AGBS
Studiengang Informatik
Fachbereich 3
Universität Bremen

Development and evaluation of a hard real-time scheduling modification for Linux 2.6

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker

vorgelegt von

Christof Efkemann

Bremen 2005

Eingereicht am 04. April 2005

1. Gutachter: Prof. Dr. Jan Peleska

2. Gutachter: Dr. Jan Bredereke

Erklärung

Die vorliegende Diplomarbeit wurde ohne fremde Hilfe angefertigt. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Datum Unterschrift

Contents

List of Figures

Chapter 1

Introduction

For many embedded systems their capability of adhering to hard real-time constraints is crucial to avoiding catastrophic consequences. Controllers used in the aerospace, railway, automotive as well as the medical sector or in industrial production serve as examples. In order to test the conformance of such systems to their specified requirements one must be able to check the adherence to hard real-time constraints.

For this task various specialised hardware and software are available, yet their costs are very high. A much more cost-effective approach is to use standard PC-hardware in combination with the Linux operating system. The problem that arises here is the fact that Linux per se does not have built-in support for hard real-time applications. But it can be achieved by modifying the Linux kernel.

Several modifications pursuing different approaches exist already. One of these was originally developed for Linux 2.4 by Klaas-Henning Zweck in his diploma thesis [Zwe02] and is based on a new scheduling class that can be activated with a command-line parameter or a *clone*-flag. A process using this scheduling class runs on its own exclusively reserved CPU and is not suspended by hardware interrupts.

The students' project *HaRTLinC* created an improved version of this modification, called the HLRT-patch [HL03]. Apart from several bugfixes the HLRT-patch allows simpler use of the scheduling class, optionally also for non-root processes, as well as a correct release of the reserved CPU and restoration of interrupt routing in case of unexpected termination of the process.

Verified Systems International successfully uses this version of the kernel modification with its Myrinet-based PC clusters for embedded systems testing.

Now that Linux 2.6 has become stable and also popular, it is desirable to also have such a modification for it. The main reasons are better support for newer hardware and a much improved scalability.

This diploma thesis will provide

- a description of the CPU reservation mechanism as a design pattern
- a description of a periodic scheduling mechanism as a design pattern
- the development of a modification that provides CPU reservation as well as periodic scheduling for Linux 2.6 and its O(1) scheduler based on the developed design patterns
- the utilisation of a separate system call table in order to avoid conflicts with existing and future system calls of the standard Linux kernel

Chapter 2

Theoretical Background

This chapter will describe the theoretical considerations which are necessary to understand the design and implementation of the hard real-time kernel modification.

2.1 Scheduling

Linux, like most modern operating systems, allows to have more than one process to be running at the same time (*multitasking*). Because this means that you can have more processes running than CPUs are available, the CPUs must be shared among the processes. By dividing the available CPU time into short quanta and assigning them to different processes the user gets the impression of processes running in parallel. This assignment is done by the *process scheduler*. As soon as a time quantum ends the currently executing process is interrupted (*preemption*) and the scheduler selects another (or possibly the same) process for execution during the next time quantum.

The scheduler's decision is based on the employed *scheduling algorithm*. Different algorithms exist that optimise CPU time assignment for various (contradictory) goals (cf. [TW97]):

- fairness: do not let processes starve, even if they have low priority
- CPU utilisation: keep the CPU busy
- interactivity: keep response time of interactive processes short
- turnaround: minimise processing time of batch jobs

The Linux process scheduler, whose implementation is explained in more detail in section 3.3, uses priority scheduling with round-robin scheduling in each priority class. Processes that sleep much are considered "interactive" and are therefore awarded a priority bonus to increase their response time. Additionally, each process has a timeslice, which, when used up, gets only refilled after all other processes have also used up their slices. This guarantees fairness because high-priority processes cannot block the CPU indefinitely[1].

For a hard real-time system such dynamic scheduling algorithms are problematic because they introduce unpredictable delays in process execution. The approach that is taken in this thesis (as before in [Zwe02] and [HL03]) is to disable scheduling on a dedicated CPU and let only the hard real-time process run on it, without interruptions.

Further information on different UNIX process schedulers can be found in [Vah95]. For more information on real-time scheduling refer to [Jos01].

2.2 Event-Triggered vs. Time-Triggered

When designing hard real-time systems, two different conceptual approaches can be taken. As described in [Kop97], an *event-triggered* (ET) system starts acting whenever a significant event happens. For example, when a button is pressed, an interrupt request is generated, which causes the interrupt service routine to be executed. The interrupt service routine reacts on the event. In contrast, a *time-triggered* (TT) system starts all of its activities at predetermined points in time. No interrupts are allowed to occur, with the exception of a periodic clock interrupt. For example, the state of the aforementioned button could be checked periodically every ten milliseconds. If a state change is detected, the system reacts accordingly.

While the interrupt mechanism used in ET systems offers flexibility and low response time, it bears a high risk: If there is no upper limit to the frequency of interrupts and the system is completely occupied with interrupt handler execution, the load becomes too high and the system cannot respond in time.

For example, imagine an ET system that needs one millisecond to process an interrupt. As soon as more than 1000 events that trigger an interrupt occur per second, the system is overloaded. In contrast to that, a TT system can run with a constant load (see figure 2.1).

The same considerations can be applied to real-time communication. An ET system would send one message for each event or state change it has to report/react on. This can cause bandwidth problems on the communication channel. Instead, a TT system would send all relevant state data, changed or not, at predetermined points in time, resulting in a constant bandwidth

[1]with the exception of `SCHED_FIFO` processes, which have infinite timeslices

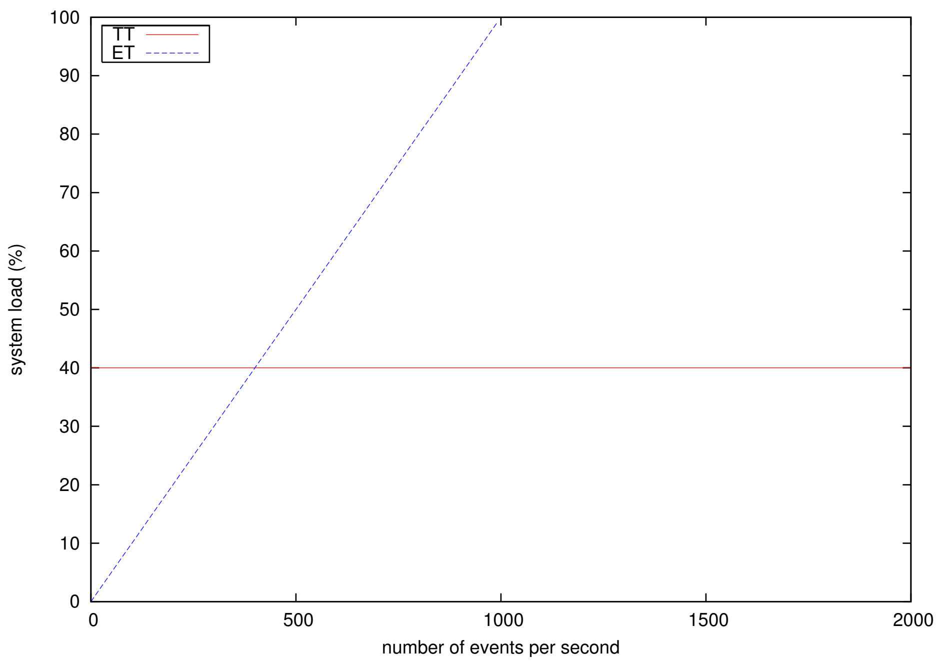

Figure 2.1: ET vs. TT: load comparison

utilisation[2].

As a consequence, the concept of the *time-triggered* system is a much better choice when designing hard real-time systems because it guarantees that no load problems will occur.

2.3 Design Patterns

In this section two design patterns (according to [GHJV95]) will give an abstract description of the features that the hard real-time kernel modification provides.

2.3.1 CPU Reservation

Pattern Name and Classification

CPU reservation

[2]with the additional benefit of the receiver being able to detect a sender/channel failure through missing status messages

Intent

Provide hard real-time scheduling for a process on a UNIX-like system.

Also Known As

None.

Motivation

Normally, a process is interrupted regularly by hardware interrupts and the scheduler. The scheduler can decide to run a different process, thereby delaying the first process. These delays and interruptions are not predictable for the process and introduce an indeterminism which prevents hard real-time execution.

The CPU reservation design pattern eliminates this indeterminism by allowing a process to run without interruptions.

Applicability

This pattern can be applied to any UNIX-like kernel with a process scheduler that supports different scheduling classes, SMP support, interrupt routing, and CPU affinities for processes.

It only works well on SMP machines, where one CPU always remains unreserved to allow normal applications to be executed in parallel with hard real-time applications.

Structure

See figure 2.2.

Participants

Process Has the attributes *scheduling class* and *CPU affinity*. Apart from the normal values, *scheduling class* can take the newly introduced value `SCHED_HLRT`. The *CPU affinity* is normally set to all CPUs.

hlrt_cpus_allowed This new global variable contains all CPUs that are not currently reserved.

Scheduler The scheduler decides which process to run on a CPU.

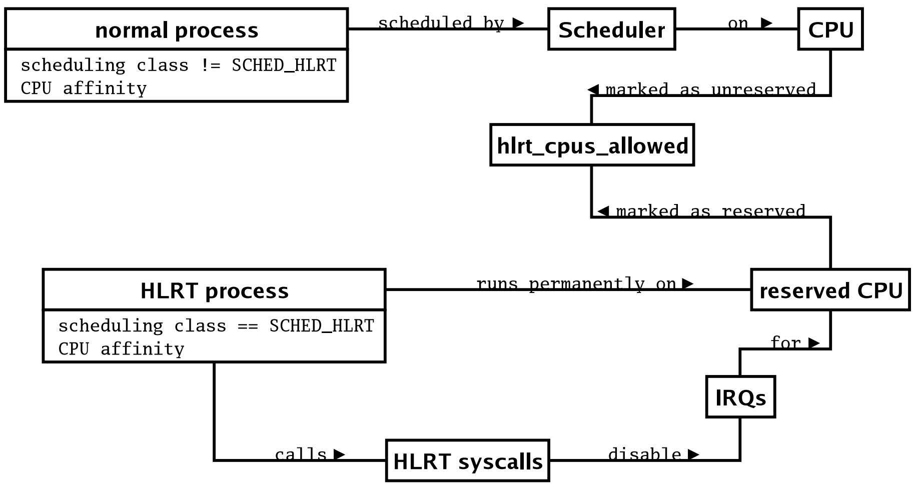

Figure 2.2: Structure of CPU reservation

System calls At least three additional system calls are provided: `hlrt_-request_irq` and `hlrt_release_irq` allow a hard real-time process to route hardware interrupts to or away from the reserved CPU. `hlrt_-set_local_apic_timer` is used to disable the timer interrupt of the CPU, thereby disabling the scheduler.

Collaborations

Whenever the scheduler evaluates a normal process' CPU affinity to determine whether a process can run on a certain CPU, it must also take `hlrt_cpus_allowed` into account: a process can use a CPU if and only if the process' CPU affinity *and* `hlrt_cpus_allowed` contain this CPU.

If a process switches to the `SCHED_HLRT` policy, the scheduler must determine a currently unreserved CPU and reserve it for the process by removing it from `hlrt_cpus_allowed`. However, it must always leave one CPU unreserved for all normal processes to run on. The process' CPU affinity is set to contain only its reserved CPU; the process must not be allowed to change it. Before running the hard real-time process on its reserved CPU it must ensure that all normal processes have left this CPU.

Consequences

On a system with n CPUs this pattern allows $n-1$ processes to run with hard real-time scheduling. A reserved CPU is not available for other processes.

Implementation

An implementation must take special care of any CPU-bound processes, especially kernel threads. If the CPU of a bound process becomes reserved, the process cannot be executed. It should then be allowed to run on another (unreserved) CPU. This, however, can be dangerous if the process is a kernel thread that makes use of per-CPU variables. Special care must be taken to avoid concurrency and race conditions. Variables that may be accessed from different CPUs must be protected by locks.

It must be assured that the child does not inherit the scheduling class and the CPU affinity if a hard real-time process uses the `fork` system call.

Sample Code

See appendix A.

Known Uses

- the HLRT patch for Linux 2.6 (this thesis)

Note: [Zwe02] and [HL03] use a similar pattern without the global `hlrt_cpus_allowed`.

Related Patterns

None.

2.3.2 Time-Triggered Architecture

Pattern Name and Classification

Time-triggered architecture

Intent

Provide periodic execution with fixed intervals and detection of missed deadlines.

Also Known As

TTA

Motivation

As described in section 2.2, the time-triggered approach is the recommended design for a hard real-time system. An implementation of such system requires the ability to execute at predetermined points in time. In order to achieve this, the time-triggered architecture design pattern allows a process to be executed periodically with a fixed but user-defined interval.

Applicability

This pattern can be applied to any UNIX-like kernel that meets the requirements for the *CPU reservation* pattern on a platform that provides a programmable timer interrupt for each CPU.

Structure

N/A

Participants

Process A hard real-time process with a reserved CPU.

Timer interrupt The programmable timer interrupt of the reserved CPU.

System calls The functionality of the system call `hlrt_set_local_apic_timer` (introduced by the *CPU reservation* design pattern) must be extended to allow a period for the timer interrupt to be specified. The new system call `hlrt_wait_for_apic_timer` must be used by the process to signal completion of its current period and to wait for the next period.

Collaborations

If a hard real-time process chooses to activate periodic execution it must use the `hlrt_set_local_apic_timer` system call to define a period length. The system call programs the interrupt timer of the reserved CPU.

The `hlrt_wait_for_apic_timer` system call, called by the process to wait for the next period, puts the process to sleep. As soon as the timer interrupt is triggered, its interrupt service routine checks if the process is sleeping. If so, the process is woken and can execute its next period. If not, the process has missed its deadline because it is still executing although the next period has started. In that case a `SIGALRM` signal is sent to the process to inform it of its missed deadline.

Consequences

A process using this feature must not use any system calls that might sleep because it would certainly miss its deadline. Instead, only polling calls should be used.

Implementation

An implementation must ensure correct restoration of the programmable timer interrupt when the process exits. Otherwise the CPU will become unusable until reboot.

Only processes that have a reserved CPU can be allowed to use this feature. Otherwise the timer interrupt would be in use by the process scheduler and the process might be interrupted by other processes, missing its deadline.

Sample Code

See appendix A.

Known Uses

- the HLRT patch for Linux 2.6 (this thesis)

Note: [Zwe02] and [HL03] do not provide this feature.

Related Patterns

- CPU reservation

Chapter 3

The Linux Kernel

This chapter describes some implementation details of the Linux kernel which are related to the hard real-time kernel modification. First we will take a quick general look on Linux and then go further into the relevant details.

3.1 What is Linux?

From the Linux `README` file:

> "Linux is a Unix clone written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX compliance.
>
> It has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management and TCP/IP networking.
>
> It is distributed under the GNU General Public License - see the accompanying COPYING file for more details."

3.2 Design

Linux is a monolithic kernel. The different subsystems do not communicate via messages like micro-kernel systems do, but instead via function calls. That means each subsystem can make function calls into other subsystems and all kernel code uses the same address space. Since micro-kernels were considered superior due to their high modularity, this design was heavily criticised. But the advantage of a monolithic design is a higher performance when compared with micro-kernels because the latter normally have to cope with a high message-passing overhead.

While originally being written for the Intel i386 PC-compatible platform, Linux has been ported to more than 20 different platforms, for example SPARC, PowerPC, Motorola 68000, and ARM-based systems. The Linux source code is therefore divided into architecture-dependent and independent code.

The Linux kernel consists of the following main subsystems:

- memory management
- process scheduler
- inter-process communication
- virtual filesystem
- networking and protocol stacks

Further information regarding each of these subsystems can be found in many Linux-related books, for example [BC01].

3.3 O(1) Scheduler

Linux 2.6 employs the O(1) process scheduler written by Ingo Molnar. Its goal is to improve some shortcomings of the old Linux scheduler:

- O(1) scheduling algorithm: no goodness and recalculation loops, whose execution time depended on the number of runnable processes
- SMP scalability: use per-CPU runqueues to avoid a “big” runqueue lock
- better SMP affinity: avoid random bouncing of processes caused by the fact that all processes had to use up their timeslices before a recalculation could be done

The O(1) scheduler can be found in the file `kernel/sched.c`. It defines a runqueue of type `struct runqueue` for each CPU. Since the scheduler running on CPU n only needs the runqueue of CPU n for its calculations, it is completely independent from the scheduler running on CPU m. Lock contention is avoided and the scheduler scales much better on SMP systems.

Each runqueue contains two priority arrays, named `active` and `expired`, of type `struct prio_array`. Their element `queue` is an array of lists of

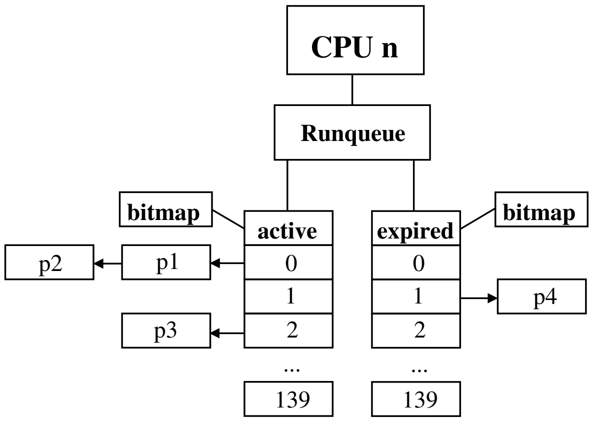

Figure 3.1: O(1) scheduler runqueue

processes. Each list is associated with a process priority. If a process of priority p becomes runnable, it is added to the list `queue[`p`]`. The priority array also has an element `bitmap`. If a list in `queue` is not empty, the corresponding bit in `bitmap` is set. This makes it possible to find the next runnable process with the highest priority by simply scanning the bit field of the `active` array for the next set bit (which can be done without a loop using an appropriate assembler instruction[1]) and taking the first process from the associated list.

Each process has a timeslice. Its specific length depends on its interactivity, i. e. how much the process sleeps. The more a process sleeps, the more it is considered "interactive" and the longer is the timeslice it gets. When a process has used up its timeslice, it is put from the `active` array into the `expired` array. As soon as the `active` array is empty, `active` and `expired` are switched (which can be done quickly because they are just pointers), and now all processes have new timeslices.

Since this algorithm does not contain any loops iterating over all runnable processes, its complexity is O(1).

On an SMP system, each CPU has its own runqueue. Occasionally, the load (i. e. the number of runnable processes) between CPUs can differ significantly. To improve the overall throughput, a load balancing algorithm is used. At regular intervals the scheduler checks if its runqueue is much smaller (at least 25%) than the largest runqueue. If so, it pulls a number of processes (half the imbalance) from the largest runqueue. Here it first tries to take processes from the `expired` array, as these are most likely not "cache-hot". Note that locking is required here because the runqueue of

[1]for example `bsf` on x86 CPUs

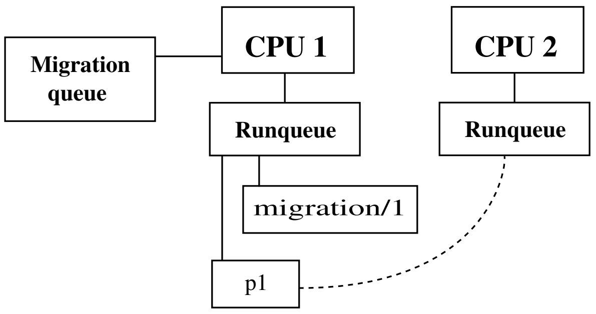

Figure 3.2: Process migration

another CPU is modified.

If a process wishes (or is forced) to run on another than its current CPU via the `sched_set_affinity()` system call, some precautions have to be taken. The process cannot simply be added to the runqueue of the destination CPU because it might still be executing on its previous CPU. Here the help of a *migration thread* is necessary: The CPU executing the `sched_set_affinity()` system call sets the new `cpus_allowed` mask of the process and places a migration request (of type `migration_req_t`) into the migration queue of the CPU the process currently executes on. Then it wakes up the migration thread of that CPU. This thread immediately forces the process off that CPU because it has the highest possible priority. Then it safely adds the process to the runqueue of one of the allowed CPUs and signals the completion of the migration to the process that executed the system call.

For more information on the O(1) scheduler and load balancing in Linux 2.6 refer to [Lov03].

Chapter 4

Implementation

This chapter describes the implementation of the hard real-time kernel modification. It is split into a kernel patch and a user-space library.

4.1 Kernel Patch

The hard real-time kernel patch that was developed in the course of this thesis consists of modifications in several parts of the kernel source code. They shall be introduced and explained here. Two different types of modifications can be distinguished:

- modifications introducing new features
- necessary adjustments to existing code which is affected by those modifications

In the following, portions of the kernel patch will be reproduced in *unified diff* format because this allows to see precisely where changes have been made. The fact that they only contain "+"-lines (i. e. lines that have been added to the original code) make them easy to read because you will not see any code that is not actually there any more. Files marked with † are new and not included in a standard Linux kernel.

4.1.1 New Features

4.1.1.1 Kernel configuration

Affected files:

- `arch/i386/defconfig`

- arch/i386/Kconfig

Two kernel configuration options have been added to the kernel configuration system: “HLRT CPU reservation” (CONFIG_HLRT) and “Allow CPU reservation for non-root users” (CONFIG_HLRT_NOROOT). If CONFIG_HLRT is disabled, no code of this patch is actually compiled and the kernel behaves like a standard kernel. Otherwise, all features of this patch are enabled. Note that it can only be activated if SMP support is enabled and kernel IRQ balancing and kernel preemption are disabled (the latter two options have not yet been tested in combination with this patch).

```
+config HLRT
+       bool "HLRT CPU reservation"
+       depends on SMP && !IRQBALANCE && !PREEMPT
+       default n
+       help
+         The HaRTLinC Real-Time extension allows a process to reserve
+         a CPU exclusively for its own use by switching to the HLRT
+         scheduling class.  It is guaranteed to not be interrupted by
+         other processes or interrupt handlers.
```

The option CONFIG_HLRT_NOROOT controls whether processes of normal users can use the hard real-time scheduling policy. Although, in general, this is not recommended, it can make work much easier in a closed laboratory environment.

```
+config HLRT_NOROOT
+       bool "Allow CPU reservation for non-root users"
+       depends on HLRT
+       default n
+       help
+         This option allows all users to use the HLRT scheduling class
+         (default: only root).  Warning: Enable this only if you trust
+         all users of your system.
```

4.1.1.2 System call table and vector

Affected files:

- arch/i386/kernel/hlrt_entry.S[†]
- arch/i386/kernel/i8259.c
- arch/i386/kernel/io_apic.c
- arch/i386/kernel/traps.c
- drivers/pci/msi.c

- `include/asm-i386/hlrt_unistd.h`†
- `include/asm-i386/mach-default/irq_vectors.h`

Because this hard real-time modification is not part of the standard Linux source code, it uses its own system call vector and table in order to avoid having to change the system call numbers introduced by this patch whenever the standard Linux kernel gets a new system call. This also ensures binary compatibility across different kernel versions.

To create a new system call vector that can be used from user space via the `int` instruction, a *trap gate* [Int03] must be added to the IDT (interrupt descriptor table) in `arch/i386/kernel/traps.c`:

```
+#ifdef CONFIG_HLRT
+       set_system_gate(HLRT_SYSCALL_VECTOR, &hlrt_system_call);
+#endif
```

The vector number is defined in `include/asm-i386/mach-default/-irq_vectors.h`:

```
 #define SYSCALL_VECTOR          0x80

+#ifdef CONFIG_HLRT
+#define HLRT_SYSCALL_VECTOR     0x83
+#endif
```

The following files contain code that assigns interrupt vectors to hardware interrupt lines: `arch/i386/kernel/i8259.c`, `arch/i386/kernel/-io_apic.c` and `drivers/pci/msi.c`. In those files it must be assured that the `HLRT_SYSCALL_VECTOR` is not overwritten.

Finally, we need our own handler routine for the trap gate: `hlrt_system_call()`. It is implemented in `arch/i386/kernel/hlrt_entry.S`. This file is basically a copy of `arch/i386/kernel/entry.S` with several symbols renamed to their `hlrt` counterpart. It also contains the actual system call table with the currently assigned HLRT system calls:

```
+.data
+ENTRY(hlrt_sys_call_table)
+       .long sys_hlrt_request_irq          /* 0 */
+       .long sys_hlrt_release_irq
+       .long sys_hlrt_set_local_apic_timer
+       .long sys_hlrt_wait_for_apic_timer
+       .long sys_ni_syscall
+       .long sys_ni_syscall                /* 5 */
+       .long sys_hlrt_get_version_info     /* 6: don't move! */
```

The HLRT system call mechanism is identical to that of the normal Linux system calls[1]: The user space program loads the system call number into the `eax` register and performs an `int 0x83` instruction, thereby using the trap gate to enter kernel mode. The `eax` register is used as an index into the system call table to find the address of the system call function which is then executed.

4.1.1.3 Set/unset HLRT policy – allocate/release CPUs

Affected files:

- `arch/i386/kernel/hlrt.c`†
- `include/asm-i386/hlrt.h`†
- `include/linux/sched.h`
- `kernel/exit.c`
- `kernel/sched.c`

The system call `sched_setscheduler()` has been extended to support the scheduling class `SCHED_HLRT`. It is implemented in `kernel/sched.c` in the function `setscheduler()`. Its parameter checks have been extended to accept `SCHED_HLRT`:

```
                retval = -EINVAL;
                if (policy != SCHED_FIFO && policy != SCHED_RR &&
+#ifdef CONFIG_HLRT
+                               policy != SCHED_HLRT &&
+#endif
                                policy != SCHED_NORMAL)
                        goto out_unlock;
```

The permission checks have been changed to allow the use of `SCHED_HLRT` only if the process has `CAP_SYS_NICE` capability (i. e. runs in the context of the superuser) or `CONFIG_HLRT_NOROOT` was enabled in the kernel configuration (see above).

```
        retval = -EPERM;
+#if defined(CONFIG_HLRT) && !defined(CONFIG_HLRT_NOROOT)
+       if ((policy == SCHED_HLRT) && !capable(CAP_SYS_NICE))
+               goto out_unlock;
+#endif
        if ((policy == SCHED_FIFO || policy == SCHED_RR) &&
            !capable(CAP_SYS_NICE))
                goto out_unlock;
```

[1] with one exception: the use of `sysenter`/`sysexit` is not supported by HLRT system calls

Apart from the checks the following code block has been added:

```
+#ifdef CONFIG_HLRT
+       if (policy == SCHED_HLRT) {
+               task_rq_unlock(rq, &flags);
+               retval = hlrt_set_hlrt_policy(p);
+               goto out_unlock_tasklist;
+       } else if (p->policy == SCHED_HLRT) {
+               task_rq_unlock(rq, &flags);
+               retval = hlrt_unset_hlrt_policy(p);
+               if (retval)
+                       goto out_unlock_tasklist;
+               rq = task_rq_lock(p, &flags);
+       }
+#endif /* CONFIG_HLRT */
```

If the new policy is `SCHED_HLRT`, the (previously locked) runqueue which contains the process is unlocked and `hlrt_set_hlrt_policy()` is called. Otherwise, if the previous scheduling policy is `SCHED_HLRT`, the runqueue is unlocked and `hlrt_unset_hlrt_policy()` is called. These two functions are implemented in `arch/i386/kernel/hlrt.c`.

`hlrt_set_hlrt_policy()` performs a CPU reservation for a process `p` for which it was called:

```
+/*
+ * must be called with tasklist_lock held, runqueue must be unlocked
+ */
+int hlrt_set_hlrt_policy(struct task_struct *p)
+{
+       int cpu;
```

It first checks if the process is already using the `SCHED_HLRT` scheduling class, in which case nothing more is to be done (a process cannot have two reserved CPUs).

```
+       if (p->policy == SCHED_HLRT)
+               return 0;
```

Now it calls `hlrt_allocate_next_cpu()` which returns the number of the reserved CPU or `NR_CPUS` if no more CPUs are available (see below).

```
+       cpu = hlrt_allocate_next_cpu();
+       if (cpu < NR_CPUS) {
+               /*
+                * kick all other processes from the reserved
+                * CPU's runqueue
+                */
+               struct task_struct *tsk;
```

Even though the CPU has been reserved now, there might still be processes executing on it (but processes not currently executing on this CPU cannot change back to it). They must be moved to other CPUs. In order to do this, the whole task list is scanned for tasks (different from `p`) executing on this CPU. For each of those tasks `set_cpus_allowed()` is called with their normal `cpus_allowed` mask. Because this function takes into account the reserved CPUs (see section 4.1.2.3), it is guaranteed that the process has left the CPU upon function return. Unfortunately, this function requires the task list and runqueues to be unlocked (because it might sleep if the migration task is needed (see section 3.3)) and therefore we must restart scanning the task list. The upper bound of repetitions is the number of tasks in the task list when starting the scan because no new processes can start execution on the reserved CPU.

```
+repeat_tasklist_scan:
+               for_each_process(tsk) {
+                       if (task_cpu(tsk) == cpu && tsk != p) {
+                               get_task_struct(tsk);
+                               read_unlock_irq(&tasklist_lock);
+                               set_cpus_allowed(tsk, tsk->cpus_allowed);
+                               read_lock_irq(&tasklist_lock);
+                               put_task_struct(tsk);
+                               /*
+                                * unfortunately, we had to drop the
+                                * task_list lock, so we have to start
+                                * over again
+                                */
+                               goto repeat_tasklist_scan;
+                       }
+               }
```

Finally, the scheduling policy of process `p` is changed and `set_cpus_allowed()` is used to move it to its reserved CPU.

```
+               p->policy = SCHED_HLRT;
+               p->rt_priority = 0;
+               get_task_struct(p);
+               read_unlock_irq(&tasklist_lock);
+               set_cpus_allowed(p, cpumask_of_cpu(cpu));
+               read_lock_irq(&tasklist_lock);
+               put_task_struct(p);
+               printk(KERN_NOTICE "HLRT: reserved CPU %i for process %i\n",
+                      cpu, p->pid);
+               return 0;
+       }
+       return -EBUSY;
+}
```

The function `hlrt_unset_hlrt_policy()` is the counterpart to the previous function. It releases a reserved CPU for a process `p`:

```
+/*
+ * must be called with tasklist_lock held, runqueue must be unlocked
+ */
+int hlrt_unset_hlrt_policy(struct task_struct *p)
+{
+       int cpu;
+       int irq;
+       struct task_struct *tsk;
```

It first checks if the calling process wants to change its own scheduling policy because releasing the CPU reservation of a different process is not permitted.

```
+       if (p != current)
+               return -EPERM;
+
+       /*
+        * disable preemtion to ensure that we stay on the reserved CPU
+        * until we're done releasing the IRQs
+        */
+       preempt_disable();
+
```

Now the reserved CPU is released by calling `hlrt_release_cpu()` (see below) and the scheduling policy of the process is (temporarily[2]) changed to `SCHED_NORMAL`. The function `set_cpus_allowed()` is used to allow the process to execute on all CPUs.

```
+       /* revoke CPU reservation */
+       cpu = first_cpu(p->cpus_allowed);
+       hlrt_release_cpu(cpu);
+
+       p->policy = SCHED_NORMAL;
+       p->rt_priority = 0;
+
+       if (!(p->flags & PF_EXITING)) {
+               /* drop locks for set_cpus_allowed() call */
+               get_task_struct(p);
+               read_unlock_irq(&tasklist_lock);
+               set_cpus_allowed(p, CPU_MASK_ALL);
+               read_lock_irq(&tasklist_lock);
+               put_task_struct(p);
+       }
```

Some processes (mainly kernel threads) are bound to one specific CPU. They were moved off the CPU when it was reserved and must now be moved back. This is done by scanning the whole task list for processes whose

[2]might be changed to a different policy when `setscheduler()` continues

`cpus_allowed` mask contains only the CPU we just released and who are executing on a different CPU. For each of those tasks `set_cpus_allowed()` is called again with their normal `cpus_allowed` mask. And again we have the locking problem because `set_cpus_allowed()` might sleep. Even worse, when we start over scanning the task list, the CPU might have already been reserved again, in which case we would race against the other tasklist-scanning loop, causing a livelock. To avoid this, we abort the loop if the CPU gets reserved again.

```
+repeat_tasklist_scan:
+       /*
+        * Stop moving back processes if the CPU has already been reserved
+        * again.  This would result in a livelock in the for_each_process
+        * loop because we have to restart it whenever we find a process
+        * that must be moved.
+        */
+       if (cpu_isset(cpu, hlrt_cpus_allowed)) {
+               for_each_process(tsk) {
+                       if (cpus_equal(tsk->cpus_allowed,
+                                      cpumask_of_cpu(cpu)) &&
+                           task_cpu(tsk) != cpu) {
+                               get_task_struct(tsk);
+                               read_unlock_irq(&tasklist_lock);
+                               set_cpus_allowed(tsk, tsk->cpus_allowed);
+                               read_lock_irq(&tasklist_lock);
+                               put_task_struct(tsk);
+                               /*
+                                * unfortunately, we had to drop the
+                                * task_list lock, so we have to start
+                                * over again
+                                */
+                               goto repeat_tasklist_scan;
+                       }
+               }
+       }
```

Finally, changes of the interrupt routing and the local APIC timer that the process might have made are undone.

```
+       /*
+        * Note: the following two parts work because hlrt_unset_hlrt_-
+        * policy() always runs on the previously reserved CPU
+        */
+
+       /* re-enable Local APIC Timer */
+       sys_hlrt_set_local_apic_timer(HLRT_LAPIC_TIMER_MODE_NORMAL, 0, 0);
+
+       /* allow all non-reserved IRQs to be routed to this CPU */
+       for (irq = 0; irq < NR_IRQS; irq++) {
+               sys_hlrt_release_irq(irq);
+       }
```

```
+
+       /* now we can re-enable preemption safely */
+       preempt_enable();
+
+       printk(KERN_NOTICE "HLRT: released CPU %i from process %i\n",
+              cpu, p->pid);
+
+       return 0;
+}
```

Now we will take a quick look at two functions that were already mentioned: `hlrt_allocate_next_cpu()` and `hlrt_release_cpu()`. Their task is the administration of the global `hlrt_cpus_allowed` bitfield. Every CPU whose bit is set in this field is "allowed" for use by normal processes. Those CPUs whose bits are cleared are reserved by an HLRT process. Wherever the `cpus_allowed` mask of a normal process is used to determine the CPUs it is allowed to use, a is bitwise AND with `hlrt_cpus_allowed` is applied (see section 4.1.2).

```
+cpumask_t hlrt_cpus_allowed = CPU_MASK_ALL;
```

The function `hlrt_allocate_next_cpu()` finds a free CPU and reserves it. It scans the variable `cpu_possible_map` which was initialised at boot time and contains those CPUs that actually exist. The first CPU is always skipped to allow normal processes to continue execution. For the next CPU that is contained in `cpu_possible_map` as well as in `hlrt_cpus_allowed` (i. e. physically present and not reserved) its bit in `hlrt_cpus_allowed` is cleared, thereby reserving this CPU. The whole procedure is protected by a spin lock to avoid having two different processes reserve the same CPU concurrently.

```
+static int hlrt_allocate_next_cpu(void)
+{
+       int cpu = first_cpu(cpu_possible_map);
+
+       spin_lock(&hlrt_cpus_lock);
+
+       /* Note: the first CPU will never be reserved */
+       while (cpu < NR_CPUS) {
+               cpu = next_cpu(cpu, cpu_possible_map);
+               if (cpu < NR_CPUS && cpu_isset(cpu, hlrt_cpus_allowed)) {
+                       cpu_clear(cpu, hlrt_cpus_allowed);
+                       break;
+               }
+       }
+
+       spin_unlock(&hlrt_cpus_lock);
+
+       return cpu;
+}
```

In contrast, the function `hlrt_release_cpu()` simply releases a CPU by setting its bit in the `hlrt_cpus_allowed` field:

```
+static int hlrt_release_cpu(int cpu)
+{
+       spin_lock(&hlrt_cpus_lock);
+       cpu_set(cpu, hlrt_cpus_allowed);
+       spin_unlock(&hlrt_cpus_lock);
+
+       return 0;
+}
```

4.1.1.4 Controlling IRQ routing

Affected files:

- `arch/i386/kernel/hlrt.c`†

The patch allows an HLRT process to control which IRQs it receives on its reserved CPU. This feature is implemented via two system calls: `hlrt_request_irq()` and `hlrt_release_irq()`. Their implementation is similar to that of the old HLRT patch [HL03] but has been rewritten to utilise `cpumask_t` bitfields internally.

`sys_hlrt_request_irq()` changes the affinity of an IRQ to the current CPU if its current affinity does not contain any reserved CPUs:

```
+int sys_hlrt_request_irq(int irq)
+{
+       int retval;
+       cpumask_t affinity, reserved_cpus, result;
+
+       spin_lock(&hlrt_irqs_lock);
+       affinity = hlrt_get_irq_affinity(irq);
+       reserved_cpus = hlrt_cpus_allowed;
+       cpus_complement(reserved_cpus);
+       cpus_and(result, affinity, reserved_cpus);
+       if (cpus_empty(result))
+               retval = hlrt_set_irq_affinity(irq, current->cpus_allowed);
+       else
+               retval = -EBUSY;
+       spin_unlock(&hlrt_irqs_lock);
+
+       return retval;
+}
```

`sys_hlrt_release_irq()` changes the affinity of an IRQ to all not currently reserved CPUs if its current affinity does not contain any reserved CPUs except for the current CPU. Note that it is not necessary to have called `sys_hlrt_request_irq()` before. Instead, simply calling `sys_hlrt_release_irq()` on a newly reserved CPU causes the IRQ to not be delivered to this CPU any more.

```
+int sys_hlrt_release_irq(int irq)
+{
+       int retval = 0;
+       cpumask_t affinity, reserved_cpus, result;
+
+       spin_lock(&hlrt_irqs_lock);
+       affinity = hlrt_get_irq_affinity(irq);
+       reserved_cpus = hlrt_cpus_allowed;
+       cpus_complement(reserved_cpus);
+       cpus_and(result, affinity, reserved_cpus);
+       cpu_clear(smp_processor_id(), result);
+       if (cpus_empty(result)) {
+               affinity = hlrt_cpus_allowed;
+               retval = hlrt_set_irq_affinity(irq, affinity);
+       }
+       spin_unlock(&hlrt_irqs_lock);
+
+       return retval;
+}
```

The functions `hlrt_get_irq_affinity()` and `hlrt_set_irq_affinity()` have not changed. See [HL03] for further information.

4.1.1.5 TTA support via local APIC timer

Affected files:

- `arch/i386/kernel/apic.c`
- `arch/i386/kernel/hlrt.c`†
- `include/asm-i386/mach-default/entry_arch.h`
- `include/asm-i386/mach-default/irq_vectors.h`

In contrast to the old HLRT patch, this patch supports the implementation of a time-triggered architecture (TTA, see section 2.2). To that purpose it offers two additional system calls. The first one, `hlrt_set_local_apic_timer()`, allows to change the mode and period of the local APIC timer [Int03] of the CPU it is called on.

```
+int sys_hlrt_set_local_apic_timer(int mode, unsigned int period,
+                                  unsigned int n_cpus)
+{
+       unsigned int clocks, lvtt_value, tmp_value, ver;
+       u64 div;
```

The parameter `mode` can take three different values. If it is set to `HLRT_LAPIC_TIMER_MODE_OFF`, the local APIC timer is completely disabled and the user process runs without further interruptions by the local APIC timer. The other two parameters are ignored. Because kernel timers are implemented using the local APIC timer, all pending timers of this CPU must be moved to another CPU using `hlrt_move_timers()`. Otherwise, they would never expire (see section 4.1.2.5 for further information).

```
+       switch (mode) {
+       case HLRT_LAPIC_TIMER_MODE_OFF:
+               __get_cpu_var(hlrt_local_apic_mode) = mode;
+               disable_APIC_timer();
+               hlrt_move_timers(smp_processor_id(),
+                                first_cpu(cpu_possible_map));
+               return 0;
```

If `mode` has the value `HLRT_LAPIC_TIMER_MODE_NORMAL`, the local APIC timer is initialised to its normal behaviour. Again, the other two parameters are ignored. This means that the process is interrupted by the local APIC timer and the scheduler once per millisecond.

```
+       case HLRT_LAPIC_TIMER_MODE_NORMAL:
+               setup_secondary_APIC_clock();
+               __get_cpu_var(hlrt_local_apic_mode) = mode;
+               return 0;
```

The last possible value for `mode` is `HLRT_LAPIC_TIMER_MODE_PERIODIC`. This allows to set the periodic time of the local APIC timer to a custom value (unit: ns) via the parameter `period`. As the time base for the timer is derived from the processor's bus clock, the actual number of cycles is determined from the calibration that was performed at boot time (variable `calibration_result`).

```
+       case HLRT_LAPIC_TIMER_MODE_PERIODIC:
+               div = (u64)period * (u64)calibration_result;
+               do_div(div, (1000000000/HZ));
+               clocks = (u32)div;
+
+               /* sanity check */
+               if (period < 1000 || clocks < 200)
+                       return -EINVAL;
```

Again, the timers are moved to another CPU. Note that the hard real-time process *can* use timers, but this is neither necessary nor desirable, as it would then depend on another CPU.

```
+               __get_cpu_var(hlrt_local_apic_mode) = mode;
+               hlrt_move_timers(smp_processor_id(),
+                                first_cpu(cpu_possible_map));
```

Now the wait queue that is used to wake up the process when the next period starts is initialised and the APIC registers are programmed.

```
+               /* initialize the wait queue */
+               init_waitqueue_head(&__get_cpu_var(
+                       hlrt_apic_timer_wait_queue));
+
+               /* write to the APIC registers */
+               ver = GET_APIC_VERSION(apic_read(APIC_LVR));
+               lvtt_value = APIC_LVT_TIMER_PERIODIC |
+                            HLRT_LOCAL_TIMER_VECTOR;
+               if (!APIC_INTEGRATED(ver))
+                       lvtt_value |=
+                               SET_APIC_TIMER_BASE(APIC_TIMER_BASE_DIV);
+               apic_write_around(APIC_LVTT, lvtt_value);
+
+               tmp_value = apic_read(APIC_TDCR);
+               apic_write_around(APIC_TDCR,
+                                 (tmp_value & ~(APIC_TDR_DIV_1 |
+                                                APIC_TDR_DIV_TMBASE))
+                                 | APIC_TDR_DIV_1);
```

With the third parameter (`n_cpus`) it is possible to synchronise the periods of the local APIC timers of several CPUs. All processes whose CPUs shall participate in the synchronisation must make the same `hlrt_set_local_apic_timer()` call, with `n_cpus` being the total number of participating CPUs. The first CPU that makes this call initialises a global counter. Each additional CPU decreases this counter by one. As soon as the counter reaches zero, all CPUs leave the loop in which they spun and start their local APIC timer. (Note that due to lack of suitable hardware[3] the following code block could not be tested yet.)

```
+               if (n_cpus > 1) {
+                       /* achieve phase synchronisation across
+                          n_cpus timers */
+                       spin_lock(&sync_cpu_lock);
+                       if (sync_cpu_counter == 0) {
+                               sync_cpu_counter = n_cpus;
+                       }
```

[3] 4-way SMP system

```
+                       spin_unlock(&sync_cpu_lock);
+
+                       __asm__ __volatile__(
+                               "lock; decl %0\n\t"
+                               "jz 2f\n"
+                               "1:\n\t"
+                               "cmpl $0,%0\n\t"
+                               "jnz 1b\n"
+                               "2:\n"
+                               :"=m" (sync_cpu_counter)
+                               :"m" (sync_cpu_counter));
+               }
+
+               apic_write_around(APIC_TMICT, clocks);
+
+               return 0;
+
+       default:
+               return -EINVAL;
+       }
+}
```

The second system call is `hlrt_wait_for_apic_timer()`, also implemented in `arch/i386/kernel/hlrt.c`. It checks if the local APIC timer is in `HLRT_LAPIC_TIMER_MODE_PERIODIC` mode and then puts the process to sleep in the wait queue which we initialised previously. As soon as the process is woken up, it returns back to user space.

```
+int sys_hlrt_wait_for_apic_timer(void)
+{
+       if (__get_cpu_var(hlrt_local_apic_mode) !=
+           HLRT_LAPIC_TIMER_MODE_PERIODIC)
+               return -EINVAL;
+
+       interruptible_sleep_on(&__get_cpu_var(hlrt_apic_timer_wait_queue));
+       return 0;
+}
```

When the local APIC timer registers were programmed (see above), the vector of the timer entry of the APIC's local vector table was changed to `HLRT_LOCAL_TIMER_VECTOR`. It is defined in `include/asm-i386/mach-default/irq_vectors.h`:

```
+#ifdef CONFIG_HLRT
+#define HLRT_LOCAL_TIMER_VECTOR 0xf1
+#endif
```

Upon initialisation of the local APIC at boot time an *interrupt gate* [Int03] is added to the IDT in `arch/i386/kernel/apic.c`:

```
        /* self generated IPI for local APIC timer */
        set_intr_gate(LOCAL_TIMER_VECTOR, apic_timer_interrupt);
+#ifdef CONFIG_HLRT
+       set_intr_gate(HLRT_LOCAL_TIMER_VECTOR, hlrt_apic_timer_interrupt);
+#endif
```

The handler for the interrupt gate is implemented in `arch/i386/kernel/hlrt.c`. It is called whenever the local APIC timer creates an interrupt while in `HLRT_LAPIC_TIMER_MODE_PERIODIC` mode.

```
+void smp_hlrt_apic_timer_interrupt(struct pt_regs regs)
+{
+       wait_queue_head_t *q;
```

First it acknowledges reception of the interrupt to the local APIC.

```
+       ack_APIC_irq();
+       /* use nmi_enter/_exit because irq_exit would execute pending
+          SoftIRQs, which we want to avoid */
+       nmi_enter();
```

Then it checks if the hard real-time process is sleeping in the wait queue. If so, it is woken. Otherwise, a `SIGALRM` signal is sent to the process because it missed its deadline and did not call `hlrt_wait_for_apic_timer()` in due time.

```
+       q = &__get_cpu_var(hlrt_apic_timer_wait_queue);
+       if (waitqueue_active(q)) {
+               wake_up(q);
+       } else {
+               send_sig(SIGALRM, current, 0);
+               printk(KERN_WARNING "HLRT: process %i missed its "
+                                       "deadline\n",
+                       current->pid);
+       }
+       nmi_exit();
+}
```

4.1.2 Necessary Adjustments

4.1.2.1 wakeup

Affected files:

- `kernel/sched.c`

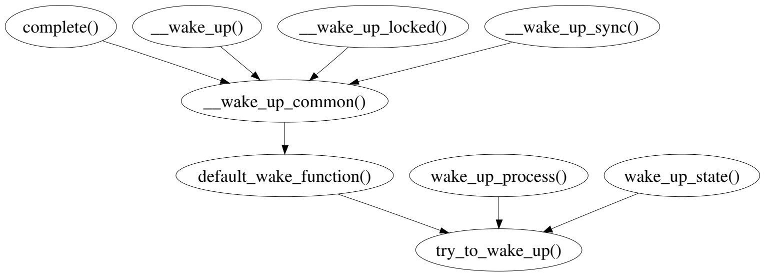

Figure 4.1: Callers of `try_to_wake_up()`

When waking up sleeping processes, care must be taken to put them on the right CPU. While HLRT processes must continue to run on their reserved CPU, normal processes must continue on unreserved CPUs. The function responsible for this is `try_to_wake_up()` in `kernel/sched.c`. It is called by the various other functions that wake up processes (see figure 4.1), and therefore it is sufficient to change this function.

In case that the to-be-woken-up process shall be migrated to the current CPU because it is not in a runqueue and not currently running, the CPU affinity check must be extended to ensure that the current CPU is not reserved or the process uses the `SCHED_HLRT` class. Note that this does not allow an HLRT process to "hop" between reserved CPUs because it has only its own reserved CPU in its `cpus_allowed` mask (which is checked here anyway).

```
                /*
                 * Fast-migrate the task if it's not running or runnable
                 * currently. Do not violate hard affinity.
                 */
                if (unlikely(sync && !task_running(rq, p) &&
                        (task_cpu(p) != smp_processor_id()) &&
+#ifdef CONFIG_HLRT
+                       (!hlrt_cpu_reserved() ||
+                        (p->policy == SCHED_HLRT)) &&
+#endif
                                cpu_isset(smp_processor_id(),
                                                p->cpus_allowed))) {

                        set_task_cpu(p, smp_processor_id());
                        task_rq_unlock(rq, &flags);
                        goto repeat_lock_task;
                }
```

The second and more important scenario is when a non-HLRT process previously ran on a CPU that has now become reserved by another process and is now supposed to be woken on that CPU. In that case it is necessary to find a new CPU for the process by taking any CPU from the intersection of the `cpus_allowed` mask of the process and the global `hlrt_cpus_allowed` mask. If the result is empty, the first, never reserved CPU is taken. The new CPU is assigned to the process using `set_task_cpu()`.

```
+#ifdef CONFIG_HLRT
+               else if (unlikely(!task_running(rq, p) &&
+                        (p->policy != SCHED_HLRT) &&
+                        !cpu_isset(task_cpu(p), hlrt_cpus_allowed))) {
+                       cpumask_t cpus_allowed;
+                       int dest_cpu;
+
+                       cpus_and(cpus_allowed, p->cpus_allowed,
+                                hlrt_cpus_allowed);
+                       cpus_and(cpus_allowed, cpus_allowed,
+                                cpu_online_map);
+                       dest_cpu = cpus_empty(cpus_allowed) ?
+                               first_cpu(cpu_possible_map) :
+                               any_online_cpu(cpus_allowed);
+                       set_task_cpu(p, dest_cpu);
+                       task_rq_unlock(rq, &flags);
+                       goto repeat_lock_task;
+               }
+#endif
```

4.1.2.2 fork

Affected files:

- `kernel/sched.c`

If a process performs a `fork()` or `clone()` system call, the resulting processes are initially identical[4]. In case of an HLRT process this must be avoided because we do not want two processes to share the same reserved CPU. Therefore we do not allow the `SCHED_HLRT` class to be inherited. The correct place for this is the function `sched_fork()` which does scheduler-related initialisations of a newly-forked process `p`. Here we check if its (inherited) scheduling policy is `SCHED_HLRT` and, if so, set it to `SCHED_NORMAL` and allow it to run on all CPUs.

```
+#ifdef CONFIG_HLRT
+       /* no inheritance of hard real-time scheduling class */
+       if (p->policy == SCHED_HLRT) {
+               p->policy = SCHED_NORMAL;
+               set_user_nice(p, 0);
+               p->rt_priority = 0;
+               p->cpus_allowed = CPU_MASK_ALL;
```

[4]except for their PID, of course

```
+       }
+#endif
```

A forked process starts its execution on the CPU on which the parent process runs at the time of the fork. For a child of an HLRT process special care must be taken to put it on a different CPU. This is done in the function `wake_up_forked_process()`. If `current` (the parent) is an HLRT process, the CPU for child `p` is determined by taking any CPU from the intersection of the child's `cpus_allowed` and the global `hlrt_cpus_allowed` mask. As before, if the result is empty, the first CPU is used.

```
+#ifdef CONFIG_HLRT
+       /* put child on a different CPU if we're a HLRT process */
+       if (current->policy == SCHED_HLRT) {
+               cpumask_t cpus_allowed;
+               int dest_cpu;
+
+               cpus_and(cpus_allowed, p->cpus_allowed, hlrt_cpus_allowed);
+               cpus_and(cpus_allowed, cpus_allowed, cpu_online_map);
+               dest_cpu = cpus_empty(cpus_allowed) ?
+                       first_cpu(cpu_possible_map) :
+                       any_online_cpu(cpus_allowed);
+               set_task_cpu(p, dest_cpu);
+               task_rq_unlock(rq, &flags);
+               rq = task_rq_lock(p, &flags);
+               __activate_task(p, rq);
+               task_rq_unlock(rq, &flags);
+               return;
+       }
+#endif
        set_task_cpu(p, smp_processor_id());
```

4.1.2.3 set_cpus_allowed

Affected files:

- `kernel/sched.c`

Changing the `cpus_allowed` mask of a process (i. e. its CPU affinity) can be done via the `sched_setaffinity()` system call. Yet, an HLRT process cannot be allowed to do so because it must not leave its reserved CPU (which is the only enabled CPU in its `cpus_allowed` mask). The system call handler has been extended by the following check:

```
        if ((current->euid != p->euid) && (current->euid != p->uid) &&
                        !capable(CAP_SYS_NICE))
                goto out_unlock;
+#ifdef CONFIG_HLRT
```

```
+       if (p->policy == SCHED_HLRT)
+               goto out_unlock;
+#endif

        retval = set_cpus_allowed(p, new_mask);
 out_unlock:
```

The actual implementation is split into the two functions `set_cpus_allowed()` and `__set_cpus_allowed()`. The latter decides whether a CPU change is necessary and how it must be done. Here it is important that the global `hlrt_cpus_allowed` mask be considered as well. Therefore, if the process is not an HLRT process, the intersection of the new mask and `hlrt_cpus_allowed` is used. If it is empty, the first, never reserved CPU is used. If the process is actually an HLRT process, no further checks are needed here because `hlrt_set_hlrt_policy()` must be able to use this function to move the process to its reserved CPU, and the process itself cannot call this function (see above).

```
        p->cpus_allowed = new_mask;
+
+#ifdef CONFIG_HLRT
+       if (p->policy != SCHED_HLRT) {
+               cpus_and(new_mask, new_mask, hlrt_cpus_allowed);
+               cpus_and(new_mask, new_mask, cpu_online_map);
+               if (cpus_empty(new_mask))
+                       new_mask = cpumask_of_cpu(
+                               first_cpu(cpu_possible_map));
+       }
+#endif
```

4.1.2.4 Load balancing

Affected files:

- `kernel/sched.c`

As explained in section 3.3, load balancing is another way for a process to change its CPU. Again, caution is necessary to not violate any CPU reservations. On the one hand, a reserved CPU must not pull processes from other runqueues, even if these are much longer than its own runqueue. The following block has been added to the function `load_balance()`:

```
+#ifdef CONFIG_HLRT
+       /* Do not perform load balancing on a reserved CPU */
+       if (hlrt_cpu_reserved())
+               return;
+#endif
```

On the other hand, we do not want to pull tasks from the runqueue of a reserved CPU to another CPU. The function `find_busiest_runqueue()`, called by `load_balance()`, checks this before examining another CPU's runqueue:

```
        for (i = 0; i < NR_CPUS; i++) {
                if (!cpu_isset(i, cpumask))
                        continue;
+#ifdef CONFIG_HLRT
+               /* do not touch the runqueue of a reserved CPU */
+               if (!cpu_isset(i, hlrt_cpus_allowed))
+                       continue;
+#endif

                rq_src = cpu_rq(i);
                if (idle || (rq_src->nr_running <
                             this_rq->prev_cpu_load[i]))
                        load = rq_src->nr_running;
```

Anyway, the second modification is mostly an optimisation since an HLRT process cannot be pulled from its CPU due to its `cpus_allowed` mask. Yet this avoids unnecessary locking of the reserved CPU's runqueue.

4.1.2.5 Timers

Affected files:

- `include/linux/timer.h`
- `kernel/timer.c`

As already mentioned, timers need special treatment on a reserved CPU. One reason is that expired timers are normally processed by the interrupt handler of the local APIC timer. If we use our own handler or the local APIC timer is completely disabled, timers will not work any more. Another reason is the fact that we do not want our hard real-time process to be interrupted by other code, and that includes kernel timers.

So, basically two situations must be handled. Already activated timers must be moved away when the local APIC timer is changed to a non-standard mode, and new timers must be activated on a different CPU if the local APIC timer mode has already been changed.

The first situation is dealt with, as already indicated, in the function `hlrt_move_timers()`, implemented in `kernel/timer.c` and called when the local APIC timer is changed to a non-standard mode. Each CPU has a structure `tvec_bases` of type `tvec_base_t` that contains five arrays of timer lists in which the timers are sorted by their expiration time. After determining the source and destination bases and checking that they differ, both structures are locked in the appropriate order.

```
+void hlrt_move_timers(int from_cpu, int to_cpu)
+{
+       tvec_base_t *from_base, *to_base;
+       unsigned long flags;
+       int i;
+
+       from_base = &per_cpu(tvec_bases, from_cpu);
+       to_base = &per_cpu(tvec_bases, to_cpu);
+
+       if (from_base == to_base)
+               return;
+
+       local_irq_save(flags);
+       if (from_base < to_base) {
+               spin_lock(&to_base->lock);
+               spin_lock(&from_base->lock);
+       } else {
+               spin_lock(&from_base->lock);
+               spin_lock(&to_base->lock);
+       }
```

Now the function `__hlrt_move_timers_list()` is called for each of the source timer lists. Finally, the locks are released.

```
+       for (i = 0; i < TVR_SIZE; i++) {
+               __hlrt_move_timers_list(from_base->tv1.vec + i, to_base);
+       }
+       for (i = 0; i < TVN_SIZE; i++) {
+               __hlrt_move_timers_list(from_base->tv2.vec + i, to_base);
+               __hlrt_move_timers_list(from_base->tv3.vec + i, to_base);
+               __hlrt_move_timers_list(from_base->tv4.vec + i, to_base);
+               __hlrt_move_timers_list(from_base->tv5.vec + i, to_base);
+       }
+
+       spin_unlock(&to_base->lock);
+       spin_unlock(&from_base->lock);
+       local_irq_restore(flags);
+}
```

The function `__hlrt_move_timers_list()` simply walks through the list of timers for which it was called and adds each timer to the target timer base by calling `internal_add_timer()`. This function takes care of finding the right list in the new timer base. Afterwards, the whole source timer list is cleared.

```
+static void __hlrt_move_timers_list(struct list_head *head,
+                                    tvec_base_t *to_base)
+{
+       struct list_head *curr;
+       struct timer_list *timer;
+
+       curr = head->next;
+       while (curr != head) {
+               timer = list_entry(curr, struct timer_list, entry);
+               curr = curr->next;
+               internal_add_timer(to_base, timer);
+               timer->base = to_base;
+       }
+       INIT_LIST_HEAD(head);
+}
```

The second situation, adding a new timer, is even simpler to handle. It is sufficient to change the function `__mod_timer()` because it is called by `mod_timer()` as well as `add_timer()`. The following code has been added: After setting `new_base` to the timer base structure of the current CPU, it is changed to the timer base of the first CPU if the local APIC timer is not in normal mode.

```
        new_base = &__get_cpu_var(tvec_bases);
+#ifdef CONFIG_HLRT
+       if (__get_cpu_var(hlrt_local_apic_mode) !=
+           HLRT_LAPIC_TIMER_MODE_NORMAL)
+               new_base = &per_cpu(tvec_bases,
+                                   first_cpu(cpu_possible_map));
+#endif
```

4.1.2.6 SoftIRQs

Affected files:

- `drivers/scsi/scsi.c`
- `include/linux/interrupt.h`
- `include/linux/netdevice.h`
- `kernel/softirq.c`
- `net/core/dev.c`

Apart from timers, another way of deferring work to a later point in time is using SoftIRQs—the difference being that the amount of time that is to be waited is unspecified. A very good and up-to-date description of SoftIRQs, tasklets and workqueues can be found in [Lov03].

A SoftIRQ is a function that gets called some time after it has been activated (i. e. the SoftIRQ has been *raised*). An activated SoftIRQ can be executed in any of the following situations:

- when returning from a hardware interrupt
- by the `ksoftirqd` kernel thread
- by code that explicitly checks for pending SoftIRQs

Just like with timers, our goal is to not have hard real-time processes be delayed by SoftIRQs. And again, because SoftIRQs are raised on each CPU individually, they must be transferred to the first, unreserved CPU. This is a nontrivial task because the handlers make heavy use of per-CPU variables[5]. As an example, let us take a look at the SCSI SoftIRQ in `drivers/scsi/scsi.c`. It has a per-CPU list called `scsi_done_q`:

```
static DEFINE_PER_CPU(struct list_head, scsi_done_q);
```

The function `scsi_done()` enqueues finished SCSI commands into the done queue and raises the SCSI SoftIRQ:

```
void scsi_done(struct scsi_cmnd *cmd)
{
        [...]
        local_irq_save(flags);
        list_add_tail(&cmd->eh_entry, &__get_cpu_var(scsi_done_q));
        raise_softirq_irqoff(SCSI_SOFTIRQ);
        local_irq_restore(flags);
}
```

Since `scsi_done_q` is a per-CPU variable, no locking apart from disabling local interrupts is needed[6]. Afterwards, because the SCSI SoftIRQ has been raised on this CPU, it is executed (also on this CPU). The SoftIRQ handler can simply disable local interrupts, take the contents of the list, reenable local interrupts, and then process the elements of the list without any locking:

```
static void scsi_softirq(struct softirq_action *h)
{
        int disposition;
        LIST_HEAD(local_q);
```

[5]a variable of which one instance exists for each CPU, so that basically no locking is needed

[6]to avoid reentrancy on the same CPU

```
        local_irq_disable();
        list_splice_init(&__get_cpu_var(scsi_done_q), &local_q);
        local_irq_enable();

        while (!list_empty(&local_q)) {
                [...]
        }
}
```

Here we have a problem: We do not want to process the SoftIRQ on a reserved CPU, but calling the handler on another CPU only processes that CPU's done queue. So, if we are on a reserved CPU, it is necessary to add the finished SCSI commands to the list of another CPU and then raise that CPU's SCSI SoftIRQ. But for this to be safe, we must re-introduce additional locking with spin locks.

At this point it is important to notice that all newly introduced regions protected by locks in this patch run in constant time. This is important because we must not jeopardise our goal of hard real-time execution by introducing indeterminable latencies waiting for spin locks. The only two exceptions are those locks used when handling the task list and moving the timers. But this is not a problem because they are only used upon initialisation of a hard real-time process.

The kernel patch changes the excerpts from above into the following: Each `scsi_done_q` variable is protected by a corresponding spin lock.

```
 static DEFINE_PER_CPU(struct list_head, scsi_done_q);
+#ifdef CONFIG_HLRT
+static DEFINE_PER_CPU(spinlock_t, scsi_done_q_lock);
+#endif
```

The function `scsi_done()` chooses an appropriate CPU depending on whether the local CPU is reserved, takes the corresponding spin lock, adds the element to the CPU's list and raises the SoftIRQ on the correct CPU by calling the new function `raise_softirq_irqoff_cpu()`.

```
        local_irq_save(flags);
+#ifdef CONFIG_HLRT
+       if (hlrt_cpu_reserved())
+               cpu = first_cpu(cpu_possible_map);
+       else
+               cpu = smp_processor_id();
+       spin_lock(&per_cpu(scsi_done_q_lock, cpu));
+       list_add_tail(&cmd->eh_entry, &per_cpu(scsi_done_q, cpu));
+       spin_unlock(&per_cpu(scsi_done_q_lock, cpu));
+       raise_softirq_irqoff_cpu(SCSI_SOFTIRQ, cpu);
+#else
        list_add_tail(&cmd->eh_entry, &__get_cpu_var(scsi_done_q));
        raise_softirq_irqoff(SCSI_SOFTIRQ);
+#endif
        local_irq_restore(flags);
```

The SCSI SoftIRQ handler now also has to take the spin lock before taking the elements out of the list:

```
        local_irq_disable();
+#ifdef CONFIG_HLRT
+       spin_lock(&__get_cpu_var(scsi_done_q_lock));
+#endif
        list_splice_init(&__get_cpu_var(scsi_done_q), &local_q);
+#ifdef CONFIG_HLRT
+       spin_unlock(&__get_cpu_var(scsi_done_q_lock));
+#endif
        local_irq_enable();
```

So the SCSI SoftIRQ has been fixed now, but there are more:

- timer SoftIRQ
- two tasklet SoftIRQs (normal/high priority)
- two network SoftIRQs (RX/TX)

The timer SoftIRQ is responsible for running the expired timers. As soon as the local APIC timer has either been disabled or changed to custom mode, it is of no further concern because it won't be raised any more. The two tasklet SoftIRQs will be discussed in the next section. That leaves us with the network SoftIRQs.

Each CPU has a structure of type `struct softnet_data`, containing four queues into which elements that need processing by the network SoftIRQ handlers are inserted:

Queue	Producer	Consumer
`input_pkt_queue`	`netif_rx()`	`net_rx_action()` / `process_backlog()`
`poll_list`	`__netif_rx_schedule()`	`net_rx_action()`
`output_queue`	`__netif_schedule()`	`net_tx_action()`
`completion_queue`	`dev_kfree_skb_irq()`	`net_tx_action()`

Here we take the same approach as for the SCSI SoftIRQ: To avoid raising the SoftIRQ on a reserved CPU, we use locking to safely add elements to the queue of the first CPU and raise its SoftIRQ instead.

```
struct softnet_data
```

```
{
        int                     throttle;
        int                     cng_level;
        int                     avg_blog;
        struct sk_buff_head     input_pkt_queue;
        struct list_head        poll_list;
        struct net_device       *output_queue;
        struct sk_buff          *completion_queue;
+#ifdef CONFIG_HLRT
+        spinlock_t              input_pkt_queue_lock;
+        spinlock_t              poll_list_lock;
+        spinlock_t              output_queue_lock;
+        spinlock_t              completion_queue_lock;
+#endif

        struct net_device       backlog_dev;    /* Sorry. 8) */
};
```

Each of the four producers now uses the appropriate spin lock and checks if the current CPU is reserved before adding an element to a queue (because the principle is the same for all of them, only two will be shown here):

```
 static inline void __netif_rx_schedule(struct net_device *dev)
 {
        unsigned long flags;
+#ifdef CONFIG_HLRT
+       struct softnet_data *sd;
+       int cpu;
+#endif

        local_irq_save(flags);
        dev_hold(dev);
+#ifdef CONFIG_HLRT
+       if (hlrt_cpu_reserved())
+               cpu = first_cpu(cpu_possible_map);
+       else
+               cpu = smp_processor_id();
+       sd = &per_cpu(softnet_data, cpu);
+       spin_lock(&sd->poll_list_lock);
+       list_add_tail(&dev->poll_list, &sd->poll_list);
+       spin_unlock(&sd->poll_list_lock);
+#else
        list_add_tail(&dev->poll_list,
                      &__get_cpu_var(softnet_data).poll_list);
+#endif
        if (dev->quota < 0)
                dev->quota += dev->weight;
        else
                dev->quota = dev->weight;
+#ifdef CONFIG_HLRT
+       __raise_softirq_irqoff_cpu(NET_RX_SOFTIRQ, cpu);
+#else
        __raise_softirq_irqoff(NET_RX_SOFTIRQ);
+#endif
        local_irq_restore(flags);
 }


static inline void dev_kfree_skb_irq(struct sk_buff *skb)
{
        if (atomic_dec_and_test(&skb->users)) {
                struct softnet_data *sd;
                unsigned long flags;
+#ifdef CONFIG_HLRT
+               int cpu;
+#endif

                local_irq_save(flags);
+#ifdef CONFIG_HLRT
+               if (hlrt_cpu_reserved())
+                       cpu = first_cpu(cpu_possible_map);
+               else
+                       cpu = smp_processor_id();
+               sd = &per_cpu(softnet_data, cpu);
+               spin_lock(&sd->completion_queue_lock);
+#else
                sd = &__get_cpu_var(softnet_data);
+#endif
                skb->next = sd->completion_queue;
                sd->completion_queue = skb;
+#ifdef CONFIG_HLRT
+               spin_unlock(&sd->completion_queue_lock);
+               raise_softirq_irqoff_cpu(NET_TX_SOFTIRQ, cpu);
+#else
                raise_softirq_irqoff(NET_TX_SOFTIRQ);
+#endif
                local_irq_restore(flags);
        }
}
```

The consumers use the spin locks when removing elements from the queues (for brevity's sake, only one will be shown):

```
static void net_tx_action(struct softirq_action *h)
{
        struct softnet_data *sd = &__get_cpu_var(softnet_data);

        if (sd->completion_queue) {
                struct sk_buff *clist;

                local_irq_disable();
+#ifdef CONFIG_HLRT
+               spin_lock(&sd->completion_queue_lock);
+#endif
                clist = sd->completion_queue;
                sd->completion_queue = NULL;
+#ifdef CONFIG_HLRT
+               spin_unlock(&sd->completion_queue_lock);
+#endif
                local_irq_enable();
                [...]
        }

        if (sd->output_queue) {
                struct net_device *head;

                local_irq_disable();
+#ifdef CONFIG_HLRT
+               spin_lock(&sd->output_queue_lock);
+#endif
                head = sd->output_queue;
                sd->output_queue = NULL;
+#ifdef CONFIG_HLRT
+               spin_unlock(&sd->output_queue_lock);
+#endif
                local_irq_enable();
                [...]
        }
}
```

With these changes (and those of the next section) we have eliminated the need to raise SoftIRQs on a reserved CPU. As a final optimisation, when code explicitly checks for pending SoftIRQs by calling the function `do_softirq()` directly, we return immediately from this function if we are on a reserved CPU:

```
asmlinkage void do_softirq(void)
{
        int max_restart = MAX_SOFTIRQ_RESTART;
        __u32 pending;
        unsigned long flags;

        if (in_interrupt())
                return;

+#ifdef CONFIG_HLRT
+       if (hlrt_cpu_reserved())
+               return;
+#endif
        [...]
}
```

4.1.2.7 Tasklets

Affected files:

- `kernel/softirq.c`

- net/core/flow.c

Tasklets are used by many device drivers to defer work. The are similar to SoftIRQs but can be created dynamically and are simpler to use. Yet, they are implemented via two SoftIRQs: TASKLET_SOFTIRQ executes all normal tasklets while HI_SOFTIRQ executes high-priority tasklets. Because both mechanisms are mostly identical, we will only look at TASKLET_SOFTIRQ here.

The per-CPU variable tasklet_vec contains a list of all scheduled tasklets:

```
 static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec) = { NULL };
```

The function __tasklet_schedule() adds a new tasklet to the list and raises the tasklet SoftIRQ. Like before, we introduce additional locking to add the tasklet to the list of the first CPU if the current CPU is reserved.

```
+#ifdef CONFIG_HLRT
+static DEFINE_PER_CPU(spinlock_t, tasklet_vec_lock) = SPIN_LOCK_UNLOCKED;
+#endif
```

__tasklet_schedule() checks if the current CPU is reserved and uses the first CPU instead. It takes the appropriate spin lock, adds the tasklet structure and raises the SoftIRQ of the correct CPU.

```
 void fastcall __tasklet_schedule(struct tasklet_struct *t)
 {
        unsigned long flags;
+#ifdef CONFIG_HLRT
+       int cpu;
+#endif

        local_irq_save(flags);
+#ifdef CONFIG_HLRT
+       if (hlrt_cpu_reserved())
+               cpu = first_cpu(cpu_possible_map);
+       else
+               cpu = smp_processor_id();
+       spin_lock(&per_cpu(tasklet_vec_lock, cpu));
+       t->next = per_cpu(tasklet_vec, cpu).list;
+       per_cpu(tasklet_vec, cpu).list = t;
+       spin_unlock(&per_cpu(tasklet_vec_lock, cpu));
+       raise_softirq_irqoff_cpu(TASKLET_SOFTIRQ, cpu);
+#else
        t->next = __get_cpu_var(tasklet_vec).list;
        __get_cpu_var(tasklet_vec).list = t;
        raise_softirq_irqoff(TASKLET_SOFTIRQ);
+#endif
        local_irq_restore(flags);
 }
```

The tasklet SoftIRQ handler also uses the spin lock before taking the elements from the tasklet list:

```
static void tasklet_action(struct softirq_action *a)
{
        struct tasklet_struct *list;

        local_irq_disable();
#ifdef CONFIG_HLRT
        spin_lock(&__get_cpu_var(tasklet_vec_lock));
#endif
        list = __get_cpu_var(tasklet_vec).list;
        __get_cpu_var(tasklet_vec).list = NULL;
#ifdef CONFIG_HLRT
        spin_unlock(&__get_cpu_var(tasklet_vec_lock));
#endif
        local_irq_enable();

        while (list) {
                [...]
        }
}
```

Of course, the consequence of this modification is that a tasklet is not necessarily executed on the same CPU as it was scheduled on. Luckily, all tasklets except two do not make use of per-CPU variables. One of the two that does, `flow_cache_flush_tasklet()` (implemented in `net/core/-flow.c`), is only called when changing IPsec policies. Therefore, IPsec should currently not be used in combination with this patch (a warning message is sent to the syslog if the tasklet is scheduled on a reserved CPU). The second one, `rcu_process_callbacks()`, is discussed below.

4.1.2.8 Workqueues

Affected files:

- `kernel/workqueue.c`

Using workqueues is the third way of deferring work. Here, work is always processed in the context of a kernel thread. The fact that, even with a standard kernel, there is no guarantee that work will be processed on the CPU it was queued on, makes the modifications relatively simple. Also, the internal function that adds work to a queue, `__queue_work()`, already uses sufficient locking. The only necessary modification is to choose an appropriate CPU to queue the work on.

Two interface functions can be used to queue work: `queue_work()` and `queue_delayed_work()`, implemented in `kernel/workqueue.c`. The former immediately adds the work to the queue, and only the following block has to be added to its implementation:

```
 int fastcall queue_work(struct workqueue_struct *wq,
                         struct work_struct *work)
 {
        int ret = 0, cpu = get_cpu();

+#ifdef CONFIG_HLRT
+       if (hlrt_cpu_reserved()) {
+               cpu = first_cpu(cpu_possible_map);
+       }
+#endif
        if (!test_and_set_bit(0, &work->pending)) {
                BUG_ON(!list_empty(&work->entry));
                __queue_work(wq->cpu_wq + cpu, work);
                ret = 1;
        }
        put_cpu();
        return ret;
 }
```

The latter uses a timer function that, when the timer expires, adds the work to the queue. This timer function was modified as follows:

```
 static void delayed_work_timer_fn(unsigned long __data)
 {
        struct work_struct *work = (struct work_struct *)__data;
        struct workqueue_struct *wq = work->wq_data;

+#ifdef CONFIG_HLRT
+       int cpu;
+
+       if (hlrt_cpu_reserved())
+               cpu = first_cpu(cpu_possible_map);
+       else
+               cpu = smp_processor_id();
+       __queue_work(wq->cpu_wq + cpu, work);
+#else
        __queue_work(wq->cpu_wq + smp_processor_id(), work);
+#endif
 }
```

Yet, this is only a precaution because timer functions are not executed on reserved CPUs due to our previous modifications.

4.1.2.9 Read-Copy Update

Affected files:

- `arch/i386/kernel/entry.S`
- `arch/i386/kernel/hlrt_entry.S`
- `include/linux/rcupdate.h`
- `kernel/rcupdate.c`

The Read-Copy Update mechanism (RCU) is a mutual exclusion algorithm designed to work around the problem of fast modern CPUs wasting lots of cycles in spin locks, especially on large SMP machines. Using this algorithm allows readers to access a shared data structure without any locking. Writers, on the other hand, apart from serialising themselves by classical mutual exclusion, must make a copy of the data, modify this copy and then change all global references to the old copy to point to the new copy. As soon as it is guaranteed that all CPUs have lost any local references to the old copy, a special callback is used to free the old copy.

The callbacks are processed in batches. This happens as soon as all CPUs have gone through a *quiescent state*, i. e. a state where all local references shared to data structures have been lost and no assumptions are made based on their previous contents. The Linux kernel knows three of such states:

- a context switch
- running in the idle-loop
- running in user space

Because writing data is much more expensive, the RCU mechanism is best suited for data that is more often read than written, e. g. caches or routing tables.

The function `call_rcu()`, implemented in `kernel/rcupdate.c`, is used to register a callback function for processing in the next batch in a per-CPU list variable, `RCU_nxtlist`. Employing the same scheme as for the SoftIRQ handling, we introduce a spin lock to protect this list and can now add callbacks to the list of the first CPU if necessary:

```
 void fastcall call_rcu(struct rcu_head *head, void (*func)(void *arg),
                        void *arg)
 {
        int cpu;
        unsigned long flags;

        head->func = func;
        head->arg = arg;
        local_irq_save(flags);
+#ifdef CONFIG_HLRT
+       if (hlrt_cpu_reserved())
+               cpu = first_cpu(cpu_possible_map);
+       else
+               cpu = smp_processor_id();
+       spin_lock(&RCU_nxtlist_lock(cpu));
+#else
        cpu = smp_processor_id();
+#endif
        list_add_tail(&head->list, &RCU_nxtlist(cpu));
+#ifdef CONFIG_HLRT
+       spin_unlock(&RCU_nxtlist_lock(cpu));
+#endif
        local_irq_restore(flags);
 }
```

Each CPU has an RCU tasklet, named `rcu_process_callbacks()`, that processes its `RCU_nxtlist`. It must now also use the spin lock to protect access to the list:

```
        local_irq_disable();
+#ifdef CONFIG_HLRT
+       spin_lock(&RCU_nxtlist_lock(cpu));
+#endif
        if (!list_empty(&RCU_nxtlist(cpu)) &&
            list_empty(&RCU_curlist(cpu))) {
                list_splice(&RCU_nxtlist(cpu), &RCU_curlist(cpu));
                INIT_LIST_HEAD(&RCU_nxtlist(cpu));
+#ifdef CONFIG_HLRT
+               spin_unlock(&RCU_nxtlist_lock(cpu));
+#endif
                local_irq_enable();
```

But this modification alone is not sufficient to keep the RCU mechanism working. If the local APIC timer is disabled and so the scheduler is not invoked regularly, the function `rcu_check_callbacks()` does not get called and it is not detected that the CPU is in a quiescent state, although it may be executing in user space all the time. And even if we did not disable the local APIC timer on a reserved CPU, we do not want the RCU tasklet to be scheduled on it. The latter can be remedied by changing `rcu_check_callbacks()` as follows:

```
+#ifdef CONFIG_HLRT
+       if (hlrt_cpu_reserved())
+               rcu_check_quiescent_state();
+       else
+               tasklet_schedule(&RCU_tasklet(cpu));
+#else
        tasklet_schedule(&RCU_tasklet(cpu));
+#endif
```

Instead of scheduling the tasklet we call `rcu_check_quiescent_state()` directly if running on a reserved CPU. This function will start a new batch if this CPU was the last to pass through a quiescent state. Normally, the tasklet will call this function, apart from executing the callbacks.

But what if we disabled the local APIC timer? We need another way of determining that a CPU passed through a quiescent state. A good moment is when a process returns from a system call back to user space. This can be easily detected by adding a call into the system call handlers (both the normal (vector 0x80) and our own (vector 0x83)):

```
        call *sys_call_table(,%eax,4)
        movl %eax,EAX(%esp)             # store the return value
+#ifdef CONFIG_HLRT
+       pushl %eax
+       call hlrt_quiescent_state
+       popl %eax
+#endif
```

The function `hlrt_quiescent_state()` basically works in the same way as `rcu_check_quiescent_state()`, i. e. it will start a new batch if necessary. Of course it will not process callbacks on this CPU. Yet, one should realise that this solution is not optimal because it can take a significantly longer time than normal for a CPU to pass through a *detected* quiescent state if its local APIC timer is disabled and the process remains in user space for a long time (without doing system calls).

4.2 User-Space Library

A user-space library, called `libhlrt`, is provided for ease of use of the features of the HLRT kernel patch. It is based upon the library developed by the *HaRTLinC* project. Its interface functions will be introduced in the following. See sections 4.3 and 5.1.1 for usage examples.

4.2.1 Functions

- int **hlrt_begin_rt_process** (void)
- int **hlrt_end_rt_process** (void)
- int **hlrt_enable_irq** (int interrupt)
- int **hlrt_disable_irq** (int interrupt)
- int **hlrt_disable_local_timer_irq** (void)
- int **hlrt_enable_local_timer_irq** (void)
- int **hlrt_set_periodic_local_timer_irq** (unsigned int period)
- void **hlrt_yield** (void)

4.2.2 Function Documentation

4.2.2.1 int hlrt_begin_rt_process (void)

Try to give the calling process hard real-time status (i. e. reserve a CPU).

Returns:
0 on success, != 0 on error (see <errno.h>)

4.2.2.2 int hlrt_disable_irq (int *interrupt*)

If a hard real-time process does not want to receive a previously enabled IRQ any longer, it can use this function to disable the delivery of that IRQ to the reserved CPU. Afterwards, the IRQ will be delivered to all non-reserved CPUs.

Parameters:
interrupt IRQ number to release

Returns:
0 on success, != 0 on error (see <errno.h>)

4.2.2.3 int hlrt_disable_local_timer_irq (void)

Disable the Local APIC Timer of the CPU the calling hard real-time process runs on.

Returns:
0 on success, != 0 on error (see <errno.h>)

4.2.2.4 int hlrt_enable_irq (int *interrupt*)

After a process has been given hard real-time status, no interrupt requests are relayed to the reserved CPU any more. If the process stills want to receive interrupt requests, it can enable single IRQs by means of this function. The requested interrupts will be delivered exclusively to the reserved CPU.

Parameters:
interrupt IRQ number to request

Returns:
0 on success, != 0 on error (see <errno.h>)

4.2.2.5 int hlrt_enable_local_timer_irq (void)

Re-enable the Local APIC Timer of the CPU the calling hard real-time process runs on.

Returns:
0 on success, != 0 on error (see <errno.h>)

4.2.2.6 int hlrt_end_rt_process (void)

If the calling process has hard real-time status, this function returns it to normal status.

Returns:
0 on success, != 0 on error (see <errno.h>)

4.2.2.7 int hlrt_set_periodic_local_timer_irq (unsigned int *period*)

Start the Local APIC Timer of the CPU the calling hard real-time process runs on in periodic mode with the specified period. When the period elapses, the process must have performed a **hlrt_yield()**(p. 60) call, from which it will then be woken. If it did not, a SIGALRM signal will be sent to the process. The periodic mode can be stopped by **hlrt_enable_local_timer_irq()**(p. 60) or **hlrt_disable_local_timer_irq()**(p. 59).

Parameters:
period the timer period in nanoseconds

Returns:
0 on success, != 0 on error (see <errno.h>)

4.2.2.8 void hlrt_yield (void)

After setting a timer period with **hlrt_set_periodic_local_timer_irq()**(p. 60), this function must be called once per period. It will return as soon as the period has elapsed.

4.3 Basic Structure of an HLRT Process

This section shows an example structure of a “classic” HLRT application without utilisation of the TTA support. It is basically an empty framework that can be filled according to one’s needs. The application demonstrates the correct order of HLRT function calls and their effects. Refer to section 5.1.1 for a usage example of the TTA functions.

The application is split in two parts: One real-time thread that performs the CPU reservation and the processing of data in hard real-time as well as one normal thread that performs "off-line" processing, e. g. data visualisation. The two threads communicate via ringbuffers. This avoids IPC system calls and blocking in the kernel.

4.3.1 test/hlrt_template.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <signal.h>
#include <sched.h>

#include "hlrt.h"

#include "hardware.h"


#define RINGBUF_SIZE 1024


volatile int term_flag = 0;


void sighandler(int sig) {
  term_flag = 1;
}


void *create_ringbuffer(unsigned int size) {
  /* allocate ringbuffer with ‘size’ elements */
  /* ... */
  return NULL;
}


int read_from_ringbuffer(void *buf, int *value) {
  /* get integer from ringpuffer and return it in ‘value’.
     If none is available, return 0, otherwise 1. */
  /* ... */
  return 1;
}


int write_to_ringbuffer(void *buf, int value) {
  /* put ‘value’ into ringpuffer.  return 1 on success,
     otherwise 0. */
  /* ... */
  return 1;
}


int open_hardware_device(void) {
  /* initialise some special hardware device */
  /* ... */

  /* optional: route its IRQ to the reserved CPU */
  hlrt_enable_irq(hw_irq);

  return 1;
}


int close_hardware_device(void) {
  /* close special hardware device */
  /* ... */

  /* optional: route its IRQ to the non-reserved CPUs */
  hlrt_disable_irq(hw_irq);
}


void *rt_thread_func(void *buf) {
  /* initialisation */
  int invalue = 0;
  int outvalue = 0;
  int old_invalue = -1;

  /* become HLRT process */
  hlrt_begin_rt_process();

  open_hardware_device();

  /* hard real-time activities */
  while (!term_flag) {
    invalue = read_from_hardware_device();
    outvalue = do_calculation(invalue);
    write_to_hardware_device(outvalue);

    if (invalue != old_invalue) {
      write_to_ringbuffer(buf, invalue);
      old_invalue = invalue;
    }
  }

  close_hardware_device();

  /* finish HLRT process */
  hlrt_end_rt_process();

  return NULL;
}
```

```
int main() {
  /* initialisation */
  pthread_t rt_thread;
  struct sigaction sigact;
  void *buf;
  int value;

  /* register signal handler */
  sigact.sa_handler = sighandler;
  sigemptyset(&sigact.sa_mask);
  sigact.sa_flags = 0;
  sigaction(SIGINT, &sigact, NULL);
  sigaction(SIGQUIT, &sigact, NULL);
  sigaction(SIGTERM, &sigact, NULL);

  /* allocate ringbuffer */
  buf = create_ringbuffer(RINGBUF_SIZE);

  /* create HLRT thread */
  pthread_create(&rt_thread, NULL, rt_thread_func, buf);

  /* non-hard-real-time activities */
  while (!term_flag) {
    if (read_from_ringbuffer(buf, &value)) {
      printf("%i\n", value);
    }

    /* let other (non-HLRT-) processes run */
    sched_yield();
  }

  return 0;
}
```

Chapter 5

Evaluation

5.1 Example Application

We will now take a look at an example application that demonstrates the correct operation of the kernel patch. Since any complex real-world example would go beyond the scope of this work, this application is kept very short and simple.

5.1.1 hlrt_tta

The example application uses the previously introduced user-space API to reserve a CPU. Then it does a TSC calibration by reading its current value, sleeping one second[1], and reading its value again. Now it registers a signal handler that would be called if the process missed its deadline, and initialises the local APIC timer in periodic mode with a one millisecond period.

The following loop tries to use up almost all CPU cycles of each period[2], then calls `hlrt_yield()` to meet its deadline and then stores the current TSC value as soon as the new period starts (upon return from `hlrt_yield()`).

Finally, the reserved CPU is released and the differences between the recorded TSC values are calculated and printed.

5.1.1.1 test/hlrt_tta.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
```

[1]of course this is only an estimate and not very accurate

[2]the upper bound of the inner loop was chosen for a 1.80 GHz Xeon

```
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include "hlrt.h"

#define rdtscl(low) \
     __asm__ __volatile__("rdtsc" : "=a" (low) : : "edx")

int stop = 0;

void sighandler(int sig) {
  stop = 1;
  fputs("Signal handler called!\n", stderr);
}

int main(int argc, char *argv[]) {
  struct timespec req = { 1, 0 };
  struct sigaction sa;
  uint32_t t[200];
  int i;

  /* reserve CPU */
  if (hlrt_begin_rt_process() != 0) {
    printf("hlrt_begin_rt_process failed\n");
    return EXIT_FAILURE;
  }

  /* calibrate TSC */
  rdtscl(t[0]);
  nanosleep(&req, NULL);
  rdtscl(t[1]);

  /* register signal handler for deadline violation */
  sa.sa_handler = sighandler;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGALRM, &sa, NULL);

  /* start local APIC timer in periodic mode */
  hlrt_set_periodic_local_timer_irq(1000000 /* 1 ms */);

  /* eat CPU cycles, call hlrt_yield() and store the TSC value
   * for each new period
   */
  for (i = 2; !stop && i < 200; i++) {
    unsigned int j;

    for (j = 0; j < 300000; j++)
      ;
    hlrt_yield();
    rdtscl(t[i]);
  }

  /* release CPU */
  if (hlrt_end_rt_process() != 0) {
    printf("hlrt_end_rt_process failed\n");
    return EXIT_FAILURE;
  }

  /* print differences between TSC values */
  for (i = 1; i < 200; i++) {
    printf("%3i: %u\n", i, t[i] - t[i - 1]);
  }

  return EXIT_SUCCESS;
}
```

5.1.1.2 Output

```
$ ./hlrt_tta
  1: 1799983452
  2: 1847764
  3: 1795782
  4: 1796606
  5: 1796618
  6: 1796600
  7: 1796998
  8: 1796810
  9: 1797020
 10: 1796030
 11: 1796432
 12: 1796462
 13: 1796546
 14: 1797142
 15: 1796390
 16: 1796446
 17: 1797186
 18: 1796194
 19: 1796544
 20: 1796944
 21: 1796616
 22: 1796252
 23: 1796684
 24: 1796392
 25: 1796640
 26: 1796570
 27: 1796618
 28: 1796614
 29: 1796630
 30: 1796604
 31: 1796616
 32: 1796614
 33: 1796662
 34: 1796570
 35: 1796618
 36: 1796614
 37: 1796618
 38: 1796616
 39: 1796626
 40: 1796604
 41: 1796618
 42: 1796614
 43: 1796616
 44: 1796618
 45: 1796616
 46: 1796664
 47: 1796568
 48: 1796616
 49: 1796616
 50: 1796634
 51: 1796596
 52: 1796632
 53: 1796602
 54: 1796616
 55: 1796630
 56: 1796600
 57: 1796618
 58: 1796592
 59: 1796660
 60: 1796594
 61: 1796618
 62: 1796582
 63: 1796780
 64: 1796462
 65: 1796658
 66: 1796606
 67: 1796574
 68: 1796646
 69: 1796620
 70: 1796636
 71: 1796596
 72: 1796582
 73: 1796648
 74: 1796616
 75: 1796616
 76: 1796618
 77: 1796616
 78: 1796614
 79: 1796596
 80: 1796652
 81: 1796622
 82: 1796596
 83: 1796614
 84: 1796616
 85: 1796594
 86: 1796638
 87: 1796618
 88: 1796636
 89: 1796616
 90: 1796594
 91: 1796638
 92: 1796596
 93: 1796614
 94: 1796658
 95: 1796574
 96: 1796628
 97: 1796604
 98: 1796616
 99: 1796618
100: 1796616
101: 1796590
102: 1796666
103: 1796590
104: 1796618
105: 1796534
106: 1796698
107: 1796602
108: 1796628
109: 1796560
110: 1796674
```

```
111: 1796634          | 141: 1796624          | 171: 1796640
112: 1796618          | 142: 1796610          | 172: 1796422
113: 1796596          | 143: 1797358          | 173: 1796566
114: 1796616          | 144: 1796544          | 174: 1796600
115: 1796634          | 145: 1796242          | 175: 1797182
116: 1796596          | 146: 1796804          | 176: 1796606
117: 1796616          | 147: 1796160          | 177: 1796252
118: 1796616          | 148: 1796586          | 178: 1796466
119: 1796616          | 149: 1796616          | 179: 1796802
120: 1796618          | 150: 1796636          | 180: 1796424
121: 1796602          | 151: 1796618          | 181: 1796980
122: 1796642          | 152: 1796604          | 182: 1796208
123: 1796602          | 153: 1796646          | 183: 1796626
124: 1796636          | 154: 1796554          | 184: 1796650
125: 1796596          | 155: 1796660          | 185: 1796596
126: 1796636          | 156: 1796574          | 186: 1796604
127: 1796596          | 157: 1796624          | 187: 1796626
128: 1796616          | 158: 1796648          | 188: 1796618
129: 1796640          | 159: 1796784          | 189: 1796616
130: 1796594          | 160: 1796468          | 190: 1796580
131: 1796616          | 161: 1796554          | 191: 1796652
132: 1796616          | 162: 1796648          | 192: 1796616
133: 1796614          | 163: 1796636          | 193: 1796614
134: 1796632          | 164: 1797132          | 194: 1796616
135: 1796586          | 165: 1796612          | 195: 1796602
136: 1796630          | 166: 1796396          | 196: 1796724
137: 1796638          | 167: 1796522          | 197: 1796524
138: 1796594          | 168: 1796930          | 198: 1796634
139: 1796636          | 169: 1796580          | 199: 1796618
140: 1796596          | 170: 1796332          | $
```

The program was executed while performing a kernel build to produce load on the machine. The first difference of the output is the calibration result: The TSC counts roughly 1,800,000,000 clock cycles per second ($\Rightarrow$ 1.80 GHz). The second difference is slightly larger than the rest because it includes the time for setting up the signal handler and the local APIC timer, so it will not be considered for statistical analysis.

The remaining 197 values will be considered for statistical analysis. They have been converted from CPU cycles into nanoseconds (1.8 cycles per nanosecond) for better comprehensibility. The values have a maximal value of 998,532 ns, a mean value of 998,118 ns and a standard deviation of 90 ns (see figures 5.1 and 5.2).

5.1.2 itimer

To show that this is actually an innovation and improvement, there is a second example program that basically does the same thing, but without using the HLRT API functions. Instead, it uses an interval timer, programmed with `setitimer()`, and the `select()` system call to wait for the `SIGALRM` signal indicating the start of a new period. Of course, here we can detect a missed deadline only indirectly by looking at the TSC differences.

5.1.2.1 test/itimer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <sys/select.h>

#define rdtscl(low) \
     __asm__ __volatile__("rdtsc" : "=a" (low) : : "edx")

void sighandler(int sig) {
}

int main(int argc, char *argv[]) {
  struct itimerval itv = { { 0, 1000 }, { 0, 1000 } };
  struct timespec req = { 1, 0 };
  struct sigaction sa;
  uint32_t t[200];
  int i;

  /* calibrate TSC */
  rdtscl(t[0]);
  nanosleep(&req, NULL);
  rdtscl(t[1]);

  /* register signal handler for period end */
  sa.sa_handler = sighandler;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGALRM, &sa, NULL);

  /* start interval timer */
  setitimer(ITIMER_REAL, &itv, NULL);

  /* eat CPU cycles, call select() and store the TSC value
   * for each new period
   */
  for (i = 2; i < 200; i++) {
    unsigned int j;

    for (j = 0; j < 300000; j++)
      ;
    select(0, NULL, NULL, NULL, NULL);
    rdtscl(t[i]);
  }

  /* print differences between TSC values */
  for (i = 1; i < 200; i++) {
    printf("%3i: %u\n", i, t[i] - t[i - 1]);
  }

  return EXIT_SUCCESS;
}
```

5.1.2.2 Output

```
$ ./itimer                 40: 3593658                80: 3593834
  1: 1798519340            41: 3593312                81: 3592614
  2: 3610052               42: 3591930                82: 3593166
  3: 3574290               43: 3593840                83: 3592962
  4: 3594080               44: 5389164                84: 3593298
  5: 3593208               45: 3593696                85: 3594354
  6: 3592352               46: 5393234                86: 3592468
  7: 3594214               47: 5386702                87: 3591774
  8: 3593430               48: 5388806                88: 3593982
  9: 3592494               49: 5389962                89: 3592916
 10: 4341922               50: 5390076                90: 3593066
 11: 2861240               51: 5389068                91: 3592866
 12: 3595828               52: 5391028                92: 3593290
 13: 3818824               53: 5387656                93: 3596282
 14: 8072250               54: 3593732                94: 3589990
 15: 2471518               55: 3592998                95: 3574338
 16: 3585460               56: 3592552                96: 3592810
 17: 3593646               57: 3595382                97: 3593100
 18: 3593766               58: 3592180                98: 3593078
 19: 3592106               59: 3592742                99: 3614638
 20: 3594822               60: 3593906               100: 3571474
 21: 3592414               61: 3594024               101: 3593260
 22: 3595608               62: 3591296               102: 3592958
 23: 3592614               63: 3591640               103: 3593234
 24: 3591602               64: 3594506               104: 3593332
 25: 3593556               65: 3592676               105: 3592832
 26: 3593438               66: 3594222               106: 3592996
 27: 3594668               67: 3593188               107: 3593232
 28: 3592420               68: 3596618               108: 3592950
 29: 3593936               69: 3589262               109: 3593098
 30: 3591760               70: 3591378               110: 3593148
 31: 3593272               71: 3593504               111: 3593112
 32: 3593226               72: 3593648               112: 3592914
 33: 3592772               73: 3593484               113: 3593402
 34: 3593160               74: 3592892               114: 3592848
 35: 3594042               75: 3592392               115: 3593066
 36: 3593902               76: 3595224               116: 3593044
 37: 3592618               77: 3590366               117: 3593156
 38: 3592768               78: 3593046               118: 3596626
 39: 3592296               79: 3593160               119: 3589628
```

```
120: 3593364
121: 3592782
122: 3593162
123: 3592906
124: 3593152
125: 3593218
126: 3593036
127: 3593114
128: 3592978
129: 3593036
130: 3593184
131: 3593022
132: 3593118
133: 3595148
134: 3591106
135: 3593018
136: 3594308
137: 3592044
138: 3593036
139: 3594006
140: 3592172
141: 3594866
142: 3597284
143: 4361474
144: 2829454
145: 3591610
146: 3592590
147: 3592072
148: 3594792
149: 3592132
150: 3592014
151: 3596060
152: 3592322
153: 3594630
154: 3591142
155: 3594032
156: 3592814
157: 3592136
158: 3901340
159: 4990132
160: 1902218
161: 6882966
162: 3898588
163: 3590364
164: 3584446
165: 3591758
166: 3603628
167: 3592450
168: 5424970
169: 1741732
170: 3593118
171: 3593488
172: 5508784
173: 1686712
174: 3583408
175: 3589792
176: 3592888
177: 3592900
178: 3593216
179: 3593216
180: 3593084
181: 3592922
182: 3593672
183: 3592616
184: 3593110
185: 3592992
186: 3593338
187: 3592850
188: 3593474
189: 3592786
190: 3593004
191: 3593254
192: 3598254
193: 3587852
194: 3592980
195: 3593090
196: 3593260
197: 3592958
198: 5155078
199: 2031602
$
```

This output was also created while performing a kernel build. These numbers immediately show a problem of the interval timer: it always needs one additional millisecond before returning[3]. This can easily be demonstrated by changing the interval timer period (variable `itv`) from 1000 microseconds to e. g. 2000 or 3000 microseconds.

As before, the differences have been converted into nanoseconds for statistical analysis. Ignoring the first two values, the remaining 197 differences have a maximal value of 4,484,583 ns, a mean value of 2,061,939 ns and a standard deviation of 365,241 ns (see figures 5.1 and 5.2). The latter value indicates that the dispersion is much higher than with the HLRT API and the maximal value shows us clearly that the interval timer is not suitable for a hard real-time program because several deadlines have been missed (even if we ignore the fact that basically *all* have been missed due to the one millisecond offset).

[3]There are some lower values, but these are always preceded by a missed deadline, i. e. this is only the rest of the following period.

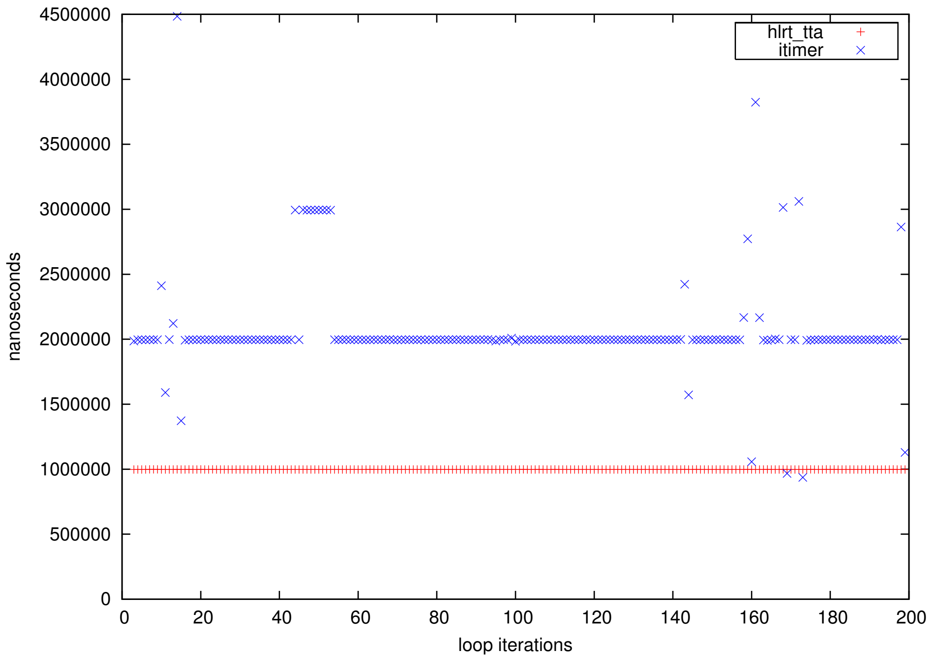

Figure 5.1: TSC value differences

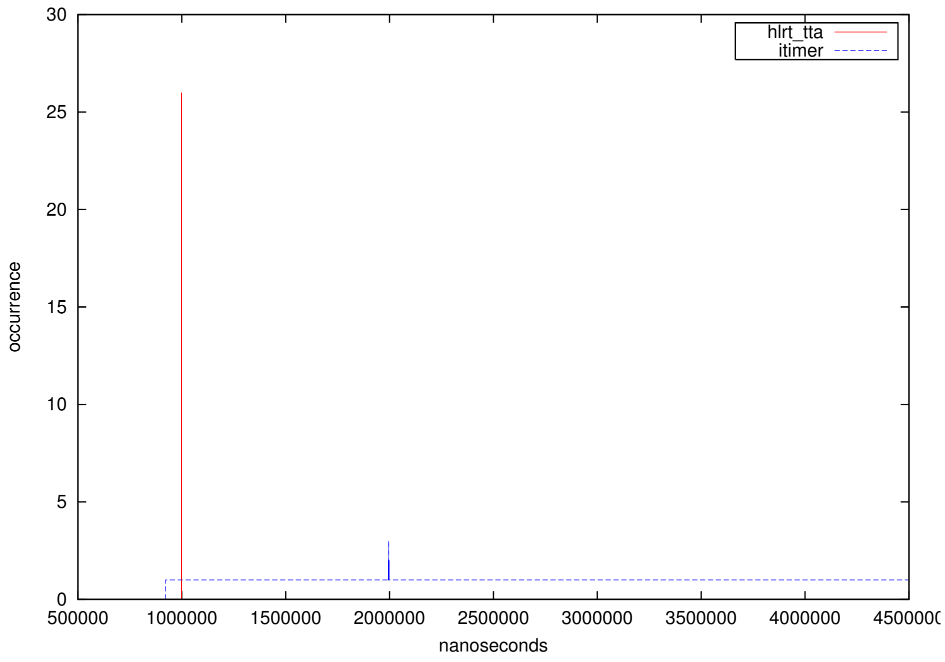

Figure 5.2: Histogram of TSC value differences

Chapter 6

Conclusion

6.1 Comparison to other real-time modifications

In the following section we will take a look at other hard real-time modifications for the Linux kernel. Since each of these projects is at least as complex as this one, I apologise for any unjustified simplifications.

There are at least two other approaches to adding hard real-time capabilities to the Linux kernel: improving kernel preemption to reduce response times and latency as well as adding another layer between the hardware and the kernel to gain control of interrupt handling and blocking.

The first approach is pursued by MontaVista: their *Real-Time Linux Project* integrates several features developed by MontaVista and others:

- Voluntary Preemption by Ingo Molnar
- IRQ thread patches by Scott Wood and Ingo Molnar
- BKL mutex patch by Ingo Molnar (with MontaVista extensions)
- PMutex from Universität der Bundeswehr, München
- MontaVista mutex abstraction layer replacing spinlocks with mutexes

The goal is the elimination of IRQ disabling and spin locks in favour of mutexes. This reduces the problem of latencies resulting from the kernel being in an uninterruptible section (IRQs disabled) and therefore being unable to respond immediately to external events. This approach, being universal in its application, is also very complex due to its intrusiveness. Currently, an experimental kernel patch (consisting of four parts) for Linux 2.6.9 is available.

The other approach is pursued by two projects: RTLinux and RTAI, the latter being based on the former. They add an additional abstraction layer, a real-time kernel, between the hardware and the Linux kernel. This layer controls the CPU and the interrupt handling. The Linux kernel is only executed as the real-time kernel's idle task. Real-time programs, written as special kernel modules, are executed in kernel space. This means that they have full access to the hardware and system calls do not require expensive context switches, but also that there is no memory protection as known for user-space applications. Therefore, a programming error is often fatal. The real-time kernel offers an API different from the standard C library.

In contrast to RTAI, RTLinux does not seem to be available for Linux 2.6. Recent development seems to have been done only for the proprietary RTLinuxPro.

The approach that was taken in this thesis is probably also found in Concurrent's RedHawk Linux, which contains a feature called *CPU shielding*, although only guesses can be made here because source code is not freely available.

Now let us compare these different approaches: The goal of MontaVista's *Real-Time Linux Project* is to make Linux more suitable for embedded devices. The HLRT patch, requiring SMP machines, is not suitable for embedded devices. RTAI and RTLinux can be run on uniprocessor machines, which an HLRT kernel cannot. The MontaVista patch and the HLRT patch both allow the real-time process to use standard C library functions in user space. RTAI and RTLinux have support for periodic processes, similar to the TTA support of the HLRT patch.

	Embedded	SMP	Standard C Library	Periodic processes
RTLP	yes	yes	yes	no
RTLinux	yes	yes	no	yes
RTAI	yes	yes	no	yes
HLRT	no	yes	yes	yes

As we can see, the different Linux real-time modifications have quite different characteristics, with several overlaps. All have strengths and weaknesses that make them more or less suitable for different fields of application.

6.2 Summary

Finally, let us reflect on the achievements of this thesis. The intention was to provide a modification for the Linux 2.6 kernel that provides hard real-time execution for user processes by means of CPU reservation. This goal has been fulfilled. Furthermore, this implementation provides a new mechanism for building time-triggered systems. Both features are available in the provided patch that can be applied to a kernel source tree (see appendix A). All problems that were identified during development have been solved (see section 4.1.2).

The code has been tested extensively and an example application is provided that shows the correct operation of the kernel modification—yet it still has to show its fitness in practice.

As indicated in the introduction, the primary field of application is the automated testing of embedded systems. It can be used as a drop-in replacement for the old HLRT patch when employing Verified's RT-Tester on a Linux 2.6-based cluster. This will increase the potentially usable hardware interfaces by any for which only drivers for Linux 2.6 are available.

Not unmentioned should remain what made this work possible: The commitment and effort of the whole Linux developers community.

> “regression testing”? What's that? If it compiles, it is good, if it boots up it is perfect.
>
> — Linus Torvalds

Appendix A

Complete kernel modification code

This appendix contains the complete source code of the hard real-time kernel modification as a unified diff against Linux 2.6.4. It is also available from http://www.tzi.de/~chref/hlrt/. The accompanying README file describes how to build an HLRT-enabled kernel.

```
diff -uNr linux-2.6.4/arch/i386/defconfig linux-2.6.4-hlrt/arch/i386/defconfig
--- linux-2.6.4/arch/i386/defconfig 2004-03-11 03:55:36.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/defconfig 2004-12-16 16:26:55.000000000 +0100
@@ -89,6 +89,8 @@
 # CONFIG_HPET_EMULATE_RTC is not set
 CONFIG_SMP=y
 CONFIG_NR_CPUS=8
+# CONFIG_HLRT is not set
+# CONFIG_HLRT_NOROOT is not set
 CONFIG_PREEMPT=y
 CONFIG_X86_LOCAL_APIC=y
 CONFIG_X86_IO_APIC=y
diff -uNr linux-2.6.4/arch/i386/Kconfig linux-2.6.4-hlrt/arch/i386/Kconfig
--- linux-2.6.4/arch/i386/Kconfig 2004-03-11 03:55:22.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/Kconfig 2004-12-16 16:26:55.000000000 +0100
@@ -478,6 +478,25 @@
    This is purely to save memory - each supported CPU adds
    approximately eight kilobytes to the kernel image.

+config HLRT
+ bool "HLRT CPU reservation"
+ depends on SMP && !IRQBALANCE && !PREEMPT
+ default n
+ help
+   The HaRTLinC Real-Time extension allows a process to reserve
+   a CPU exclusively for its own use by switching to the HLRT
+   scheduling class.  It is guaranteed to not be interrupted by
+   other processes or interrupt handlers.
```

```
+
+config HLRT_NOROOT
+ bool "Allow CPU reservation for non-root users"
+ depends on HLRT
+ default n
+ help
+   This option allows all users to use the HLRT scheduling class
+   (default: only root).  Warning: Enable this only if you trust
+   all users of your system.
+
 config PREEMPT
  bool "Preemptible Kernel"
  help
diff -uNr linux-2.6.4/arch/i386/kernel/apic.c linux-2.6.4-hlrt/arch/i386/kernel/apic.c
--- linux-2.6.4/arch/i386/kernel/apic.c 2004-03-11 03:55:54.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/kernel/apic.c 2004-12-16 16:26:55.000000000 +0100
@@ -36,6 +36,10 @@
 #include <asm/arch_hooks.h>
 #include <asm/hpet.h>

+#ifdef CONFIG_HLRT
+#include <asm/hlrt.h>
+#endif
+
 #include <mach_apic.h>

 #include "io_ports.h"
@@ -49,6 +53,9 @@
 #endif
  /* self generated IPI for local APIC timer */
  set_intr_gate(LOCAL_TIMER_VECTOR, apic_timer_interrupt);
+#ifdef CONFIG_HLRT
+ set_intr_gate(HLRT_LOCAL_TIMER_VECTOR, hlrt_apic_timer_interrupt);
+#endif

  /* IPI vectors for APIC spurious and error interrupts */
  set_intr_gate(SPURIOUS_APIC_VECTOR, spurious_interrupt);
@@ -940,7 +947,11 @@
  return result;
 }

+#ifdef CONFIG_HLRT
+unsigned int calibration_result;
+#else
 static unsigned int calibration_result;
+#endif

 void __init setup_boot_APIC_clock(void)
 {
@@ -958,14 +969,22 @@
  local_irq_enable();
 }

+#ifdef CONFIG_HLRT
```

```
+void setup_secondary_APIC_clock(void)
+#else
 void __init setup_secondary_APIC_clock(void)
+#endif
 {
  local_irq_disable(); /* FIXME: Do we need this? --RR */
  setup_APIC_timer(calibration_result);
  local_irq_enable();
 }

+#ifdef CONFIG_HLRT
+void disable_APIC_timer(void)
+#else
 void __init disable_APIC_timer(void)
+#endif
 {
  if (using_apic_timer) {
  unsigned long v;
diff -uNr linux-2.6.4/arch/i386/kernel/entry.S linux-2.6.4-hlrt/arch/i386/kernel/entry.S
--- linux-2.6.4/arch/i386/kernel/entry.S 2004-03-11 03:55:24.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/kernel/entry.S 2004-12-16 16:26:55.000000000 +0100
@@ -268,6 +268,11 @@
  jnz syscall_trace_entry
  call *sys_call_table(,%eax,4)
  movl %eax,EAX(%esp)
+#ifdef CONFIG_HLRT
+ pushl %eax
+ call hlrt_quiescent_state
+ popl %eax
+#endif
  cli
  movl TI_FLAGS(%ebp), %ecx
  testw $_TIF_ALLWORK_MASK, %cx
@@ -292,6 +297,11 @@
 syscall_call:
  call *sys_call_table(,%eax,4)
  movl %eax,EAX(%esp) # store the return value
+#ifdef CONFIG_HLRT
+ pushl %eax
+ call hlrt_quiescent_state
+ popl %eax
+#endif
 syscall_exit:
  cli # make sure we don't miss an interrupt
  # setting need_resched or sigpending
diff -uNr linux-2.6.4/arch/i386/kernel/hlrt.c linux-2.6.4-hlrt/arch/i386/kernel/hlrt.c
--- linux-2.6.4/arch/i386/kernel/hlrt.c 1970-01-01 01:00:00.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/kernel/hlrt.c 2004-12-16 16:26:55.000000000 +0100
@@ -0,0 +1,405 @@
+/*
+ * HaRTLinC CPU reservation -- administrative and helper functions
+ *
+ * Authors:
+ * Kai Thomsen <kthomsen@tzi.de>
+ * Christof Efkemann <chref@tzi.de>
+ *
+ * based on the ‘LinuxRT’ patch by
+ * Klaas-Henning Zweck
+ * Mark Hapke
+ *
+ * ported to Linux 2.6 by Christof Efkemann
+ *
+ */
+
+
+#include <linux/sched.h>
+#include <linux/cpumask.h>
+#include <linux/timer.h>
+#include <linux/irq.h>
+#include <linux/module.h>
+#include <asm/types.h>
+#include <asm/errno.h>
+#include <asm/hardirq.h>
+#include <asm/div64.h>
+#include <asm/hlrt.h>
+
+#define HLRT_ABI_REVISION  3
+static char __attribute_used__ hlrt_version[] =
+ "\n$Id: $ HLRT ABI rev. 3\n";
+
+
+extern cpumask_t irq_affinity[NR_IRQS];
+extern unsigned int calibration_result;
+
+
+/*
+ * bit mask representing the reservation of CPUs for hard real-time processes
+ * 1 == CPU is available for non-RT processes
+ * 0 == CPU is reserved for an RT process
+ */
+cpumask_t hlrt_cpus_allowed = CPU_MASK_ALL;
+
+DEFINE_PER_CPU(wait_queue_head_t, hlrt_apic_timer_wait_queue);
+DEFINE_PER_CPU(int, hlrt_local_apic_mode) = HLRT_LAPIC_TIMER_MODE_NORMAL;
+
+static spinlock_t hlrt_cpus_lock = SPIN_LOCK_UNLOCKED;
+static spinlock_t hlrt_irqs_lock = SPIN_LOCK_UNLOCKED;
+
+
+static int hlrt_allocate_next_cpu(void)
+{
+ int cpu = first_cpu(cpu_possible_map);
+
+ spin_lock(&hlrt_cpus_lock);
+
+ /* Note: the first CPU will never be reserved */
+ while (cpu < NR_CPUS) {
+ cpu = next_cpu(cpu, cpu_possible_map);
+ if (cpu < NR_CPUS && cpu_isset(cpu, hlrt_cpus_allowed)) {
+ cpu_clear(cpu, hlrt_cpus_allowed);
+ break;
+ }
+ }
+
+ spin_unlock(&hlrt_cpus_lock);
+
+ return cpu;
+}
+
+
+static int hlrt_release_cpu(int cpu)
+{
+ spin_lock(&hlrt_cpus_lock);
+ cpu_set(cpu, hlrt_cpus_allowed);
+ spin_unlock(&hlrt_cpus_lock);
+
+ return 0;
+}
+
+
+static int hlrt_set_irq_affinity(int irq, cpumask_t mask)
+{
+ /*
+  * check if IRQ handler allows CPU mask
+  */
+ if (!irq_desc[irq].handler->set_affinity)
+ return -EIO;
+
+ /*
+  * check if masked CPUs are actually in use by the system
+  */
+ if (!any_online_cpu(mask))
+ return -EINVAL;
+
+ irq_affinity[irq] = mask;
+ irq_desc[irq].handler->set_affinity(irq, mask);
+
+ return 0;
+}
+
+
+static cpumask_t hlrt_get_irq_affinity(int irq)
+{
+ return irq_affinity[irq];
+}
+
+
+int hlrt_cpu_reserved(void) {
+ return !cpu_isset(smp_processor_id(), hlrt_cpus_allowed);
+}
+EXPORT_SYMBOL(hlrt_cpu_reserved);
+
+
+/*
+ * must be called with tasklist_lock held, runqueue must be unlocked
+ */
+int hlrt_set_hlrt_policy(struct task_struct *p)
+{
+ int cpu;
+
+ if (p->policy == SCHED_HLRT)
+ return 0;
+
+ cpu = hlrt_allocate_next_cpu();
+ if (cpu < NR_CPUS) {
+ /*
+  * kick all other processes from the reserved
+  * CPU's runqueue
+  */
+ struct task_struct *tsk;
+
+repeat_tasklist_scan:
+ for_each_process(tsk) {
+ if (task_cpu(tsk) == cpu && tsk != p) {
+ get_task_struct(tsk);
+ read_unlock_irq(&tasklist_lock);
+ set_cpus_allowed(tsk, tsk->cpus_allowed);
+ read_lock_irq(&tasklist_lock);
+ put_task_struct(tsk);
+ /*
+  * unfortunately, we had to drop the
+  * task_list lock, so we have to start
+  * over again
+  */
+ goto repeat_tasklist_scan;
+ }
+ }
+
+ p->policy = SCHED_HLRT;
+ p->rt_priority = 0;
+ get_task_struct(p);
+ read_unlock_irq(&tasklist_lock);
+ set_cpus_allowed(p, cpumask_of_cpu(cpu));
+ read_lock_irq(&tasklist_lock);
+ put_task_struct(p);
+ printk(KERN_NOTICE "HLRT: reserved CPU %i for process %i\n",
+        cpu, p->pid);
+ return 0;
+ }
+ return -EBUSY;
+}
+
+
+/*
+ * must be called with tasklist_lock held, runqueue must be unlocked
+ */
```

```
+int hlrt_unset_hlrt_policy(struct task_struct *p)
+{
+ int cpu;
+ int irq;
+ struct task_struct *tsk;
+
+ if (p != current)
+ return -EPERM;
+
+ /*
+  * disable preemtion to ensure that we stay on the reserved CPU
+  * until we’re done releasing the IRQs
+  */
+ preempt_disable();
+
+ /* revoke CPU reservation */
+ cpu = first_cpu(p->cpus_allowed);
+ hlrt_release_cpu(cpu);
+
+ p->policy = SCHED_NORMAL;
+ p->rt_priority = 0;
+
+ if (!(p->flags & PF_EXITING)) {
+ /* drop locks for set_cpus_allowed() call */
+ get_task_struct(p);
+ read_unlock_irq(&tasklist_lock);
+ set_cpus_allowed(p, CPU_MASK_ALL);
+ read_lock_irq(&tasklist_lock);
+ put_task_struct(p);
+ }
+
+repeat_tasklist_scan:
+ /*
+  * Stop moving back processes if the CPU has already been reserved
+  * again.  This would result in a livelock in the for_each_process
+  * loop because we have to restart it whenever we find a process
+  * that must be moved.
+  */
+ if (cpu_isset(cpu, hlrt_cpus_allowed)) {
+ for_each_process(tsk) {
+ if (cpus_equal(tsk->cpus_allowed,
+        cpumask_of_cpu(cpu)) &&
+     task_cpu(tsk) != cpu) {
+ get_task_struct(tsk);
+ read_unlock_irq(&tasklist_lock);
+ set_cpus_allowed(tsk, tsk->cpus_allowed);
+ read_lock_irq(&tasklist_lock);
+ put_task_struct(tsk);
+ /*
+  * unfortunately, we had to drop the
+  * task_list lock, so we have to start
+  * over again
+  */
+ goto repeat_tasklist_scan;
+ }
+ }
+ }
+
+ /*
+  * Note: the following two parts work because hlrt_unset_hlrt_policy()
+  * always runs on the previously reserved CPU
+  */
+
+ /* re-enable Local APIC Timer */
+ sys_hlrt_set_local_apic_timer(HLRT_LAPIC_TIMER_MODE_NORMAL, 0, 0);
+
+ /* allow all non-reserved IRQs to be routed to this CPU */
+ for (irq = 0; irq < NR_IRQS; irq++) {
+ sys_hlrt_release_irq(irq);
+ }
+
+ /* now we can re-enable preemption safely */
+ preempt_enable();
+
+ printk(KERN_NOTICE "HLRT: released CPU %i from process %i\n",
+        cpu, p->pid);
+
+ return 0;
+}
+
+
+void smp_hlrt_apic_timer_interrupt(struct pt_regs regs)
+{
+ wait_queue_head_t *q;
+
+ ack_APIC_irq();
+ /* use nmi_enter/_exit because irq_exit would execute pending
+    SoftIRQs, which we want to avoid */
+ nmi_enter();
+ q = &__get_cpu_var(hlrt_apic_timer_wait_queue);
+ if (waitqueue_active(q)) {
+ wake_up(q);
+ } else {
+ send_sig(SIGALRM, current, 0);
+ printk(KERN_WARNING "HLRT: process %i missed its deadline\n",
+        current->pid);
+ }
+ nmi_exit();
+}
+
+
+static unsigned int sync_cpu_counter = 0;
+static spinlock_t sync_cpu_lock = SPIN_LOCK_UNLOCKED;
+
+
+int sys_hlrt_set_local_apic_timer(int mode, unsigned int period,
+   unsigned int n_cpus)
+{
```

```
+ unsigned int clocks, lvtt_value, tmp_value, ver;
+ u64 div;
+
+ switch (mode) {
+ case HLRT_LAPIC_TIMER_MODE_NORMAL:
+ setup_secondary_APIC_clock();
+ __get_cpu_var(hlrt_local_apic_mode) = mode;
+ return 0;
+
+ case HLRT_LAPIC_TIMER_MODE_OFF:
+ __get_cpu_var(hlrt_local_apic_mode) = mode;
+ disable_APIC_timer();
+ hlrt_move_timers(smp_processor_id(),
+  first_cpu(cpu_possible_map));
+ return 0;
+
+ case HLRT_LAPIC_TIMER_MODE_PERIODIC:
+ div = (u64)period * (u64)calibration_result;
+ do_div(div, (1000000000/HZ));
+ clocks = (u32)div;
+
+ /* sanity check */
+ if (period < 1000 || clocks < 200)
+ return -EINVAL;
+
+ __get_cpu_var(hlrt_local_apic_mode) = mode;
+ hlrt_move_timers(smp_processor_id(),
+  first_cpu(cpu_possible_map));
+
+ /* initialize the wait queue */
+ init_waitqueue_head(&__get_cpu_var(hlrt_apic_timer_wait_queue));
+
+ /* write to the APIC registers */
+ ver = GET_APIC_VERSION(apic_read(APIC_LVR));
+ lvtt_value = APIC_LVT_TIMER_PERIODIC | HLRT_LOCAL_TIMER_VECTOR;
+ if (!APIC_INTEGRATED(ver))
+ lvtt_value |= SET_APIC_TIMER_BASE(APIC_TIMER_BASE_DIV);
+ apic_write_around(APIC_LVTT, lvtt_value);
+
+ tmp_value = apic_read(APIC_TDCR);
+ apic_write_around(APIC_TDCR,
+   (tmp_value & ~(APIC_TDR_DIV_1 |
+  APIC_TDR_DIV_TMBASE))
+   | APIC_TDR_DIV_1);
+
+ if (n_cpus > 1) {
+ /* achieve phase synchronisation across n_cpus timers */
+ spin_lock(&sync_cpu_lock);
+ if (sync_cpu_counter == 0) {
+ sync_cpu_counter = n_cpus;
+ }
+ spin_unlock(&sync_cpu_lock);
+
+ __asm__ __volatile__(
+                                 "lock; decl %0\n\t"
+ "jz 2f\n"
+ "1:\n\t"
+ "cmpl $0,%0\n\t"
+ "jnz 1b\n"
+ "2:\n"
+ :"=m" (sync_cpu_counter)
+ :"m" (sync_cpu_counter));
+ }
+
+ apic_write_around(APIC_TMICT, clocks);
+
+ return 0;
+
+ default:
+ return -EINVAL;
+ }
+}
+
+
+int sys_hlrt_wait_for_apic_timer(void)
+{
+ if (__get_cpu_var(hlrt_local_apic_mode) !=
+     HLRT_LAPIC_TIMER_MODE_PERIODIC)
+ return -EINVAL;
+
+ interruptible_sleep_on(&__get_cpu_var(hlrt_apic_timer_wait_queue));
+ return 0;
+}
+
+
+int sys_hlrt_request_irq(int irq)
+{
+ int retval;
+ cpumask_t affinity, reserved_cpus, result;
+
+ spin_lock(&hlrt_irqs_lock);
+ affinity = hlrt_get_irq_affinity(irq);
+ reserved_cpus = hlrt_cpus_allowed;
+ cpus_complement(reserved_cpus);
+ cpus_and(result, affinity, reserved_cpus);
+ if (cpus_empty(result))
+ retval = hlrt_set_irq_affinity(irq, current->cpus_allowed);
+ else
+ retval = -EBUSY;
+ spin_unlock(&hlrt_irqs_lock);
+
+ return retval;
+}
+
+
+int sys_hlrt_release_irq(int irq)
+{
+ int retval = 0;
+ cpumask_t affinity, reserved_cpus, result;
+
+ spin_lock(&hlrt_irqs_lock);
+ affinity = hlrt_get_irq_affinity(irq);
+ reserved_cpus = hlrt_cpus_allowed;
+ cpus_complement(reserved_cpus);
+ cpus_and(result, affinity, reserved_cpus);
+ cpu_clear(smp_processor_id(), result);
+ if (cpus_empty(result)) {
+ affinity = hlrt_cpus_allowed;
+ retval = hlrt_set_irq_affinity(irq, affinity);
+ }
+ spin_unlock(&hlrt_irqs_lock);
+
+ return retval;
+}
+
+
+unsigned int sys_hlrt_get_version_info(void)
+{
+ return HLRT_ABI_REVISION;
+}
diff -uNr linux-2.6.4/arch/i386/kernel/hlrt_entry.S linux-2.6.4-hlrt/arch/i386/kernel/hlrt_entry.S
--- linux-2.6.4/arch/i386/kernel/hlrt_entry.S 1970-01-01 01:00:00.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/kernel/hlrt_entry.S 2004-12-16 16:26:55.000000000 +0100
@@ -0,0 +1,226 @@
+/*
+ *  linux/arch/i386/hlrt_entry.S
+ *
+ *  Copyright (C) 1991, 1992  Linus Torvalds
+ *  Copyright (C) 2003  Christof Efkemann, Kai Thomsen
+ *  Copyright (C) 2004  Christof Efkemann
+ */
+
+#include <linux/config.h>
+#include <linux/linkage.h>
+#include <asm/thread_info.h>
+#include <asm/errno.h>
+#include <asm/segment.h>
+#include <asm/smp.h>
+#include <asm/page.h>
+#include "irq_vectors.h"
+
+#define nr_hlrt_syscalls ((hlrt_syscall_table_size)/4)
+
+EBX = 0x00
+ECX = 0x04
+EDX = 0x08
+ESI = 0x0C
+EDI = 0x10
+EBP = 0x14
+EAX = 0x18
+DS = 0x1C
+ES = 0x20
```

```
+ORIG_EAX = 0x24
+EIP = 0x28
+CS = 0x2C
+EFLAGS = 0x30
+OLDESP = 0x34
+OLDSS = 0x38
+
+CF_MASK = 0x00000001
+TF_MASK = 0x00000100
+IF_MASK = 0x00000200
+DF_MASK = 0x00000400
+NT_MASK = 0x00004000
+VM_MASK = 0x00020000
+
+/*
+ * ESP0 is at offset 4. 0x200 is the size of the TSS, and
+ * also thus the top-of-stack pointer offset of SYSENTER_ESP
+ */
+TSS_ESP0_OFFSET = (4 - 0x200)
+
+#ifdef CONFIG_PREEMPT
+#define preempt_stop cli
+#else
+#define preempt_stop
+#define resume_kernel restore_all
+#endif
+
+#define SAVE_ALL \
+ cld; \
+ pushl %es; \
+ pushl %ds; \
+ pushl %eax; \
+ pushl %ebp; \
+ pushl %edi; \
+ pushl %esi; \
+ pushl %edx; \
+ pushl %ecx; \
+ pushl %ebx; \
+ movl $(__USER_DS), %edx; \
+ movl %edx, %ds; \
+ movl %edx, %es;
+
+#define RESTORE_INT_REGS \
+ popl %ebx; \
+ popl %ecx; \
+ popl %edx; \
+ popl %esi; \
+ popl %edi; \
+ popl %ebp; \
+ popl %eax
+
+#define RESTORE_REGS \
+ RESTORE_INT_REGS; \
+1: popl %ds; \
+2: popl %es; \
+.section .fixup,"ax"; \
+3: movl $0,(%esp); \
+ jmp 1b; \
+4: movl $0,(%esp); \
+ jmp 2b; \
+.previous; \
+.section __ex_table,"a";\
+ .align 4; \
+ .long 1b,3b; \
+ .long 2b,4b; \
+.previous
+
+
+#define RESTORE_ALL \
+ RESTORE_REGS \
+ addl $4, %esp; \
+1: iret; \
+.section .fixup,"ax";   \
+2: sti; \
+ movl $(__USER_DS), %edx; \
+ movl %edx, %ds; \
+ movl %edx, %es; \
+ pushl $11; \
+ call do_exit; \
+.previous; \
+.section __ex_table,"a";\
+ .align 4; \
+ .long 1b,2b; \
+.previous
+
+
+
+ # system call handler stub
+ENTRY(hlrt_system_call)
+ pushl %eax # save orig_eax
+ SAVE_ALL
+ GET_THREAD_INFO(%ebp)
+ cmpl $(nr_hlrt_syscalls), %eax
+ jae syscall_badsys
+syscall_call:
+ call *hlrt_sys_call_table(,%eax,4)
+ movl %eax,EAX(%esp) # store the return value
+#ifdef CONFIG_HLRT
+ pushl %eax
+ call hlrt_quiescent_state
+ popl %eax
+#endif
+syscall_exit:
+ cli # make sure we don't miss an interrupt
+ # setting need_resched or sigpending
+ # between sampling and the iret
+ movl TI_FLAGS(%ebp), %ecx
+ testw $_TIF_ALLWORK_MASK, %cx # current->work
+ jne syscall_exit_work
+restore_all:
+ RESTORE_ALL
+
+ # perform work that needs to be done immediately before resumption
+ ALIGN
+work_pending:
+ testb $_TIF_NEED_RESCHED, %cl
+ jz work_notifysig
+work_resched:
+ call schedule
+ cli # make sure we don't miss an interrupt
+ # setting need_resched or sigpending
+ # between sampling and the iret
+ movl TI_FLAGS(%ebp), %ecx
+ andl $_TIF_WORK_MASK, %ecx # is there any work to be done other
+ # than syscall tracing?
+ jz restore_all
+ testb $_TIF_NEED_RESCHED, %cl
+ jnz work_resched
+
+work_notifysig: # deal with pending signals and
+ # notify-resume requests
+ testl $VM_MASK, EFLAGS(%esp)
+ movl %esp, %eax
+ jne work_notifysig_v86 # returning to kernel-space or
+ # vm86-space
+ xorl %edx, %edx
+ call do_notify_resume
+ jmp restore_all
+
+ ALIGN
+work_notifysig_v86:
+ pushl %ecx
+ call save_v86_state
+ popl %ecx
+ movl %eax, %esp
+ xorl %edx, %edx
+ call do_notify_resume
+ jmp restore_all
+
+ # perform syscall exit tracing
+ ALIGN
+syscall_trace_entry:
+ movl $-ENOSYS,EAX(%esp)
+ movl %esp, %eax
+ xorl %edx,%edx
+ call do_syscall_trace
+ movl ORIG_EAX(%esp), %eax
+ cmpl $(nr_hlrt_syscalls), %eax
+ jnae syscall_call
+ jmp syscall_exit
+
+ # perform syscall exit tracing
+ ALIGN
+syscall_exit_work:
+ testb $_TIF_SYSCALL_TRACE, %cl
+ jz work_pending
+ sti # could let do_syscall_trace() call
+ # schedule() instead
+ movl %esp, %eax
+ movl $1, %edx
+ call do_syscall_trace
+ jmp resume_userspace
+
+ ALIGN
+syscall_fault:
+ pushl %eax # save orig_eax
+ SAVE_ALL
+ GET_THREAD_INFO(%ebp)
+ movl $-EFAULT,EAX(%esp)
+ jmp resume_userspace
+
+ ALIGN
+syscall_badsys:
+ movl $-ENOSYS,EAX(%esp)
+ jmp resume_userspace
+
+
+.data
+ENTRY(hlrt_sys_call_table)
+   .long sys_hlrt_request_irq           /* 0 */
+   .long sys_hlrt_release_irq
+   .long sys_hlrt_set_local_apic_timer
+   .long sys_hlrt_wait_for_apic_timer
+ .long sys_ni_syscall
+ .long sys_ni_syscall                 /* 5 */
+   .long sys_hlrt_get_version_info      /* 6: don't move! */
+
+hlrt_syscall_table_size=(.-hlrt_sys_call_table)
diff -uNr linux-2.6.4/arch/i386/kernel/i8259.c linux-2.6.4-hlrt/arch/i386/kernel/i8259.c
--- linux-2.6.4/arch/i386/kernel/i8259.c 2004-03-11 03:55:44.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/kernel/i8259.c 2004-12-16 16:26:55.000000000 +0100
@@ -423,6 +423,10 @@
  int vector = FIRST_EXTERNAL_VECTOR + i;
  if (i >= NR_IRQS)
  break;
+#ifdef CONFIG_HLRT
+ if (vector == HLRT_SYSCALL_VECTOR)
+ continue;
+#endif
  if (vector != SYSCALL_VECTOR)
  set_intr_gate(vector, interrupt[i]);
  }
diff -uNr linux-2.6.4/arch/i386/kernel/io_apic.c linux-2.6.4-hlrt/arch/i386/kernel/io_apic.c
--- linux-2.6.4/arch/i386/kernel/io_apic.c 2004-03-11 03:55:25.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/kernel/io_apic.c 2004-12-16 16:26:55.000000000 +0100
@@ -1161,6 +1161,11 @@
```

```
  if (current_vector == SYSCALL_VECTOR)
  goto next;

+#ifdef CONFIG_HLRT
+ if (current_vector == HLRT_SYSCALL_VECTOR)
+ goto next;
+#endif
+
  if (current_vector >= FIRST_SYSTEM_VECTOR) {
  offset = (offset + 1) & 7;
  current_vector = FIRST_DEVICE_VECTOR + offset;
diff -uNr linux-2.6.4/arch/i386/kernel/Makefile linux-2.6.4-hlrt/arch/i386/kernel/Makefile
--- linux-2.6.4/arch/i386/kernel/Makefile 2004-03-11 03:55:23.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/kernel/Makefile 2004-12-16 16:26:55.000000000 +0100
@@ -32,6 +32,7 @@
 obj-$(CONFIG_HPET_TIMER)  += time_hpet.o
 obj-$(CONFIG_EFI)  += efi.o efi_stub.o
 obj-$(CONFIG_EARLY_PRINTK) += early_printk.o
+obj-$(CONFIG_HLRT) += hlrt.o hlrt_entry.o

 EXTRA_AFLAGS   := -traditional

diff -uNr linux-2.6.4/arch/i386/kernel/traps.c linux-2.6.4-hlrt/arch/i386/kernel/traps.c
--- linux-2.6.4/arch/i386/kernel/traps.c 2004-03-11 03:55:23.000000000 +0100
+++ linux-2.6.4-hlrt/arch/i386/kernel/traps.c 2004-12-16 16:26:55.000000000 +0100
@@ -58,6 +58,10 @@
 asmlinkage void lcall7(void);
 asmlinkage void lcall27(void);

+#ifdef CONFIG_HLRT
+asmlinkage int hlrt_system_call(void);
+#endif
+
 struct desc_struct default_ldt[] = { { 0, 0 }, { 0, 0 }, { 0, 0 },
  { 0, 0 }, { 0, 0 } };

@@ -898,6 +902,10 @@
  set_call_gate(&default_ldt[0],lcall7);
  set_call_gate(&default_ldt[4],lcall27);

+#ifdef CONFIG_HLRT
+ set_system_gate(HLRT_SYSCALL_VECTOR, &hlrt_system_call);
+#endif
+
  /*
   * Should be a barrier for any external CPU state.
   */
diff -uNr linux-2.6.4/drivers/pci/msi.c linux-2.6.4-hlrt/drivers/pci/msi.c
--- linux-2.6.4/drivers/pci/msi.c 2004-03-11 03:55:44.000000000 +0100
+++ linux-2.6.4-hlrt/drivers/pci/msi.c 2004-12-16 16:26:55.000000000 +0100
@@ -305,6 +305,11 @@
  if (current_vector == SYSCALL_VECTOR)
  goto next;
```

```
+#ifdef CONFIG_HLRT
+ if (current_vector == HLRT_SYSCALL_VECTOR)
+ goto next;
+#endif
+
  if (current_vector > FIRST_SYSTEM_VECTOR) {
  offset++;
  current_vector = FIRST_DEVICE_VECTOR + offset;
diff -uNr linux-2.6.4/drivers/scsi/scsi.c linux-2.6.4-hlrt/drivers/scsi/scsi.c
--- linux-2.6.4/drivers/scsi/scsi.c 2004-03-11 03:55:27.000000000 +0100
+++ linux-2.6.4-hlrt/drivers/scsi/scsi.c 2004-12-16 16:26:55.000000000 +0100
@@ -54,6 +54,10 @@
 #include <linux/kmod.h>
 #include <linux/interrupt.h>

+#ifdef CONFIG_HLRT
+#include <asm/hlrt_util.h>
+#endif
+
 #include <scsi/scsi_host.h>
 #include "scsi.h"

@@ -671,6 +675,9 @@
  * Per-CPU I/O completion queue.
  */
 static DEFINE_PER_CPU(struct list_head, scsi_done_q);
+#ifdef CONFIG_HLRT
+static DEFINE_PER_CPU(spinlock_t, scsi_done_q_lock);
+#endif

 /**
  * scsi_done - Enqueue the finished SCSI command into the done queue.
@@ -688,6 +695,9 @@
 void scsi_done(struct scsi_cmnd *cmd)
 {
  unsigned long flags;
+#ifdef CONFIG_HLRT
+ int cpu;
+#endif

  /*
   * We don’t have to worry about this one timing out any more.
@@ -714,8 +724,19 @@
   * and need no spinlock.
   */
  local_irq_save(flags);
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ cpu = first_cpu(cpu_possible_map);
+ else
+ cpu = smp_processor_id();
+ spin_lock(&per_cpu(scsi_done_q_lock, cpu));
+ list_add_tail(&cmd->eh_entry, &per_cpu(scsi_done_q, cpu));
+ spin_unlock(&per_cpu(scsi_done_q_lock, cpu));
+ raise_softirq_irqoff_cpu(SCSI_SOFTIRQ, cpu);
+#else
  list_add_tail(&cmd->eh_entry, &__get_cpu_var(scsi_done_q));
  raise_softirq_irqoff(SCSI_SOFTIRQ);
+#endif
  local_irq_restore(flags);
 }

@@ -734,7 +755,13 @@
  LIST_HEAD(local_q);

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&__get_cpu_var(scsi_done_q_lock));
+#endif
  list_splice_init(&__get_cpu_var(scsi_done_q), &local_q);
+#ifdef CONFIG_HLRT
+ spin_unlock(&__get_cpu_var(scsi_done_q_lock));
+#endif
  local_irq_enable();

  while (!list_empty(&local_q)) {
@@ -1161,6 +1188,10 @@

  for (i = 0; i < NR_CPUS; i++)
  INIT_LIST_HEAD(&per_cpu(scsi_done_q, i));
+#ifdef CONFIG_HLRT
+ for (i = 0; i < NR_CPUS; i++)
+ spin_lock_init(&per_cpu(scsi_done_q_lock, i));
+#endif

  devfs_mk_dir("scsi");
  open_softirq(SCSI_SOFTIRQ, scsi_softirq, NULL);
diff -uNr linux-2.6.4/include/asm-i386/hlrt.h linux-2.6.4-hlrt/include/asm-i386/hlrt.h
--- linux-2.6.4/include/asm-i386/hlrt.h 1970-01-01 01:00:00.000000000 +0100
+++ linux-2.6.4-hlrt/include/asm-i386/hlrt.h 2004-12-16 16:26:55.000000000 +0100
@@ -0,0 +1,101 @@
+#ifndef _LINUX_HLRT_H
+#define _LINUX_HLRT_H
+
+#include <linux/sched.h>
+
+
+/*
+ * HaRTLinC CPU reservation -- administrative and helper functions
+ *
+ * Authors:
+ * Kai Thomsen <kthomsen@tzi.de>
+ * Christof Efkemann <chref@tzi.de>
+ *
+ * based on the ‘LinuxRT’ patch by
+ * Klaas-Henning Zweck
+ * Mark Hapke
+ *
+ * ported to Linux 2.6 by Christof Efkemann
+ *
+ * This header file contains function prototypes and variable declarations
+ * used to implement the hard-realtime extension of the Linux kernel.
+ *
+ */
+
+
+/*
+ * bit mask representing the reservation of CPUs for hard-realtime processes
+ * 1 == CPU is available for non-RT processes
+ * 0 == CPU is reserved for an RT process
+ */
+extern cpumask_t hlrt_cpus_allowed;
+
+
+asmlinkage void hlrt_apic_timer_interrupt(void);
+
+
+int hlrt_set_hlrt_policy(struct task_struct *p);
+
+
+/**
+ * Release this reserved CPU for use by non-RT processes and re-enable
+ * routing of interrupt requests to this CPU.
+ *
+ * @param p pointer to task_struct of the hard real-time process
+ */
+
+int hlrt_unset_hlrt_policy(struct task_struct *p);
+
+
+/**
+ * Alter the routing of interrupt requests so that interrupt ‘irq’ is
+ * routed exclusively to the current (reserved) CPU.
+ *
+ * @param irq requested IRQ number
+ * @return 0 on success
+ */
+
+int sys_hlrt_request_irq(int irq);
+
+
+/**
+ * Alter the routing of interrupt requests so that interrupt ‘irq’ is
+ * routed to all non-reserved CPUs again.
+ *
+ * @param irq IRQ number to release
+ * @return 0 on success
+ */
+int sys_hlrt_release_irq(int irq);
+
+
+/**
```

```
+ * Enable or disable the Local APIC Timer of the CPU the calling
+ * process runs on.
+ *
+ * @param mode 1 to disable, 0 to enable the normal timer as
+    used by the kernel, 2 to activate a custom periodic timer
+   @param period only relevant if mode is 2: defines the timer
+    period in nanoseconds
+   @param n_cpus only relevant if mode is 2: number of CPUs of
+    which to achieve phase synchronisation the Local APIC Timer
+    (all of these CPUs must make this call with the same
+    parameters).  If n_cpus is smaller than 2, the CPU's Local
+    APIC timer is started alone.
+ */
+#define HLRT_LAPIC_TIMER_MODE_NORMAL    0
+#define HLRT_LAPIC_TIMER_MODE_OFF       1
+#define HLRT_LAPIC_TIMER_MODE_PERIODIC  2
+
+int sys_hlrt_set_local_apic_timer(int mode, unsigned int period,
+   unsigned int n_cpus);
+
+
+/**
+ * Return the ABI revision number of the HLRT syscall interface.
+ *
+ * @return numeric ABI version
+ */
+
+unsigned int sys_hlrt_get_version_info(void);
+
+#endif /* _LINUX_HLRT_H */
diff -uNr linux-2.6.4/include/asm-i386/hlrt_unistd.h linux-2.6.4-hlrt/include/asm-i386/hlrt_unist
--- linux-2.6.4/include/asm-i386/hlrt_unistd.h 1970-01-01 01:00:00.000000000 +0100
+++ linux-2.6.4-hlrt/include/asm-i386/hlrt_unistd.h 2004-12-16 16:26:55.000000000 +0100
@@ -0,0 +1,103 @@
+#ifndef _ASM_I386_HLRT_UNISTD_H_
+#define _ASM_I386_HLRT_UNISTD_H_
+
+/*
+ * This file contains the HLRT system call numbers.
+ */
+
+#define __NR_hlrt_request_irq 0
+#define __NR_hlrt_release_irq 1
+#define __NR_hlrt_set_local_apic_timer 2
+#define __NR_hlrt_wait_for_apic_timer 3
+#define __NR_hlrt_get_version_info 6  /* don't move! */
+
+/* user-visible error numbers are in the range -1 - -124: see <asm-i386/errno.h> */
+
+#define __hlrt_syscall_return(type, res) \
+do { \
+ if ((unsigned long)(res) >= (unsigned long)(-125)) { \
+ errno = -(res); \
+ res = -1; \
+ } \
+ return (type) (res); \
+} while (0)
+
+/* XXX - _foo needs to be __foo, while __NR_bar could be _NR_bar. */
+#define _hlrt_syscall0(type,name) \
+type name(void) \
+{ \
+long __res; \
+__asm__ volatile ("int $0x83" \
+ : "=a" (__res) \
+ : "0" (__NR_##name)); \
+__hlrt_syscall_return(type,__res); \
+}
+
+#define _hlrt_syscall1(type,name,type1,arg1) \
+type name(type1 arg1) \
+{ \
+long __res; \
+__asm__ volatile ("int $0x83" \
+ : "=a" (__res) \
+ : "0" (__NR_##name),"b" ((long)(arg1))); \
+__hlrt_syscall_return(type,__res); \
+}
+
+#define _hlrt_syscall2(type,name,type1,arg1,type2,arg2) \
+type name(type1 arg1,type2 arg2) \
+{ \
+long __res; \
+__asm__ volatile ("int $0x83" \
+ : "=a" (__res) \
+ : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2))); \
+__hlrt_syscall_return(type,__res); \
+}
+
+#define _hlrt_syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
+type name(type1 arg1,type2 arg2,type3 arg3) \
+{ \
+long __res; \
+__asm__ volatile ("int $0x83" \
+ : "=a" (__res) \
+ : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
+   "d" ((long)(arg3))); \
+__hlrt_syscall_return(type,__res); \
+}
+
+#define _hlrt_syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
+type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
+{ \
+long __res; \
+__asm__ volatile ("int $0x83" \
+ : "=a" (__res) \
+ : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
+   "d" ((long)(arg3)),"S" ((long)(arg4))); \
+__hlrt_syscall_return(type,__res); \
+}
+
+#define _hlrt_syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
+   type5,arg5) \
+type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5) \
+{ \
+long __res; \
+__asm__ volatile ("int $0x83" \
+ : "=a" (__res) \
+ : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
+   "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5))); \
+__hlrt_syscall_return(type,__res); \
+}
+
+#define _hlrt_syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
+   type5,arg5,type6,arg6) \
+type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5,type6 arg6) \
+{ \
+long __res; \
+__asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int $0x83 ; pop %%ebp" \
+ : "=a" (__res) \
+ : "i" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
+   "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)), \
+   "0" ((long)(arg6))); \
+__hlrt_syscall_return(type,__res); \
+}
+
+#endif /* _ASM_I386_HLRT_UNISTD_H_ */
diff -uNr linux-2.6.4/include/asm-i386/hlrt_util.h linux-2.6.4-hlrt/include/asm-i386/hlrt_util.h
--- linux-2.6.4/include/asm-i386/hlrt_util.h 1970-01-01 01:00:00.000000000 +0100
+++ linux-2.6.4-hlrt/include/asm-i386/hlrt_util.h 2004-12-16 16:26:55.000000000 +0100
@@ -0,0 +1,28 @@
+#ifndef _LINUX_HLRT_UTIL_H
+#define _LINUX_HLRT_UTIL_H
+
+
+/*
+ * HaRTLinC CPU reservation -- exported helper functions
+ *
+ * Author:
+ * Christof Efkemann <chref@tzi.de>
+ *
+ * ported to Linux 2.6 by Christof Efkemann
+ *
+ * This header file contains function prototypes of exported functions
+ * that are needed in device drivers etc.
+ *
+ */
+
+
+/**
+ * Find out whether the CPU on which this function is called is reserved.
+ *
+ * @return 0 if it is not reserved, or a value different from 0 if it
+ *         is reserved
+ */
+
+int hlrt_cpu_reserved(void);
+
+#endif /* _LINUX_HLRT_UTIL_H */
diff -uNr linux-2.6.4/include/asm-i386/mach-default/entry_arch.h linux-2.6.4-hlrt/include/asm-i386/mach-def
--- linux-2.6.4/include/asm-i386/mach-default/entry_arch.h 2004-03-11 03:55:20.000000000 +0100
+++ linux-2.6.4-hlrt/include/asm-i386/mach-default/entry_arch.h 2004-12-16 16:26:55.000000000 +0100
@@ -31,4 +31,8 @@
 BUILD_INTERRUPT(thermal_interrupt,THERMAL_APIC_VECTOR)
 #endif

+#ifdef CONFIG_HLRT
+BUILD_INTERRUPT(hlrt_apic_timer_interrupt,HLRT_LOCAL_TIMER_VECTOR)
+#endif
+
 #endif
diff -uNr linux-2.6.4/include/asm-i386/mach-default/irq_vectors.h linux-2.6.4-hlrt/include/asm-i386/mach-de
--- linux-2.6.4/include/asm-i386/mach-default/irq_vectors.h 2004-03-11 03:55:24.000000000 +0100
+++ linux-2.6.4-hlrt/include/asm-i386/mach-default/irq_vectors.h 2004-12-16 16:26:55.000000000 +0100
@@ -30,6 +30,10 @@

 #define SYSCALL_VECTOR 0x80

+#ifdef CONFIG_HLRT
+#define HLRT_SYSCALL_VECTOR     0x83
+#endif
+
 /*
  * Vectors 0x20-0x2f are used for ISA interrupts.
  */
@@ -49,6 +53,10 @@
 #define RESCHEDULE_VECTOR 0xfc
 #define CALL_FUNCTION_VECTOR 0xfb

+#ifdef CONFIG_HLRT
+#define HLRT_LOCAL_TIMER_VECTOR 0xf1
+#endif
+
 #define THERMAL_APIC_VECTOR 0xf0
 /*
  * Local APIC timer IRQ vector is on a different priority level,
diff -uNr linux-2.6.4/include/linux/interrupt.h linux-2.6.4-hlrt/include/linux/interrupt.h
--- linux-2.6.4/include/linux/interrupt.h 2004-03-11 03:55:28.000000000 +0100
+++ linux-2.6.4-hlrt/include/linux/interrupt.h 2004-12-16 16:26:55.000000000 +0100
@@ -95,7 +95,22 @@
 asmlinkage void do_softirq(void);
 extern void open_softirq(int nr, void (*action)(struct softirq_action*), void *data);
 extern void softirq_init(void);
+#ifdef CONFIG_HLRT
+extern spinlock_t hlrt_softirq_pending_lock[NR_CPUS];
+/*
```

```
+ * here we need additional locking when raising softirqs
+ * across different CPUs
+ */
+#define __raise_softirq_irqoff_cpu(nr, cpu) do { \
+ spin_lock(&hlrt_softirq_pending_lock[cpu]); \
+ softirq_pending(cpu) |= 1UL << (nr); \
+ spin_unlock(&hlrt_softirq_pending_lock[cpu]); \
+ } while (0)
+#define __raise_softirq_irqoff(nr) __raise_softirq_irqoff_cpu((nr), smp_processor_id())
+extern void FASTCALL(raise_softirq_irqoff_cpu(unsigned int nr, int cpu));
+#else
 #define __raise_softirq_irqoff(nr) do { local_softirq_pending() |= 1UL << (nr); } while (0)
+#endif
 extern void FASTCALL(raise_softirq_irqoff(unsigned int nr));
 extern void FASTCALL(raise_softirq(unsigned int nr));

diff -uNr linux-2.6.4/include/linux/netdevice.h linux-2.6.4-hlrt/include/linux/netdevice.h
--- linux-2.6.4/include/linux/netdevice.h 2004-03-11 03:55:44.000000000 +0100
+++ linux-2.6.4-hlrt/include/linux/netdevice.h 2004-12-16 17:45:39.000000000 +0100
@@ -148,6 +148,10 @@

 #include <linux/cache.h>
 #include <linux/skbuff.h>
+#ifdef CONFIG_HLRT
+#include <linux/cpumask.h>
+#include <asm/hlrt_util.h>
+#endif

 struct neighbour;
 struct neigh_parms;
@@ -562,6 +566,12 @@
  struct list_head poll_list;
  struct net_device *output_queue;
  struct sk_buff *completion_queue;
+#ifdef CONFIG_HLRT
+ spinlock_t              input_pkt_queue_lock;
+ spinlock_t poll_list_lock;
+ spinlock_t output_queue_lock;
+ spinlock_t completion_queue_lock;
+#endif

  struct net_device backlog_dev; /* Sorry. 8) */
 };
@@ -575,12 +585,29 @@
  if (!test_and_set_bit(__LINK_STATE_SCHED, &dev->state)) {
  unsigned long flags;
  struct softnet_data *sd;
+#ifdef CONFIG_HLRT
+ int cpu;
+#endif

  local_irq_save(flags);
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ cpu = first_cpu(cpu_possible_map);
+ else
+ cpu = smp_processor_id();
+ sd = &per_cpu(softnet_data, cpu);
+ spin_lock(&sd->output_queue_lock);
+#else
  sd = &__get_cpu_var(softnet_data);
+#endif
  dev->next_sched = sd->output_queue;
  sd->output_queue = dev;
+#ifdef CONFIG_HLRT
+ spin_unlock(&sd->output_queue_lock);
+ raise_softirq_irqoff_cpu(NET_TX_SOFTIRQ, cpu);
+#else
  raise_softirq_irqoff(NET_TX_SOFTIRQ);
+#endif
  local_irq_restore(flags);
  }
 }
@@ -626,12 +653,29 @@
  if (atomic_dec_and_test(&skb->users)) {
  struct softnet_data *sd;
  unsigned long flags;
+#ifdef CONFIG_HLRT
+ int cpu;
+#endif

  local_irq_save(flags);
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ cpu = first_cpu(cpu_possible_map);
+ else
+ cpu = smp_processor_id();
+ sd = &per_cpu(softnet_data, cpu);
+ spin_lock(&sd->completion_queue_lock);
+#else
  sd = &__get_cpu_var(softnet_data);
+#endif
  skb->next = sd->completion_queue;
  sd->completion_queue = skb;
+#ifdef CONFIG_HLRT
+ spin_unlock(&sd->completion_queue_lock);
+ raise_softirq_irqoff_cpu(NET_TX_SOFTIRQ, cpu);
+#else
  raise_softirq_irqoff(NET_TX_SOFTIRQ);
+#endif
  local_irq_restore(flags);
  }
 }
@@ -789,15 +833,34 @@
 static inline void __netif_rx_schedule(struct net_device *dev)
 {
  unsigned long flags;
+#ifdef CONFIG_HLRT
```

```
+ struct softnet_data *sd;
+ int cpu;
+#endif

  local_irq_save(flags);
  dev_hold(dev);
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ cpu = first_cpu(cpu_possible_map);
+ else
+ cpu = smp_processor_id();
+ sd = &per_cpu(softnet_data, cpu);
+ spin_lock(&sd->poll_list_lock);
+ list_add_tail(&dev->poll_list, &sd->poll_list);
+ spin_unlock(&sd->poll_list_lock);
+#else
  list_add_tail(&dev->poll_list, &__get_cpu_var(softnet_data).poll_list);
+#endif
  if (dev->quota < 0)
  dev->quota += dev->weight;
  else
  dev->quota = dev->weight;
+#ifdef CONFIG_HLRT
+ __raise_softirq_irqoff_cpu(NET_RX_SOFTIRQ, cpu);
+#else
  __raise_softirq_irqoff(NET_RX_SOFTIRQ);
+#endif
  local_irq_restore(flags);
 }

@@ -816,12 +879,28 @@
 {
  if (netif_rx_schedule_prep(dev)) {
  unsigned long flags;
+#ifdef CONFIG_HLRT
+ struct softnet_data *sd;
+ int cpu;
+#endif

  dev->quota += undo;

  local_irq_save(flags);
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ cpu = first_cpu(cpu_possible_map);
+ else
+ cpu = smp_processor_id();
+ sd = &per_cpu(softnet_data, cpu);
+ spin_lock(&sd->poll_list_lock);
+ list_add_tail(&dev->poll_list, &sd->poll_list);
+ spin_unlock(&sd->poll_list_lock);
+ __raise_softirq_irqoff_cpu(NET_RX_SOFTIRQ, cpu);
+#else
  list_add_tail(&dev->poll_list, &__get_cpu_var(softnet_data).poll_list);
  __raise_softirq_irqoff(NET_RX_SOFTIRQ);
+#endif
  local_irq_restore(flags);
  return 1;
  }
@@ -839,7 +918,13 @@

  local_irq_save(flags);
  BUG_ON(!test_bit(__LINK_STATE_RX_SCHED, &dev->state));
+#ifdef CONFIG_HLRT
+ spin_lock(&__get_cpu_var(softnet_data).poll_list_lock);
+#endif
  list_del(&dev->poll_list);
+#ifdef CONFIG_HLRT
+ spin_unlock(&__get_cpu_var(softnet_data).poll_list_lock);
+#endif
  smp_mb__before_clear_bit();
  clear_bit(__LINK_STATE_RX_SCHED, &dev->state);
  local_irq_restore(flags);
@@ -865,7 +950,13 @@
 static inline void __netif_rx_complete(struct net_device *dev)
 {
  BUG_ON(!test_bit(__LINK_STATE_RX_SCHED, &dev->state));
+#ifdef CONFIG_HLRT
+ spin_lock(&__get_cpu_var(softnet_data).poll_list_lock);
+#endif
  list_del(&dev->poll_list);
+#ifdef CONFIG_HLRT
+ spin_unlock(&__get_cpu_var(softnet_data).poll_list_lock);
+#endif
  smp_mb__before_clear_bit();
  clear_bit(__LINK_STATE_RX_SCHED, &dev->state);
 }
diff -uNr linux-2.6.4/include/linux/rcupdate.h linux-2.6.4-hlrt/include/linux/rcupdate.h
--- linux-2.6.4/include/linux/rcupdate.h 2004-03-11 03:55:24.000000000 +0100
+++ linux-2.6.4-hlrt/include/linux/rcupdate.h 2004-12-16 16:26:55.000000000 +0100
@@ -96,6 +96,9 @@
        long           batch;           /* Batch # for current RCU batch */
        struct list_head  nxtlist;
        struct list_head  curlist;
+#ifdef CONFIG_HLRT
+ spinlock_t nxtlist_lock;
+#endif
 };

 DECLARE_PER_CPU(struct rcu_data, rcu_data);
@@ -106,6 +109,9 @@
 #define RCU_batch(cpu)  (per_cpu(rcu_data, (cpu)).batch)
 #define RCU_nxtlist(cpu)  (per_cpu(rcu_data, (cpu)).nxtlist)
 #define RCU_curlist(cpu)  (per_cpu(rcu_data, (cpu)).curlist)
+#ifdef CONFIG_HLRT
+#define RCU_nxtlist_lock(cpu)  (per_cpu(rcu_data, (cpu)).nxtlist_lock)
+#endif
```

```
 #define RCU_QSCTR_INVALID 0

diff -uNr linux-2.6.4/include/linux/sched.h linux-2.6.4-hlrt/include/linux/sched.h
--- linux-2.6.4/include/linux/sched.h 2004-03-11 03:55:22.000000000 +0100
+++ linux-2.6.4-hlrt/include/linux/sched.h 2004-12-16 16:26:55.000000000 +0100
@@ -127,6 +127,12 @@
 #define SCHED_FIFO 1
 #define SCHED_RR 2

+#ifdef CONFIG_HLRT
+#define SCHED_HLRT              3  /* additional scheduling policy for
+       hard-realtime processes */
+#endif
+
+
 struct sched_param {
  int sched_priority;
 };
diff -uNr linux-2.6.4/include/linux/timer.h linux-2.6.4-hlrt/include/linux/timer.h
--- linux-2.6.4/include/linux/timer.h 2004-03-11 03:55:36.000000000 +0100
+++ linux-2.6.4-hlrt/include/linux/timer.h 2004-12-16 16:26:55.000000000 +0100
@@ -90,6 +90,10 @@
 # define del_timer_sync(t) del_timer(t)
 #endif

+#ifdef CONFIG_HLRT
+extern void hlrt_move_timers(int from_cpu, int to_cpu);
+#endif
+
 extern void init_timers(void);
 extern void run_local_timers(void);
 extern void it_real_fn(unsigned long);
diff -uNr linux-2.6.4/kernel/exit.c linux-2.6.4-hlrt/kernel/exit.c
--- linux-2.6.4/kernel/exit.c 2004-03-11 03:55:44.000000000 +0100
+++ linux-2.6.4-hlrt/kernel/exit.c 2004-12-16 16:26:55.000000000 +0100
@@ -27,6 +27,10 @@
 #include <asm/pgtable.h>
 #include <asm/mmu_context.h>

+#ifdef CONFIG_HLRT
+#include <asm/hlrt.h>
+#endif /* CONFIG_HLRT */
+
 extern void sem_exit (void);
 extern struct task_struct *child_reaper;

@@ -771,6 +775,13 @@
  }

  acct_process(code);
+#ifdef CONFIG_HLRT
+ if (tsk->policy == SCHED_HLRT) {
+ read_lock_irq(&tasklist_lock);
+ hlrt_unset_hlrt_policy(tsk);
+ read_unlock_irq(&tasklist_lock);
+ }
+#endif /* CONFIG_HLRT */
  __exit_mm(tsk);

  exit_sem(tsk);
diff -uNr linux-2.6.4/kernel/rcupdate.c linux-2.6.4-hlrt/kernel/rcupdate.c
--- linux-2.6.4/kernel/rcupdate.c 2004-03-11 03:55:21.000000000 +0100
+++ linux-2.6.4-hlrt/kernel/rcupdate.c 2004-12-16 16:26:55.000000000 +0100
@@ -45,6 +45,10 @@
 #include <linux/rcupdate.h>
 #include <linux/cpu.h>

+#ifdef CONFIG_HLRT
+#include <asm/hlrt_util.h>
+#endif
+
 /* Definition for rcupdate control block. */
 struct rcu_ctrlblk rcu_ctrlblk =
  { .mutex = SPIN_LOCK_UNLOCKED, .curbatch = 1,
@@ -74,8 +78,19 @@
  head->func = func;
  head->arg = arg;
  local_irq_save(flags);
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ cpu = first_cpu(cpu_possible_map);
+ else
+ cpu = smp_processor_id();
+ spin_lock(&RCU_nxtlist_lock(cpu));
+#else
  cpu = smp_processor_id();
+#endif
  list_add_tail(&head->list, &RCU_nxtlist(cpu));
+#ifdef CONFIG_HLRT
+ spin_unlock(&RCU_nxtlist_lock(cpu));
+#endif
  local_irq_restore(flags);
 }

@@ -153,6 +168,38 @@
  spin_unlock(&rcu_ctrlblk.mutex);
 }

+#ifdef CONFIG_HLRT
+/*
+ * This function is similar to rcu_check_quiescent_state,
+ * but here we omit checking the qsctr because we do not
+ * have the local timer interrupt that would increment it
+ * and we know that we are in a quiescent state right now.
+ */
+asmlinkage void hlrt_quiescent_state(void) {
+ if (unlikely(hlrt_cpu_reserved())) {
+ int cpu = smp_processor_id();
```

```
+
+ if (!cpu_isset(cpu, rcu_ctrlblk.rcu_cpu_mask))
+ return;
+
+ spin_lock(&rcu_ctrlblk.mutex);
+ if (!cpu_isset(cpu, rcu_ctrlblk.rcu_cpu_mask))
+ goto out_unlock;
+
+ cpu_clear(cpu, rcu_ctrlblk.rcu_cpu_mask);
+ RCU_last_qsctr(cpu) = RCU_QSCTR_INVALID;
+ if (!cpus_empty(rcu_ctrlblk.rcu_cpu_mask))
+ goto out_unlock;
+
+ rcu_ctrlblk.curbatch++;
+ rcu_start_batch(rcu_ctrlblk.maxbatch);
+
+out_unlock:
+ spin_unlock(&rcu_ctrlblk.mutex);
+ }
+}
+#endif
+

 /*
  * This does the RCU processing work from tasklet context.
@@ -169,9 +216,15 @@
  }

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&RCU_nxtlist_lock(cpu));
+#endif
  if (!list_empty(&RCU_nxtlist(cpu)) && list_empty(&RCU_curlist(cpu))) {
  list_splice(&RCU_nxtlist(cpu), &RCU_curlist(cpu));
  INIT_LIST_HEAD(&RCU_nxtlist(cpu));
+#ifdef CONFIG_HLRT
+ spin_unlock(&RCU_nxtlist_lock(cpu));
+#endif
  local_irq_enable();

  /*
@@ -182,6 +235,9 @@
  rcu_start_batch(RCU_batch(cpu));
  spin_unlock(&rcu_ctrlblk.mutex);
  } else {
+#ifdef CONFIG_HLRT
+ spin_unlock(&RCU_nxtlist_lock(cpu));
+#endif
  local_irq_enable();
  }
  rcu_check_quiescent_state();
@@ -195,7 +251,14 @@
      (idle_cpu(cpu) && !in_softirq() &&
  hardirq_count() <= (1 << HARDIRQ_SHIFT)))
  RCU_qsctr(cpu)++;
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ rcu_check_quiescent_state();
+ else
+ tasklet_schedule(&RCU_tasklet(cpu));
+#else
  tasklet_schedule(&RCU_tasklet(cpu));
+#endif
 }

 static void __devinit rcu_online_cpu(int cpu)
@@ -204,6 +267,9 @@
  tasklet_init(&RCU_tasklet(cpu), rcu_process_callbacks, 0UL);
  INIT_LIST_HEAD(&RCU_nxtlist(cpu));
  INIT_LIST_HEAD(&RCU_curlist(cpu));
+#ifdef CONFIG_HLRT
+ spin_lock_init(&RCU_nxtlist_lock(cpu));
+#endif
 }

 static int __devinit rcu_cpu_notify(struct notifier_block *self,
diff -uNr linux-2.6.4/kernel/sched.c linux-2.6.4-hlrt/kernel/sched.c
--- linux-2.6.4/kernel/sched.c 2004-03-11 03:55:43.000000000 +0100
+++ linux-2.6.4-hlrt/kernel/sched.c 2004-12-16 16:26:55.000000000 +0100
@@ -40,6 +40,11 @@
 #include <linux/percpu.h>
 #include <linux/kthread.h>

+#ifdef CONFIG_HLRT
+#include <asm/hlrt.h>
+#include <asm/hlrt_util.h>
+#endif /* CONFIG_HLRT */
+
 #ifdef CONFIG_NUMA
 #define cpu_to_node_mask(cpu) node_to_cpumask(cpu_to_node(cpu))
 #else
@@ -558,6 +563,16 @@
  runqueue_t *rq = task_rq(p);

  p->cpus_allowed = new_mask;
+
+#ifdef CONFIG_HLRT
+ if (p->policy != SCHED_HLRT) {
+ cpus_and(new_mask, new_mask, hlrt_cpus_allowed);
+ cpus_and(new_mask, new_mask, cpu_online_map);
+ if (cpus_empty(new_mask))
+ new_mask = cpumask_of_cpu(first_cpu(cpu_possible_map));
+ }
+#endif
+
  /*
   * Can the task run on the task's current CPU? If not then
   * migrate the thread off to a proper CPU.
```

```
@@ -570,7 +585,11 @@
   * it is sufficient to simply update the task's cpu field.
   */
  if (!p->array && !task_running(rq, p)) {
+#ifdef CONFIG_HLRT
+ set_task_cpu(p, any_online_cpu(new_mask));
+#else
  set_task_cpu(p, any_online_cpu(p->cpus_allowed));
+#endif
  return 0;
  }

@@ -671,6 +690,10 @@
   */
  if (unlikely(sync && !task_running(rq, p) &&
  (task_cpu(p) != smp_processor_id()) &&
+#ifdef CONFIG_HLRT
+ (!hlrt_cpu_reserved() ||
+  (p->policy == SCHED_HLRT)) &&
+#endif
  cpu_isset(smp_processor_id(),
  p->cpus_allowed))) {

@@ -678,6 +701,25 @@
  task_rq_unlock(rq, &flags);
  goto repeat_lock_task;
  }
+#ifdef CONFIG_HLRT
+ else if (unlikely(!task_running(rq, p) &&
+  (p->policy != SCHED_HLRT) &&
+  !cpu_isset(task_cpu(p), hlrt_cpus_allowed))) {
+ cpumask_t cpus_allowed;
+ int dest_cpu;
+
+ cpus_and(cpus_allowed, p->cpus_allowed,
+  hlrt_cpus_allowed);
+ cpus_and(cpus_allowed, cpus_allowed,
+  cpu_online_map);
+ dest_cpu = cpus_empty(cpus_allowed) ?
+ first_cpu(cpu_possible_map) :
+ any_online_cpu(cpus_allowed);
+ set_task_cpu(p, dest_cpu);
+ task_rq_unlock(rq, &flags);
+ goto repeat_lock_task;
+ }
+#endif
  if (old_state == TASK_UNINTERRUPTIBLE) {
  rq->nr_uninterruptible--;
  /*
@@ -739,6 +781,17 @@
   */
  p->thread_info->preempt_count = 1;
 #endif
+
+#ifdef CONFIG_HLRT
+ /* no inheritance of hard-realtime scheduling class */
+ if (p->policy == SCHED_HLRT) {
+ p->policy = SCHED_NORMAL;
+ set_user_nice(p, 0);
+ p->rt_priority = 0;
+ p->cpus_allowed = CPU_MASK_ALL;
+ }
+#endif
+
  /*
   * Share the timeslice between parent and child, thus the
   * total amount of pending timeslices in the system doesn't change,
@@ -795,6 +848,25 @@
  p->interactive_credit = 0;

  p->prio = effective_prio(p);
+#ifdef CONFIG_HLRT
+ /* put child on a different CPU if we're a HLRT process */
+ if (current->policy == SCHED_HLRT) {
+ cpumask_t cpus_allowed;
+ int dest_cpu;
+
+ cpus_and(cpus_allowed, p->cpus_allowed, hlrt_cpus_allowed);
+ cpus_and(cpus_allowed, cpus_allowed, cpu_online_map);
+ dest_cpu = cpus_empty(cpus_allowed) ?
+ first_cpu(cpu_possible_map) :
+ any_online_cpu(cpus_allowed);
+ set_task_cpu(p, dest_cpu);
+ task_rq_unlock(rq, &flags);
+ rq = task_rq_lock(p, &flags);
+ __activate_task(p, rq);
+ task_rq_unlock(rq, &flags);
+ return;
+ }
+#endif
  set_task_cpu(p, smp_processor_id());

  if (unlikely(!current->array))
@@ -1204,6 +1276,11 @@
  for (i = 0; i < NR_CPUS; i++) {
  if (!cpu_isset(i, cpumask))
  continue;
+#ifdef CONFIG_HLRT
+ /* do not touch the runqueue of a reserved CPU */
+ if (!cpu_isset(i, hlrt_cpus_allowed))
+ continue;
+#endif

  rq_src = cpu_rq(i);
  if (idle || (rq_src->nr_running < this_rq->prev_cpu_load[i]))
@@ -1305,6 +1382,12 @@
  struct list_head *head, *curr;
  task_t *tmp;
```

```
+#ifdef CONFIG_HLRT
+ /* Do not perform load balancing on a reserved CPU */
+ if (hlrt_cpu_reserved())
+ return;
+#endif
+
  busiest = find_busiest_queue(this_rq, this_cpu, idle,
       &imbalance, cpumask);
  if (!busiest)
@@ -2145,6 +2228,9 @@
  else {
  retval = -EINVAL;
  if (policy != SCHED_FIFO && policy != SCHED_RR &&
+#ifdef CONFIG_HLRT
+ policy != SCHED_HLRT &&
+#endif
  policy != SCHED_NORMAL)
  goto out_unlock;
  }
@@ -2157,9 +2243,16 @@
  if (lp.sched_priority < 0 || lp.sched_priority > MAX_USER_RT_PRIO-1)
  goto out_unlock;
  if ((policy == SCHED_NORMAL) != (lp.sched_priority == 0))
+#ifdef CONFIG_HLRT
+ if (policy != SCHED_HLRT)
+#endif
  goto out_unlock;

  retval = -EPERM;
+#if defined(CONFIG_HLRT) && !defined(CONFIG_HLRT_NOROOT)
+ if ((policy == SCHED_HLRT) && !capable(CAP_SYS_NICE))
+ goto out_unlock;
+#endif
  if ((policy == SCHED_FIFO || policy == SCHED_RR) &&
      !capable(CAP_SYS_NICE))
  goto out_unlock;
@@ -2171,6 +2264,20 @@
  if (retval)
  goto out_unlock;

+#ifdef CONFIG_HLRT
+ if (policy == SCHED_HLRT) {
+ task_rq_unlock(rq, &flags);
+ retval = hlrt_set_hlrt_policy(p);
+ goto out_unlock_tasklist;
+ } else if (p->policy == SCHED_HLRT) {
+ task_rq_unlock(rq, &flags);
+ retval = hlrt_unset_hlrt_policy(p);
+ if (retval)
+ goto out_unlock_tasklist;
+ rq = task_rq_lock(p, &flags);
+ }
+#endif /* CONFIG_HLRT */
```

```
+
  array = p->array;
  if (array)
  deactivate_task(p, task_rq(p));
@@ -2332,6 +2439,10 @@
  if ((current->euid != p->euid) && (current->euid != p->uid) &&
  !capable(CAP_SYS_NICE))
  goto out_unlock;
+#ifdef CONFIG_HLRT
+ if (p->policy == SCHED_HLRT)
+ goto out_unlock;
+#endif

  retval = set_cpus_allowed(p, new_mask);

@@ -2792,6 +2903,20 @@
  list_del_init(head->next);
  spin_unlock_irq(&rq->lock);

+#ifdef CONFIG_HLRT
+ if (req->task->policy != SCHED_HLRT) {
+ cpumask_t cpus_allowed;
+ int dest_cpu;
+
+ cpus_and(cpus_allowed, req->task->cpus_allowed,
+  hlrt_cpus_allowed);
+ cpus_and(cpus_allowed, cpus_allowed, cpu_online_map);
+ dest_cpu = cpus_empty(cpus_allowed) ?
+ first_cpu(cpu_possible_map) :
+ any_online_cpu(cpus_allowed);
+ move_task_away(req->task, dest_cpu);
+ } else
+#endif
  move_task_away(req->task,
         any_online_cpu(req->task->cpus_allowed));
  complete(&req->done);
diff -uNr linux-2.6.4/kernel/softirq.c linux-2.6.4-hlrt/kernel/softirq.c
--- linux-2.6.4/kernel/softirq.c 2004-03-11 03:55:24.000000000 +0100
+++ linux-2.6.4-hlrt/kernel/softirq.c 2004-12-16 16:26:55.000000000 +0100
@@ -16,6 +16,10 @@
 #include <linux/cpu.h>
 #include <linux/kthread.h>

+#ifdef CONFIG_HLRT
+#include <asm/hlrt_util.h>
+#endif
+
 /*
    - No shared variables, all the data are CPU local.
    - If a softirq needs serialization, let it serialize itself
@@ -39,6 +43,12 @@
 EXPORT_SYMBOL(irq_stat);
 #endif
```

```
+#ifdef CONFIG_HLRT
+/* locks for __softirq_pending members of irq_stat */
+spinlock_t hlrt_softirq_pending_lock[NR_CPUS] = { [0 ... NR_CPUS - 1] =
+   SPIN_LOCK_UNLOCKED };
+#endif
+
 static struct softirq_action softirq_vec[32] __cacheline_aligned_in_smp;

 static DEFINE_PER_CPU(struct task_struct *, ksoftirqd);
@@ -58,6 +68,17 @@
  wake_up_process(tsk);
 }

+#ifdef CONFIG_HLRT
+static inline void __wakeup_softirqd(int cpu)
+{
+ /* Interrupts are disabled: no need to stop preemption */
+ struct task_struct *tsk = per_cpu(ksoftirqd, cpu);
+
+ if (tsk && tsk->state != TASK_RUNNING)
+ wake_up_process(tsk);
+}
+#endif
+
 /*
  * We restart softirq processing MAX_SOFTIRQ_RESTART times,
  * and we fall back to softirqd after that.
@@ -78,7 +99,19 @@
  if (in_interrupt())
  return;

+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ return;
+#endif
+
  local_irq_save(flags);
+#ifdef CONFIG_HLRT
+ /*
+  * here we need additional locking when raising softirqs
+  * across different CPUs
+  */
+ spin_lock(&hlrt_softirq_pending_lock[smp_processor_id()]);
+#endif

  pending = local_softirq_pending();

@@ -90,6 +123,9 @@
  /* Reset the pending bitmask before enabling irqs */
  local_softirq_pending() = 0;

+#ifdef CONFIG_HLRT
+ spin_unlock(&hlrt_softirq_pending_lock[smp_processor_id()]);
+#endif
```

```
  local_irq_enable();

  h = softirq_vec;
@@ -102,6 +138,9 @@
  } while (pending);

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&hlrt_softirq_pending_lock[smp_processor_id()]);
+#endif

  pending = local_softirq_pending();
  if (pending && --max_restart)
@@ -111,6 +150,9 @@
  __local_bh_enable();
  }

+#ifdef CONFIG_HLRT
+ spin_unlock(&hlrt_softirq_pending_lock[smp_processor_id()]);
+#endif
  local_irq_restore(flags);
 }

@@ -149,6 +191,25 @@

 EXPORT_SYMBOL(raise_softirq_irqoff);

+#ifdef CONFIG_HLRT
+/*
+ * This function must run with irqs disabled!
+ */
+inline fastcall void raise_softirq_irqoff_cpu(unsigned int nr, int cpu)
+{
+ __raise_softirq_irqoff_cpu(nr, cpu);
+
+ /*
+  * Same as above, but it might also be another CPU, so schedule
+  * its softirqd.
+  */
+ if (!in_interrupt() || cpu != smp_processor_id())
+ __wakeup_softirqd(cpu);
+}
+
+EXPORT_SYMBOL(raise_softirq_irqoff_cpu);
+#endif
+
 void fastcall raise_softirq(unsigned int nr)
 {
  unsigned long flags;
@@ -178,15 +239,34 @@
    initialized -- RR */
 static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec) = { NULL };
 static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec) = { NULL };
+#ifdef CONFIG_HLRT
```

```
+static DEFINE_PER_CPU(spinlock_t, tasklet_vec_lock) = SPIN_LOCK_UNLOCKED;
+static DEFINE_PER_CPU(spinlock_t, tasklet_hi_vec_lock) = SPIN_LOCK_UNLOCKED;
+#endif

 void fastcall __tasklet_schedule(struct tasklet_struct *t)
 {
  unsigned long flags;
+#ifdef CONFIG_HLRT
+ int cpu;
+#endif

  local_irq_save(flags);
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ cpu = first_cpu(cpu_possible_map);
+ else
+ cpu = smp_processor_id();
+ spin_lock(&per_cpu(tasklet_vec_lock, cpu));
+ t->next = per_cpu(tasklet_vec, cpu).list;
+ per_cpu(tasklet_vec, cpu).list = t;
+ spin_unlock(&per_cpu(tasklet_vec_lock, cpu));
+ raise_softirq_irqoff_cpu(TASKLET_SOFTIRQ, cpu);
+#else
  t->next = __get_cpu_var(tasklet_vec).list;
  __get_cpu_var(tasklet_vec).list = t;
  raise_softirq_irqoff(TASKLET_SOFTIRQ);
+#endif
  local_irq_restore(flags);
 }

@@ -195,11 +275,26 @@
 void fastcall __tasklet_hi_schedule(struct tasklet_struct *t)
 {
  unsigned long flags;
+#ifdef CONFIG_HLRT
+ int cpu;
+#endif

  local_irq_save(flags);
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ cpu = first_cpu(cpu_possible_map);
+ else
+ cpu = smp_processor_id();
+ spin_lock(&per_cpu(tasklet_hi_vec_lock, cpu));
+ t->next = per_cpu(tasklet_hi_vec, cpu).list;
+ per_cpu(tasklet_hi_vec, cpu).list = t;
+ spin_unlock(&per_cpu(tasklet_hi_vec_lock, cpu));
+ raise_softirq_irqoff_cpu(HI_SOFTIRQ, cpu);
+#else
  t->next = __get_cpu_var(tasklet_hi_vec).list;
  __get_cpu_var(tasklet_hi_vec).list = t;
  raise_softirq_irqoff(HI_SOFTIRQ);
+#endif
```

```
  local_irq_restore(flags);
 }

@@ -210,8 +305,14 @@
  struct tasklet_struct *list;

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&__get_cpu_var(tasklet_vec_lock));
+#endif
  list = __get_cpu_var(tasklet_vec).list;
  __get_cpu_var(tasklet_vec).list = NULL;
+#ifdef CONFIG_HLRT
+ spin_unlock(&__get_cpu_var(tasklet_vec_lock));
+#endif
  local_irq_enable();

  while (list) {
@@ -231,8 +332,14 @@
  }

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&__get_cpu_var(tasklet_vec_lock));
+#endif
  t->next = __get_cpu_var(tasklet_vec).list;
  __get_cpu_var(tasklet_vec).list = t;
+#ifdef CONFIG_HLRT
+ spin_unlock(&__get_cpu_var(tasklet_vec_lock));
+#endif
  __raise_softirq_irqoff(TASKLET_SOFTIRQ);
  local_irq_enable();
  }
@@ -243,8 +350,14 @@
  struct tasklet_struct *list;

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&__get_cpu_var(tasklet_hi_vec_lock));
+#endif
  list = __get_cpu_var(tasklet_hi_vec).list;
  __get_cpu_var(tasklet_hi_vec).list = NULL;
+#ifdef CONFIG_HLRT
+ spin_unlock(&__get_cpu_var(tasklet_hi_vec_lock));
+#endif
  local_irq_enable();

  while (list) {
@@ -264,8 +377,14 @@
  }

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&__get_cpu_var(tasklet_hi_vec_lock));
```

```
+#endif
  t->next = __get_cpu_var(tasklet_hi_vec).list;
  __get_cpu_var(tasklet_hi_vec).list = t;
+#ifdef CONFIG_HLRT
+ spin_unlock(&__get_cpu_var(tasklet_hi_vec_lock));
+#endif
  __raise_softirq_irqoff(HI_SOFTIRQ);
  local_irq_enable();
  }
@@ -324,6 +443,12 @@
  __set_current_state(TASK_RUNNING);

  while (local_softirq_pending()) {
+#ifdef CONFIG_HLRT
+ if (unlikely(smp_processor_id() != cpu)) {
+ printk(KERN_ERR "HLRT: %s running on CPU %i!\n",
+        current->comm, smp_processor_id());
+ }
+#endif
  do_softirq();
  cond_resched();
  }
diff -uNr linux-2.6.4/kernel/timer.c linux-2.6.4-hlrt/kernel/timer.c
--- linux-2.6.4/kernel/timer.c 2004-03-11 03:55:43.000000000 +0100
+++ linux-2.6.4-hlrt/kernel/timer.c 2004-12-16 16:26:55.000000000 +0100
@@ -36,6 +36,10 @@
 #include <asm/div64.h>
 #include <asm/timex.h>

+#ifdef CONFIG_HLRT
+#include <asm/hlrt.h>
+#endif
+
 /*
  * per-CPU timer vector definitions:
  */
@@ -145,6 +149,10 @@
  list_add_tail(&timer->entry, vec);
 }

+#ifdef CONFIG_HLRT
+DECLARE_PER_CPU(int, hlrt_local_apic_mode);
+#endif
+
 int __mod_timer(struct timer_list *timer, unsigned long expires)
 {
  tvec_base_t *old_base, *new_base;
@@ -157,6 +165,11 @@

  spin_lock_irqsave(&timer->lock, flags);
  new_base = &__get_cpu_var(tvec_bases);
+#ifdef CONFIG_HLRT
+ if (__get_cpu_var(hlrt_local_apic_mode) !=
+     HLRT_LAPIC_TIMER_MODE_NORMAL)
+ new_base = &per_cpu(tvec_bases, first_cpu(cpu_possible_map));
+#endif
 repeat:
  old_base = timer->base;

@@ -352,6 +365,59 @@
 EXPORT_SYMBOL(del_timer_sync);
 #endif

+#ifdef CONFIG_HLRT
+static void __hlrt_move_timers_list(struct list_head *head, tvec_base_t *to_base)
+{
+ struct list_head *curr;
+ struct timer_list *timer;
+
+ curr = head->next;
+ while (curr != head) {
+ timer = list_entry(curr, struct timer_list, entry);
+ curr = curr->next;
+ internal_add_timer(to_base, timer);
+ timer->base = to_base;
+ }
+ INIT_LIST_HEAD(head);
+}
+
+void hlrt_move_timers(int from_cpu, int to_cpu)
+{
+ tvec_base_t *from_base, *to_base;
+   unsigned long flags;
+ int i;
+
+ from_base = &per_cpu(tvec_bases, from_cpu);
+ to_base = &per_cpu(tvec_bases, to_cpu);
+
+ if (from_base == to_base)
+ return;
+
+ local_irq_save(flags);
+ if (from_base < to_base) {
+ spin_lock(&to_base->lock);
+ spin_lock(&from_base->lock);
+ } else {
+ spin_lock(&from_base->lock);
+ spin_lock(&to_base->lock);
+ }
+
+ for (i = 0; i < TVR_SIZE; i++) {
+ __hlrt_move_timers_list(from_base->tv1.vec + i, to_base);
+ }
+ for (i = 0; i < TVN_SIZE; i++) {
+ __hlrt_move_timers_list(from_base->tv2.vec + i, to_base);
+ __hlrt_move_timers_list(from_base->tv3.vec + i, to_base);
+ __hlrt_move_timers_list(from_base->tv4.vec + i, to_base);
+ __hlrt_move_timers_list(from_base->tv5.vec + i, to_base);
+ }
+
+ spin_unlock(&to_base->lock);
+ spin_unlock(&from_base->lock);
+ local_irq_restore(flags);
+}
+#endif
+
 static int cascade(tvec_base_t *base, tvec_t *tv, int index)
 {
  /* cascade all the timers from tv up one level */
diff -uNr linux-2.6.4/kernel/workqueue.c linux-2.6.4-hlrt/kernel/workqueue.c
--- linux-2.6.4/kernel/workqueue.c 2004-03-11 03:55:51.000000000 +0100
+++ linux-2.6.4-hlrt/kernel/workqueue.c 2004-12-16 16:26:55.000000000 +0100
@@ -24,6 +24,10 @@
 #include <linux/slab.h>
 #include <linux/kthread.h>

+#ifdef CONFIG_HLRT
+#include <asm/hlrt_util.h>
+#endif
+
 /*
  * The per-CPU workqueue.
  *
@@ -82,6 +86,11 @@
 {
  int ret = 0, cpu = get_cpu();

+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved()) {
+ cpu = first_cpu(cpu_possible_map);
+ }
+#endif
  if (!test_and_set_bit(0, &work->pending)) {
  BUG_ON(!list_empty(&work->entry));
  __queue_work(wq->cpu_wq + cpu, work);
@@ -96,7 +105,17 @@
  struct work_struct *work = (struct work_struct *)__data;
  struct workqueue_struct *wq = work->wq_data;

+#ifdef CONFIG_HLRT
+ int cpu;
+
+ if (hlrt_cpu_reserved())
+ cpu = first_cpu(cpu_possible_map);
+ else
+ cpu = smp_processor_id();
+ __queue_work(wq->cpu_wq + cpu, work);
+#else
  __queue_work(wq->cpu_wq + smp_processor_id(), work);
+#endif
 }
 int fastcall queue_delayed_work(struct workqueue_struct *wq,
@@ -183,6 +202,12 @@
  set_task_state(current, TASK_RUNNING);
  remove_wait_queue(&cwq->more_work, &wait);

+#ifdef CONFIG_HLRT
+ if (unlikely(smp_processor_id() != cpu)) {
+ printk(KERN_ERR "HLRT: %s running on CPU %i!\n",
+        current->comm, smp_processor_id());
+ }
+#endif
  if (!list_empty(&cwq->worklist))
  run_workqueue(cwq);
  }
diff -uNr linux-2.6.4/Makefile linux-2.6.4-hlrt/Makefile
--- linux-2.6.4/Makefile 2004-03-11 03:55:35.000000000 +0100
+++ linux-2.6.4-hlrt/Makefile 2004-12-16 16:26:55.000000000 +0100
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
 SUBLEVEL = 4
-EXTRAVERSION =
+EXTRAVERSION = -hlrt
 NAME=Feisty Dunnart

 # *DOCUMENTATION*
diff -uNr linux-2.6.4/net/core/dev.c linux-2.6.4-hlrt/net/core/dev.c
--- linux-2.6.4/net/core/dev.c 2004-03-11 03:55:27.000000000 +0100
+++ linux-2.6.4-hlrt/net/core/dev.c 2004-12-16 18:38:31.000000000 +0100
@@ -1546,8 +1546,17 @@
   * short when CPU is congested, but is still operating.
   */
  local_irq_save(flags);
+#ifdef CONFIG_HLRT
+ if (hlrt_cpu_reserved())
+ this_cpu = first_cpu(cpu_possible_map);
+ else
+ this_cpu = smp_processor_id();
+ queue = &per_cpu(softnet_data, this_cpu);
+ spin_lock(&queue->input_pkt_queue_lock);
+#else
  this_cpu = smp_processor_id();
  queue = &__get_cpu_var(softnet_data);
+#endif

  __get_cpu_var(netdev_rx_stat).total++;
  if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
@@ -1561,6 +1570,9 @@
 #ifndef OFFLINE_SAMPLE
  get_sample_stats(this_cpu);
 #endif
+#ifdef CONFIG_HLRT
+ spin_unlock(&queue->input_pkt_queue_lock);
+#endif
```

```
  local_irq_restore(flags);
  return queue->cng_level;
  }
@@ -1587,6 +1599,9 @@

 drop:
  __get_cpu_var(netdev_rx_stat).dropped++;
+#ifdef CONFIG_HLRT
+ spin_unlock(&queue->input_pkt_queue_lock);
+#endif
  local_irq_restore(flags);

  kfree_skb(skb);
@@ -1611,8 +1626,14 @@
  struct sk_buff *clist;

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&sd->completion_queue_lock);
+#endif
  clist = sd->completion_queue;
  sd->completion_queue = NULL;
+#ifdef CONFIG_HLRT
+ spin_unlock(&sd->completion_queue_lock);
+#endif
  local_irq_enable();

  while (clist) {
@@ -1628,8 +1649,14 @@
  struct net_device *head;

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&sd->output_queue_lock);
+#endif
  head = sd->output_queue;
  sd->output_queue = NULL;
+#ifdef CONFIG_HLRT
+ spin_unlock(&sd->output_queue_lock);
+#endif
  local_irq_enable();

  while (head) {
@@ -1762,7 +1789,13 @@
  struct net_device *dev;

  local_irq_disable();
+#ifdef CONFIG_HLRT
+ spin_lock(&queue->input_pkt_queue_lock);
+#endif
  skb = __skb_dequeue(&queue->input_pkt_queue);
+#ifdef CONFIG_HLRT
+ spin_unlock(&queue->input_pkt_queue_lock);
+#endif
```

```
  if (!skb)
  goto job_done;
  local_irq_enable();
@@ -1779,14 +1812,28 @@
  break;

 #ifdef CONFIG_NET_HW_FLOWCONTROL
+#ifdef CONFIG_HLRT
+ local_irq_disable();
+ spin_lock(&queue->input_pkt_queue_lock);
+#endif
  if (queue->throttle &&
      queue->input_pkt_queue.qlen < no_cong_thresh ) {
  queue->throttle = 0;
+#ifdef CONFIG_HLRT
+ spin_unlock(&queue->input_pkt_queue_lock);
+ local_irq_enable();
+#endif CONFIG_HLRT
  if (atomic_dec_and_test(&netdev_dropping)) {
  netdev_wakeup();
  break;
  }
  }
+#ifdef CONFIG_HLRT
+ else {
+ spin_unlock(&queue->input_pkt_queue_lock);
+ local_irq_enable();
+ }
+#endif
 #endif
  }

@@ -1798,10 +1845,19 @@
  backlog_dev->quota -= work;
  *budget -= work;

+#ifdef CONFIG_HLRT
+ spin_lock(&queue->poll_list_lock);
+#endif
  list_del(&backlog_dev->poll_list);
+#ifdef CONFIG_HLRT
+ spin_unlock(&queue->poll_list_lock);
+#endif
  smp_mb__before_clear_bit();
  netif_poll_enable(backlog_dev);

+#ifdef CONFIG_HLRT
+ spin_lock(&queue->input_pkt_queue_lock);
+#endif
  if (queue->throttle) {
  queue->throttle = 0;
 #ifdef CONFIG_NET_HW_FLOWCONTROL
@@ -1809,6 +1865,9 @@
  netdev_wakeup();
 #endif
  }
+#ifdef CONFIG_HLRT
+ spin_unlock(&queue->input_pkt_queue_lock);
+#endif
  local_irq_enable();
  return 0;
 }
@@ -1823,6 +1882,11 @@
  preempt_disable();
  local_irq_disable();

+ /*
+  * no locking needed here because the list can only
+  * become empty via dev->poll() call and is always
+  * added to at its tail
+  */
  while (!list_empty(&queue->poll_list)) {
  struct net_device *dev;

@@ -1836,8 +1900,18 @@

  if (dev->quota <= 0 || dev->poll(dev, &budget)) {
  local_irq_disable();
+ /*
+  * now we need locking because we're modifying
+  * the list
+  */
+#ifdef CONFIG_HLRT
+ spin_lock(&queue->poll_list_lock);
+#endif
  list_del(&dev->poll_list);
  list_add_tail(&dev->poll_list, &queue->poll_list);
+#ifdef CONFIG_HLRT
+ spin_unlock(&queue->poll_list_lock);
+#endif
  if (dev->quota < 0)
  dev->quota += dev->weight;
  else
@@ -3168,6 +3242,12 @@
  queue->backlog_dev.weight = weight_p;
  queue->backlog_dev.poll = process_backlog;
  atomic_set(&queue->backlog_dev.refcnt, 1);
+#ifdef CONFIG_HLRT
+ spin_lock_init(&queue->input_pkt_queue_lock);
+ spin_lock_init(&queue->poll_list_lock);
+ spin_lock_init(&queue->output_queue_lock);
+ spin_lock_init(&queue->completion_queue_lock);
+#endif
  }

 #ifdef OFFLINE_SAMPLE
diff -uNr linux-2.6.4/net/core/flow.c linux-2.6.4-hlrt/net/core/flow.c
--- linux-2.6.4/net/core/flow.c 2004-03-11 03:55:21.000000000 +0100
```

```
+++ linux-2.6.4-hlrt/net/core/flow.c 2004-12-16 16:26:55.000000000 +0100
@@ -24,6 +24,10 @@
 #include <asm/atomic.h>
 #include <asm/semaphore.h>

+#ifdef CONFIG_HLRT
+#include <asm/hlrt_util.h>
+#endif
+
 struct flow_cache_entry {
  struct flow_cache_entry *next;
  u16 family;
@@ -273,6 +277,9 @@
  int cpu;
  struct tasklet_struct *tasklet;

+#ifdef CONFIG_HLRT
+ WARN_ON(hlrt_cpu_reserved());
+#endif
  cpu = smp_processor_id();

  tasklet = flow_flush_tasklet(cpu);
```

Bibliography

[BC01] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*, chapter 1. Addison-Wesley, 1995.

[HL03] Universität Bremen, Fachbereich 3. *HaRTLinC Projektbericht*, Dezember 2003. Chapter 2. (in German).

[Int03] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2003. Chapter 5, pages 5-11 – 5-13 and chapter 8, pages 8-6 – 8-21.

[Jos01] Mathai Joseph, editor. *Real-time Systems. Specification, Verification and Analysis*. June 2001. Chapters 2, 3 and 4. http://www.tcs.com/techbytes/htdocs/book_mj.htm.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*, pages 15–16, 35–36, 83–86, 134, 164–166. Kluwer Academic Publishers, 1997.

[Lov03] Robert Love. *Linux Kernel Development*, pages 31–52, 85–102. Sams Publishing, 2003.

[TW97] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems*, chapter 24, pages 82–93. Prentice Hall, second edition edition, 1997.

[Vah95] Uresh Vahalia. *UNIX Internals: The New Frontiers*, chapter 5. Prentice Hall, 1995.

[Zwe02] Klaas-Henning Zweck. Kernelbasierte Echtzeiterweiterung eines Linux-Multiprozessor-Systems. Diploma thesis, Universität Bremen, Fachbereich 3, April 2002. (in German).