PFM_REGT_NHTCCCR}, /* MS_CCCR3 */
+ /*pmc 58*/ { {0x36a, 0}, 13, PFM_REGT_NHTCCCR}, /* FLAME_CCCR1 */
+ /*pmc 59*/ { {0x36b, 0}, 14, PFM_REGT_NHTCCCR}, /* FLAME_CCCR3 */
+ /*pmc 60*/ { {0x36e, 0}, 15, PFM_REGT_NHTCCCR}, /* IQ_CCCR1 */
+ /*pmc 61*/ { {0x36f, 0}, 16, PFM_REGT_NHTCCCR}, /* IQ_CCCR3 */
+ /*pmc 62*/ { {0x371, 0}, 17, PFM_REGT_NHTCCCR}, /* IQ_CCCR5 */
+ /*pmc 63*/ { {0x3f2, 0}, 0, PFM_REGT_NHTPEBS}, /* PEBS_MATRIX_VERT */
+ /*pmc 64*/ { {0x3f1, 0}, 0, PFM_REGT_NHTPEBS} /* PEBS_ENABLE */
+ },
+
+ .pmd_addrs = {
+ /*pmd 0*/ { {0x300, 0x302}, 0, PFM_REGT_CTR}, /* BPU_CRT0,1 */
+ /*pmd 1*/ { {0x301, 0x303}, 0, PFM_REGT_CTR}, /* BPU_CTR2,3 */
+ /*pmd 2*/ { {0x304, 0x306}, 0, PFM_REGT_CTR}, /* MS_CRT0,1 */
+ /*pmd 3*/ { {0x305, 0x307}, 0, PFM_REGT_CTR}, /* MS_CRT2,3 */
+ /*pmd 4*/ { {0x308, 0x30a}, 0, PFM_REGT_CTR}, /* FLAME_CRT0,1 */
+ /*pmd 5*/ { {0x309, 0x30b}, 0, PFM_REGT_CTR}, /* FLAME_CRT2,3 */
+ /*pmd 6*/ { {0x30c, 0x30e}, 0, PFM_REGT_CTR}, /* IQ_CRT0,1 */
+ /*pmd 7*/ { {0x30d, 0x30f}, 0, PFM_REGT_CTR}, /* IQ_CRT2,3 */
+ /*pmd 8*/ { {0x310, 0x311}, 0, PFM_REGT_CTR}, /* IQ_CRT4,5 */
+ /*
+ * non HT extensions
+ */
+ /*pmd 9*/ { {0x302, 0}, 0, PFM_REGT_NHTCTR}, /* BPU_CRT1 */
+ /*pmd 10*/ { {0x303, 0}, 0, PFM_REGT_NHTCTR}, /* BPU_CTR3 */
+ /*pmd 11*/ { {0x306, 0}, 0, PFM_REGT_NHTCTR}, /* MS_CRT1 */
+ /*pmd 12*/ { {0x307, 0}, 0, PFM_REGT_NHTCTR}, /* MS_CRT3 */
+ /*pmd 13*/ { {0x30a, 0}, 0, PFM_REGT_NHTCTR}, /* FLAME_CRT1 */
+ /*pmd 14*/ { {0x30b, 0}, 0, PFM_REGT_NHTCTR}, /* FLAME_CRT3 */
+ /*pmd 15*/ { {0x30e, 0}, 0, PFM_REGT_NHTCTR}, /* IQ_CRT1 */
+ /*pmd 16*/ { {0x30f, 0}, 0, PFM_REGT_NHTCTR}, /* IQ_CRT3 */
+ /*pmd 17*/ { {0x311, 0}, 0, PFM_REGT_NHTCTR}, /* IQ_CRT5 */
+ },
+ .pebs_ctr_idx = 8, /* thread0: IQ_CTR4, thread1: IQ_CTR5 */
+ .pmu_style = PFM_X86_PMU_P4,
+ .lps_per_core = 1
+ };
+
+ static struct pfm_reg_desc pfm_p4_pmc_desc[]={
+ /* pmc0 */ { PFM_REG_I, "BPU_ESCR0", 0x0, PFM_ESCR_RSVD},
+ /* pmc1 */ { PFM_REG_I, "IS_ESCR0", 0x0, PFM_ESCR_RSVD},

```

```

+/* pmc2 */ { PFM_REG_I, "MOB_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc3 */ { PFM_REG_I, "ITLB_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc4 */ { PFM_REG_I, "PMH_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc5 */ { PFM_REG_I, "IX_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc6 */ { PFM_REG_I, "FSB_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc7 */ { PFM_REG_I, "BSU_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc8 */ { PFM_REG_I, "MS_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc9 */ { PFM_REG_I, "TC_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc10 */ { PFM_REG_I, "TBPU_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc11 */ { PFM_REG_I, "FLAME_ESCR0", 0x0, PFM_ESCR_RSVD},
+/* pmc12 */ { PFM_REG_I, "FIRM_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc13 */ { PFM_REG_I, "SAAT_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc14 */ { PFM_REG_I, "U2L_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc15 */ { PFM_REG_I, "DAC_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc16 */ { PFM_REG_I, "IQ_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc17 */ { PFM_REG_I, "ALF_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc18 */ { PFM_REG_I, "RAT_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc19 */ { PFM_REG_I, "SSU_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc20 */ { PFM_REG_I, "CRU_ESCR0" , 0x0, PFM_ESCR_RSVD},
+/* pmc21 */ { PFM_REG_I, "CRU_ESCR2" , 0x0, PFM_ESCR_RSVD},
+/* pmc22 */ { PFM_REG_I, "CRU_ESCR4" , 0x0, PFM_ESCR_RSVD},
+/* pmc23 */ { PFM_REG_I64, "BPU_CCCR0" , PFM_CCCR_DFL, PFM_CCCR_RSVD,
PFM_P4_NO64},
+/* pmc24 */ { PFM_REG_I64, "BPU_CCCR2" , PFM_CCCR_DFL, PFM_CCCR_RSVD,
PFM_P4_NO64},
+/* pmc25 */ { PFM_REG_I64, "MS_CCCR0" , PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+/* pmc26 */ { PFM_REG_I64, "MS_CCCR2" , PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+/* pmc27 */ { PFM_REG_I64, "FLAME_CCCR0", PFM_CCCR_DFL, PFM_CCCR_RSVD,
PFM_P4_NO64},
+/* pmc28 */ { PFM_REG_I64, "FLAME_CCCR2", PFM_CCCR_DFL, PFM_CCCR_RSVD,
PFM_P4_NO64},
+/* pmc29 */ { PFM_REG_I64, "IQ_CCCR0" , PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+/* pmc30 */ { PFM_REG_I64, "IQ_CCCR2" , PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+/* pmc31 */ { PFM_REG_I64, "IQ_CCCR4" , PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+ /* No HT extension */
+/* pmc32 */ { PFM_REG_I, "BPU_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc33 */ { PFM_REG_I, "IS_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc34 */ { PFM_REG_I, "MOB_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc35 */ { PFM_REG_I, "ITLB_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc36 */ { PFM_REG_I, "PMH_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc37 */ { PFM_REG_I, "IX_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc38 */ { PFM_REG_I, "FSB_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc39 */ { PFM_REG_I, "BSU_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc40 */ { PFM_REG_I, "MS_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc41 */ { PFM_REG_I, "TC_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc42 */ { PFM_REG_I, "TBPU_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc43 */ { PFM_REG_I, "FLAME_ESCR1", 0x0, PFM_ESCR_RSVD},
+/* pmc44 */ { PFM_REG_I, "FIRM_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc45 */ { PFM_REG_I, "SAAT_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc46 */ { PFM_REG_I, "U2L_ESCR1" , 0x0, PFM_ESCR_RSVD},
+/* pmc47 */ { PFM_REG_I, "DAC_ESCR1" , 0x0, PFM_ESCR_RSVD},

```

```

+/* pmc48 */ { PFM_REG_I, "IQ_ESCR1", 0x0, PFM_ESCR_RSVD},
+/* pmc49 */ { PFM_REG_I, "ALF_ESCR1", 0x0, PFM_ESCR_RSVD},
+/* pmc50 */ { PFM_REG_I, "RAT_ESCR1", 0x0, PFM_ESCR_RSVD},
+/* pmc51 */ { PFM_REG_I, "CRU_ESCR1", 0x0, PFM_ESCR_RSVD},
+/* pmc52 */ { PFM_REG_I, "CRU_ESCR3", 0x0, PFM_ESCR_RSVD},
+/* pmc53 */ { PFM_REG_I, "CRU_ESCR5", 0x0, PFM_ESCR_RSVD},
+/* pmc54 */ { PFM_REG_I64, "BPU_CCCR1", PFM_CCCR_DFL, PFM_CCCR_RSVD,
PFM_P4_NO64},
+/* pmc55 */ { PFM_REG_I64, "BPU_CCCR3", PFM_CCCR_DFL, PFM_CCCR_RSVD,
PFM_P4_NO64},
+/* pmc56 */ { PFM_REG_I64, "MS_CCCR1", PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+/* pmc57 */ { PFM_REG_I64, "MS_CCCR3", PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+/* pmc58 */ { PFM_REG_I64, "FLAME_CCCR1", PFM_CCCR_DFL, PFM_CCCR_RSVD,
PFM_P4_NO64},
+/* pmc59 */ { PFM_REG_I64, "FLAME_CCCR3", PFM_CCCR_DFL, PFM_CCCR_RSVD,
PFM_P4_NO64},
+/* pmc60 */ { PFM_REG_I64, "IQ_CCCR1", PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+/* pmc61 */ { PFM_REG_I64, "IQ_CCCR3", PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+/* pmc62 */ { PFM_REG_I64, "IQ_CCCR5", PFM_CCCR_DFL, PFM_CCCR_RSVD, PFM_P4_NO64},
+/* pmc63 */ { PFM_REG_I, "PEBS_MATRIX_VERT", 0, 0x13},
+/* pmc64 */ { PFM_REG_I, "PEBS_ENABLE", 0, 0x3000fff},
+};
+#define PFM_P4_NUM_PMCS ARRAY_SIZE(pfm_p4_pmc_desc)
+
+/*
+ * See section 15.10.6.6 for details about the IQ block
+ */
+static struct pfm_reg_desc pfm_p4_pmd_desc[]={
+/* pmd0 */ { PFM_REG_C, "BPU_CTR0", 0x0, -1},
+/* pmd1 */ { PFM_REG_C, "BPU_CTR2", 0x0, -1},
+/* pmd2 */ { PFM_REG_C, "MS_CTR0", 0x0, -1},
+/* pmd3 */ { PFM_REG_C, "MS_CTR2", 0x0, -1},
+/* pmd4 */ { PFM_REG_C, "FLAME_CTR0", 0x0, -1},
+/* pmd5 */ { PFM_REG_C, "FLAME_CTR2", 0x0, -1},
+/* pmd6 */ { PFM_REG_C, "IQ_CTR0", 0x0, -1},
+/* pmd7 */ { PFM_REG_C, "IQ_CTR2", 0x0, -1},
+/* pmd8 */ { PFM_REG_C, "IQ_CTR4", 0x0, -1},
+/* no HT extension */
+/* pmd9 */ { PFM_REG_C, "BPU_CTR1", 0x0, -1},
+/* pmd10 */ { PFM_REG_C, "BPU_CTR3", 0x0, -1},
+/* pmd11 */ { PFM_REG_C, "MS_CTR1", 0x0, -1},
+/* pmd12 */ { PFM_REG_C, "MS_CTR3", 0x0, -1},
+/* pmd13 */ { PFM_REG_C, "FLAME_CTR1", 0x0, -1},
+/* pmd14 */ { PFM_REG_C, "FLAME_CTR3", 0x0, -1},
+/* pmd15 */ { PFM_REG_C, "IQ_CTR1", 0x0, -1},
+/* pmd16 */ { PFM_REG_C, "IQ_CTR3", 0x0, -1},
+/* pmd17 */ { PFM_REG_C, "IQ_CTR5", 0x0, -1},
+};
+#define PFM_P4_NUM_PMDS ARRAY_SIZE(pfm_p4_pmd_desc)
+
+#define MSR_IA32_MISC_ENABLE_PEBS_UNAVAIL (1<<12) /* PEBS unavailable */

```

```

+#define cpu_has_dts boot_cpu_has(X86_FEATURE_DTES)
+
+static int pfm_p4_probe_pmu(void)
+{
+ int high, low;
+ unsigned int i;+ * The "int vec" is not the right solution either
+ * because it triggers a software intr. We need
+ * to regenerate the intr. and have it pended until
+ * we unmask interrupts.
+ *
+ * Instead we send ourself an IPI on the perfmon
+ * vector.
+ */
+ val = APIC_DEST_SELF|APIC_INT_ASSERT|
+ APIC_DM_FIXED|LOCAL_PERFMON_VECTOR;
+ dest = apic_read(APIC_ID);
+ apic_write(APIC_ICR2, dest);
+ apic_write(APIC_ICR, val);
+
+}
+
+#define pfm_arch_serialize() /* nothing */
+
+static inline u64 pfm_arch_get_itc(void)
+{
+ u64 tmp;
+ rdtscll(tmp);
+ return tmp;
+}
+
+static inline void pfm_arch_write_pmc(struct pfm_context *ctx, unsigned int cnum, u64 value)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ /*
+ * we only write to the actual register when monitoring is
+ * active (pfm_start was issued)
+ */
+ if (ctx && ctx->flags.started == 0) return;
+
+ __pfm_write_reg(&arch_info->pmc_addrs[cnum], value);
+}
+
+static inline void pfm_arch_write_pmd(struct pfm_context *ctx, unsigned int cnum, u64 value)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+
+ /*
+ * force upper bit set for counter to ensure overflow
+ */
+ if (arch_info->pmd_addrs[cnum].reg_type & PFM_REGT_CTR)
+ value |= ~pfm_pmu_conf->ovfl_mask;

```

```

+
+ __pfm_write_reg(&arch_info->pmd_addrs[cnum], value);
+}
+
+static inline u64 pfm_arch_read_pmd(struct pfm_context *ctx, unsigned int cnum)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ u64 tmp;
+ __pfm_read_reg(&arch_info->pmd_addrs[cnum], &tmp);
+ return tmp;
+}
+
+static inline u64 pfm_arch_read_pmc(struct pfm_context *ctx, unsigned int cnum)
+{
+ struct pfm_arch_pmu_info *arch_info = pfm_pmu_conf->arch_info;
+ u64 tmp;
+ __pfm_read_reg(&arch_info->pmc_addrs[cnum], &tmp);
+ return tmp;
+}
+/*
+ * At certain points, perfmon needs to know if monitoring has been
+ * explicitly started/stopped by user via pfm_start/pfm_stop. The
+ * information is tracked in flags.started. However on certain
+ * architectures, it may be possible to start/stop directly from
+ * user level with a single assembly instruction bypassing
+ * the kernel. This function must be used to determine by
+ * an arch-specific mean if monitoring is actually started/stopped.
+ * If there is no other way but to go through pfm_start/pfm_stop
+ * then this function can simply return 0
+ */
+static inline int pfm_arch_is_active(struct pfm_context *ctx)
+{
+ return 0;
+}
+
+void pfm_arch_init_percpu(void);
+void pfm_arch_ctxswout(struct task_struct *task,
+ struct pfm_context *ctx, struct pfm_event_set *set);
+void pfm_arch_ctxswin(struct task_struct *task,
+ struct pfm_context *ctx, struct pfm_event_set *set);
+void pfm_arch_stop(struct task_struct *task,
+ struct pfm_context *ctx, struct pfm_event_set *set);
+void pfm_arch_start(struct task_struct *task,
+ struct pfm_context *ctx, struct pfm_event_set *set);
+void pfm_arch_restore_pmds(struct pfm_context *ctx, struct pfm_event_set *set);
+void pfm_arch_save_pmds(struct pfm_context *ctx, struct pfm_event_set *set);
+void pfm_arch_restore_pmcs(struct pfm_context *ctx, struct pfm_event_set *set);
+void pfm_arch_intr_freeze_pmu(struct pfm_context *ctx);
+void pfm_arch_intr_unfreeze_pmu(struct pfm_context *ctx);
+int pfm_arch_pmu_config_check(struct pfm_pmu_config *cfg);
+void pfm_arch_pmu_config_init(void);

```

```
+int pfm_arch_initialize(void);  
+char *pfm_arch_get_pmu_module_name(void);  
+void pfm_arch_mask_monitoring(struct pfm_context *ctx);  
+void pfm_arch_unmask_monitoring(struct pfm_context *ctx);  
+void pfm_arch_context_initialize(str
```