CHEMNITZ UNIVERSITY OF TECHNOLOGY

Department of Computer Science

Real-Time Systems Group

Seminar Paper

A Quantitative Comparison of Realtime Linux Solutions

Markus Franke

Chemnitz, March 5, 2007

supervisor: Dr. Robert Baumgartl

Contents

List of Figures

1 Introduction

1.1 Realtime Operating Systems

The fast evolution in the field of Computer Science and Microelectronics led to a broad variety of new device used in everyday life. The Information- and Communication-Industries are hereby trend-setting. Cellular phones, MP3-players, personal digital assistants (PDA's), notebooks, and navigation systems are just a few examples of innovative product divisions. But also the automobile industry gains advantages from the new technologies, integrating more and more mostly interconnected functionalities in a car.

The diversity of available devices and the complexity of these systems made it highly desirable to use the advantages of an operating system. It should be capable to offer applications as well as programmers an interface to the underlying architecture. A well-defined Application Programmers Interface (API) enables programming of applications based on this API without taking care of system-specific properties.

Many applications especially in the field of digital signal processing have to meet time constraints in order to work properly. Crunching sounds while playing music on an MP3-player, or delayed video playback should be avoided although these are just soft realtime applications. But there are also fields which require hard realtime constraints. Programs for controlling a space vehicle, or for observing the correct function of powerplants, fall in this category.

Up to now, specialised commercial realtime operating systems (RTOS) have been used widely. Examples are VxWorks, which was used during the Mars Pathfinder mission, or pSOS, and LynxOS. The performance of these systems is of course very good, and they are perfectly suited to meet the requirements in hard realtime environments. However, shortcomings are for instance the price and the probably small user/programmer community.[1]

The idea now is to use and adapt general-purpose operating systems (GPOS) to provide a realtime environment which is comparable to specialised RTOS. Sometimes it is enough to meet soft realtime constraints for applications. But even hard realtime functionality is achievable. GPOS are much cheaper than specialised RTOS, and look retrospectively at long-term development experiences. Moreover, existing hardware can be used further on because device drivers are already available.

1.2 Objectives

The focus of this paper is on the GPOS Linux. The question is how Linux can be extended to provide realtime capabilities. Windows is also a widely known GPOS. It can also be analysed considering its usability for realtime applications, but this is beyond the scope of this work. Linux is an open source operating system and that is the big difference to Windows. This enables programmers to analyse the system and to introduce new mechanisms for providing realtime capabilities. Besides, the costs of using Linux in realtime environments is limited to the education of staff members rather than paying any license fees. A drawback can be the missing reliable vendor support, which can be very important, especially for companies which are using realtime Linux as a part of their products.

Limiting to Linux, this paper considers 2 Linux realtime distributions namely Xenomai, and RTAI. Before discussing these extensions to the Linux kernel in chapter 3, chapter 2 deals with native support of realtime capabilities in Linux. Chapter 4 presents some basic measurements for each distribution in order to give a quantitative comparison, as promised in the title of this paper. The final chapter 5 will summarize the measurments from chapter 4 and tries to give some industrial guidelines. These rules can be helpful for deciding which realtime Linux extension should be used with respect to the requirements of the applications.

If not other stated the term Linux always refers to the Linux kernel in version 2.6.

2 Native Support for Realtime Demands

Before starting the discussion about several Realtime Linux distributions, it is a good idea to have a look at native support for realtime in the standard Linux kernel.

2.1 Problems in Standard Linux

The biggest problem at all is the historical non-preemptibility of the kernel.[1] It means that, if one process enters the kernel it acquires the so called "Big Kernel Lock" (BKL) and prevents other processes from entering in the kernel. (e.g. through a system call) This was done in order to protect critical sections in kernel code. In this example, the whole Linux kernel is a critical section by itself. The kernel till version 2.0 worked in such a way. With respect to realtime requirements this behaviour is completely unacceptable. For example an interrupt from an I/O-device is triggered. The activation of the associated I/O-task as effect of the interrupt is delayed as long as any other process is in kernel mode.

During work on version 2.2 and 2.4 the BKL was slightly removed and replaced by several different datastructure specific spin-locks for protecting small critical sections.[2] Over time, the kernel gained a finer granularity, which is very important for providing short latencies. Nevertheless, the BKL is still available in Linux kernel 2.6 (as a semaphore) mostly to protect legacy code. (VFS and several filesystems) In addition to that, some important extensions to the Linux kernel 2.4/2.6 exist, improving its granularity and therefore its realtime capabilities. These solutions will be presented throughout the next sections.

Another issue, which has to be considered, is the available clock and timer resolution.[1] Linux as a GPOS receives every 4ms (standard configuration) an interrupt for maintaining the system time and checking for the expiration of timers. To provide a higher resolution, Linux stores the timestamp-counter on x86 CPU's at each timer interrupt. When a system call like gettimeofday() occurs, the actual timestamp-counter is read again and from the difference of the two timestamp-values the current time is computed in a microsecond range. Actually, the Linux kernel works with so called jiffies, but this is just an implementation issue, which is omitted here for simplicity. As an addition to the reading of the timestamp counter, the kernel also has to check for timer expirations. The complexity of this test and the execution time of the timer routine may cause high latencies. This could be a problem in a realtime environment. Solutions for overcoming this issue are not part of this paper. Rather, in this chapter it will be focussed on improvements of granularity of the Linux kernel by itself.

2.2 Introduction of explicit Preemption Points

In the year 2001, Ingo Molnar started his work on the Linux kernel, trying to increase its preemptibility. The idea was to introduce explicit preemption points in blocks which will possibly run for long stretches of time.[4] These situations occur for example when iterating over large data structures. The question is therefore, whether there can be safely placed preemption points in order to call the scheduler again. An example from the Linux kernel is depicted in listing 2.1. With "DEFINE_RESCHED_COUNT" a threshold is defined at the beginning. During execution of the loop a counter is checked against this threshold and on exceedance, rescheduling is performed if needed. Therefore, the lock is released, the scheduler is called and after that a jump back to the label "redo:" is done. From this point the spinlock has to be acquired again. This technique is called "lock-breaking preemption".

The problem of this approach is to find the long running code fragments and insert the preemption points. Andrew Morton's rtc-debug patch can be a good tool for doing that. As the kernel is undergoing a steady development the maintenance of these extensions can grow to a full-time job. Another drawback is the overhead which is needed at each preemption point. The listing in 2.1 shows the amount of code which has to be executed at every preemption point.

These extensions of Ingo Molnar are widely known as the "Low-Latency Patches". However, it is not correct to speak about a patch with respect to the recent kernel. The patches have been integrated into Linux kernel version 2.6.11 and can be activated via the switch "CONFIG_PREEMPT_VOLUNTARY" during kernel configuration. Andrew Morton is currently responsible for the maintenance of this feature.

```
void prune_dcache(int count)
{
  spin_lock(&dcache_lock);

  for (;;) {
       /* do some work*
  }

  spin_unlock(&dcache_lock);
}
```

```
void prune_dcache(int count)
{
  DEFINE_RESCHED_COUNT;
redo:
  spin_lock(&dcache_lock);
  for (;;) {
   if (TEST_RESCHED_COUNT(100)) {
     RESET_RESCHED_COUNT();
     if (conditional_schedule_needed())
     {
         spin_unlock(&dcache_lock);
         unconditional_schedule();
         goto redo;
     }
   }
   /* do some work */
  }
  spin_unlock(&dcache_lock);
}
```

Figure 2.1: Introduction of explicit Preemption Points

2.3 Introduction of implicit Preemption Points

A straight forward approach to gain more granularity is making the Linux kernel preemptable in general. [3] This idea was presented and carried out by MontaVista. In literature these modifications are often called the "Preemption Patches". The basic idea is to modify the spinlock macros and the interrupt return code in a sense that the Linux scheduler gets the chance to be invoked more often. Modification means that everytime when there is a scheduling request pending, and it is safe to preempt the current process, the scheduler is called. If the current code beeing executed is not an interrupt handler, or the scheduler itself and no spinlock is beeing held it is safe to make a context switch. Before the kernel enters one of the previous mentioned sections, it increments a global Preemption-Lock-Counter, which indicates that preemption is now forbidden. When leaving, the state of this counter is decremented again, and the kernel checks whether preemption has been called, meanwhile. In this case a context switch will be made. That means, each time when a spinlock is released or a return from an interrupt happens there is the possiblity for a preemption, and withit decreasing the time between the occurence of an event and the invocation of the scheduler. The big advantage of this approach is that no explicit preemption points have to be introduced in the kernel. Instead the finer granularity is achieved by means of implicit preemption points. Actually, its not appropriate to speak about a "Preemption Patch", because this features was migrated to the Linux kernel mainline with version 2.5.4-pre6. It can be activated via the "CONFIG_PREEMPT"-switch during kernel configuration.

2.4 Towards a fully Preemptible Design

The two above mentioned techniques, for achieving a more responsive kernel, lead to considerable improvements. However, these designs are still not fully preemptible, with respect to priority inheritance, preemptible hardware interrupts, and preemptible soft interrupts. All these types of critical sections are not preemptible, if using "CONFIG_PREEMPT" or "CONFIG_PREEMPT_VOLUNTARY". The solution are the "Realtime Preemption Patches" by Ingo Molnar, which are currently under high development focussing the incorporation in the standard Linux kernel. It is a consequent continuation of the ideas playing a role in the Low-Latency Patches. The new introduced preemption model is fully deterministic, with respect to scheduling, interrupt handling, and high-resolution timers. At the time of writing this paper, there is still a lot of work to do for integration in the mainline. Nevertheless, a separate patch is available for kernel versions 2.6.13 upwards. After patching the kernel, the feature can be enabled via the switch "CONFIG_PREEMPT_RT". The philosophy of "PREEMPT_RT" is to leverage the symmetric multiprocessing (SMP) capabilities of the Linux kernel in order to provide extra preemptibility. A good overview about the "PREEMPT_RT"-patch is given on [5]. The discussion of each feature would be a paper by itself, and is therefore beyond the scope of this work.

3 Realtime Linux Distributions

3.1 Introduction

Having discussed the realtime capabilities provided by the native Linux kernel itself, the following chapter will deal with dedicated Linux realtime distributions. All of the presented systems follow the principles of using a specialised microkernel to achieve realtime performance. This mircrokernel is responsible for the core functions of an operating system like interrupt handling, scheduling, timer handling, and interprocess communication. On top of this microkernel several domains coexist, which underlie the microkernel scheduler. Each of the domains owns a fixed priority, which allows to distinguish between tasks of different qualities during scheduling. Usually, the hard realtime tasks are running with a high priority, while a very low priority is assigned to the standard Linux kernel. This means, everytime a realtime process is runnable the Linux kernel can be preempted, thus meaning the standard Linux kernel plays the role of an idle-task.

Another function of the realtime kernel is interrupt handling. Usually, the standard Linux kernel deals with real hardware interrupts. But this is not possible anymore, as the realtime microkernel is working directly on the hardware now. Because of this, a technique called "Interrupt Virtualisation" is introduced, which means that the realtime microkernel emulates hardware interrupts to the Linux kernel. In order to work together the standard Linux kernel usually has to be adapted to the underlying realtime-layer.

3.2 Connections and Influences

The presented realtime distributions must not be considered independently. Rather, they are based on and influenced by each other. The diagram in figure 3.1 gives an overview about the temporal connections between Xenomai, RTAI, and RTLinux.

The RTLinux project started in 1997 and was the first kind of a realtime Linux distribution. In 1999 the FSMLabs company was founded, and from this point in time two products began to exist namely RTLinux and RTLinux/Free. As a competitive system, the development of RTAI started in the year 2000. However, license issues of RTLinux forced the RTAI project to change their hardware abstraction layer in the year 2003. In 2001, the Xenomai project arises and fused with RTAI to the RTAI/Fusion project in 2002. The big goal was to reach hard realtime capabilities in User-Space applications. Finally, RTAI/Fusion splitted up again in 2005. Xenomai restarted to exist independently again, following the design of an abstract realtime core. RTAI/LXRT tries to achieve realtime constraints in User-Space in another way than Xenomai does.

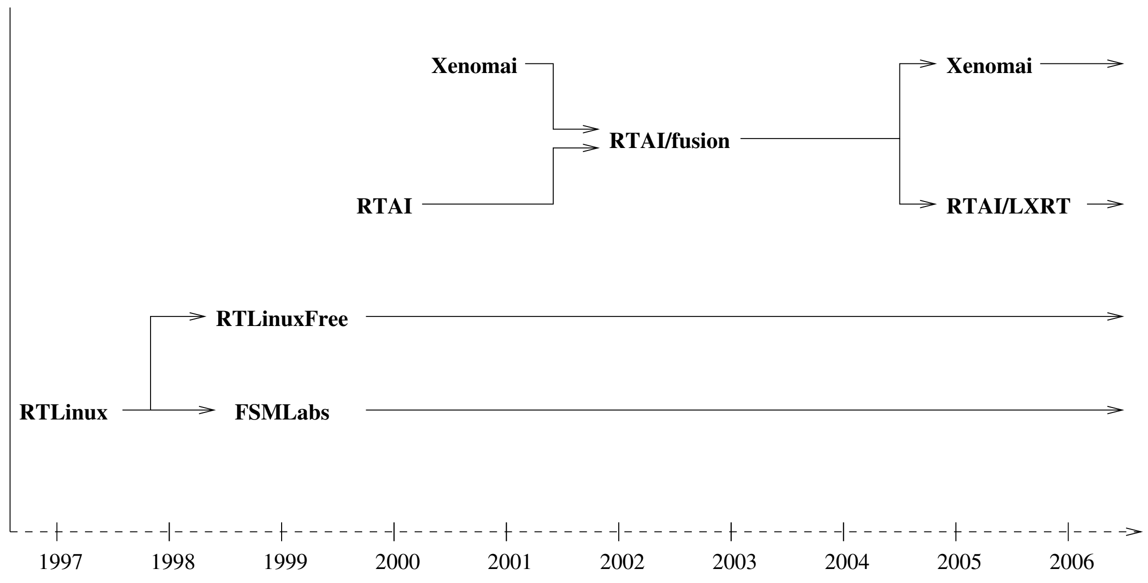

Figure 3.1: Timeline of realtime Linux distributions

3.3 Xenomai

The Xenomai project[1] was launched in the year 2001. The head maintainers are currently Philippe Gerum, Bruno Rouchouse and Gilles Chanteperdrix. The main goal is to provide real-time capabilities based on some fundamental functionalities exported by an abstract RTOS core. Furthermore, mechanisms are provided to support applications which are originally intended to be used with traditional RTOS. In general, Xenomai aims on flexibility, extensibility, and maintainability rather than trying to achieve lowest technically feasible latencies like RTAI does. [6] The Xenomai kernel components underlie the GNU General Public License (GPL). However, it is still possible to distribute proprietary applications by using the User-Space libraries, which are under the GNU Lesser General Public License (LGPL).

3.3.1 Principles

When discussing the principles of Xenomai, it is necessary to have a closer look at the architecture. Figure 3.2 gives a good overview about the vertical structure of the overall system. Xenomai consists of several different abstract layers, which introduce an enormous flexibility.

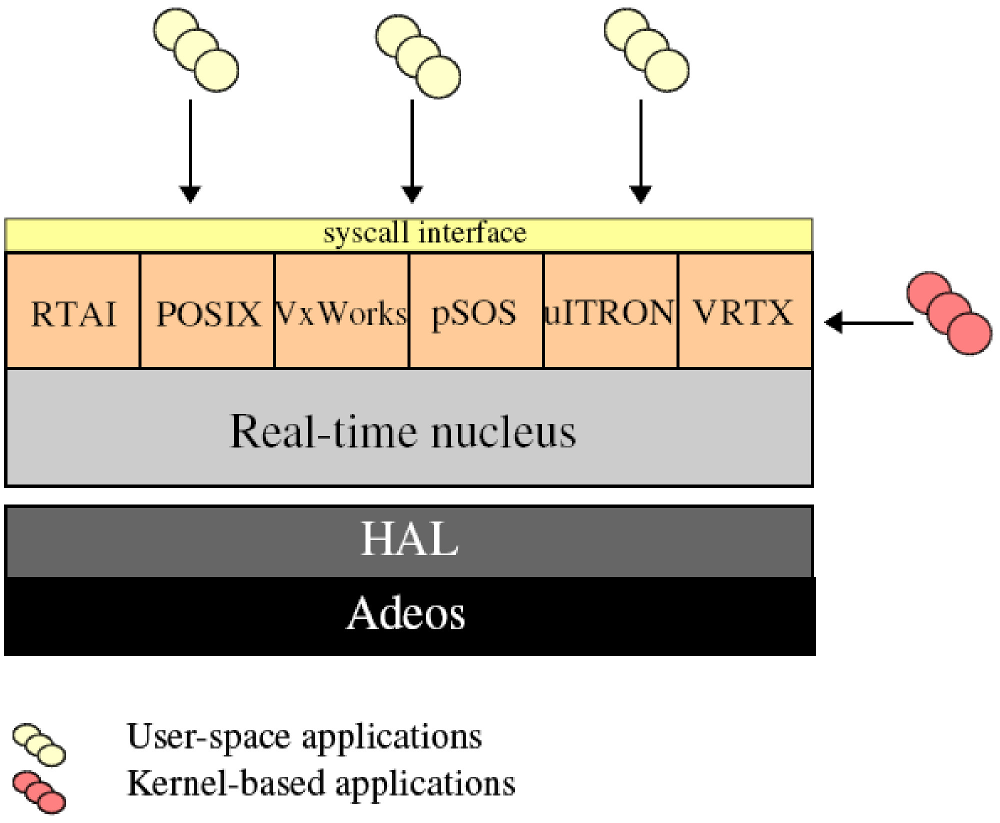

Figure 3.2: Architecture of Xenomai [7]

[1]http://www.xenomai.org

On top of the hardware the Adeos nanokernel is working. Section 3.4.2 gives an overview about the aims it focuses on and functionalities it provides. The hardware abstraction layer(HAL) lying above Adeos is the architecture-dependent part of Xenomai. When porting Xenomai to be runnable on another hardware platform, the HAL is the layer which has to be adapted, provided that Adeos is available for the new architecture. The central part of the system is the abstract RTOS nucleus which runs on top of the HAL. It implements a set of generic RTOS services, which are common for the most systems. The services can be either accessed through a Xenomai native API or via the different RTOS-API's build on top of the nucleus. These additional so called API skins for traditional RTOS make it possible to run even legacy software on top of the Xenomai core. This is especially useful when migrating to the Xenomai architecture because applications need not to be rewritten entirely. Xenomai enables users to run their applications either in Kernel- or in User-Space, though tasks living in the latter one have guaranteed but worse execution times. Running tasks in the User-Space context ensures the reliability of the system, because the crash of one process in Kernel-Space can lead to a breakdown of the entire system. The next section will focus on some characteristics of the native API.

3.3.2 API

Despite the fact that Xenomai offers a wide variety of traditional RTOS API's there is also a native API, which can be used fully independently. It is implemented in the kernel module "xeno_native.ko". The following services are exported:[7]

- Task managment
- Timing services
- Synchronisation support (Counting semaphores, Mutexes, Condition variables, Event flag groups
- Messaging and Communication (Message queues, Message pipes, Memory heaps
- Device I/O handling
- Registry support (enables seamless access to system calls from different execution spaces)

The API is context-independent, which means that nearly the same set of services can be used either from Kernel- or from User-Space. A task for example is always represented by an RT_TASK deskriptor and a message queue by a RT_QUEUE deskriptor regardless of the execution space from which they are used. Moreover, these deskriptors can be shared between execution spaces allowing to call services from objects which are created from the opposite space.

Xenomai supports mixable execution modes for tasks embodied in User-Space. That means after starting a new realtime process it is running in the Xenomai domain, also named "Primary Domain". The task benefits from memory protection but is scheduled by Xenomai directly, rather than by the Linux kernel. Running in this mode worst-case scheduling latency is always near to hardware limits because no synchronisation with the Linux kernel is necessary. Any Linux activity can be preempted everytime with no delay.

When issuing a Linux system call the task is migrated transparently to the Linux domain ("Secondary Domain"). It underlies now the scheduler of Linux which means that execution can start as soon as the next rescheduling point of the Linux kernel is reached. Due to this, Xenomai benefits of recent improvements in granularity of the Linux kernel which are outlined in chapter 2. It allows even realtime tasks running in the Secondary Domain to reach predictability.

A task which is running in the Secondary Domain has access to all regular Linux services. However, system calls which are designed for fairness and predictability will even cause some problems here. When entering the Secondary Domain, the realtime task does not loose its priority. The priority scheme is consistent across domains, and this means that the Linux kernel as a whole inherits the priority of the realtime task. It now competes for CPU resources with the remaining realtime processes. This is a fundamental difference to the RTAI/LXRT approach, where such a priority inheritance is omitted. In RTAI/LXRT the realtime task gets the lowest priority defined by the RTAI scheduler.

When the task enters the Secondary Domain there would still be the possiblity for beeing preempted by a standard Linux interrupt handler. This can cause unwanted latencies thus making it necessary to apply a new technique called "Interrupt Shield". It delays every Linux interrupt handler as long as any realtime task is running in the Secondary Domain. The Interrupt Shield can be enabled during configuration of the patched kernel via the switch "CONFIG_XENO_OPT_ISHIELD". Besides, it is possible to enable/disable the Interrupt Shield during runtime through the Xenomai system call "rt_task_set_mode". (flag: T_SHIELD) The Interrupt Shield can be seen as a Adeos domain by itself. (see figure 3.3)

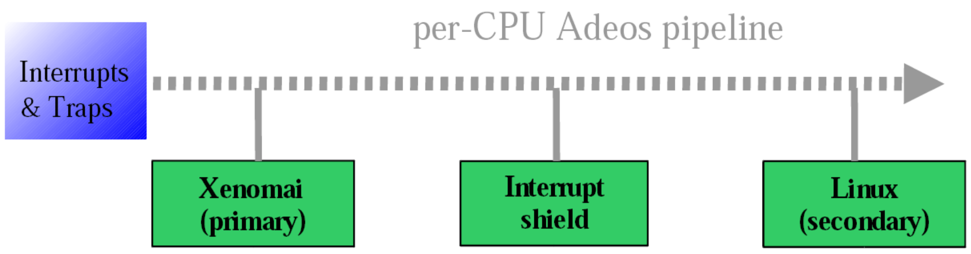

Figure 3.3: Interrupt Shield in the Interrupt Pipeline [12]

But this feature can also lead to a priority inversion problem. A realtime task can block on the execution of an I/O operation (e.g. a regular Linux device driver). Since there is the Interrupt Shield enabled, the delivery of an interrupt can be delayed as long as any other realtime task is active in the Secondary Domain. This can cause a priority inversion and makes it necessary to write new device drivers based upon the Xenomai or Primary Domain. The Realtime Device Driver Model(RTDM) supports the development of these drivers. A drawback is that drivers, which are already available for the standard Linux kernel, are not suitable for being used in a realtime environment and thus have to be rewritten.

Figure 3.4 shows the process of migrating a realtime task between Primary and Secondary mode.

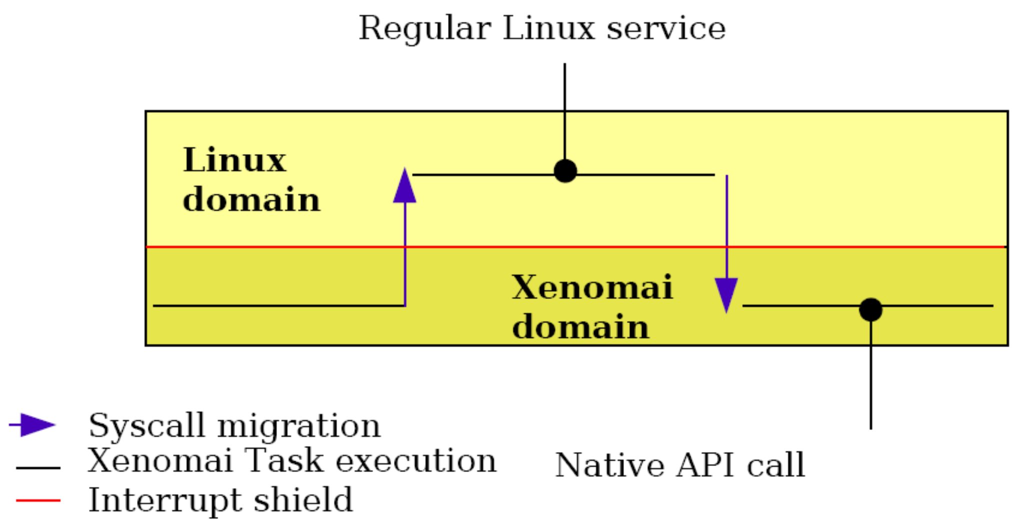

Figure 3.4: Migration between Primary and Secondary Mode[7]

3.3.3 Features

The following section will outline some important features of the Xenomai-nucleus without commenting them in detail. Moreover, some key points are presented to make a comparison with other Linux realtime distributions possible. More precise information can be found in [7] and [8].

1. Multithreading support

   - preemptive scheduling algorithm
   - 32 bit integer priorities
   - round-robin scheduling for threads with same priority (individual time quantums)
   - per thread watchdogs
   - 5-dimensional state modell for processes
   - support for priority inheritance

   - per-thread FPU support

2. Basic synchronisation support
   - support for priority inheritance, implementation shared with scheduler code
   - support for time-bounded wait and forcible deletion with waiters awakening

3. Timer and clock management
   - periodic and aperiodic mode (if hardware support available)
   - nucleus transparently switches from periodic jiffies to time-stamp counter values depending on timer operating mode
   - bounded worst-case time for starting, stopping and maintaining timers
   - based on the timer wheel algorithm[9]

4. Basic memory allocation
   - dynamic memory allocation with realtime guarantees (realtime heap device)
   - based on McKusick's and Karels' proposal[10]

3.3.4 Getting Started

Before running the first realtime task, the system must be set up properly. The first task is to get the most recent Xenomai release from http://www.xenomai.org. After extracting the tarball archive the Linux kernel patches for different architectures can be found in the directory "<XENOMAI-ROOT>/ksrc/arch/<ARCH>/patches". By the time of writing this paper the following platforms were supported:

- ARM
- Blackfin DSP
- x86, x86_64 under development
- IA64
- Power PC

With the add of the latest kernel patch, a plain Linux kernel from http://kernel.org can be downloaded and patched. During this procedure the standard Linux kernel is prepared for working together with Adeos (see section 3.4.2). The patched kernel can now be configured in order to adjust Xenomai parameters. Therefore, a new configuration node named "Real-time sub-system" was added by the kernel patch. In addition to that, some kernel parameters influence Linux realtime capabilities in a negative sense. They are documented in "<XENOMAI-ROOT>/TROUBLESHOOTING", and its use should be avoided. Finally, the kernel can be compiled and installed. Having set up the kernel, the Xenomai User-Space tools have to be installed, providing things like libraries, header files and a Xenomai testsuite.

3.4 RTAI - Realtime Application Interface

3.4.1 Principles

The development of the Realtime Application Interface (RTAI) was started around the year 2000 by Professor Mantegazza at Dipartimento di Ingegneria Aerospaziale in Mailand. As already illustrated in figure 3.1, it is a fork from the RTLinux project. The following architectures are supported by RTAI 3.4:

- x86
  - with and without timestamp counter (TSC)
  - floating point support
  - manifold patches available for Linux kernel 2.4 and 2.6
  - x86_64 is in beta status
- PowerPC
  - PowerPC G5
  - patches available only for Linux kernel 2.4
- ARM
  - StrongARM 11x0
  - ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712, PXA25x
  - ARM9: Cirrus EDB9301 development board with EP9301 CPU (ARM920T)
  - patches not always for both Linux kernel versions (2.4 / 2.6) available

RTAI is very similar to Xenomai as a consequence of its merging to RTAI/Fusion project in the year 2002. Many API calls of Xenomai's native API have their counterparts in the RTAI API, distinguishable only by their names. Nevertheless, RTAI does not follow the concept of offering multiple API's to the user, like Xenomai does. In fact, it is out of scope to provide the ability to run legacy RTOS applications over RTAI. As a consequence, there are not as many different abstraction layers stacked on top of each other as in the Xenomai case. Figure 3.5 gives an idea about the architecture of RTAI:

Till the end of 2003, the hardware abstraction layer relied on the so called Realtime Hardware Abstraction Layer (RTHAL). Basically, it consists of a set of function pointers, which are patched into the Linux kernel. In the beginning, the pointers refer to standard Linux functions and thus have no impact on the operations of the Linux kernel.(fig.3.6(A)) After loading the RTAI modules, the function pointers are changed to work under the control of RTAI. Figure 3.6(B) illustrates this procedure. This process incorporates mainly basic interrupt handling and trap handling. The advantage of this approach is the very slight impact on the Linux kernel. Only about 70 lines of code (LOC) had to be changed or added. However, license issues

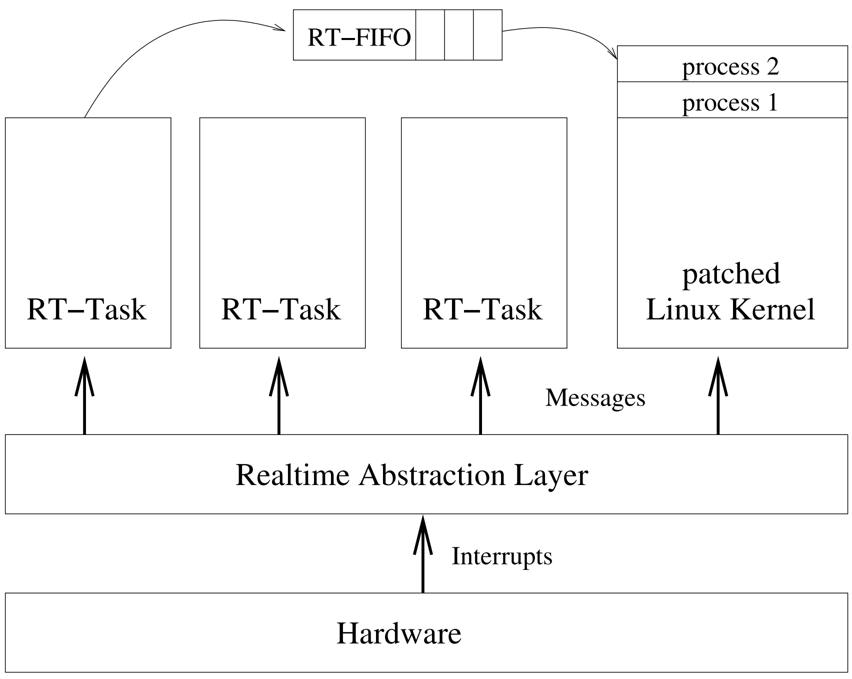

Figure 3.5: Architecture of RTAI

with the RTLinux project forced RTAI to move their HAL to the Adeos nanokernel. (see section 3.4.2) The codebase has nothing anymore in common with RTLinux. The kernel components of the RTAI project are under the GNU General Public License (GPL). The User-Space libraries are under the terms of the GNU Lesser General Public License (LGPL). Thus, it is possible to distribute proprietary hard realtime User-Space applications based on RTAI/LXRT and/or RTDM.

Like in Xenomai, RTAI regards the standard Linux kernel as the idle process with lowest priority, executed only when there are no other realtime processes schedulable. Timers in RTAI can be programmed to work either in periodic or in oneshot mode. The oneshot mode offers a finer granularity with slightly more overhead, caused by the reprogramming of the timer.

RTAI offers a testsuite called "Showroom", which covers almost every feature available in RTAI. It is very useful for testing the proper functionality of RTAI and can be also used for basic measurements, e.g. scheduling latencies, under different circumstances.

Furthermore, the so called "RTAI-Lab Toolchain" enables users to convert block diagrams into RTAI executables and to monitor their operation on various targets. The diagrams can be developed using either Scilab/Scicos or Matlab/Simulink/RTW. Especially, this functionality is a fundamental advantage to Xenomai, where such a feature is not available.

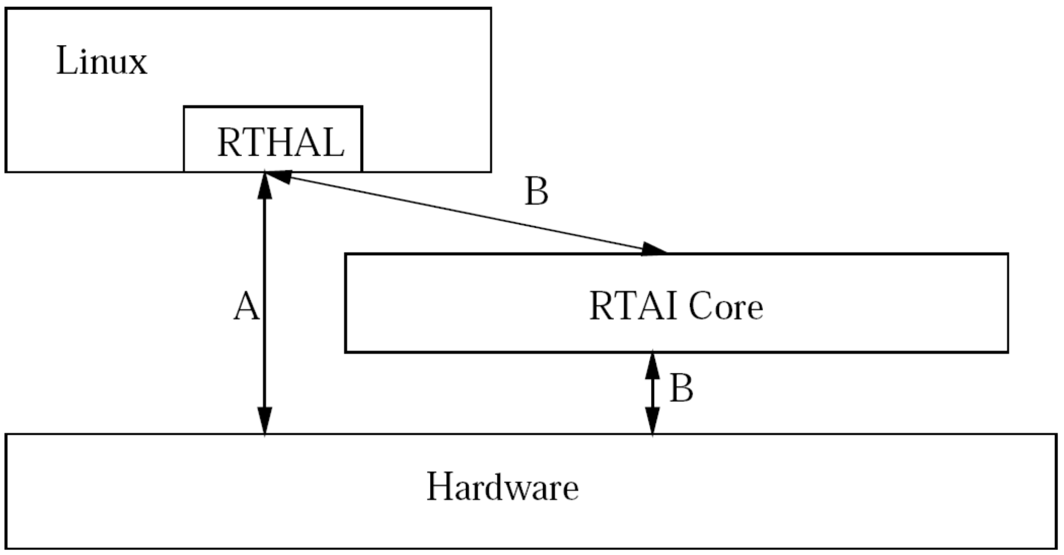

Figure 3.6: Legacy RTHAL approach in RTAI [15]

3.4.2 Adeos

The general design of the Adeos nanokernel [2] has been proposed by Karim Yaghmour in the year 2001 [11]. Adeos is a resource virtualisation layer available as a Linux kernel patch[12]. It allows several different so called "domains" to coexist on the same hardware. A domain could be a complete operating system like Linux, but also realtime tasks not based on the Linux kernel belong to this category. The domains by themselves do not see each other, but every domain sees Adeos. Thats why the standard Linux kernel needs to be patched for working together with Adeos. The domains compete each other for the processing of external events (e.g. interrupts) and internal events (e.g. traps and exceptions) Another key feature of Adeos is its ability to export a generic API to client domains, which is independent of the underlying CPU architecture. If a client domain has to be ported to another platform, much of the porting effort has to be done on the Adeos layer.

Each domain is attached to a central datastructure called the "event pipeline" or "I-Pipe", offering the possiblity to notify the domains for external interrupts, system calls issued by Linux or other system events triggered by kernel code (see figure 3.7). Moreover, each domain is assigned a static priority, which is used for a proper dispatchment order of events. Once, e.g. an external interrupt occurs it is first handled by the domain owning the highest priority. After its treatment, it is progressed down along all the other attached domains. The Linux kernel by itself still plays a special role as a root domain. Other domains rely on its functionality, basically loading of kernel-modules, to be put into operation.

In order to dispatch external events in a prioritized manner, Adeos offers the possiblity for stalling events. That means, during interrupt handling in stage "n", domain "n" can block inter-

[2]http://home.gna.org/adeos/

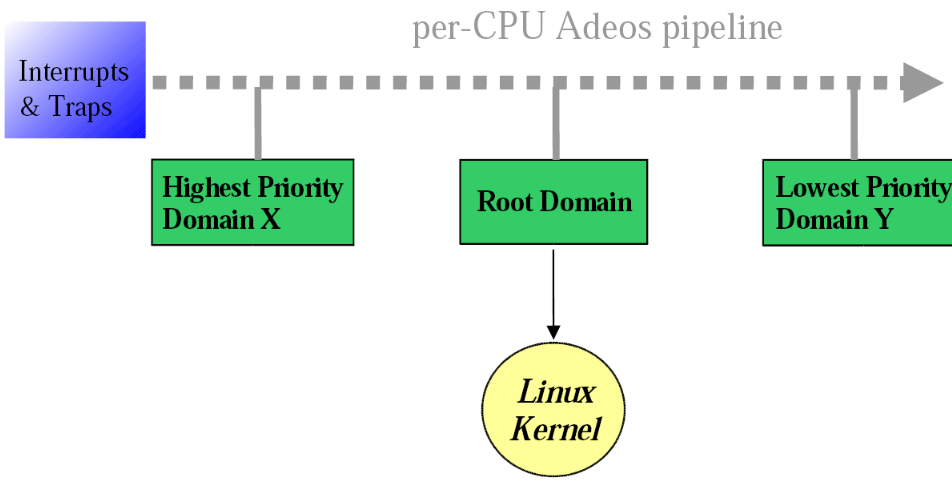

Figure 3.7: Adeos I-PIPE [12]

rupt propagation for all stages "k, k > n", providing domain "0" is the one owning the highest priority. Following this principle, a domain with a lower priority is not allowed to preempt another higher prioritized one. Interrupts are simply stored in a per-CPU log file and are dispatched after calling a special Adeos unstall routine. As a consequence of this, a standard Linux kernel blocking interrupts for performing critical sections, does not influence realtime tasks embodied in a domain lying ahead of it in the interrupt pipeline.

This "deferred" interrupt handling is limited to external events. Internal events generated by the standard Linux kernel, such as a page fault or other kind of traps, are synchronous by essence. There is no way to stall them in the interrupt pipeline since they need to be handled immediately.

Implementation details for the x86-platform can be found in [11].

3.4.3 Scheduling

RTAI comes along with its own schedulers, in order not to be limited by the capabilities of the standard Linux scheduler. It offers the possiblity to choose between two different schedulers, implemented in the modules rtai_sched.ko and rtai_lxrt.ko. These two approaches differ only in the type of objects they can schedule in Kernel-Space.[14] In User-Space only applications both schedulers are equal.

With rtai_sched.ko, specialised lightweight RTAI kernel threads can be scheduled as well as any schedulable Linux object like Linux processes, Linux threads and Linux kernel threads.

The purpose of the LinuX RealTime (LXRT) Scheduler in rtai_lxrt.ko is to offer a symmetric API to realtime RTAI tasks and Linux processes. User-Space applications based on this API can simply be transfered to hard realtime context by calling "rt_make_hard_realtime()". Basically, the usage of LXRT creates always kernel-threads which can then be scheduled by the realtime scheduler of RTAI. This allows to achieve hard realtime performance with slightly worse execution times, caused by management overhead, than that of RTAI realtime tasks. Obviously, this is a great advantage when developing new realtime applications. User-Space debugging tools are available and during the development process the code runs in a safe environment. After reaching a stable release the application finally may be migrated to a RTAI realtime task easily because of the symmetry of the LXRT API.

However, when making a Linux system call the User-Space application is migrated back to Linux and underlies the standard Linux scheduler. When the service request is fulfilled the task is handed back to the RTAI scheduler. Thus, using standard POSIX system calls should be avoided because the task will loose its hard realtime context. Instead, the RTAI API should be used in order not to be switched to Linux. Due to the fact, that the overall granularity of the Linux kernel steadily increases (see chapter 2), these switches will get less time consuming and more deterministic in future.

More detailed information about how the two different schedulers are working can be found in [14].

3.4.4 API

The RTAI API offers various functions for constructing, managing, and destructing realtime tasks as well as dealing with external interrupts. Moreover, the LXRT API provides hard realtime tasks in User-Space. The RTAI API is distributed among 22 kernel modules in version 3.4. During configuration progress, each feature can be enabled/disabled separately. The following list gives an idea about the major functionalities.

- realtime interrupt handling and clocking services (rtai_hal.ko)
- safely managing interrupts in User-Space (rtai_usi.ko)
- scheduling services
  - preemptive scheduling policy, fixed external priorities, round robin or fifo in the same priority class
  - hard realtime scheduling of RTAI lightweight realtime tasks (rtai_sched.ko)
  - hard realtime scheduling of all Linux schedulable objects (rtai_lxrt.ko)
- interprocess data transfer
  - mail boxes (rtai_mbx.ko)
  - messages (rtai_msg.ko)
  - RTAI message queues (rtai_tbx.ko)
  - POSIX like message queues (rtai_mq.ko)
  - shared memory (rtai_shm.ko)
  - realtime FIFOs (rtai_fifos.ko)

  - NetRPC for realtime network communication (rtai_netrpc.ko)
- interprocess synchronisation
  - semaphores (rtai_sem.ko), support for full priority inheritance
  - event flags (rtai_bits.ko)
  - signals (rtai_signals.ko)
  - tasklets (simple and timed), used when functions are needed to be called from User- and Kernel-Space (rtai_tasklets.ko)
- realtime driver model (RTDM) support, offers a unique API for device driver development (rtai_rtdm.ko)
- realtime drivers for serial port (rtai_serial.ko) and 16550A UART (rtai_16550A.ko, RTDM based)
- realtime memory managment (rtai_malloc.ko)
- in kernel floating point support (rtai_math.ko)
- software watchdog functionality for monitoring realtime tasks (rtai_wd.ko)
- LED debugging support (rtai_leds.ko)

A POSIX conform API is always a highly desirable design goal. In fact, RTAI offers a subset of the Realtime POSIX Extensions proposed in IEEE 1003.1b-1993 and IEEE 1003.1d-1999. By the time of writing this paper, RTAI 3.4 offers the following POSIX functions:

- POSIX threads
- POSIX semaphores
- POSIX mutex
- POSIX barriers
- POSIX condition variables
- POSIX locks (reader/writer locks and spinlocks)
- POSIX message queues

The implementation of POSIX conform shared memory and timer functions is planned for the next releases.

3.4.5 Getting Started

Starting with a plain vanilla Linux kernel[3], a suitable kernel patch, included in the RTAI distribution, has to be applied. During kernel configuration, some options may have a bad influence on realtime characteristics of the system and should be therefore changed. The following list gives an idea about relevant parameters, but it is not exhaustive.

Interrupt pipeline (Processor Type and Features) enables the usage of the Adeos I-Pipe

Use register arguments (Processor Type and Features) should be disabled

Module versioning support (Loadable module support) should be disabled

Power Management support (Power management options (ACPI, APM)) should be disabled

CPU Frequency scaling ((Power management options (ACPI, APM))) should be disabled

After compiling and installing the patched kernel, the RTAI modules can be compiled and installed. There were no serious problems during the setup of the RTAI based system. The only thing, which had to be fixed, were the non-persistent realtime devices in the "/dev"-directory. They had to be recreated at every system boot by a simple bash script.

[3]http://kernel.org

3.5 RTLinux/Free

RTLinux has been developed by Professor Victor Yodaiken and Michael Barabanov during his master thesis, in the year 1997, at New Mexico Institute of Mining and Technology. [16] As already illustrated in figure 3.1, it was the first one of the so called realtime Linux extensions. In the year 1999, the FSMLabs [4] company was founded, starting with a commercial RTLinux distribution (RTLinux/Pro). Nevertheless, a community RTLinux version named RTLinux/Free [5], under the terms of the GPL and Open RTLinux Patent License is still available.

In the beginning of writing this paper, the goal was also to evaluate RTLinux/Free. However, a systematic investigation of recent RTLinux/Free distributions turned out that this project seems not to be well maintained anymore. The latest release (3.2rc1) dates from March 15, 2005, and a Linux kernel patch 2.6.9 was the only one available for standard Linux kernel 2.6. A quantitative comparison with Xenomai and RTAI would not have been very useful because of the different kernel versions. The latest release provides support for i386, Alpha, MIPS and Power PC.

It seems as if maintenance of RTLinux/Free has been given up at the expense of RTLinux/Pro. Another reason could be that some developers have moved to the RTAI or Xenomai project. A free evaluation version of RTLinux/Pro can be downloaded at [17]. An evaluation of this commercial product, regarding its realtime capabilities in comparison with the non-commercial distributions, would be interesting. It could be the focus of another subsequent paper.

[4] `http://www.fsmlabs.com/`

[5] `http://www.rtlinuxfree.com`, `http://www.rtlinux-gpl.org`

4 Measurements

The purpose of this work is mostly to present a quantitative comparison of different Linux realtime extensions. Throughout the last chapter, functionality aspects of Xenomai, and RTAI have been discussed with respect to the basic principles which they are based on. In order to draw conclusions about the performance, parameters have to be defined which are important in a realtime environment. Basically, this paper deals with external interrupt latencies and scheduling latencies. Other parameters which can give information about the quality of realtime capabilities are for example context switch times, process dispatch latencies or the time that is needed to allocate a certain amount of memory. Furthermore, the terms and conditions for tests have to be well defined e.g. architecture of the test system, system load, etc..

4.1 Test System

In the beginning, a standard desktop pc (x86) has been used for the measurements. The appropriate setup of the Linux distribution is much easier and suitable kernel patches are available. Later on, these measurements are done on an embedded Linux system (Coldfire CPU), as far as each distribution is runnable on this platform. The test system consists of:

- Intel(R) Pentium(R) 4 CPU 2.80GHz
- 1024 MB main memory
- base kernel: 2.6.17 and 2.6.18
- 100 Mbit/s Ethernet

The realtime Linux distributions were used in the following versions.

- Realtime Preemption Patches by Ingo Molnar, Patch "patch-2.6.18-rt7" in connection with vanilla Linux 2.6.18
- Xenomai 2.2.4, Adeos I-PIPE Patch "adeos-ipipe-2.6.17-i386-1.5-00.patch"
- RTAI 3.4, Patch "hal-linux-2.6.17-i386-1.3-08.patch"

The load on the x86 test system was applied in the following way:

- I/O-load: Flood-Ping to the test system and usage of "dd"
- CPU-load: CPU Burn-in v1.00 [1]

[1] http://users.bigpond.net.au/cpuburn/

4.2 Interrupt Latencies

4.2.1 Parallel-Port Loopback Connector

The measurement of the response time to external interrupts gives a good idea about realtime capabilities. The time between the occurrence of an external signal and the start of the corresponding interrupt service routine (ISR) is measured. Therefore, the author used a parallelport-loopback connector which was already used by Thomas Wiedemann in [13] for measuring interrupt latencies with plain Linux. The modified connector is shown in figure 4.1.

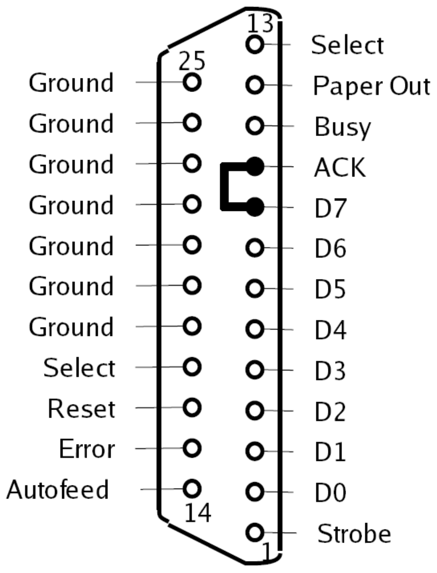

Figure 4.1: Parallel-Port Loopback Connector [13]

In a first cycle, latencies were measured over a standard Linux 2.6.17 kernel without any preemption switched on. Afterwards, "CONFIG_PREEMPT" was enabled in order to evaluate the improvements of preemptibility of the Linux kernel. Finally, the kernel module has been reeimplemented to work on top of the different realtime Linux API's. All measurements were made according to an interrupt frequency of $1ms$ and under heavy stress, by means of I/O- and CPU-load. The results are illustrated throughout the histograms 4.2 to 4.4.

The results from standard Linux are in conflict with the theory. The kernel without native preemption shows the lowest maximum latencies followed by the one with native preemption activated and the Linux kernel 2.6.18 in connection with the Realtime Preemption Patch from Ingo Molnar. A reason for this behaviour could be that the modifications in the Linux kernel induce some management overhead, as already stated in chapter 2. However, in summary we will get bounded worst case response times, which may be higher than the one without the preemption patches, which are not guaranteed. The measurements have been done over a period of around 1 minute. This time is probably also too short in order to trap into the worst case response time of the kernel without any preemption.

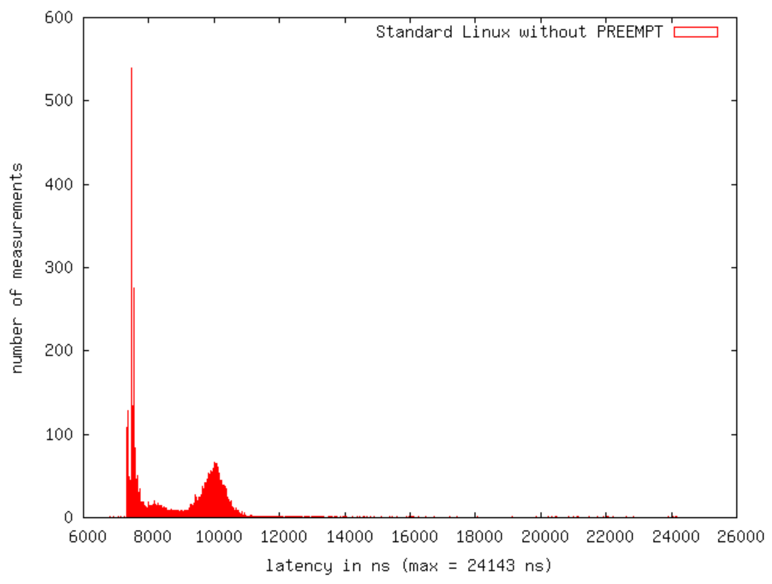

Figure 4.2: Standard Linux 2.6.17 without native preemption

The histograms in figures 4.5 to 4.8 show the interrupt response times of Xenomai and RTAI. The maximum latencies are about 30 % lower than the unpatched vanilla Linux kernel without any preemption and 95 % lower than in histogram 4.4. This is the most important thing for realtime applications. Nevertheless, the minimum and average case response times are slightly above the once of the unpatched Linux kernel. This could be explained by the management overhead, which is incured by the Adeos interrupt pipeline.

Xenomai shows higher latencies if native preemptibility of the Linux kernel is activated, although the interrupt reply is initiated in interrupt handler context. The native preemption should only provide better results and only when a switch to the Secondary Domain occurs . This could be a sign that "CONFIG_PREEMPT" worsens the cache hit rate. It has to be kept in mind that it does not make any sense to switch on "CONFIG_PREEMPT" if the realtime thread does not issue any Linux system call, which is the case in the interrupt latency benchmark.

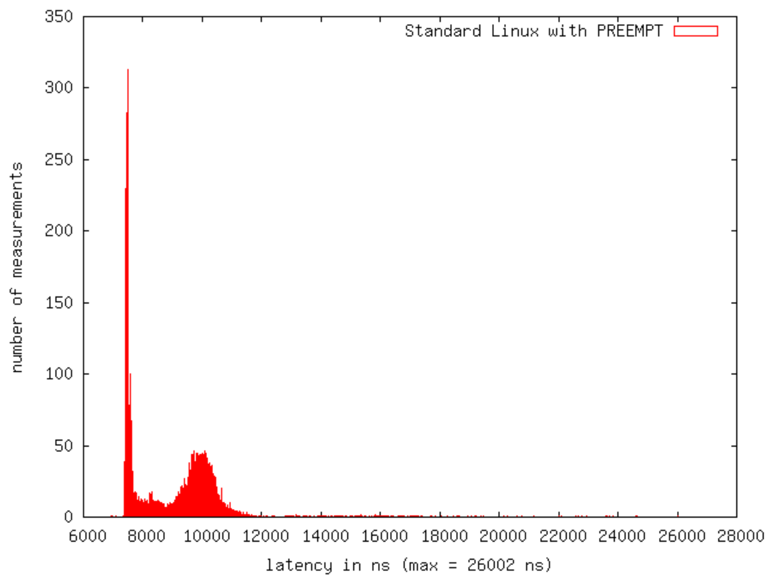

Figure 4.3: Standard Linux 2.6.17 with native preemption

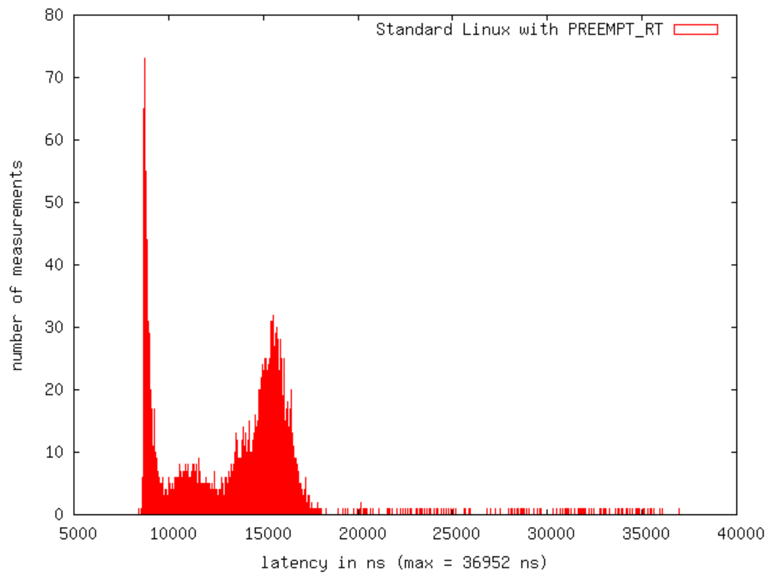

Figure 4.4: Standard Linux 2.6.18 with realtime preemption patch

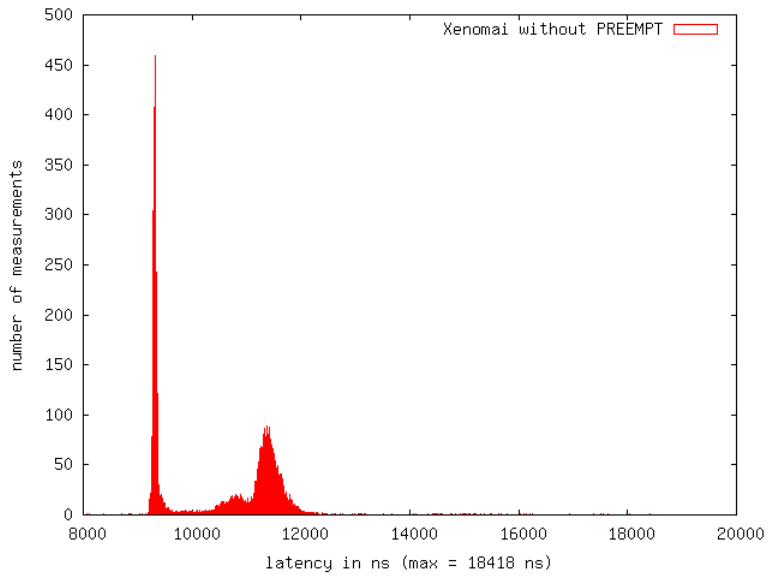

Figure 4.5: Xenomai without native preemption

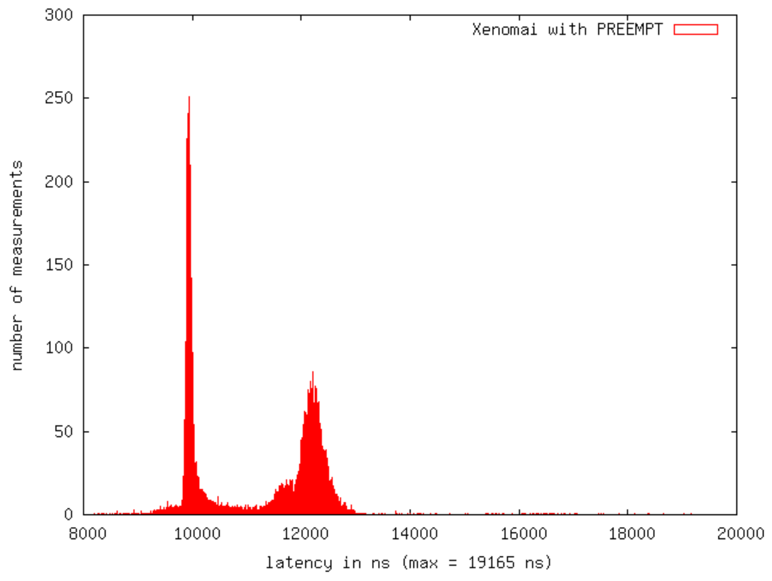

Figure 4.6: Xenomai with native preemption

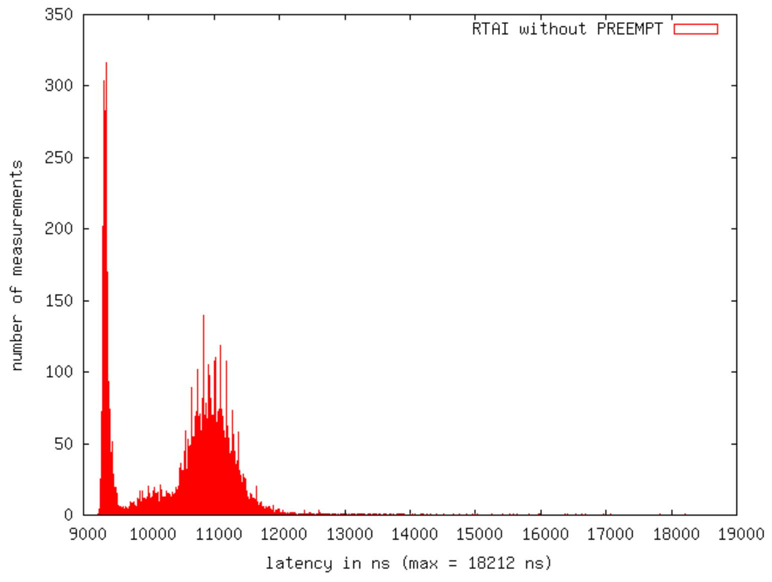

Figure 4.7: RTAI without native preemption

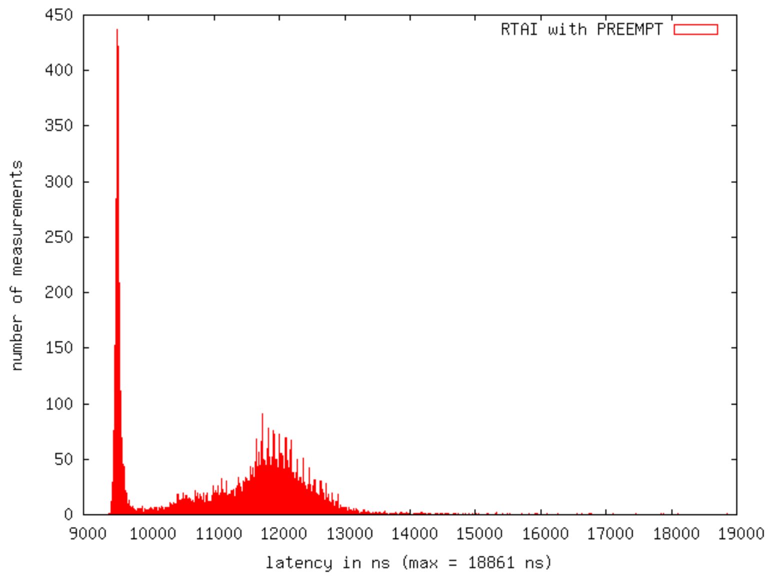

Figure 4.8: RTAI with native preemption

4.2.2 Serial-Port Crossover Cable

The Xenomai distribution includes a standard testsuite, which offers various types of programs for evaluating realtime performance. The measurement of external interrupt latencies is also provided. Therefore, 2 systems are needed, connected via a serial port crossover cable (pins "CTS" and "RTS" are crossed). The first system, named log system, generates interrupts on the serial port line, which the test system has to acknowledge. The time between the generation of the interrupt and its corresponding acknowledgement by the test system is measured.

There are 4 different test modes available, and they differ in the context where the interrupt reply occurs. In User-Space and Kernel-Space the procedure is obvious. The "IRQ Handler"-test replies to interrupts directly in the interrupt service routine. The last mode, named "hard-IRQ Handler", installs an own Adeos domain on top of the interrupt pipeline. This mode always gives highest realtime performance, but lacks therefore any realtime operating system support from Xenomai. That's why it is only suitable for small jobs, which can be done easily in an interrupt handler. Latency measurements for this method have been omitted, because it can be used only in a very limited number of applications.

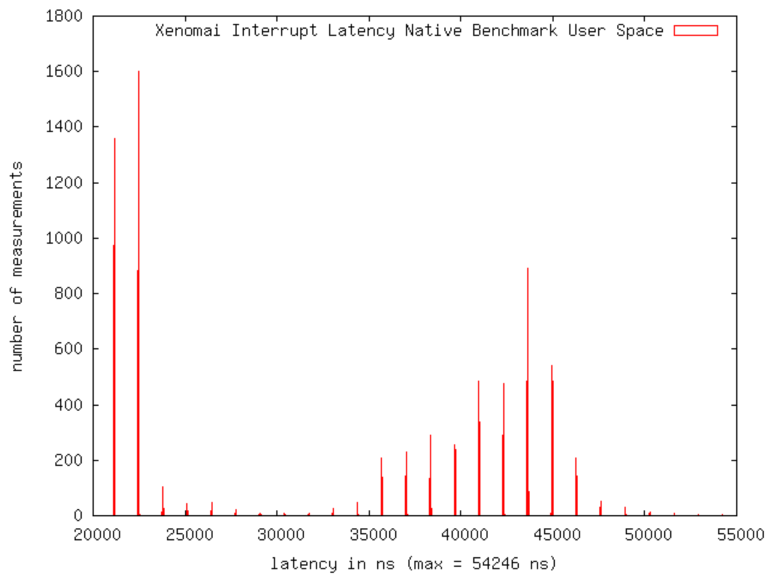

Figure 4.9: Xenomai Native Interrupt Latency Test (User-Space Task)

The histograms from figure 4.9 to 4.11 show the results of the interrupt latency measurements resolved from the Xenomai testsuite. They are a few microseconds above the latencies from the parallel port loopback device (see figures 4.5 to 4.6). However, the values should be more appropriate with this method of measuring. The advantage of this approach is that the part of the interrupt generation is fully independent, and therefore can not influence the measurements. If the measurements of external interrupts is performed via the parallelport loopback connector, the system is always moved into the same state right before interrupt generation. Therefore, the results in figures 4.9 to 4.11 could be more reasonable.

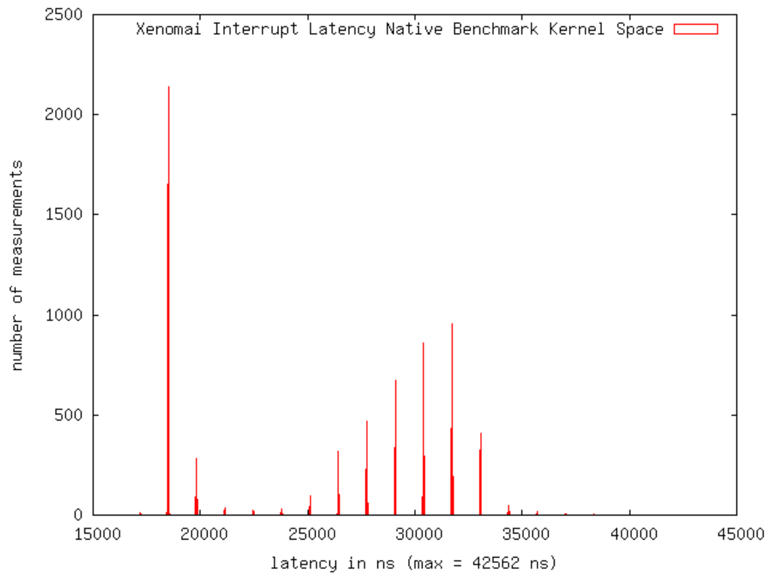

Figure 4.10: Xenomai Native Interrupt Latency Test (Kernel-Space Task)

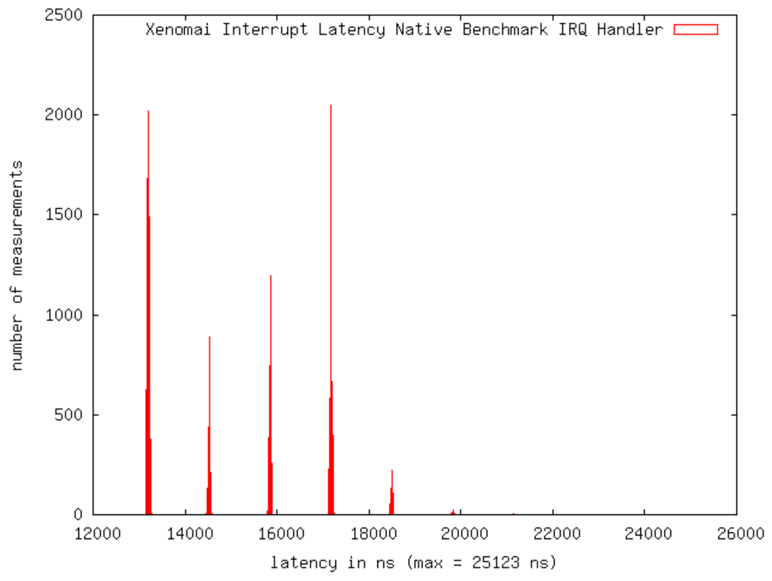

Figure 4.11: Xenomai Native Interrupt Latency Test (IRQ Handler)

4.3 Scheduling Latencies

In a second step, a periodic task was set up and the jitter of activation times has been monitored. The standard Linux kernel was compiled with the standard timer accuracy of 250 Hz ($4ms$). However, the resolution is quite coarse grained for realtime applications. That's why the period of the task was adjusted to $10ms$. The measurements have been done on a clean vanilla Linux kernel and on a kernel after the application of the Realtime Preemption Patches, by Ingo Molnar. Furthermore, the periodic job was started once with "SCHED_FIFO" and once with "SCHED_OTHER" policy. The results are shown in histograms 4.12 to 4.15.

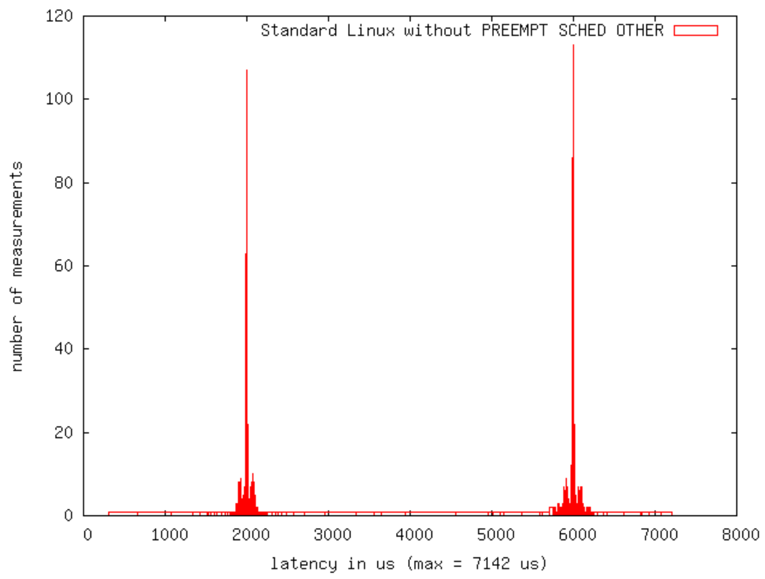

Figure 4.12: Standard Linux without Preemption ("SCHED_OTHER")

The measurements of the standard Linux kernel (patched or unpatched) show that the "SCHED_FIFO" policy has a positive influence on the worst case scheduling jitter. The gap of $4ms$ between the two peaks represents the timer resolution of 250 Hz. The "PREEMPT_RT" patched kernel shows the lowest worst case scheduling jitter with "SCHED_FIFO" policy and around two-third of the samples are in the interval of around 2 ms. Also the scattering of the measurements is much lower than with "SCHED_OTHER" policy.

With Xenomai and RTAI the timer resolution is much more accurate, allowing to set up a task running with a period of $100\mu s$. All measurements have been done over a period of 1 minute (600000 samples) and under heavy stress, by means of I/O- and CPU-load. The results are illustrated throughout the histograms 4.16 to 4.20:

With Xenomai and RTAI much better results are achievable. The measurements with a periodic timer interrupt handler show the lowest scheduling jitter. The User- and Kernel-Space task is slightly worse but delivers also reasonable latencies.

The results of the periodic User-Space task with RTAI are similar to its Xenomai counterpart. Both are nearly equivalent. Nevertheless, the measurements of the RTAI Kernel-Space task look

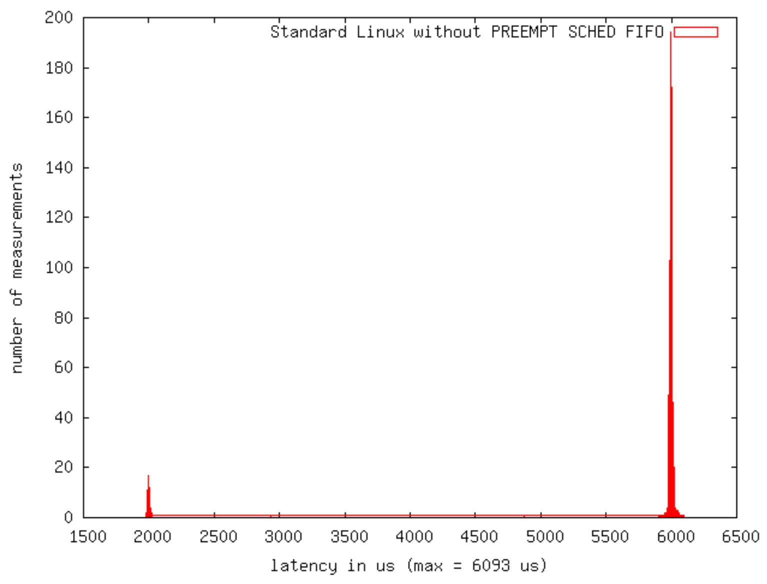

Figure 4.13: Standard Linux without Preemption ("SCHED_FIFO")

quite impressive. Nearly all latencies are below $2\mu s$ which is much better than the Xenomai Kernel-Space task. A reason therefore could be the different implementation of the Kernel-Space latency task in RTAI.

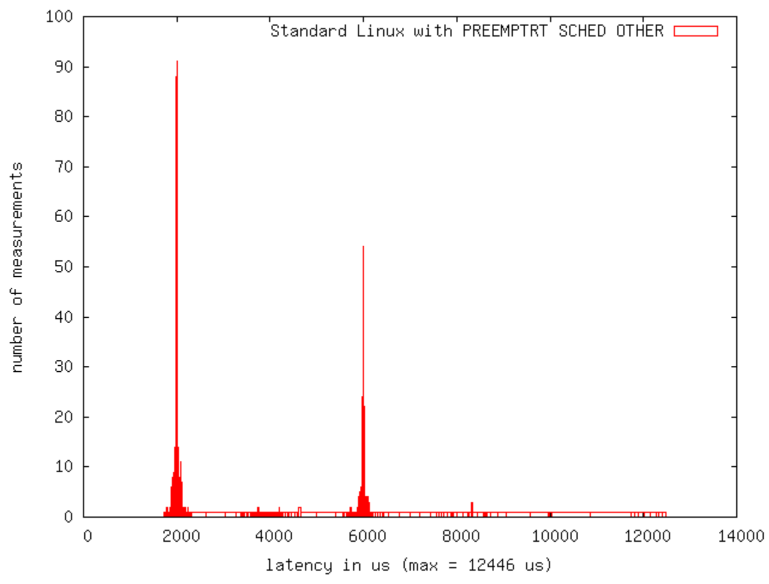

Figure 4.14: Standard Linux with Realtime Preemption Patch ("SCHED_OTHER")

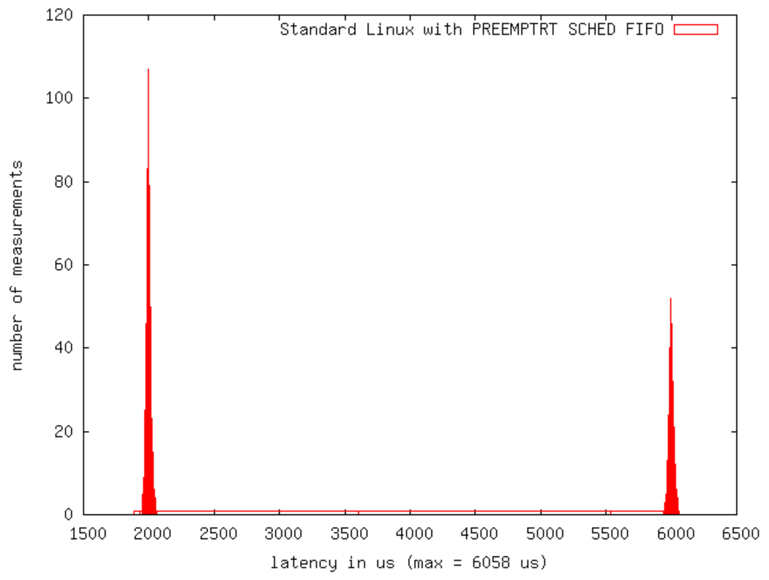

Figure 4.15: Standard Linux with Realtime Preemption Patch ("SCHED_FIFO")

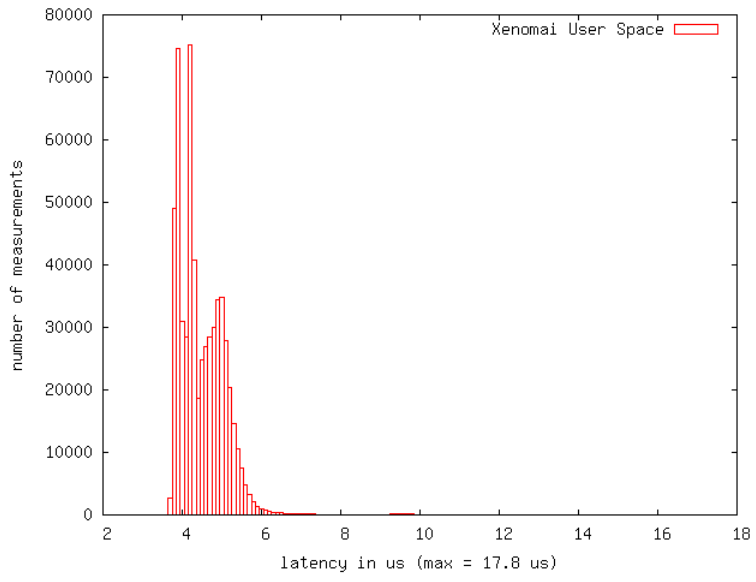

Figure 4.16: Xenomai User-Space Task

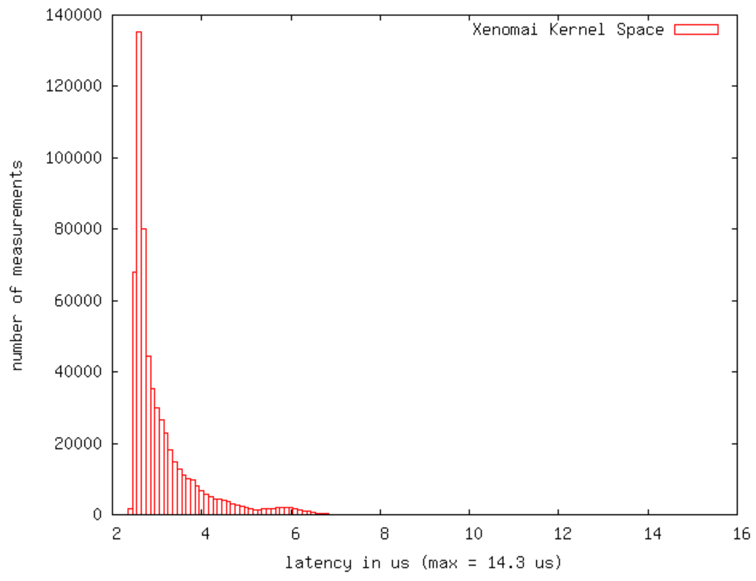

Figure 4.17: Xenomai Kernel-Space Task

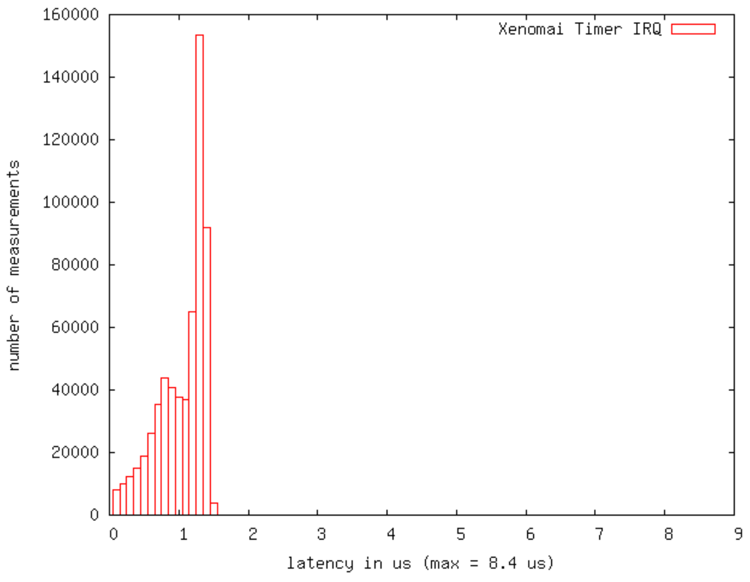

Figure 4.18: Xenomai Timer IRQ

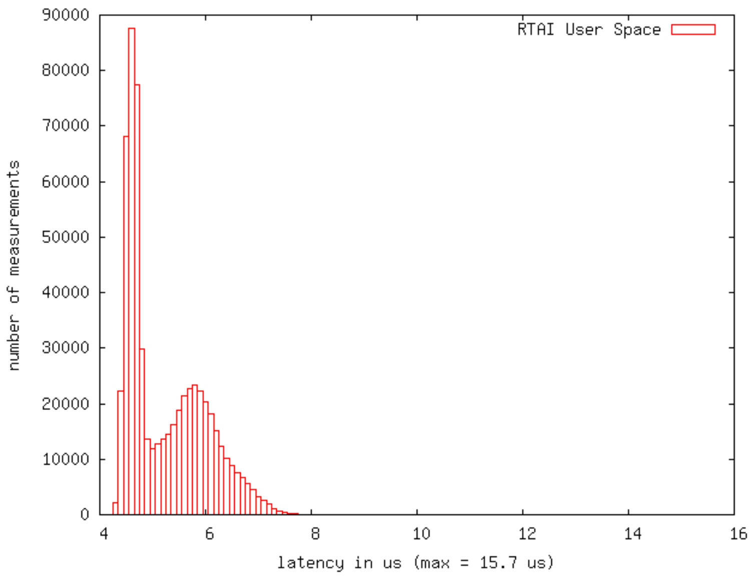

Figure 4.19: RTAI User-Space Task

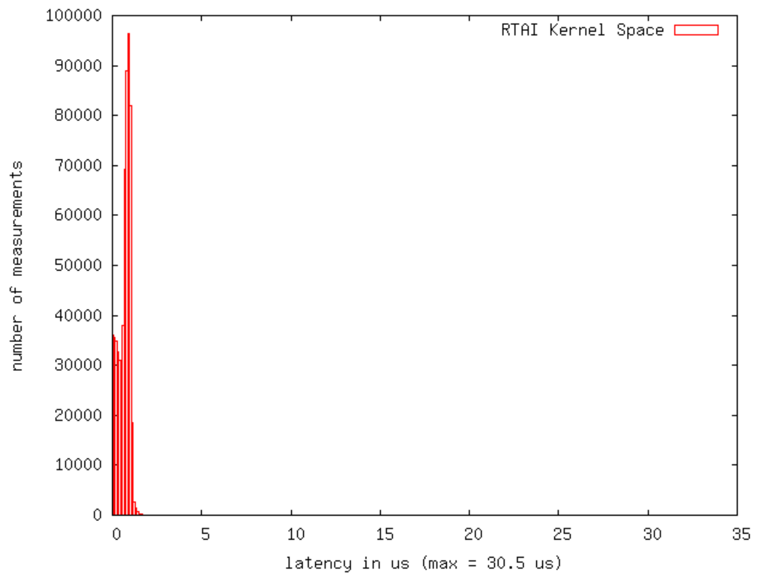

Figure 4.20: RTAI Kernel-Space Task

4.4 Context Switch Time

Xenomai and RTAI can benefit from the improvements of native preemptibility in the Linux kernel by means of "CONFIG_PREEMPT" and "CONFIG_PREEMPT_VOLUNTARY". If Linux kernel functions are used, a realtime task is transparently migrated from hard realtime to soft realtime context. Soft realtime means that the task is now under the control of the Linux scheduler. After the service delivery, the task will be migrated back into hard realtime context, as soon as possible.

In order to measure the improvements of native preemptibility of the Linux kernel, a periodic task (1 ms) was set up, which switches between RTAI/Xenomai context and Linux context, by doing a Linux system call (gettimeofday). The time for this mode switch was measured once with a RTAI/Xenomai kernel with "CONFIG_PREEMPT" enabled and once without preemption. (see figures 4.21 to 4.24) For clarification reasons, the x-coordinate in these histograms is limited to $50\mu s$, although the worst case latency is much higher. The measurements were done under high stress by means of I/O- and CPU-load. An application of Ingo Molnar's Realtime Preemption Patch was not possible because of incompatibilities with the Xenomai/RTAI Adeos I-PIPE patch. Nevertheless, the collaboration of these two patches is focussed with Xenomai in the future.

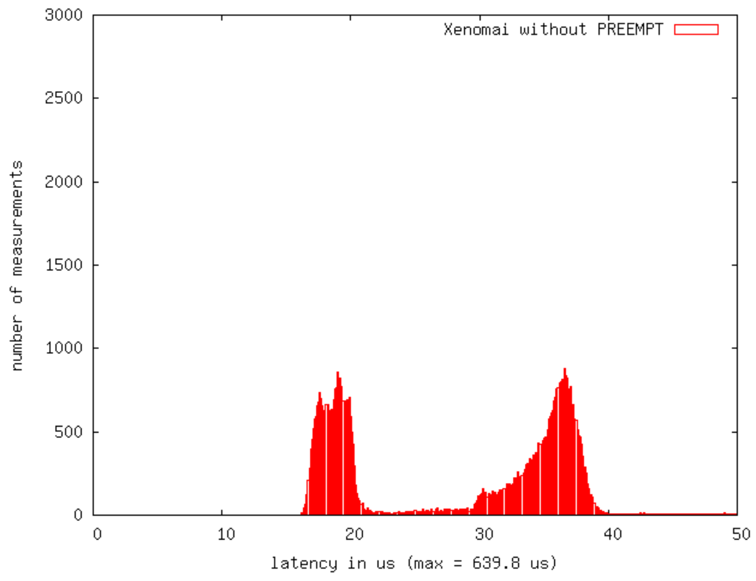

Figure 4.21: Xenomai Context Switch Times (without Preemption)

The results show that native preemption support provides better RTAI/Xenomai <-> Linux switch times. Also, the worst case switch times are lower, if a Linux kernel with native preemption support is used. The performance with Xenomai is slightly better in comparison with RTAI. With respect to future development and integration of Ingo Molnar's Realtime Preemption Patches, the mode switches will be much more cheaper as the preemptibility of the Linux kernel steadily increases. Nevertheless, the results show that worst case switch times are pretty

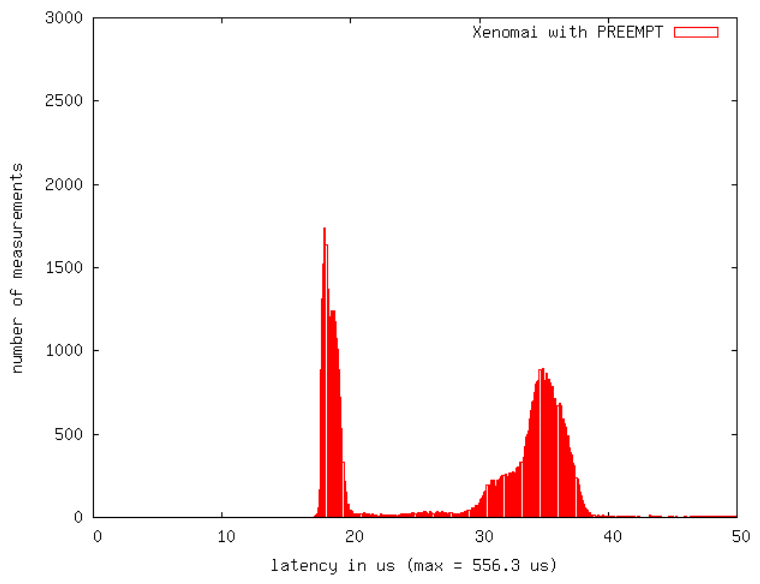

Figure 4.22: Xenomai Context Switch Times (with Preemption)

high ($1463,5\mu s$) with respect to the period of the task ($1ms$). That means, native preemptibility only improves the average case, but it is not suitable for guaranteeing worst case switch times.

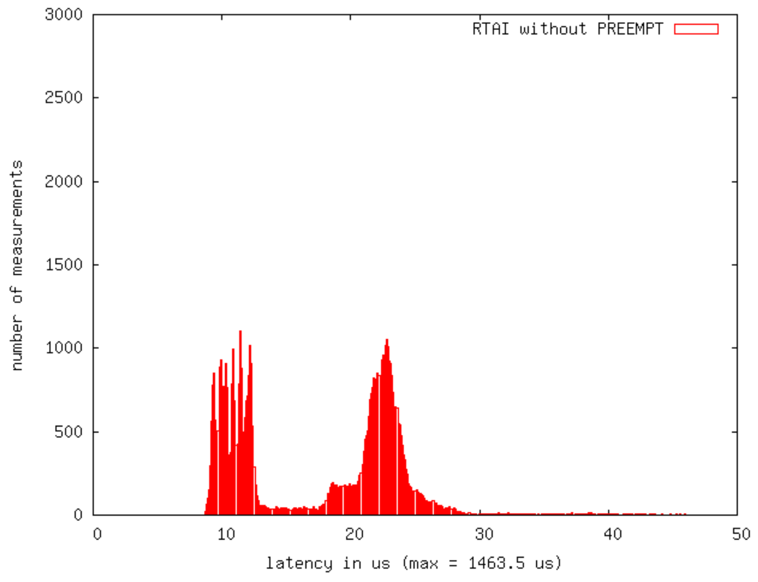

Figure 4.23: RTAI Context Switch Times (without Preemption)

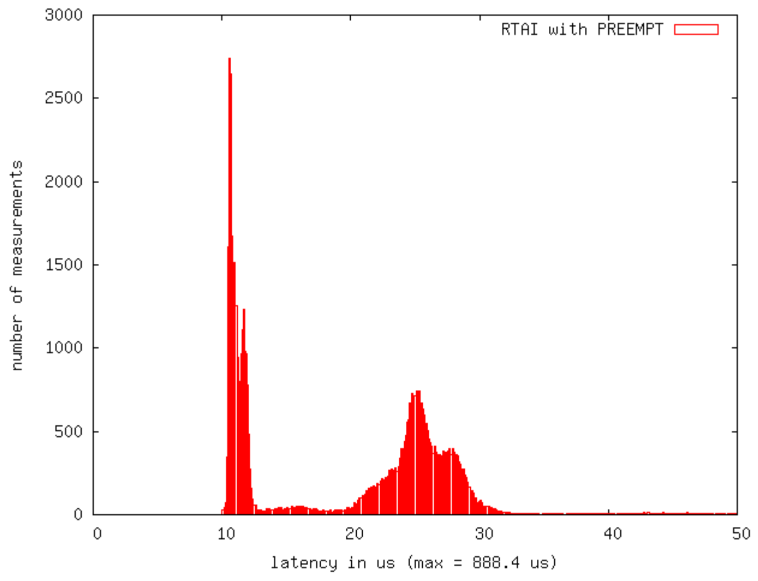

Figure 4.24: RTAI Context Switch Times (with Preemption)

5 Summary

The paper has mainly discussed two major realtime Linux extension throughout chapter 3. A short overview about the state-of-the-art development in RTAI, Xenomai and RTLinux/Free has been given. Especially, RTAI and Xenomai are under very high development and a lot of literature, which can be found, is already obsolete. Nevertheless, the presented information refer to the latest available releases of each realtime Linux distribution.

The analysis of RTLinux/Free was finally omitted in this paper, because the project seems to be not mainained anymore, with respect to patches for recent kernel versions. At the time of writing this paper, the mailing list archives were not accessible, and there were only very few postings throughout the last weeks. In conclusion, RTLinux/Free is not suitable for beeing used within industrial projects. The termination of the project's development process is too risky. Furthermore, RTLinux/Free can only be used for free within GPL projects. For commercial closed source applications a license from FSMLabs has to be obtained.

In contrast to this, Xenomai and RTAI are maintained very well. Questions on the mailing list are always answered very quickly, mainly by the core developers of each project. The list of supported architectures is much longer and kernel patches are available for lots of kernel versions, either Linux 2.6 or 2.4. Xenomai and RTAI are also well suited to be used in closed source commercial products. The User-Space libraries are under the terms of the GNU Lesser General Public Licence (LGPL), which means that every User-Space application linked against these libraries may be distributed under a non-GPL license. Nevertheless, realtime applications embedded into Linux kernel modules have to be redistributed under GPL again.

Chapter 4 illustrated some representative measurements of realtime capabilities. There are lots of more parameters which could be measured, in order to evaluate the realtime performance. Some measurements have been done also with plain Linux without any support of a realtime extension. These results are used as a base for the comparison of the improvements with the dedicated distributions. The response time to external interrupts are very similar with Xenomai and RTAI. The results of the scheduling latency measurements are a bit different. In User-Space there is no difference between Xenomai and RTAI. However, if using a Kernel-Space task, RTAI provides much better performance than Xenomai. Having a look at the context switch times to Linux, Xenomai is slightly better than RTAI.

To sum up, Xenomai and RTAI are nearly equivalent in performance and feature support even in the field of User-Space realtime applications. The only advantage of Xenomai is the more flexible architecture by means of several different skins. It owns therefore the possibility to run legacy RTOS over Xenomai, offering a soft migration process to the Xenomai architecture.

Bibliography

[1] **Jane W. S. Liu:** *Real-Time Systems* Prentice Hall (2000). ISBN 0-13-099651-3

[2] **Daniel P. Bovet, Marco Cesati:** *Understanding the Linux Kernel, 3rd Edition* O'Reilly (2005). ISBN 0-596-00565-2

[3] **Arnd C. Heursch, Alexander Horstkotte and Helmut Rzehak:** *Preemption concepts, Rhealstone Benchmark and scheduler analysis of Linux 2.4* published on the Real-Time & Embedded Computing Conference, Milan, November 27-28, 2001.

[4] **Clark Williams, Red Hat, Inc.:** *Linux Scheduler Latency* March 2002.

[5] `http://lwn.net/Articles/146861/` date: 08/12/2006

[6] **Xenomai FAQ:** `http://www.xenomai.org/index.php/FAQs` date: 09/12/2006

[7] **Xenomai Native API Tour:** `http://www.xenomai.org/documentation/branches/v2.0.x/pdf/Native-API-Tour.pdf` date: 09/12/2006

[8] **Xenomai Whitepaper:** `http://www.xenomai.org/documentation/branches/v2.0.x/pdf/xenomai.pdf` date: 13/12/2006

[9] **Adam M. Costello and George Varghese:** *Redesigning the BSD Timer Facilities* published in SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(8), 883–896 (10 JULY 1998).

[10] **Marshall Kirk McKusick, Michael J. Karels:** *Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel* published in Proceedings of the San Francisco USENIX Conference, pp. 295-303, June 1988.

[11] **Karim Yaghmour:** *Adaptive Domain Environment for Operating Systems*

[12] **Life with Adeos:** `http://www.xenomai.org/documentation/branches/v2.0.x/pdf/Life-with-Adeos.pdf` date: 13/12/2006

[13] **Thomas Wiedemann:** *Seminar Paper: How Fast Can Computers React?* date: 21/12/2005

[14] **RTAI 3.4 User Manual rev 0.3** date: 10/04/2006

[15] **Karim Yaghmour:** *The Real-Time Application Interface* date: published in 2001

[16] **Michael Barabanov:** *A Linux-based Real-Time Operating System* date: published in June 1, 1997

[17] **Free Evaluation Version of RTLinux/Pro** `http://www.fsmlabs.com/order.html` date: 20/02/2007

Declaration of Authorship

I hereby declare that the whole of this document is my own work unless explicitly stated otherwise in the text.

I declare that this work has not been submitted in whole or in part for any other degree.

Chemnitz, March 5, 2007

____

Franke Markus