

[patch] x86, ptrace: PEBS support

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-04/msg06799.html>

- *From:* Markus Metzger <markus.t.metzger@xxxxxxxxxx>
 - *Date:* Thu, 17 Apr 2008 16:50:04 +0200
-

This is meant to update the previous version of this patch with commit number ae452ae05c96d0924d163831f5681babaf2dd0fd on x86.git.

It adds a few bugfixes that were detected during the perfmon2 adaptation.

Polish the ds.h interface and add support for PEBS.

Ds.c is meant to be the resource allocator for per-thread and per-cpu BTS and PEBS recording.

It is used by ptrace/utrace to provide execution tracing of debugged tasks.

It will be used by profilers (e.g. perfmon2).

It may be used by kernel debuggers to provide a kernel execution trace.

Changes in detail:

- guard DS and ptrace by CONFIG macros
- separate DS and BTS more clearly
- simplify field accesses
- add functions to manage PEBS buffers
- add simple protection/allocation mechanism
- added support for Atom

Opens:

- buffer overflow handling

Currently, only circular buffers are supported. This is all we need for debugging. Profilers would want an overflow notification.

This is planned to be added when perfmon2 is made to use the ds.h interface.

- utrace intermediate layer

Signed-off-by: Markus Metzger <markus.t.metzger@xxxxxxxxxx>

diff --git a/arch/x86/Kconfig.cpu b/arch/x86/Kconfig.cpu

index 7d44c4b..2237152 100644

--- a/arch/x86/Kconfig.cpu

[patch] x86, ptrace: PEBS support

```
+++ b/arch/x86/Kconfig.cpu
@@ -411,3 +411,21 @@ config X86_MINIMUM_CPU_FAMILY
config X86_DEBUGCTLMR
def_bool y
depends on !(M586MMX || M586TSC || M586 || M486 || M386)
+
+config X86_DS
+ bool "Debug Store support"
+ default y
+ help
+ Add support for Debug Store.
+ This allows the kernel to provide a memory buffer to the hardware
+ to store various profiling and tracing events.
+
+config X86_PTRACE_BTS
+ bool "ptrace interface to Branch Trace Store"
+ default y
+ depends on (X86_DS && X86_DEBUGCTLMR)
+ help
+ Add a ptrace interface to allow collecting an execution trace
+ of the traced task.
+ This collects control flow changes in a (cyclic) buffer and allows
+ debuggers to fill in the gaps and show an execution trace of the debuggee.
diff --git a/arch/x86/kernel/cpu/intel.c b/arch/x86/kernel/cpu/intel.c
index fe9224c..cbffa2a 100644
--- a/arch/x86/kernel/cpu/intel.c
+++ b/arch/x86/kernel/cpu/intel.c
@@ -222,10 +222,11 @@ static void __cpuintel_init_intel(struct cpuinfo_x86 *c)
set_cpu_cap(c, X86_FEATURE_BTS);
if (!(11 & (1<<12)))
set_cpu_cap(c, X86_FEATURE_PEBS);
+ ds_init_intel(c);
}

if (cpu_has_bts)
- ds_init_intel(c);
+ ptrace_bts_init_intel(c);
}

static unsigned int __cpuintel_size_cache(struct cpuinfo_x86 *c, unsigned int size)
diff --git a/arch/x86/kernel/ds.c b/arch/x86/kernel/ds.c
index 11c11b8..e787ac7 100644
--- a/arch/x86/kernel/ds.c
+++ b/arch/x86/kernel/ds.c
@@ -2,26 +2,49 @@
* Debug Store support
*
* This provides a low-level interface to the hardware's Debug Store
- * feature that is used for last branch recording (LBR) and
+ * feature that is used for branch trace store (BTS) and
* precise-event based sampling (PEBS).
```

[patch] x86, ptrace: PEBS support

[patch] x86, ptrace: PEBS support

```
*
- * Different architectures use a different DS layout/pointer size.
- * The below functions therefore work on a void*.
+ * It manages:
+ * - per-thread and per-cpu allocation of BTS and PEBS
+ * - buffer memory allocation (optional)
+ * - buffer overflow handling
+ * - buffer access
*
+ * It assumes:
+ * - get_task_struct on all parameter tasks
+ * - current is allowed to trace parameter tasks
*
- * Since there is no user for PEBS, yet, only LBR (or branch
- * trace store, BTS) is supported.
*
- *
- * Copyright (C) 2007 Intel Corporation.
- * Markus Metzger <markus.t.metzger@xxxxxxxx>, Dec 2007
+ * Copyright (C) 2007-2008 Intel Corporation.
+ * Markus Metzger <markus.t.metzger@xxxxxxxx>, 2007-2008
*/

+
+#ifdef CONFIG_X86_DS
+
+#include <asm/ds.h>

#include <linux/errno.h>
#include <linux/string.h>
#include <linux/slab.h>
+#include <linux/sched.h>
+#include <linux/module.h>
+
+
+/*
+ * The configuration for a particular DS hardware implementation.
+ */
+struct ds_configuration {
+ /* the size of the DS structure in bytes */
+ unsigned char sizeof_ds;
+ /* the size of one pointer-typed field in the DS structure in bytes;
+ this covers the first 8 fields related to buffer management. */
+ unsigned char sizeof_field;
+ /* the size of a BTS/PEBS record in bytes */
+ unsigned char sizeof_rec[2];
+};
+static struct ds_configuration ds_cfg;

/*
```

[patch] x86, ptrace: PEBS support

```
@@ -44,378 +67,786 @@
* (interrupt occurs when write pointer passes interrupt pointer)
* - value to which counter is reset following counter overflow
*
- * On later architectures, the last branch recording hardware uses
- * 64bit pointers even in 32bit mode.
- *
- *
- * Branch Trace Store (BTS) records store information about control
- * flow changes. They at least provide the following information:
- * - source linear address
- * - destination linear address
+ * Later architectures use 64bit pointers throughout, whereas earlier
+ * architectures use 32bit pointers in 32bit mode.
*
- * Netburst supported a predicated bit that had been dropped in later
- * architectures. We do not support it.
*
+ * We compute the base address for the first 8 fields based on:
+ * - the field size stored in the DS configuration
+ * - the relative field position
+ * - an offset giving the start of the respective region
*
- * In order to abstract from the actual DS and BTS layout, we describe
- * the access to the relevant fields.
- * Thanks to Andi Kleen for proposing this design.
+ * This offset is further used to index various arrays holding
+ * information for BTS and PEBS at the respective index.
*
- * The implementation, however, is not as general as it might seem. In
- * order to stay somewhat simple and efficient, we assume an
- * underlying unsigned type (mostly a pointer type) and we expect the
- * field to be at least as big as that type.
+ * On later 32bit processors, we only access the lower 32bit of the
+ * 64bit pointer fields. The upper halves will be zeroed out.
*/

-/*
- * A special from_ip address to indicate that the BTS record is an
- * info record that needs to be interpreted or skipped.
- */
-#define BTS_ESCAPE_ADDRESS (-1)
+enum ds_field {
+ ds_buffer_base = 0,
+ ds_index,
+ ds_absolute_maximum,
+ ds_interrupt_threshold,
+};

-/*
- * A field access descriptor
```

[patch] x86, ptrace: PEBS support

```
– */
–struct access_desc {
– unsigned char offset;
– unsigned char size;
+enum ds_qualifier {
+ ds_bts = 0,
+ ds_pebs
};

+static inline unsigned long ds_get(const unsigned char *base,
+ enum ds_qualifier qual, enum ds_field field)
+{
+ base += (ds_cfg.sizeof_field * (field + (4 * qual)));
+ return *(unsigned long *)base;
+}
+
+static inline void ds_set(unsigned char *base, enum ds_qualifier qual,
+ enum ds_field field, unsigned long value)
+{
+ base += (ds_cfg.sizeof_field * (field + (4 * qual)));
+ (*(unsigned long *)base) = value;
+}
+
+
+/*
– * The configuration for a particular DS/BTS hardware implementation.
– * Locking is done only for allocating BTS or PEBS resources and for
– * guarding context and buffer memory allocation.
– *
– * Most functions require the current task to own the ds context part
– * they are going to access. All the locking is done when validating
– * access to the context.
*/
–struct ds_configuration {
– /* the DS configuration */
– unsigned char sizeof_ds;
– struct access_desc bts_buffer_base;
– struct access_desc bts_index;
– struct access_desc bts_absolute_maximum;
– struct access_desc bts_interrupt_threshold;
– /* the BTS configuration */
– unsigned char sizeof_bts;
– struct access_desc from_ip;
– struct access_desc to_ip;
– /* BTS variants used to store additional information like
– timestamps */
– struct access_desc info_type;
– struct access_desc info_data;
– unsigned long debugctl_mask;
–};
+static spinlock_t ds_lock = __SPIN_LOCK_UNLOCKED(ds_lock);
```

[patch] x86, ptrace: PEBS support

```
/*
- * The global configuration used by the below accessor functions
+ * Validate that the current task is allowed to access the BTS/PEBS
+ * buffer of the parameter task.
+ *
+ * Returns 0, if access is granted; -Eerrno, otherwise.
*/
-static struct ds_configuration ds_cfg;
+static inline int ds_validate_access(struct ds_context *context,
+ enum ds_qualifier qual)
+{
+ if (!context)
+ return -EPERM;
+
+ /* The one who did the ds_request() is OK */
+ if (context->owner[qual] == current)
+ return 0;
+
+ /* Every task may access its own configuration */
+ if (context->task == current)
+ return 0;
+
+ return -EPERM;
+}
+

/*
- * Accessor functions for some DS and BTS fields using the above
- * global ptrace_bts_cfg.
+ * We either support (system-wide) per-cpu or per-thread allocation.
+ * We distinguish the two based on the task_struct pointer, where a
+ * NULL pointer indicates per-cpu allocation for the current cpu.
+ *
+ * Allocations are use-counted. As soon as resources are allocated,
+ * further allocations must be of the same type (per-cpu or
+ * per-thread). We model this by counting allocations (i.e. the number
+ * of tracers of a certain type) for one type negatively:
+ * =0 no tracers
+ * >0 number of per-thread tracers
+ * <0 number of per-cpu tracers
+ *
+ * The below functions to get and put tracers and to check the
+ * allocation type require the ds_lock to be held by the caller.
+ *
+ * Tracers essentially gives the number of ds contexts for a certain
+ * type of allocation.
*/
-static inline unsigned long get_bts_buffer_base(char *base)
+static long tracers;
+
```

[patch] x86, ptrace: PEBS support

```
+static inline void get_tracer(struct task_struct *task)
{
- return *(unsigned long *)(base + ds_cfg.bts_buffer_base.offset);
+ tracers += (task ? 1 : -1);
}
-static inline void set_bts_buffer_base(char *base, unsigned long value)
+
+static inline void put_tracer(struct task_struct *task)
{
- (*(unsigned long *)(base + ds_cfg.bts_buffer_base.offset)) = value;
+ tracers -= (task ? 1 : -1);
}
-static inline unsigned long get_bts_index(char *base)
+
+static inline int check_tracer(struct task_struct *task)
{
- return *(unsigned long *)(base + ds_cfg.bts_index.offset);
+ return (task ? (tracers >= 0) : (tracers <= 0));
}
-static inline void set_bts_index(char *base, unsigned long value)
+
+
+/*
+ * The DS context is either attached to a thread or to a cpu:
+ * - in the former case, the thread_struct contains a pointer to the
+ * attached context.
+ * - in the latter case, we use a static array of per-cpu context
+ * pointers.
+ *
+ * Contexts are use-counted. They are allocated on first access and
+ * deallocated when the last user puts the context.
+ *
+ * We distinguish between an allocating and a non-allocating get of a
+ * context:
+ * - the allocating get is used for requesting BTS/PEBS resources. It
+ * requires the caller to hold the global ds_lock.
+ * - the non-allocating get is used for all other cases. A
+ * non-existing context indicates an error. It acquires and releases
+ * the ds_lock itself for obtaining the context.
+ *
+ * A context and its DS configuration are allocated and deallocated
+ * together. A context always has a DS configuration of the
+ * appropriate size.
+ */
+static DEFINE_PER_CPU(struct ds_context *, system_context);
+
+#define this_system_context per_cpu(system_context, smp_processor_id())
+
+/*
+ * Returns the pointer to the parameter task's context or to the
+ * system-wide context, if task is NULL.
```

[patch] x86, ptrace: PEBS support

```
+ *
+ * Increases the use count of the returned context, if not NULL.
+ */
+static inline struct ds_context *ds_get_context(struct task_struct *task)
{
- (*(unsigned long *) (base + ds_cfg.bts_index.offset)) = value;
+ struct ds_context *context = 0;
+
+ spin_lock(&ds_lock);
+
+ context = (task ? task->thread.ds_ctx : this_system_context);
+ if (context)
+ context->count++;
+
+ spin_unlock(&ds_lock);
+
+ return context;
}
-static inline unsigned long get_bts_absolute_maximum(char *base)
+
+/*
+ * Same as ds_get_context, but allocates the context and it's DS
+ * structure, if necessary; returns NULL; if out of memory.
+ *
+ * pre: requires ds_lock to be held
+ */
+static inline struct ds_context *ds_alloc_context(struct task_struct *task)
{
- return *(unsigned long *) (base + ds_cfg.bts_absolute_maximum.offset);
+ struct ds_context **p_context =
+ (task ? &task->thread.ds_ctx : &this_system_context);
+ struct ds_context *context = *p_context;
+
+ if (!context) {
+ context = kzalloc(sizeof(*context), GFP_KERNEL);
+
+ if (!context)
+ return 0;
+
+ context->ds = kzalloc(ds_cfg.sizeof_ds, GFP_KERNEL);
+ if (!context->ds) {
+ kfree(context);
+ return 0;
+ }
+
+ *p_context = context;
+
+ context->this = p_context;
+ context->task = task;
+
+ if (task)
```

[patch] x86, ptrace: PEBS support

```
+ set_tsk_thread_flag(task, TIF_DS_AREA_MSR);
+
+ if (!task || (task == current))
+ wrmsr(MSR_IA32_DS_AREA, (unsigned long)context->ds, 0);
+
+ get_tracer(task);
+ }
+
+ context->count++;
+
+ return context;
}
-static inline void set_bts_absolute_maximum(char *base, unsigned long value)
+
+/*
+ * Decreases the use count of the parameter context, if not NULL.
+ * Deallocates the context, if the use count reaches zero.
+ */
+static inline void ds_put_context(struct ds_context *context)
{
- (*(unsigned long *) (base + ds_cfg.bts_absolute_maximum.offset)) = value;
+ if (!context)
+ return;
+
+ spin_lock(&ds_lock);
+
+ if (--context->count)
+ goto out;
+
+ *(context->this) = 0;
+
+ if (context->task)
+ clear_tsk_thread_flag(context->task, TIF_DS_AREA_MSR);
+
+ if (!context->task || (context->task == current))
+ wrmsr(MSR_IA32_DS_AREA, 0);
+
+ put_tracer(context->task);
+
+ /* free any leftover buffers from tracers that did not
+ * deallocate them properly. */
+ kfree(context->buffer[ds_bts]);
+ kfree(context->buffer[ds_pebs]);
+ kfree(context->ds);
+ kfree(context);
+ out:
+ spin_unlock(&ds_lock);
}
-static inline unsigned long get_bts_interrupt_threshold(char *base)
+
+
```

[patch] x86, ptrace: PEBS support

```
+/*
+ * Handle a buffer overflow
+ *
+ * task: the task whose buffers are overflowing;
+ * NULL for a buffer overflow on the current cpu
+ * context: the ds context
+ * qual: the buffer type
+ */
+static void ds_overflow(struct task_struct *task, struct ds_context *context,
+ enum ds_qualifier qual)
{
- return *(unsigned long *) (base + ds_cfg.bts_interrupt_threshold.offset);
+ if (!context)
+ return;
+
+ if (context->callback[qual])
+ (*context->callback[qual])(task);
+
+ /* todo: do some more overflow handling */
}
-static inline void set_bts_interrupt_threshold(char *base, unsigned long value)
+
+
+/*
+ * Allocate a non-pageable buffer of the parameter size.
+ * Checks the memory and the locked memory rlimit.
+ *
+ * Returns the buffer, if successful;
+ * NULL, if out of memory or rlimit exceeded.
+ *
+ * size: the requested buffer size in bytes
+ * pages (out): if not NULL, contains the number of pages reserved
+ */
+static inline void *ds_allocate_buffer(size_t size, unsigned int *pages)
{
- (*(unsigned long *) (base + ds_cfg.bts_interrupt_threshold.offset)) = value;
+ struct mm_struct *mm = current->mm;
+ unsigned long rlim, vm, pgsz;
+ void *buffer = 0;
+
+ pgsz = PAGE_ALIGN(size) >> PAGE_SHIFT;
+
+ down_write(&mm->mmap_sem);
+
+ rlim = current->signal->rlim[RLIMIT_AS].rlim_cur >> PAGE_SHIFT;
+ vm = current->mm->total_vm + pgsz;
+ if (rlim < vm)
+ goto out;
+
+ rlim = current->signal->rlim[RLIMIT_MEMLOCK].rlim_cur >> PAGE_SHIFT;
+ vm = current->mm->locked_vm + pgsz;
```

[patch] x86, ptrace: PEBS support

```
+ if (rlim < vm)
+ goto out;
+
+ buffer = kzalloc(size, GFP_KERNEL);
+ if (!buffer)
+ goto out;
+
+ current->mm->total_vm += pgsz;
+ current->mm->locked_vm += pgsz;
+
+ if (pages)
+ *pages = pgsz;
+
+ out:
+ up_write(&mm->mmap_sem);
+ return buffer;
+ }
-static inline unsigned long get_from_ip(char *base)
+
+static int ds_request(struct task_struct *task, void *base, size_t size,
+ ds_ovfl_callback_t ovfl, enum ds_qualifier qual)
+ {
- return *(unsigned long *)(base + ds_cfg.from_ip.offset);
+ struct ds_context *context = 0;
+ unsigned long buffer, adj;
+ const unsigned long alignment = (1 << 3);
+ int error = 0;
+
+ if (!ds_cfg.sizeof_ds)
+ return -EOPNOTSUPP;
+
+ /* we require some space to do alignment adjustments below */
+ if (size < (alignment + ds_cfg.sizeof_rec[qual]))
+ return -EINVAL;
+
+ /* buffer overflow notification is not yet implemented */
+ if (ovfl)
+ return -EOPNOTSUPP;
+
+
+ spin_lock(&ds_lock);
+
+ error = -EPERM;
+ if (!check_tracer(task))
+ goto out_unlock;
+
+ error = -ENOMEM;
+ context = ds_alloc_context(task);
+ if (!context)
+ goto out_unlock;
+
+ }
```

[patch] x86, ptrace: PEBS support

```
+ error = -EALREADY;
+ if (context->owner[qual] == current)
+ goto out_unlock;
+ error = -EPERM;
+ if (context->owner[qual] != 0)
+ goto out_unlock;
+ context->owner[qual] = current;
+
+ spin_unlock(&ds_lock);
+
+
+ error = -ENOMEM;
+ if (!base) {
+ base = ds_allocate_buffer(size, &context->pages[qual]);
+ if (!base)
+ goto out_release;
+
+ context->buffer[qual] = base;
+ }
+ error = 0;
+
+ context->callback[qual] = ovfl;
+
+ /* adjust the buffer address and size to meet alignment
+ * constraints:
+ * - buffer is double-word aligned
+ * - size is multiple of record size
+ *
+ * We checked the size at the very beginning; we have enough
+ * space to do the adjustment.
+ */
+ buffer = (unsigned long)base;
+
+ adj = ALIGN(buffer, alignment) - buffer;
+ buffer += adj;
+ size -= adj;
+
+ size /= ds_cfg.sizeof_rec[qual];
+ size *= ds_cfg.sizeof_rec[qual];
+
+ ds_set(context->ds, qual, ds_buffer_base, buffer);
+ ds_set(context->ds, qual, ds_index, buffer);
+ ds_set(context->ds, qual, ds_absolute_maximum, buffer + size);
+
+ if (ovfl) {
+ /* todo: select a suitable interrupt threshold */
+ } else
+ ds_set(context->ds, qual,
+ ds_interrupt_threshold, buffer + size + 1);
+
+ /* we keep the context until ds_release */
```

[patch] x86, ptrace: PEBS support

```
+ return error;
+
+ out_release:
+ context->owner[qual] = 0;
+ ds_put_context(context);
+ return error;
+
+ out_unlock:
+ spin_unlock(&ds_lock);
+ ds_put_context(context);
+ return error;
}
-static inline void set_from_ip(char *base, unsigned long value)
+
+int ds_request_bts(struct task_struct *task, void *base, size_t size,
+ ds_ovfl_callback_t ovfl)
{
- (*(unsigned long *) (base + ds_cfg.from_ip.offset)) = value;
+ return ds_request(task, base, size, ovfl, ds_bts);
}
-static inline unsigned long get_to_ip(char *base)
+EXPORT_SYMBOL(ds_request_bts);
+
+int ds_request_pebs(struct task_struct *task, void *base, size_t size,
+ ds_ovfl_callback_t ovfl)
{
- return *(unsigned long *) (base + ds_cfg.to_ip.offset);
+ return ds_request(task, base, size, ovfl, ds_pebs);
}
-static inline void set_to_ip(char *base, unsigned long value)
+EXPORT_SYMBOL(ds_request_pebs);
+
+static int ds_release(struct task_struct *task, enum ds_qualifier qual)
{
- (*(unsigned long *) (base + ds_cfg.to_ip.offset)) = value;
+ struct mm_struct *mm = current->mm;
+ struct ds_context *context = 0;
+ int error;
+
+ context = ds_get_context(task);
+ error = ds_validate_access(context, qual);
+ if (error < 0)
+ goto out;
+
+ if (context->buffer[qual]) {
+ kfree(context->buffer[qual]);
+ context->buffer[qual] = 0;
+
+ down_write(&mm->mmap_sem);
+
+ current->mm->total_vm -= context->pages[qual];
```

[patch] x86, ptrace: PEBS support

```
+ current->mm->locked_vm -= context->pages[qual];
+
+ up_write(&mm->mmap_sem);
+
+ context->pages[qual] = 0;
+ context->owner[qual] = 0;
+ }
+
+ /*
+ * we put the context twice:
+ * once for the ds_get_context
+ * once for the corresponding ds_request
+ */
+ ds_put_context(context);
+ out:
+ ds_put_context(context);
+ return error;
+ }
-static inline unsigned char get_info_type(char *base)
+
+int ds_release_bts(struct task_struct *task)
+ {
- return *(unsigned char *)(base + ds_cfg.info_type.offset);
+ return ds_release(task, ds_bts);
+ }
-static inline void set_info_type(char *base, unsigned char value)
+EXPORT_SYMBOL(ds_release_bts);
+
+int ds_release_pebs(struct task_struct *task)
+ {
- (*(unsigned char *)(base + ds_cfg.info_type.offset)) = value;
+ return ds_release(task, ds_pebs);
+ }
-static inline unsigned long get_info_data(char *base)
+EXPORT_SYMBOL(ds_release_pebs);
+
+static int ds_get_index(struct task_struct *task, size_t *pos,
+ enum ds_qualifier qual)
+ {
- return *(unsigned long *)(base + ds_cfg.info_data.offset);
+ struct ds_context *context = 0;
+ unsigned long base, index;
+ int error;
+
+ context = ds_get_context(task);
+ error = ds_validate_access(context, qual);
+ if (error < 0)
+ goto out;
+
+ base = ds_get(context->ds, qual, ds_buffer_base);
+ index = ds_get(context->ds, qual, ds_index);
```

[patch] x86, ptrace: PEBS support

```
+
+ error = ((index - base) / ds_cfg.sizeof_rec[qual]);
+ if (pos)
+ *pos = error;
+ out:
+ ds_put_context(context);
+ return error;
+ }
-static inline void set_info_data(char *base, unsigned long value)
+
+int ds_get_bts_index(struct task_struct *task, size_t *pos)
+ {
+ - (*(unsigned long *) (base + ds_cfg.info_data.offset)) = value;
+ return ds_get_index(task, pos, ds_bts);
+ }
+EXPORT_SYMBOL(ds_get_bts_index);

+int ds_get_pebs_index(struct task_struct *task, size_t *pos)
+ {
+ return ds_get_index(task, pos, ds_pebs);
+ }
+EXPORT_SYMBOL(ds_get_pebs_index);

-int ds_allocate(void **dsp, size_t bts_size_in_bytes)
+static int ds_get_end(struct task_struct *task, size_t *pos,
+ enum ds_qualifier qual)
+ {
+ - size_t bts_size_in_records;
+ - unsigned long bts;
+ - void *ds;
+ struct ds_context *context = 0;
+ unsigned long base, end;
+ int error;
+
+ context = ds_get_context(task);
+ error = ds_validate_access(context, qual);
+ if (error < 0)
+ goto out;
+
+ base = ds_get(context->ds, qual, ds_buffer_base);
+ end = ds_get(context->ds, qual, ds_absolute_maximum);
+
+ error = ((end - base) / ds_cfg.sizeof_rec[qual]);
+ if (pos)
+ *pos = error;
+ out:
+ ds_put_context(context);
+ return error;
+ }

- if (!ds_cfg.sizeof_ds || !ds_cfg.sizeof_bts)
```

[patch] x86, ptrace: PEBS support

```
- return -EOPNOTSUPP;
+int ds_get_bts_end(struct task_struct *task, size_t *pos)
+{
+ return ds_get_end(task, pos, ds_bts);
+}
+EXPORT_SYMBOL(ds_get_bts_end);

- if (bts_size_in_bytes < 0)
- return -EINVAL;
+int ds_get_pebs_end(struct task_struct *task, size_t *pos)
+{
+ return ds_get_end(task, pos, ds_pebs);
+}
+EXPORT_SYMBOL(ds_get_pebs_end);

- bts_size_in_records =
- bts_size_in_bytes / ds_cfg.sizeof_bts;
- bts_size_in_bytes =
- bts_size_in_records * ds_cfg.sizeof_bts;
+static int ds_access(struct task_struct *task, size_t index,
+ const void **record, enum ds_qualifier qual)
+{
+ struct ds_context *context = 0;
+ unsigned long base, idx;
+ int error;

- if (bts_size_in_bytes <= 0)
+ if (!record)
return -EINVAL;

- bts = (unsigned long)kzalloc(bts_size_in_bytes, GFP_KERNEL);
-
- if (!bts)
- return -ENOMEM;
-
- ds = kzalloc(ds_cfg.sizeof_ds, GFP_KERNEL);
+ context = ds_get_context(task);
+ error = ds_validate_access(context, qual);
+ if (error < 0)
+ goto out;

- if (!ds) {
- kfree((void *)bts);
- return -ENOMEM;
- }
+ base = ds_get(context->ds, qual, ds_buffer_base);
+ idx = base + (index * ds_cfg.sizeof_rec[qual]);

- set_bts_buffer_base(ds, bts);
- set_bts_index(ds, bts);
- set_bts_absolute_maximum(ds, bts + bts_size_in_bytes);
```

[patch] x86, ptrace: PEBS support

```
- set_bts_interrupt_threshold(ds, bts + bts_size_in_bytes + 1);
+ error = -EINVAL;
+ if (idx > ds_get(context->ds, qual, ds_absolute_maximum))
+ goto out;

- *dsp = ds;
- return 0;
+ *record = (const void *)idx;
+ error = ds_cfg.sizeof_rec[qual];
+ out:
+ ds_put_context(context);
+ return error;
}

-int ds_free(void **dsp)
+int ds_access_bts(struct task_struct *task, size_t index, const void **record)
{
- if (*dsp) {
- kfree((void *)get_bts_buffer_base(*dsp));
- kfree(*dsp);
- *dsp = NULL;
- }
- return 0;
+ return ds_access(task, index, record, ds_bts);
}
+EXPORT_SYMBOL(ds_access_bts);

-int ds_get_bts_size(void *ds)
+int ds_access_pebs(struct task_struct *task, size_t index, const void **record)
{
- int size_in_bytes;
-
- if (!ds_cfg.sizeof_ds || !ds_cfg.sizeof_bts)
- return -EOPNOTSUPP;
-
- if (!ds)
- return 0;
-
- size_in_bytes =
- get_bts_absolute_maximum(ds) -
- get_bts_buffer_base(ds);
- return size_in_bytes;
+ return ds_access(task, index, record, ds_pebs);
}
+EXPORT_SYMBOL(ds_access_pebs);

-int ds_get_bts_end(void *ds)
+static int ds_write(struct task_struct *task, const void *record, size_t size,
+ enum ds_qualifier qual, int force)
{
- int size_in_bytes = ds_get_bts_size(ds);
```

[patch] x86, ptrace: PEBS support

```
+ struct ds_context *context = 0;
+ int error;

- if (size_in_bytes <= 0)
- return size_in_bytes;
+ if (!record)
+ return -EINVAL;

- return size_in_bytes / ds_cfg.sizeof_bts;
-}
+ error = -EPERM;
+ context = ds_get_context(task);
+ if (!context)
+ goto out;

-int ds_get_bts_index(void *ds)
-{
- int index_offset_in_bytes;
-
- if (!ds_cfg.sizeof_ds || !ds_cfg.sizeof_bts)
- return -EOPNOTSUPP;
+ if (!force) {
+ error = ds_validate_access(context, qual);
+ if (error < 0)
+ goto out;
+ }

- index_offset_in_bytes =
- get_bts_index(ds) -
- get_bts_buffer_base(ds);
+ error = 0;
+ while (size) {
+ unsigned long base, index, end, write_end, int_th;
+ unsigned long write_size, adj_write_size;
+
+ /*
+ * write as much as possible without producing an
+ * overflow interrupt.
+ *
+ * interrupt_threshold must either be
+ * - bigger than absolute_maximum or
+ * - point to a record between buffer_base and absolute_maximum
+ *
+ * index points to a valid record.
+ */
+ base = ds_get(context->ds, qual, ds_buffer_base);
+ index = ds_get(context->ds, qual, ds_index);
+ end = ds_get(context->ds, qual, ds_absolute_maximum);
+ int_th = ds_get(context->ds, qual, ds_interrupt_threshold);
+
+ write_end = min(end, int_th);
```

[patch] x86, ptrace: PEBS support

```
+
+ /* if we are already beyond the interrupt threshold,
+ * we fill the entire buffer */
+ if (write_end <= index)
+ write_end = end;
+
+ if (write_end <= index)
+ goto out;
+
+ write_size = min((unsigned long) size, write_end - index);
+ memcpy((void *)index, record, write_size);
+
+ record = (const char *)record + write_size;
+ size -= write_size;
+ error += write_size;
+
+ adj_write_size = write_size / ds_cfg.sizeof_rec[qual];
+ adj_write_size *= ds_cfg.sizeof_rec[qual];
+
+ /* zero out trailing bytes */
+ memset((char *)index + write_size, 0,
+ adj_write_size - write_size);
+ index += adj_write_size;
+
+ if (index >= end)
+ index = base;
+ ds_set(context->ds, qual, ds_index, index);
+
+ if (index >= int_th)
+ ds_overflow(task, context, qual);
+ }

- return index_offset_in_bytes / ds_cfg.sizeof_bts;
+ out:
+ ds_put_context(context);
+ return error;
}

-int ds_set_overflow(void *ds, int method)
+int ds_write_bts(struct task_struct *task, const void *record, size_t size)
{
- switch (method) {
- case DS_O_SIGNAL:
- return -EOPNOTSUPP;
- case DS_O_WRAP:
- return 0;
- default:
- return -EINVAL;
- }
+ return ds_write(task, record, size, ds_bts, /* force = */ 0);
}
```

[patch] x86, ptrace: PEBS support

```
+EXPORT_SYMBOL(ds_write_bts);

-int ds_get_overflow(void *ds)
+int ds_write_pebs(struct task_struct *task, const void *record, size_t size)
{
- return DS_O_WRAP;
+ return ds_write(task, record, size, ds_pebs, /* force = */ 0);
}
+EXPORT_SYMBOL(ds_write_pebs);

-int ds_clear(void *ds)
+int ds_unchecked_write_bts(struct task_struct *task,
+ const void *record, size_t size)
{
- int bts_size = ds_get_bts_size(ds);
- unsigned long bts_base;
-
- if (bts_size <= 0)
- return bts_size;
-
- bts_base = get_bts_buffer_base(ds);
- memset((void *)bts_base, 0, bts_size);
-
- set_bts_index(ds, bts_base);
- return 0;
+ return ds_write(task, record, size, ds_bts, /* force = */ 1);
}
+EXPORT_SYMBOL(ds_unchecked_write_bts);

-int ds_read_bts(void *ds, int index, struct bts_struct *out)
+int ds_unchecked_write_pebs(struct task_struct *task,
+ const void *record, size_t size)
{
- void *bts;
+ return ds_write(task, record, size, ds_pebs, /* force = */ 1);
+}
+EXPORT_SYMBOL(ds_unchecked_write_pebs);

- if (!ds_cfg.sizeof_ds || !ds_cfg.sizeof_bts)
- return -EOPNOTSUPP;
+static int ds_reset_or_clear(struct task_struct *task,
+ enum ds_qualifier qual, int clear)
+{
+ struct ds_context *context = 0;
+ unsigned long base, end;
+ int error;

- if (index < 0)
- return -EINVAL;
+ context = ds_get_context(task);
+ error = ds_validate_access(context, qual);
```

[patch] x86, ptrace: PEBS support

```
+ if (error < 0)
+ goto out;

- if (index >= ds_get_bts_size(ds))
- return -EINVAL;
+ base = ds_get(context->ds, qual, ds_buffer_base);
+ end = ds_get(context->ds, qual, ds_absolute_maximum);

- bts = (void *) (get_bts_buffer_base(ds) + (index * ds_cfg.sizeof_bts));
+ if (clear)
+ memset((void *)base, 0, end - base);

- memset(out, 0, sizeof(*out));
- if (get_from_ip(bts) == BTS_ESCAPE_ADDRESS) {
- out->qualifier = get_info_type(bts);
- out->variant.jiffies = get_info_data(bts);
- } else {
- out->qualifier = BTS_BRANCH;
- out->variant.lbr.from_ip = get_from_ip(bts);
- out->variant.lbr.to_ip = get_to_ip(bts);
- }
+ ds_set(context->ds, qual, ds_index, base);

- return sizeof(*out);;
+ error = 0;
+ out:
+ ds_put_context(context);
+ return error;
}

-int ds_write_bts(void *ds, const struct bts_struct *in)
+int ds_reset_bts(struct task_struct *task)
{
- unsigned long bts;
-
- if (!ds_cfg.sizeof_ds || !ds_cfg.sizeof_bts)
- return -EOPNOTSUPP;
-
- if (ds_get_bts_size(ds) <= 0)
- return -ENXIO;
+ return ds_reset_or_clear(task, ds_bts, /* clear = */ 0);
+}
+EXPORT_SYMBOL(ds_reset_bts);

- bts = get_bts_index(ds);
+int ds_reset_pebs(struct task_struct *task)
+{
+ return ds_reset_or_clear(task, ds_pebs, /* clear = */ 0);
+}
+EXPORT_SYMBOL(ds_reset_pebs);
```

[patch] x86, ptrace: PEBS support

```
- memset((void *)bts, 0, ds_cfg.sizeof_bts);
- switch (in->qualifier) {
- case BTS_INVALID:
- break;
+int ds_clear_bts(struct task_struct *task)
+{
+ return ds_reset_or_clear(task, ds_bts, /* clear = */ 1);
+}
+EXPORT_SYMBOL(ds_clear_bts);

- case BTS_BRANCH:
- set_from_ip((void *)bts, in->variant.lbr.from_ip);
- set_to_ip((void *)bts, in->variant.lbr.to_ip);
- break;
+int ds_clear_pebs(struct task_struct *task)
+{
+ return ds_reset_or_clear(task, ds_pebs, /* clear = */ 1);
+}
+EXPORT_SYMBOL(ds_clear_pebs);

- case BTS_TASK_ARRIVES:
- case BTS_TASK_DEPARTS:
- set_from_ip((void *)bts, BTS_ESCAPE_ADDRESS);
- set_info_type((void *)bts, in->qualifier);
- set_info_data((void *)bts, in->variant.jiffies);
- break;
+int ds_get_pebs_reset(struct task_struct *task, u64 *value)
+{
+ struct ds_context *context = 0;
+ int error;

- default:
+ if (!value)
return -EINVAL;
- }

- bts = bts + ds_cfg.sizeof_bts;
- if (bts >= get_bts_absolute_maximum(ds))
- bts = get_bts_buffer_base(ds);
- set_bts_index(ds, bts);
+ context = ds_get_context(task);
+ error = ds_validate_access(context, ds_pebs);
+ if (error < 0)
+ goto out;
+
+ *value = *(u64 *) (context->ds + (ds_cfg.sizeof_field * 8));

- return ds_cfg.sizeof_bts;
+ error = 0;
+ out:
+ ds_put_context(context);
```

[patch] x86, ptrace: PEBS support

```
+ return error;
}
+EXPORT_SYMBOL(ds_get_pebs_reset);

-unsigned long ds_debugctl_mask(void)
+int ds_set_pebs_reset(struct task_struct *task, u64 value)
{
- return ds_cfg.debugctl_mask;
-}
+ struct ds_context *context = 0;
+ int error;

-#ifdef __i386__
-static const struct ds_configuration ds_cfg_netburst = {
- .sizeof_ds = 9 * 4,
- .bts_buffer_base = { 0, 4 },
- .bts_index = { 4, 4 },
- .bts_absolute_maximum = { 8, 4 },
- .bts_interrupt_threshold = { 12, 4 },
- .sizeof_bts = 3 * 4,
- .from_ip = { 0, 4 },
- .to_ip = { 4, 4 },
- .info_type = { 4, 1 },
- .info_data = { 8, 4 },
- .debugctl_mask = (1<<2)|(1<<3)
-};
+ context = ds_get_context(task);
+ error = ds_validate_access(context, ds_pebs);
+ if (error < 0)
+ goto out;
+
+ *(u64 *) (context->ds + (ds_cfg.sizeof_field * 8)) = value;

-static const struct ds_configuration ds_cfg_pentium_m = {
- .sizeof_ds = 9 * 4,
- .bts_buffer_base = { 0, 4 },
- .bts_index = { 4, 4 },
- .bts_absolute_maximum = { 8, 4 },
- .bts_interrupt_threshold = { 12, 4 },
- .sizeof_bts = 3 * 4,
- .from_ip = { 0, 4 },
- .to_ip = { 4, 4 },
- .info_type = { 4, 1 },
- .info_data = { 8, 4 },
- .debugctl_mask = (1<<6)|(1<<7)
+ error = 0;
+ out:
+ ds_put_context(context);
+ return error;
+}
+EXPORT_SYMBOL(ds_set_pebs_reset);
```

[patch] x86, ptrace: PEBS support

```
+
+static const struct ds_configuration ds_cfg_var = {
+ .sizeof_ds = sizeof(long) * 12,
+ .sizeof_field = sizeof(long),
+ .sizeof_rec[ds_bts] = sizeof(long) * 3,
+ .sizeof_rec[ds_pebs] = sizeof(long) * 10
+};
-#endif /* _i386_ */
-
-static const struct ds_configuration ds_cfg_core2 = {
- .sizeof_ds = 9 * 8,
- .bts_buffer_base = { 0, 8 },
- .bts_index = { 8, 8 },
- .bts_absolute_maximum = { 16, 8 },
- .bts_interrupt_threshold = { 24, 8 },
- .sizeof_bts = 3 * 8,
- .from_ip = { 0, 8 },
- .to_ip = { 8, 8 },
- .info_type = { 8, 1 },
- .info_data = { 16, 8 },
- .debugctl_mask = (1<<6)|(1<<7)|(1<<9)
+static const struct ds_configuration ds_cfg_64 = {
+ .sizeof_ds = 8 * 12,
+ .sizeof_field = 8,
+ .sizeof_rec[ds_bts] = 8 * 3,
+ .sizeof_rec[ds_pebs] = 8 * 18
+};

static inline void
@@ -429,14 +860,13 @@ void __cpunit ds_init_intel(struct cpuinfo_x86 *c)
switch (c->x86) {
case 0x6:
switch (c->x86_model) {
-#ifdef __i386__
case 0xD:
case 0xE: /* Pentium M */
- ds_configure(&ds_cfg_pentium_m);
+ ds_configure(&ds_cfg_var);
break;
-#endif /* _i386_ */
case 0xF: /* Core2 */
- ds_configure(&ds_cfg_core2);
+ case 0x1C: /* Atom */
+ ds_configure(&ds_cfg_64);
break;
default:
/* sorry, don't know about them */
@@ -445,13 +875,11 @@ void __cpunit ds_init_intel(struct cpuinfo_x86 *c)
break;
case 0xF:
switch (c->x86_model) {
```

[patch] x86, ptrace: PEBS support

```

-#ifdef __i386__
case 0x0:
case 0x1:
case 0x2: /* Netburst */
- ds_configure(&ds_cfg_netburst);
+ ds_configure(&ds_cfg_var);
break;
-#endif /* __i386_ */
default:
/* sorry, don't know about them */
break;
@@ -462,3 +890,16 @@ void __cpunit ds_init_intel(struct cpuinfo_x86 *c)
break;
}
}
+
+void ds_free(struct ds_context *context)
+{
+ /* This is called when the task owning the parameter context
+ * is dying. There should not be any user of that context left
+ * to disturb us, anymore. */
+ unsigned long leftovers = context->count;
+ while (leftovers-->0)
+ ds_put_context(context);
+}
+EXPORT_SYMBOL(ds_free);
+
+#endif /* CONFIG_X86_DS */
diff --git a/arch/x86/kernel/process_32.c b/arch/x86/kernel/process_32.c
index 63dca4b..eda2304 100644
--- a/arch/x86/kernel/process_32.c
+++ b/arch/x86/kernel/process_32.c
@@ -426,6 +426,14 @@ void exit_thread(void)
tss->x86_tss.io_bitmap_base = INVALID_IO_BITMAP_OFFSET;
put_cpu();
}
+#ifdef CONFIG_X86_DS
+ /* Free any DS contexts that have not been properly released. */
+ if (unlikely(current->thread.ds_ctx)) {
+ /* we clear debugctl to make sure DS is not used. */
+ update_debugctlmsr(0);
+ ds_free(current->thread.ds_ctx);
+ }
+#endif /* CONFIG_X86_DS */
}

void flush_thread(void)
@@ -451,6 +459,16 @@ void release_thread(struct task_struct *dead_task)
{
BUG_ON(dead_task->mm);
release_vm86_irqs(dead_task);

```

[patch] x86, ptrace: PEBS support

```
+
+ #ifdef CONFIG_X86_DS
+ /* Free any DS contexts that have not been properly released. */
+ if (unlikely(dead_task->thread.ds_ctx)) {
+ /* we clear debugctl to make sure DS is not used. */
+ if (dead_task == current)
+ update_debugctlmsr(0);
+ ds_free(dead_task->thread.ds_ctx);
+ }
+ #endif /* CONFIG_X86_DS */
}

/*
@@ -592,18 +610,27 @@ __switch_to_xtra(struct task_struct *prev_p, struct task_struct *next_p,
{
struct thread_struct *prev, *next;
unsigned long debugctl;
+ unsigned long ds_prev = 0, ds_next = 0;

prev = &prev_p->thread;
next = &next_p->thread;

debugctl = prev->debugctlmsr;
- if (next->ds_area_msr != prev->ds_area_msr) {
+
+ #ifdef CONFIG_X86_DS
+ if (prev->ds_ctx)
+ ds_prev = (unsigned long)prev->ds_ctx->ds;
+ if (next->ds_ctx)
+ ds_next = (unsigned long)next->ds_ctx->ds;
+
+ if (ds_next != ds_prev) {
/* we clear debugctl to make sure DS
* is not in use when we change it */
debugctl = 0;
update_debugctlmsr(0);
- wrmsr(MSR_IA32_DS_AREA, next->ds_area_msr, 0);
+ wrmsr(MSR_IA32_DS_AREA, ds_next, 0);
}
+ #endif /* CONFIG_X86_DS */

if (next->debugctlmsr != debugctl)
update_debugctlmsr(next->debugctlmsr);
@@ -627,13 +654,13 @@ __switch_to_xtra(struct task_struct *prev_p, struct task_struct *next_p,
hard_enable_TSC();
}

- #ifdef X86_BTS
+ #ifdef CONFIG_X86_PTRACE_BTS
if (test_tsk_thread_flag(prev_p, TIF_BTS_TRACE_TS))
ptrace_bts_take_timestamp(prev_p, BTS_TASK_DEPARTS);
```

[patch] x86, ptrace: PEBS support

```
if (test_tsk_thread_flag(next_p, TIF_BTS_TRACE_TS))
ptrace_bts_take_timestamp(next_p, BTS_TASK_ARRIVES);
-#endif
+#endif /* CONFIG_X86_PTRACE_BTS */

if (!test_tsk_thread_flag(next_p, TIF_IO_BITMAP)) {
diff --git a/arch/x86/kernel/process_64.c b/arch/x86/kernel/process_64.c
index efa98af..24916c2 100644
--- a/arch/x86/kernel/process_64.c
+++ b/arch/x86/kernel/process_64.c
@@ -400,6 +400,14 @@ void exit_thread(void)
t->io_bitmap_max = 0;
put_cpu();
}
+#ifdef CONFIG_X86_DS
+ /* Free any DS contexts that have not been properly released. */
+ if (unlikely(t->ds_ctx)) {
+ /* we clear debugctl to make sure DS is not used. */
+ update_debugctlmsr(0);
+ ds_free(t->ds_ctx);
+ }
+#endif /* CONFIG_X86_DS */
}

void flush_thread(void)
@@ -625,18 +633,27 @@ static inline void __switch_to_xtra(struct task_struct *prev_p,
{
struct thread_struct *prev, *next;
unsigned long debugctl;
+ unsigned long ds_prev = 0, ds_next = 0;

prev = &prev_p->thread,
next = &next_p->thread;

debugctl = prev->debugctlmsr;
- if (next->ds_area_msr != prev->ds_area_msr) {
+
+#ifdef CONFIG_X86_DS
+ if (prev->ds_ctx)
+ ds_prev = (unsigned long)prev->ds_ctx->ds;
+ if (next->ds_ctx)
+ ds_next = (unsigned long)next->ds_ctx->ds;
+
+ if (ds_next != ds_prev) {
+ /* we clear debugctl to make sure DS
+ * is not in use when we change it */
debugctl = 0;
update_debugctlmsr(0);
- wrmsrl(MSR_IA32_DS_AREA, next->ds_area_msr);
```

[patch] x86, ptrace: PEBS support

```
+ wrmsrl(MSR_IA32_DS_AREA, ds_next);
}
+ #endif /* CONFIG_X86_DS */

if (next->debugctlmsr != debugctl)
update_debugctlmsr(next->debugctlmsr);
@@ -674,13 +691,13 @@ static inline void __switch_to_xtra(struct task_struct *prev_p,
memset(tss->io_bitmap, 0xff, prev->io_bitmap_max);
}

-#ifdef X86_BTS
+#ifdef CONFIG_X86_PTRACE_BTS
if (test_tsk_thread_flag(prev_p, TIF_BTS_TRACE_TS))
ptrace_bts_take_timestamp(prev_p, BTS_TASK_DEPARTS);

if (test_tsk_thread_flag(next_p, TIF_BTS_TRACE_TS))
ptrace_bts_take_timestamp(next_p, BTS_TASK_ARRIVES);
-#endif
+ #endif /* CONFIG_X86_PTRACE_BTS */
}

/*
diff --git a/arch/x86/kernel/ptrace.c b/arch/x86/kernel/ptrace.c
index 559c1b0..3ec17b5 100644
--- a/arch/x86/kernel/ptrace.c
+++ b/arch/x86/kernel/ptrace.c
@@ -554,45 +554,118 @@ static int ptrace_set_debugreg(struct task_struct *child,
return 0;
}

-#ifdef X86_BTS
+#ifdef CONFIG_X86_PTRACE_BTS
+ /*
+ * The configuration for a particular BTS hardware implementation.
+ */
+ struct bts_configuration {
+ /* the size of a BTS record in bytes; at most BTS_MAX_RECORD_SIZE */
+ unsigned char sizeof_bts;
+ /* the size of a field in the BTS record in bytes */
+ unsigned char sizeof_field;
+ /* a bitmask to enable/disable BTS in DEBUGCTL MSR */
+ unsigned long debugctl_mask;
+ };
+ static struct bts_configuration bts_cfg;
+
+ #define BTS_MAX_RECORD_SIZE (8 * 3)
+
+ /*
+ * Branch Trace Store (BTS) uses the following format. Different
+ * architectures vary in the size of those fields.
```

[patch] x86, ptrace: PEBS support

```
+ * – source linear address
+ * – destination linear address
+ * – flags
+ *
+ * Later architectures use 64bit pointers throughout, whereas earlier
+ * architectures use 32bit pointers in 32bit mode.
+ *
+ * We compute the base address for the first 8 fields based on:
+ * – the field size stored in the DS configuration
+ * – the relative field position
+ *
+ * In order to store additional information in the BTS buffer, we use
+ * a special source address to indicate that the record requires
+ * special interpretation.
+ *
+ * Netburst indicated via a bit in the flags field whether the branch
+ * was predicted; this is ignored.
+ */
+
+enum bts_field {
+ bts_from = 0,
+ bts_to,
+ bts_flags,
+
+ bts_escape = (unsigned long)-1,
+ bts_qual = bts_to,
+ bts_jiffies = bts_flags
+};
+
+static inline unsigned long bts_get(const char *base, enum bts_field field)
+{
+ base += (bts_cfg.sizeof_field * field);
+ return *(unsigned long *)base;
+}

-static int ptrace_bts_get_size(struct task_struct *child)
+static inline void bts_set(char *base, enum bts_field field, unsigned long val)
+{
+ if (!child->thread.ds_area_msr)
+ return -ENXIO;
+ base += (bts_cfg.sizeof_field * field);
+ (*(unsigned long *)base) = val;
+}

- return ds_get_bts_index((void *)child->thread.ds_area_msr);
+ */
+ * Translate a BTS record from the raw format into the bts_struct format
+ *
+ * out (out): bts_struct interpretation
+ * raw: raw BTS record
+ */
```

[patch] x86, ptrace: PEBS support

```
+static void ptrace_bts_translate_record(struct bts_struct *out, const void *raw)
+{
+  memset(out, 0, sizeof(*out));
+  if (bts_get(raw, bts_from) == bts_escape) {
+    out->qualifier = bts_get(raw, bts_qual);
+    out->variant.jiffies = bts_get(raw, bts_jiffies);
+  } else if ((bts_get(raw, bts_from) == 0) &&
+    (bts_get(raw, bts_to) == 0)) {
+    out->qualifier = BTS_INVALID;
+  } else {
+    out->qualifier = BTS_BRANCH;
+    out->variant.lbr.from_ip = bts_get(raw, bts_from);
+    out->variant.lbr.to_ip = bts_get(raw, bts_to);
+  }
+}
```

```
-static int ptrace_bts_read_record(struct task_struct *child,
- long index,
+static int ptrace_bts_read_record(struct task_struct *child, size_t index,
struct bts_struct __user *out)
{
  struct bts_struct ret;
  - int retval;
  - int bts_end;
  - int bts_index;
  -
  - if (!child->thread.ds_area_msr)
  - return -ENXIO;
  + const void *bts_record;
  + size_t bts_index, bts_end;
  + int error;

  - if (index < 0)
  - return -EINVAL;
  + error = ds_get_bts_end(child, &bts_end);
  + if (error < 0)
  + return error;

  - bts_end = ds_get_bts_end((void *)child->thread.ds_area_msr);
  if (bts_end <= index)
  return -EINVAL;

  + error = ds_get_bts_index(child, &bts_index);
  + if (error < 0)
  + return error;
  +
  /* translate the ptrace bts index into the ds bts index */
  - bts_index = ds_get_bts_index((void *)child->thread.ds_area_msr);
  - bts_index -= (index + 1);
  - if (bts_index < 0)
  - bts_index += bts_end;
```

[patch] x86, ptrace: PEBS support

```
+ bts_index += bts_end - (index + 1);
+ if (bts_end <= bts_index)
+ bts_index -= bts_end;

- retval = ds_read_bts((void *)child->thread.ds_area_msr,
- bts_index, &ret);
- if (retval < 0)
- return retval;
+ error = ds_access_bts(child, bts_index, &bts_record);
+ if (error < 0)
+ return error;
+
+ ptrace_bts_translate_record(&ret, bts_record);

if (copy_to_user(out, &ret, sizeof(ret)))
return -EFAULT;
@@ -600,101 +673,106 @@ static int ptrace_bts_read_record(struct task_struct *child,
return sizeof(ret);
}

-static int ptrace_bts_clear(struct task_struct *child)
-{
- if (!child->thread.ds_area_msr)
- return -ENXIO;
-
- return ds_clear((void *)child->thread.ds_area_msr);
-}
-
static int ptrace_bts_drain(struct task_struct *child,
long size,
struct bts_struct __user *out)
{
- int end, i;
- void *ds = (void *)child->thread.ds_area_msr;
-
- if (!ds)
- return -ENXIO;
+ struct bts_struct ret;
+ const unsigned char *raw;
+ size_t end, i;
+ int error;

- end = ds_get_bts_index(ds);
- if (end <= 0)
- return end;
+ error = ds_get_bts_index(child, &end);
+ if (error < 0)
+ return error;

if (size < (end * sizeof(struct bts_struct)))
return -EIO;
```

[patch] x86, ptrace: PEBS support

```
- for (i = 0; i < end; i++, out++) {
- struct bts_struct ret;
- int retval;
+ error = ds_access_bts(child, 0, (const void **)&raw);
+ if (error < 0)
+ return error;

- retval = ds_read_bts(ds, i, &ret);
- if (retval < 0)
- return retval;
+ for (i = 0; i < end; i++, out++, raw += bts_cfg.sizeof_bts) {
+ ptrace_bts_translate_record(&ret, raw);

if (copy_to_user(out, &ret, sizeof(ret)))
return -EFAULT;
}

- ds_clear(ds);
+ error = ds_clear_bts(child);
+ if (error < 0)
+ return error;

return end;
}

+static void ptrace_bts_ovfl(struct task_struct *child)
+{
+ send_sig(child->thread.bts_ovfl_signal, child, 0);
+}
+
static int ptrace_bts_config(struct task_struct *child,
long cfg_size,
const struct ptrace_bts_config __user *ucfg)
{
struct ptrace_bts_config cfg;
- int bts_size, ret = 0;
- void *ds;
+ int error = 0;
+
+ error = -EOPNOTSUPP;
+ if (!bts_cfg.sizeof_bts)
+ goto errout;

+ error = -EIO;
if (cfg_size < sizeof(cfg))
- return -EIO;
+ goto errout;

+ error = -EFAULT;
if (copy_from_user(&cfg, ucfg, sizeof(cfg)))
```

[patch] x86, ptrace: PEBS support

```
- return -EFAULT;
+ goto errout;

- if ((int)cfg.size < 0)
- return -EINVAL;
+ error = -EINVAL;
+ if ((cfg.flags & PTRACE_BTS_O_SIGNAL) &&
+ !(cfg.flags & PTRACE_BTS_O_ALLOC))
+ goto errout;

- bts_size = 0;
- ds = (void *)child->thread.ds_area_msr;
- if (ds) {
- bts_size = ds_get_bts_size(ds);
- if (bts_size < 0)
- return bts_size;
- }
- cfg.size = PAGE_ALIGN(cfg.size);
+ if (cfg.flags & PTRACE_BTS_O_ALLOC) {
+ ds_ovfl_callback_t ovfl = 0;
+ unsigned int sig = 0;
+
+ /* we ignore the error in case we were not tracing child */
+ (void)ds_release_bts(child);

- if (bts_size != cfg.size) {
- ret = ptrace_bts_realloc(child, cfg.size,
- cfg.flags & PTRACE_BTS_O_CUT_SIZE);
- if (ret < 0)
+ if (cfg.flags & PTRACE_BTS_O_SIGNAL) {
+ if (!cfg.signal)
+ goto errout;
+
+ sig = cfg.signal;
+ ovfl = ptrace_bts_ovfl;
+ }
+
+ error = ds_request_bts(child, /* base = */ 0, cfg.size, ovfl);
+ if (error < 0)
goto errout;

- ds = (void *)child->thread.ds_area_msr;
+ child->thread.bts_ovfl_signal = sig;
}

- if (cfg.flags & PTRACE_BTS_O_SIGNAL)
- ret = ds_set_overflow(ds, DS_O_SIGNAL);
- else
- ret = ds_set_overflow(ds, DS_O_WRAP);
- if (ret < 0)
+ error = -EINVAL;
```

[patch] x86, ptrace: PEBS support

```
+ if (!child->thread.ds_ctx && cfg.flags)
goto errout;

if (cfg.flags & PTRACE_BTS_O_TRACE)
- child->thread.debugctlmsr |= ds_debugctl_mask();
+ child->thread.debugctlmsr |= bts_cfg.debugctl_mask;
else
- child->thread.debugctlmsr &= ~ds_debugctl_mask();
+ child->thread.debugctlmsr &= ~bts_cfg.debugctl_mask;

if (cfg.flags & PTRACE_BTS_O_SCHED)
set_tsk_thread_flag(child, TIF_BTS_TRACE_TS);
else
clear_tsk_thread_flag(child, TIF_BTS_TRACE_TS);

- ret = sizeof(cfg);
+ error = sizeof(cfg);

out:
if (child->thread.debugctlmsr)
@@ -702,10 +780,10 @@ out:
else
clear_tsk_thread_flag(child, TIF_DEBUGCTLMR);

- return ret;
+ return error;

errout:
- child->thread.debugctlmsr &= ~ds_debugctl_mask();
+ child->thread.debugctlmsr &= ~bts_cfg.debugctl_mask;
clear_tsk_thread_flag(child, TIF_BTS_TRACE_TS);
goto out;
}
@@ -714,29 +792,40 @@ static int ptrace_bts_status(struct task_struct *child,
long cfg_size,
struct ptrace_bts_config __user *ucfg)
{
- void *ds = (void *)child->thread.ds_area_msr;
struct ptrace_bts_config cfg;
+ size_t end;
+ const void *base, *max;
+ int error;

if (cfg_size < sizeof(cfg))
return -EIO;

- memset(&cfg, 0, sizeof(cfg));
+ error = ds_get_bts_end(child, &end);
+ if (error < 0)
+ return error;
+
+

```

[patch] x86, ptrace: PEBS support

```
+ error = ds_access_bts(child, /* index = */ 0, &base);
+ if (error < 0)
+ return error;

- if (ds) {
- cfg.size = ds_get_bts_size(ds);
+ error = ds_access_bts(child, /* index = */ end, &max);
+ if (error < 0)
+ return error;

- if (ds_get_overflow(ds) == DS_O_SIGNAL)
- cfg.flags |= PTRACE_BTS_O_SIGNAL;
+ memset(&cfg, 0, sizeof(cfg));
+ cfg.size = (max - base);
+ cfg.signal = child->thread.bts_ovfl_signal;
+ cfg.bts_size = sizeof(struct bts_struct);

- if (test_tsk_thread_flag(child, TIF_DEBUGCTLMR) &&
- child->thread.debugctlmsr & ds_debugctl_mask())
- cfg.flags |= PTRACE_BTS_O_TRACE;
+ if (cfg.signal)
+ cfg.flags |= PTRACE_BTS_O_SIGNAL;

- if (test_tsk_thread_flag(child, TIF_BTS_TRACE_TS))
- cfg.flags |= PTRACE_BTS_O_SCHED;
- }
+ if (test_tsk_thread_flag(child, TIF_DEBUGCTLMR) &&
+ child->thread.debugctlmsr & bts_cfg.debugctl_mask)
+ cfg.flags |= PTRACE_BTS_O_TRACE;

- cfg.bts_size = sizeof(struct bts_struct);
+ if (test_tsk_thread_flag(child, TIF_BTS_TRACE_TS))
+ cfg.flags |= PTRACE_BTS_O_SCHED;

if (copy_to_user(ucfg, &cfg, sizeof(cfg)))
return -EFAULT;
@@ -744,89 +833,38 @@ static int ptrace_bts_status(struct task_struct *child,
return sizeof(cfg);
}

-
static int ptrace_bts_write_record(struct task_struct *child,
const struct bts_struct *in)
{
- int retval;
+ unsigned char bts_record[BTS_MAX_RECORD_SIZE];

- if (!child->thread.ds_area_msr)
- return -ENXIO;
+ BUG_ON(BTS_MAX_RECORD_SIZE < bts_cfg.sizeof_bts);
```

[patch] x86, ptrace: PEBS support

```
- retval = ds_write_bts((void *)child->thread.ds_area_msr, in);
- if (retval)
- return retval;
+ memset(bts_record, 0, bts_cfg.sizeof_bts);
+ switch (in->qualifier) {
+ case BTS_INVALID:
+ break;

- return sizeof(*in);
-}
+ case BTS_BRANCH:
+ bts_set(bts_record, bts_from, in->variant.lbr.from_ip);
+ bts_set(bts_record, bts_to, in->variant.lbr.to_ip);
+ break;

-static int ptrace_bts_realloc(struct task_struct *child,
- int size, int reduce_size)
-{
- unsigned long rlim, vm;
- int ret, old_size;
+ case BTS_TASK_ARRIVES:
+ case BTS_TASK_DEPARTS:
+ bts_set(bts_record, bts_from, bts_escape);
+ bts_set(bts_record, bts_qual, in->qualifier);
+ bts_set(bts_record, bts_jiffies, in->variant.jiffies);
+ break;

- if (size < 0)
+ default:
return -EINVAL;
-
- old_size = ds_get_bts_size((void *)child->thread.ds_area_msr);
- if (old_size < 0)
- return old_size;
-
- ret = ds_free((void **)&child->thread.ds_area_msr);
- if (ret < 0)
- goto out;
-
- size >>= PAGE_SHIFT;
- old_size >>= PAGE_SHIFT;
-
- current->mm->total_vm -= old_size;
- current->mm->locked_vm -= old_size;
-
- if (size == 0)
- goto out;
-
- rlim = current->signal->rlim[RLIMIT_AS].rlim_cur >> PAGE_SHIFT;
- vm = current->mm->total_vm + size;
- if (rlim < vm) {
```

[patch] x86, ptrace: PEBS support

```
- ret = -ENOMEM;
-
- if (!reduce_size)
- goto out;
-
- size = rlim - current->mm->total_vm;
- if (size <= 0)
- goto out;
- }

- rlim = current->signal->rlim[RLIMIT_MEMLOCK].rlim_cur >> PAGE_SHIFT;
- vm = current->mm->locked_vm + size;
- if (rlim < vm) {
- ret = -ENOMEM;
-
- if (!reduce_size)
- goto out;
-
- size = rlim - current->mm->locked_vm;
- if (size <= 0)
- goto out;
- }

- ret = ds_allocate((void *)&child->thread.ds_area_msr,
- size << PAGE_SHIFT);
- if (ret < 0)
- goto out;
-
- current->mm->total_vm += size;
- current->mm->locked_vm += size;
-
-out:
- if (child->thread.ds_area_msr)
- set_tsk_thread_flag(child, TIF_DS_AREA_MSR);
- else
- clear_tsk_thread_flag(child, TIF_DS_AREA_MSR);
-
- return ret;
+ /* The writing task will be the switched-to task on a context
+ * switch. It needs to write into the switched-from task's BTS
+ * buffer. */
+ return ds_unchecked_write_bts(child, bts_record, bts_cfg.sizeof_bts);
- }

void ptrace_bts_take_timestamp(struct task_struct *tsk,
@@ -839,7 +877,66 @@ void ptrace_bts_take_timestamp(struct task_struct *tsk,

ptrace_bts_write_record(tsk, &rec);
- }
-#endif /* X86_BTS */
+
```

[patch] x86, ptrace: PEBS support

```
+static const struct bts_configuration bts_cfg_netburst = {
+ .sizeof_bts = sizeof(long) * 3,
+ .sizeof_field = sizeof(long),
+ .debugctl_mask = (1<<2)|(1<<3)|(1<<5)
+};
+
+static const struct bts_configuration bts_cfg_pentium_m = {
+ .sizeof_bts = sizeof(long) * 3,
+ .sizeof_field = sizeof(long),
+ .debugctl_mask = (1<<6)|(1<<7)
+};
+
+static const struct bts_configuration bts_cfg_core2 = {
+ .sizeof_bts = 8 * 3,
+ .sizeof_field = 8,
+ .debugctl_mask = (1<<6)|(1<<7)|(1<<9)
+};
+
+static inline void bts_configure(const struct bts_configuration *cfg)
+{
+ bts_cfg = *cfg;
+}
+
+void __cpunit ptrace_bts_init_intel(struct cpuinfo_x86 *c)
+{
+ switch (c->x86) {
+ case 0x6:
+ switch (c->x86_model) {
+ case 0xD:
+ case 0xE: /* Pentium M */
+ bts_configure(&bts_cfg_pentium_m);
+ break;
+ case 0xF: /* Core2 */
+ case 0x1C: /* Atom */
+ bts_configure(&bts_cfg_core2);
+ break;
+ default:
+ /* sorry, don't know about them */
+ break;
+ }
+ break;
+ case 0xF:
+ switch (c->x86_model) {
+ case 0x0:
+ case 0x1:
+ case 0x2: /* Netburst */
+ bts_configure(&bts_cfg_netburst);
+ break;
+ default:
+ /* sorry, don't know about them */
+ break;
+ }
+ }
+}
```

[patch] x86, ptrace: PEBS support

```
+ }
+ break;
+ default:
+ /* sorry, don't know about them */
+ break;
+ }
+ }
+}
+#endif /* CONFIG_X86_PTRACE_BTS */

/*
 * Called by kernel/ptrace.c when detaching..
 @@ -852,15 +949,15 @@ void ptrace_disable(struct task_struct *child)
 #ifdef TIF_SYSCALL_EMU
 clear_tsk_thread_flag(child, TIF_SYSCALL_EMU);
 #endif
 - if (child->thread.ds_area_msr) {
 -#ifdef X86_BTS
 - ptrace_bts_realloc(child, 0, 0);
 -#endif
 - child->thread.debugctlmsr &= ~ds_debugctl_mask();
 - if (!child->thread.debugctlmsr)
 - clear_tsk_thread_flag(child, TIF_DEBUGCTLMSR);
 - clear_tsk_thread_flag(child, TIF_BTS_TRACE_TS);
 - }
 + #ifdef CONFIG_X86_PTRACE_BTS
 + (void)ds_release_bts(child);
 +
 + child->thread.debugctlmsr &= ~bts_cfg.debugctl_mask;
 + if (!child->thread.debugctlmsr)
 + clear_tsk_thread_flag(child, TIF_DEBUGCTLMSR);
 +
 + clear_tsk_thread_flag(child, TIF_BTS_TRACE_TS);
 + #endif /* CONFIG_X86_PTRACE_BTS */
 }

 #if defined CONFIG_X86_32 || defined CONFIG_IA32_EMULATION
 @@ -980,7 +1077,7 @@ long arch_ptrace(struct task_struct *child, long request, long addr, long data)
 /*
 * These bits need more cooking - not enabled yet:
 */
 -#ifdef X86_BTS
 + #ifdef CONFIG_X86_PTRACE_BTS
 case PTRACE_BTS_CONFIG:
 ret = ptrace_bts_config
 (child, data, (struct ptrace_bts_config __user *)addr);
 @@ -992,7 +1089,7 @@ long arch_ptrace(struct task_struct *child, long request, long addr, long data)
 break;

 case PTRACE_BTS_SIZE:
 - ret = ptrace_bts_get_size(child);
 + ret = ds_get_bts_index(child, /* pos = */ 0);
```

[patch] x86, ptrace: PEBS support

```
break;

case PTRACE_BTS_GET:
@@ -1001,14 +1098,14 @@ long arch_ptrace(struct task_struct *child, long request, long addr, long data)
break;

case PTRACE_BTS_CLEAR:
- ret = ptrace_bts_clear(child);
+ ret = ds_clear_bts(child);
break;

case PTRACE_BTS_DRAIN:
ret = ptrace_bts_drain
(child, data, (struct bts_struct __user *) addr);
break;
-#endif
+#endif /* CONFIG_X86_PTRACE_BTS */

default:
ret = ptrace_request(child, request, addr, data);
@@ -1254,14 +1351,14 @@ asmlinkage long sys32_ptrace(long request, u32 pid, u32 addr, u32 data)
case PTRACE_SETOPTIONS:
case PTRACE_SET_THREAD_AREA:
case PTRACE_GET_THREAD_AREA:
-#ifdef X86_BTS
+#ifdef CONFIG_X86_PTRACE_BTS
case PTRACE_BTS_CONFIG:
case PTRACE_BTS_STATUS:
case PTRACE_BTS_SIZE:
case PTRACE_BTS_GET:
case PTRACE_BTS_CLEAR:
case PTRACE_BTS_DRAIN:
-#endif
+#endif /* CONFIG_X86_PTRACE_BTS */
return sys_ptrace(request, pid, addr, data);

default:
diff --git a/arch/x86/kernel/setup_64.c b/arch/x86/kernel/setup_64.c
index 492c6a4..cd245ef 100644
--- a/arch/x86/kernel/setup_64.c
+++ b/arch/x86/kernel/setup_64.c
@@ -906,11 +906,12 @@ static void __cpuinit init_intel(struct cpuinfo_x86 *c)
set_cpu_cap(c, X86_FEATURE_BTS);
if (!(11 & (1<<12)))
set_cpu_cap(c, X86_FEATURE_PEBS);
+ ds_init_intel(c);
}

if (cpu_has_bts)
- ds_init_intel(c);
```

[patch] x86, ptrace: PEBS support

```
+ ptrace_bts_init_intel(c);

n = c->extended_cpuid_level;
if (n >= 0x80000008) {
diff --git a/include/asm-x86/ds.h b/include/asm-x86/ds.h
index 7881368..72c5a19 100644
--- a/include/asm-x86/ds.h
+++ b/include/asm-x86/ds.h
@@ -2,71 +2,237 @@
* Debug Store (DS) support
*
* This provides a low-level interface to the hardware's Debug Store
- * feature that is used for last branch recording (LBR) and
+ * feature that is used for branch trace store (BTS) and
* precise-event based sampling (PEBS).
*
- * Different architectures use a different DS layout/pointer size.
- * The below functions therefore work on a void*.
+ * It manages:
+ * - per-thread and per-cpu allocation of BTS and PEBS
+ * - buffer memory allocation (optional)
+ * - buffer overflow handling
+ * - buffer access
*
+ * It assumes:
+ * - get_task_struct on all parameter tasks
+ * - current is allowed to trace parameter tasks
*
- * Since there is no user for PEBS, yet, only LBR (or branch
- * trace store, BTS) is supported.
*
- *
- * Copyright (C) 2007 Intel Corporation.
- * Markus Metzger <markus.t.metzger@xxxxxxxx>, Dec 2007
+ * Copyright (C) 2007-2008 Intel Corporation.
+ * Markus Metzger <markus.t.metzger@xxxxxxxx>, 2007-2008
*/

#ifdef _ASM_X86_DS_H
#define _ASM_X86_DS_H

+#ifdef CONFIG_X86_DS
+
+ #include <linux/types.h>
+ #include <linux/init.h>

-struct cpuinfo_x86;

+struct task_struct;

-/* a branch trace record entry
```

[patch] x86, ptrace: PEBS support

[patch] x86, ptrace: PEBS support

```
+/ *
+ * Request BTS or PEBS
+ *
+ * Due to alignment constraints, the actual buffer may be slightly
+ * smaller than the requested or provided buffer.
+ *
- * In order to unify the interface between various processor versions,
- * we use the below data structure for all processors.
+ * Returns 0 on success; -Eerrno otherwise
+ *
+ * task: the task to request recording for;
+ * NULL for per-cpu recording on the current cpu
+ * base: the base pointer for the (non-pageable) buffer;
+ * NULL if buffer allocation requested
+ * size: the size of the requested or provided buffer
+ * ovfl: pointer to a function to be called on buffer overflow;
+ * NULL if cyclic buffer requested
+ */
-enum bts_qualifier {
- BTS_INVALID = 0,
- BTS_BRANCH,
- BTS_TASK_ARRIVES,
- BTS_TASK_DEPARTS
-};
+typedef void (*ds_ovfl_callback_t)(struct task_struct *);
+extern int ds_request_bts(struct task_struct *task, void *base, size_t size,
+ ds_ovfl_callback_t ovfl);
+extern int ds_request_pebs(struct task_struct *task, void *base, size_t size,
+ ds_ovfl_callback_t ovfl);
+
+/ *
+ * Release BTS or PEBS resources
+ *
+ * Frees buffers allocated on ds_request.
+ *
+ * Returns 0 on success; -Eerrno otherwise
+ *
+ * task: the task to release resources for;
+ * NULL to release resources for the current cpu
+ */
+extern int ds_release_bts(struct task_struct *task);
+extern int ds_release_pebs(struct task_struct *task);
+
+/ *
+ * Return the (array) index of the write pointer.
+ * (assuming an array of BTS/PEBS records)
+ *
+ * Returns -Eerrno on error
+ *
+ * task: the task to access;
+ * NULL to access the current cpu
```

[patch] x86, ptrace: PEBS support

```
+ * pos (out): if not NULL, will hold the result
+ */
+extern int ds_get_bts_index(struct task_struct *task, size_t *pos);
+extern int ds_get_pebs_index(struct task_struct *task, size_t *pos);
+
+/*
+ * Return the (array) index one record beyond the end of the array.
+ * (assuming an array of BTS/PEBS records)
+ *
+ * Returns -Eerrno on error
+ *
+ * task: the task to access;
+ * NULL to access the current cpu
+ * pos (out): if not NULL, will hold
```