Preemption concepts, Rhealstone Benchmark and scheduler analysis of Linux 2.4

Arnd C. Heursch, Alexander Horstkotte and Helmut Rzehak
Department of Computer Science
University of Federal Armed Forces, Munich
Werner-Heisenberg-Weg 39, 85577 Neubiberg, Germany
{heursch, rz}@informatik.unibw-muenchen.de, alexander.horstkotte@unibw-muenchen.de
This paper has been published first on the Real-Time & Embedded Computing Conference, Milan, November 27-28, 2001, http://www.rtcgroup.com/summitmilan/invitation.html

Abstract

In this paper we examine the soft realtime capabilities of the Linux kernel 2.4 and realtime enhancements. We discuss and visualize in the form of PDLT (process dispatch latency time) diagrams how the MontaVista Preemption Patch 6 for Linux 2.4.2 preempts a Linux system call executing in kernel mode. In similar diagrams we show the effect of introducing Conditional Preemption Points, like Ingo Molnar´s and Andrew Morton´s Low Latency Patches do. To detect possible performance changes we measure some programs of the Rhealstone Benchmark on Linux kernels and variants.
We examine the behaviour of the Linux scheduler when scheduling a SCHED_FIFO process varying the number of SCHED_OTHER load processes ready to run.

1 Introduction

This paper deals with concepts to make the Linux kernel more responsive regarding soft realtime tasks. Linux kernel code is not preemptable in general. Section 2 presents two approaches to lower the latencies induced by the non-preemptable standard Linux kernel, it discusses pros and cons of the concepts, and presents some measurements we made to test whether and how the kernel patches provided work.

In section 3 we use some programs of the Rhealstone Benchmark to test whether the kernel patches that make the kernel more preemptive may cost performance compared to the standard Linux kernels.

In section 4 we present measurements of the Linux scheduler. We create a situation, where we can analyze how fast a soft realtime process, scheduled by the standard Linux SCHED_FIFO or SCHED_RR scheduling policy, can be scheduled, when there are many non-realtime load processes.

2 Concepts to improve the preemption of the kernel

2.1 Non-preemptable regions in the Linux kernel

In Linux a SCHED_FIFO process, i.e. a soft realtime process, needs only about some microseconds to preempt a process of lower priority on a PC with only one processor and current hardware, if this process with lower priority runs in user space. This situation is shown in line A1 of Figure 1. The Tables 1 and 2 in section 3 confirm this, the average Preemption Time measured in the Rhealstone Benchmark is about 1 microsecond, if processes are locked into memory.
But if the process to preempt currently executes a system call, i.e. kernel code which is not preemptable, the process of higher priority has to wait until the system function is fulfilled - see line A2 of Figure 1 - or until a preemption point in the code of the system function is reached. Furthermore, also a kernel thread could cause a SCHED_FIFO process to wait for some time, because Linux kernel code normally is not preemptable. As several measurements, f.ex. [19], [13], [8] , have shown, the longest latencies produced by kernel code on current PC´s are on the

order of tens up to hundreds of milliseconds. Line A1 and A2 of Fig.1 illustrate the situation in standard Linux kernels at the moment.
Actually there are two different concepts that try to lower those latencies, see also Fig. 1

- introducing Preemption Points into the kernel, see line B of Figure 1. Premption Points mean, that if there is a realtime request the execution of a system call can be interrupted at special points.
- making the kernel code preemptable in general, see line C of Fig.1, but with some exceptions.

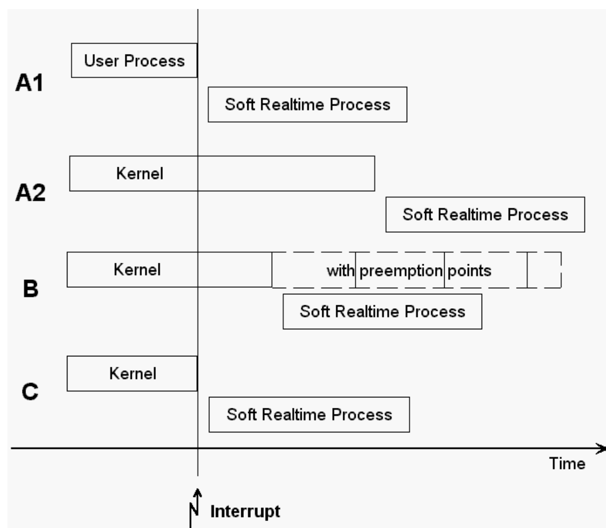

FIGURE 1: *Concepts to make the Linux kernel more responsive. A1 and A2: situation in the current standard Linux kernels 2.4. B: Preemption Points in the kernel, C: a preemptable kernel*

2.2 Introducing Preemption Points and rewriting parts of the kernel

Ingo Molnar [6] identified six sources of long latencies on the order of tens up to hundreds of milliseconds on current hardware in the Linux kernel:

- Calls to the disk buffer cache
- Memory page management
- Calls to the /proc file system
- VGA and console management
- The forking and exits of large processes
- The keyboard driver

He then rewrote parts of that code, f.ex. the keyboard driver. To reduce long latencies he introduced Conditional Preemption Points into the kernel code:

```
if (current->need_resched)
{
        current->state = TASK_RUNNING;
        schedule();
}
```

These Preemption Points only take effect, when the variable `current→need_resched` has been set, f.ex. an interrupt service routine often awakens a SCHED_FIFO process and sets this variable to '1' in order to force a rescheduling as soon as possible. Ingo Molnar provides all his changes as one kernel patch to the Linux kernel, called ´Low Latency Patch´ [6]. This patch has proven to lower several long latencies of the Linux kernel sucessfully [19, 13] to the order of 5 to 10 milliseconds on current hardware. Andrew Morton [7] wrote another patch to lower latencies in the Linux kernel 2.4 also using Preemption Points [9].
From the theoretical point of view three objections have been made to that concept:

1. Preemption Points must be placed at secure points in the kernel, badly positioned Preemption Points could cause system crashes of the whole Linux kernel because of data inconsistencies [11].
2. Ingo Molnar's 'Low Latency Patch' does not seem to affect the stability of the Linux kernel, although nobody can guarantee its total correctness, because nobody can test all execution paths in the kernel. This is a general problem of kernel patches.
3. Taking in mind the first two objections, it might be difficult to maintain these patches, also because they apply changes to many different regions of the kernel.

2.3 Trying to make the kernel preemptable in general

Another idea has been presented and carried out by MontaVista [12]. MontaVista´s preemption concept uses even on PC´s with only one processor the kernel lock concept that has been developed for SMP, for symmetric multiprocessing systems. That way, all kernel functions and kernel threads are made preemptable, but there are exceptions: Preemption can´t take place:

- while handling interrupts
- while doing Soft-IRQ and 'Bottom Half' processing

- while holding a spinlock, writelock or readlock. These locks have been put into the kernel to protect code executing on one processor from others in Symmetric Multiprocessing (SMP) Systems. While these locks are held the kernel is not preemptable for reentrancy or data protection reasons (just as in the SMP case)
- while the kernel is executing the scheduler
- while initializing a new process in the `fork()` system call

At all other times the MontaVista algorithm allows preemption. When the system enters one of the states shown above, a global Preemption-Lock-Counter is incremented, indicating that preemption is now forbidden. When leaving the state, f.ex. the lock is released, the Preemption-Lock-Counter is decremented and a test is made to see, whether preemption has been called for meanwhile. This means, after every such region, there is a Preemption Point included. If preemption has been called for, the current task is preempted. MontaVista points out that this ´polling for preemption´ at the end of a preemption-locked region can happen tens of thousands of times per second on an average system. Our measurements - see Tables 1 and 2 in section 3 - in fact show longer execution times of benchmark programs on Linux kernels patched with MontaVista´s Preemption patches.
We measured the Preemption Patch 1.5, Preemption Patch 6, and Preemption Patch 6 combined with Andrew Morton´s Low Latency Patch [12].
The following objections can be made to this concept of preemption:

- It is a good concept to be able to preempt kernel code in general. In this method no preemption points have to be introduced explicitly in the code to preempt, f.ex. into system calls as shown in line B of Fig.1. But unfortunately there are still long regions in the kernel spinlocked, which can't be preempted.

  A Preemption Point that invokes the scheduler can easily cope up with a spinlocked region, if well positioned, so that it does not affect the stability. Before it calls the function `schedule()` it unlocks the spinlock, when returning from `schedule()` it locks the spinlock again:

```
if (current->need_resched)
{
        spin_unlock(&special_lock);
        current->state = TASK_RUNNING;
        schedule();
        spin_lock(&special_lock);
}
```

  To cope with the long term spinlocked regions in the kernel - long in comparison to two context switches - MontaVista recently provided a new kernel patch where they combined their Preemption Patch 6 for Linux 2.4.2 with Andrew Morton´s 'Low Latency Patch', that uses Conditional Preemption Points to reduce long latencies.

- Another idea by MontaVista to tackle the problem of long term non-preemptable locked regions in the kernel is to change ´long´ held spinlocks -´long´ means much greater than the time taken by two context switches - into mutex locks, currently implemented using semaphores. MontaVista started this project, all these changes are released in a further ´mutex-*´ kernel patch, that can be combined with their Preemption Patch. MontaVista plans to implement a fast, priority inheriting, binary semaphore implementation of these mutex locks.

  The following objections can be made to this extended concept:

  – Recently Victor Yodaiken, the creator of RT-Linux, pointed out, that in special situations ´priority inheritance´ concepts can cause unpredictable long latencies because they can´t avoid ´priority inversion´ in all cases [4].
  – This is a long term project because many parts of the source code have to be altered, as much as in the case of the ´Low Latency Patches´, we discussed in the paragraph before, and it won´t be easier to validate such a changed kernel.

- In the Linux Kernel Mailing List kernel developers pointed out that it might be theoretically possible that MontaVista's Preemption Patches would deadlock at some parts in the current Linux 2.4 kernels. Changes necessary to the kernel code to fix the problem would need some time. [10]
- As the measurements indicate in Tables 1 and 2 in section 3, this concept of a preemptable kernel seems to cost performance.

2.4 Preemption of our test driver

In this section we present measurements in order to visualize the different concepts and to test how they work in a special case.

2.5 The measurement method

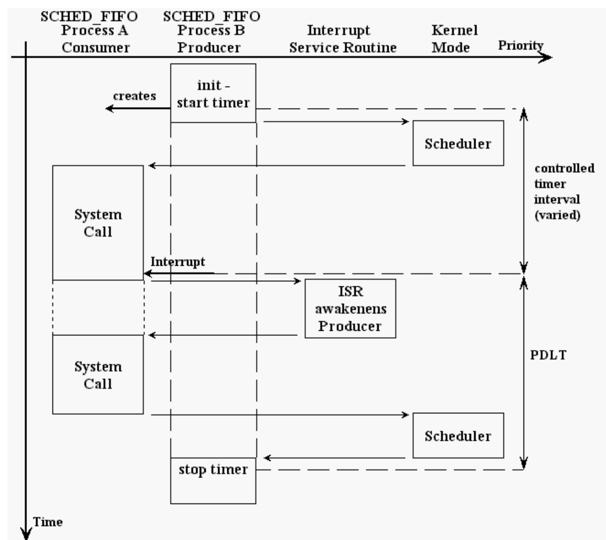

FIGURE 2: *illustration of the measurement method, called the 'Software Monitor'*

To visualize these effects we used a method developed by [16] that shows the PDLT, the process dispatch latency time, of the high priority process, when it tries to preempt a low priority process which executes f.ex. a system function. Non preemptable regions of code appear in these measurements as triangles added to the normal amount of PDLT defined by the preemption time of the Rhealstone Benchmark and by the overhead of the measurement method, see Fig.3.

2.6 Why do non-preemptable regions show up as triangles in the diagrams ?

A low priority SCHED_FIFO, i.e. a soft realtime process - called 'Consumer' in Fig.2 -, executes a system call, which runs in kernel mode. When an interrupt occurs a timer on a PCI timer card [18] is started. The interrupt service routine (ISR) following the interrupt interrupts the low priority SCHED_FIFO process - i.e. the 'Consumer' -, even if it executes kernel code, and awakens a high priority process - the 'Producer' in Fig.2 -, that preempts the low priority process as soon as possible. We assume here that the system call does not disable interrupts, if it does it disables them mostly only for short periods on the order of microseconds. After the system call is finished or a preemption point in the code of the system call is reached, the scheduler is invoked and starts the high priority SCHED_FIFO process - called 'Producer' - as the process of highest priority in the system ready to run.
As its first instruction the high priority process stops the timer and calculates the value of the PDLT, as shown in Fig.2. The PDLT is the time in between the interrupt occurs and the high priority process obtains the processor. This measurement procedure is repeated varying the length of the 'Controlled Timer Interval' in Fig. 2, i.e. varying the moment the interrupt occurs related to the start of the Consumer's system call. The interrupts are generated by our external PCI-Timer-Card [18]. That way the system call is scanned by generating an interrupt every microsecond. Every measurement generates one point in the diagram, see Figure 3. If the interrupt occurs just after the system call has been started, the PDLT triangle reaches its highest point, because the whole system call has to be fulfilled before the high priority process can start running. Interrupting later on, only shorter parts of the system call have still to be fulfilled, so the triangle decreases until it reaches the baseline of the triangle in the end of the system call. In the Figures 3-8 the execution time of the system call is marked by a thick line on the X-axis.

2.7 Preemption of a kernel driver method

In our measurement method the low priority SCHED_FIFO process - the 'Consumer' - invokes a system call. We wrote our own kernel device driver in a kernel modul to serve our own device ´/dev/measure´. The 'Consumer' process invokes the `read()` function, a system call which is handled by the Virtual File System (VFS) to the `read()` function of our kernel driver, executing the code in kernel mode, which is not preemptable in standard Linux.

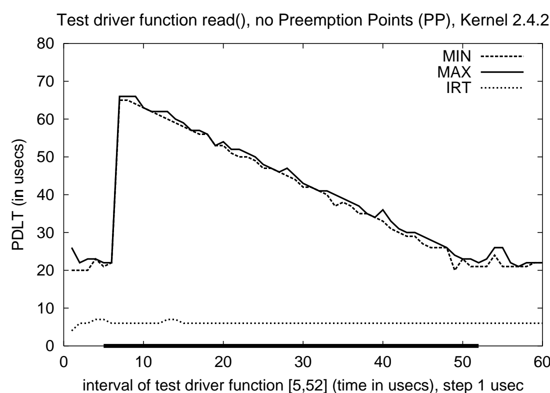

FIGURE 3: *PDLT of test driver function* `read()`*, for-1200-loop without Preemption Points, Linux 2.4.2. Shown are Minimum and Maximum of the PDLT, the Process Dispatch Latency Time, and the IRT, the Interrupt Response Time. The thick line on the X-axis indicates the time during the test driver function read() has been executed.*

The source code of the read-function of our test kernel driver, executes only a loop, it does not read any data:

```
static ssize_t read(struct file *file,
          char *user_buf, size_t count,
          loff_t *offset)
{
  int i;
  for(i=1; i<1200; i++)
  {
    if((i%1200)==0)  // modulo n function
    {
      if(current->need_resched)
          schedule();  // Preemption Point
    }
  }
  return(0);
}
```

This is the loop, the 'Consumer' process in Figure 2 executes in the kernel driver. We can control the amount of Preemption Points by altering the argument of the "modulo" function in the "if" clause. An "i%1200" means no Preemption Point whereas "i%600" stands for one Preemption Point in the middle of the "for" loop. The "modulo" function has to be checked in every loop what means the execution of the three instructions divide, subtract and compare. Now using our software monitor shown in Fig.2 we made different experiments with different lines of code in the `read()` function of our kernel driver:

1. At first we implemented the for-loop without any Preemption Points. As a driver is part of the kernel and `read()` is a system call, it can´t be preempted by the high priority process - the 'Producer'. Thus Fig.3 shows a triangle of about 65 microseconds (usecs) with a minimum PDLT baseline of about 22 usecs. This baseline is defined by the preemption time including scheduling and the measurement overhead.

2. Our test driver function `read()` is executed in kernel mode and it does not contain any spinlock. So the Linux kernel 2.4.2 with MontaVista´s Preemption Patch 6 applied to it, should be able to preempt it. Fig.4 shows that MontaVista´s patch fulfills the expectations, the system call indeed is preempted, there is no triangle shown any more. In the whole diagram Fig.4 the PDLT is in between 30 and 40 usecs.

3. Now we introduced Preemption Points into the for-loop. The method shown in 3 (a) to (c) is the way Ingo Molnar´s ´Low Latency Patch´ reduces the longest latencies caused by Linux system calls and kernel daemons from the order of hundreds of milliseconds to the order of five or ten microseconds.

   (a) First we introduced only 1 Preemption Point at half-time of the for-loop. The results are shown in Fig.5. There we can see, that the one and only triangle of Fig.3 is split into two smaller triangles, i.e. into two non preemptable regions, in Fig.5.

   (b) Next we introduced 2 and 3 Preemption Points into the `read()` function of our test kernel driver. N preemption points divide the code of the system function into $N+1$ parts so that Fig.6 and 7 show $N+1$, here 3 and 4, PDLT-triangles. But as we can see in the Figures 3 to 8 the more preemption points are introduced into the code, the smaller the PDLT triangels become.

   (c) At last we introduced 23 Preemption Points into our for-loop. Figure 8 shows the result: The triangles nearly disappear. An Preemption Point only is executed, if the variable `current→need_resched` is set to 1 by the ISR. The code of our `read()` function in the kernel driver is divided into 24 parts by our 23 Preemption Points. When the interrupt occurs, the code is executed up to the next Preemption Point, where the scheduler is invoked. The scheduler then preempts the 'Consumer' process and gives the processor to the 'Producer'. All other 'conditional Preemption Points' do not affect the code, except for testing `current→need_resched` at every Preemption Point.

When we compare Fig.8 - the driver function with 23 preemption points - and Fig.4 - the for-loop of the driver without preemption points but on a Linux 2.4.2 kernel with the Preemption Patch 6 by MontaVista -, we see that both methods result in reducing the PDLT triangle of Fig. 3, that shows a maximum PDLT of 65 microseconds (usecs) in this example. MontaVista's Patch manages to reduce it to 40 usecs without changing the code of our test driver. Also on higher latencies caused by our test driver MontaVista's Patch was able to reduce the PDLT to about 30 up to 40 usecs. By introducing 23 preemption points in our test driver function we reach a PDLT of about 22 up to 25 usecs during the system call, but we had to change the code of our driver to reach that goal.

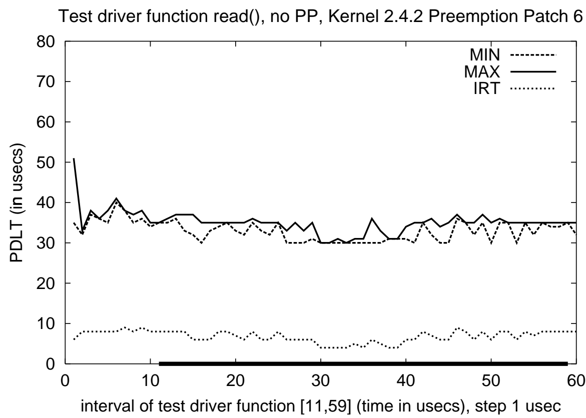

FIGURE 4: *PDLT of test driver function read(), for-1200-loop without Preemption Points, Linux 2.4.2 with Preemption Patch 6 by MontaVista*

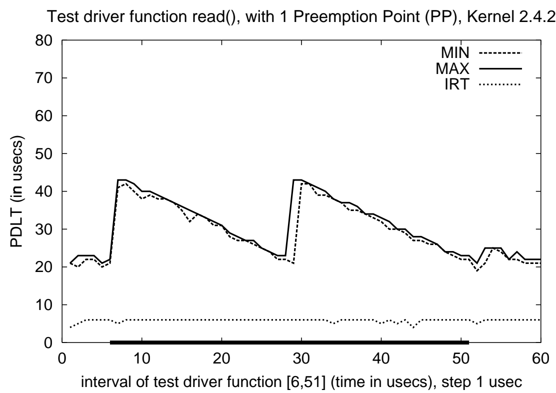

FIGURE 5: *PDLT of test driver function read(), for-1200-loop with 1 Preemption Point, Linux 2.4.2*

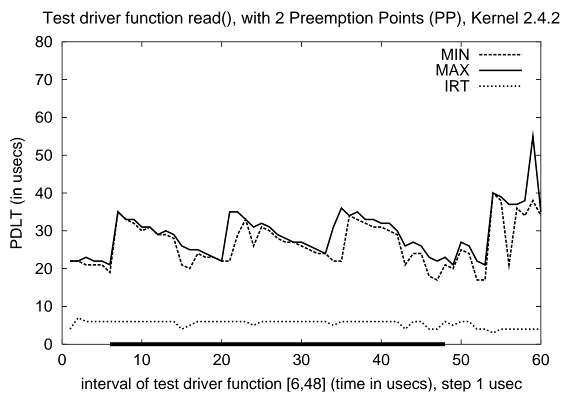

FIGURE 6: *PDLT of test driver function read(), for-1200-loop with 2 Preemption Points, Linux 2.4.2*

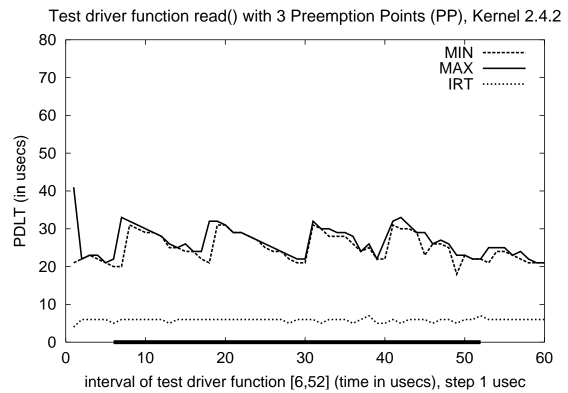

FIGURE 7: *PDLT of test driver function read(), for-1200-loop with 3 Preemption Points, Linux 2.4.2*

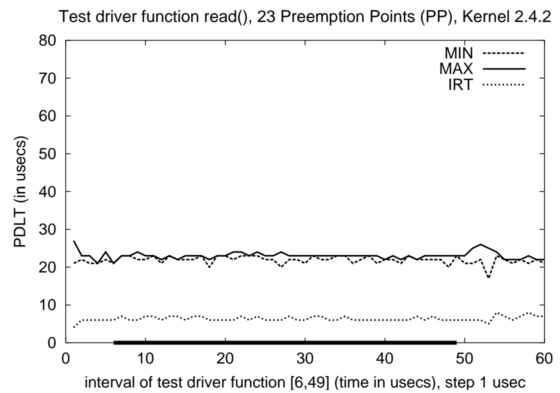

FIGURE 8: *PDLT of test driver function read(), for-1200-loop with 23 Preemption Points, Linux 2.4.2*

Comparing the thick lines on the X-Axis of Fig. 3 -Fig. 4, we see that neither the introduced Conditional Preemption Points change the execution time of the system call significantly, nor MontaVista's Preemption Patch 6 does. But in Fig. 4 we can see, that the time before the system call starts is a little bit longer using MontaVista's patch than on standard Linux kernels.

3 Examining performance changes

3.1 The Rhealstone Real-Time Benchmark

The Rhealstone Benchmark is a well known benchmark for Real-Time operating systems. It has been developed in 1989 [1, 2]. It belongs to the class of ´fine grained´ benchmarks that measure the average duration of often used basic operations of an operating system with respect to responsiveness, interrupt capability and data throughput. Using some programs of the Rhealstone Benchmark we try to examine, whether the different kernel patches affect the performance of the standard Linux kernel.
To set up the Rhealstone Benchmark we looked at [1, 2] to implement the programs in Linux. Later on we compared our programs and results to [3]. All benchmark programs use the scheduling policy SCHED_FIFO and the processes are locked into memory. In some newer Linux kernels like the kernel 2.4.5, the system call `sched_yield()` does not work sufficiently for SCHED_FIFO, i.e. for soft realtime processes. We replaced the intended call of `sched_yield()` in some benchmark programs by calling `sched_setscheduler()` with a different priority, but always with the policy SCHED_FIFO. Because `sched_yield()` did not work, we couldnt measure the ´Task or Context Switch Time´ according to the propositions of the reference implementation [2]. We did not measure the ´Deadlock Breaking Time´ either, because standard Linux does not possess an implementation of semaphores with ´priority inheritance´ [4], which is inevitable for this benchmark program to make sense. So we must admit that our benchmark programs are only similar to the benchmark programs of the original Rhealstone Benchmark [2], not identical. We didn´t apply the formula of the Rhealstone Benchmark to unify all the results up to one single value, because we measured only a part of the benchmark. Every measurement, except the IRT measurement, has been repeated for 7 million times, thereafter the average values of the measured times have been calculated. Since the underlying hardware influences the measurement results, all measurements in the Tables 1 and 2 have been performed on the same system, on an AMD K6 with a processor frequency of 400 MHz, at the runlevel ´init 1´.

- **Preemption Time**
  This is the average time a high priority process needs to preempt a process of lower priority, if the second one executes code in user space and does not hold any locks. In the benchmark program the lower priority process just executes a loop. Herein included is the time the scheduler needs to find out which process to execute next.

- **Intertask Message Passing Time (System V)**
  This is the average time, that passes in between one process sends a message of nonzero length to another process and the other process gets the message. We measured this time using the System V Message Queues implemented in Linux.

- **Semaphore Shuffle Time (System V)**
  This means the average time in between a process requests a semaphore, that is currently held by another process of lower priority, until it obtains the semaphore. The time the process of lower priority runs until it releases the semaphore is not included in the measurement. So here the implementation of the semaphores is measured. We measured the System V semaphores for processes, that Linux offers. Since the Rhealstone benchmark does not measure interthread-communication, we did not measure the POSIX semaphores implemented in the `libpthread` library to be used with threads only.

- **Interrupt Response Time (IRT)**
  This is the average time in between an extern peripheral device generates an interrupt and the first command of the interrupt service routine (ISR) as first reaction to the interrupt. This time is especially affected by the hardware used.

  The original benchmark measures the ILT - the interrupt latency time -, the time in between the CPU gets an interrupt and the execution of the first line of the interrupt handler.

  Our measurement uses the parallel port to generate an interrupt. The measurement program writes a '1' to the highest bit of the output port of the parallel port. A wire leads the electrical signal out of that bit into the interrupt entrance of the parallel port. Generating an interrupt this way, the interrupt is synchronized to the kernel-thread performing the `outb()` call to trigger the interrupt. But we can trust the IRT-results shown in Table 2, because the IRT in the Figures 3 and 4 shows similar behaviour and the same order of magnitude.

	Version	Preemption Time [usec]	Intertask Message Passing Time [usec]
A	Linux 2.2.13	1.5	3.1
B	Linux 2.4-test6	1.1	3.4
C	Linux 2.4.2	1.0	3.6
D	Linux 2.4.5	1.0	3.6
E	B + I.Molnar [6] LowLatency E2	1.2	3.4
F	C + A.Morton LowLatency [7]	1.0	3.9
G	B + Preemption Patch 1.5 [12]	1.5	5.1
H	C + Preemption Patch 6 [12]	3.5	5.4
I	H + A.Morton LowLatency [12]	5.4	5.5

TABLE 1: *Results of some of the RHEALSTONE Benchmark programs, measured on an AMD K6, 400 MHz processor, measured in the runlevel ´init 1´. Every measurement has been repeated for 7 million times.*

	Version	Semaphore Shuffle Time [usec]	Interrupt Response Time [usec]
A	Linux 2.2.13	5.6	3.5
B	Linux 2.4-test6	5.4	3.7
C	Linux 2.4.2	5.4	3.7
D	Linux 2.4.5	5.4	3.7
E	B + I.Molnar [6] LowLatency E2	5.6	3.7
F	C + A.Morton LowLatency [7]	5.3	3.8
G	B + Preemption Patch 1.5 [12]	9.0	4.7
H	C + Preemption Patch 6 [12]	11.6	5.2
I	H + A.Morton LowLatency [12]	12.5	5.1

TABLE 2: *Results of some of the RHEALSTONE Benchmark programs, the measurement conditions were the same as specified in the caption of Table 1*

3.2 RHEALSTONE Benchmark Measurements, interpretation of the results

We obtained the results shown in Tables 1 and 2 from the measurements of our Rhealstone Benchmark programs measured at the runlevel 'init 1' (all times in microseconds).

- The fine grained benchmark programs don´t show significant different results on the different versions of the Linux 2.4 kernel. The Preemption Time shows smaller values for all Linux 2.4 kernels compared to the 2.2 kernels. This may be due to the fact, that the Linux scheduler efficiency for SCHED_FIFO processes has been improved in Linux 2.4, see section 4. Also the ´Low Latency Patches´ by Ingo Molnar and Andrew Morton - introducing mostly Preemption Points - don´t show an impact on the performance of the benchmark programs. Of course benchmark programs only measure a few execution paths in the kernel.
- A Linux kernel with one of MontaVista´s Preemption Patches needs in all benchmark programs longer execution times than the standard kernel. This indicates that this method of making the Linux kernel preemptable is not free of performance costs. When looking into the changes made to the source code by MontaVista one realizes that in order to improve the respondiveness of the kernel MontaVista increased f.ex. the number of lines of code to perform at every task state transition, as a transition from a process or thread to an interrupt handler and vice versa.

3.3 Our personal view

At the moment we favorite two solutions to lower the latencies of the Linux kernel to make soft realtime applications work more efficient on Linux. Of course all soft realtime applications should be in general scheduled using the SCHED_FIFO policy and be locked into memory, if possible.

1. Ingo Molnar´s ´Low Latency Patch´ [6] has already proven its value and stability on different platforms. A disadvantage might be that it is necessary to patch a Conditional Preemption Point into every part of the kernel that could introduce a latency longer than f.ex. 10 ms. So a ´Low Latency Patch´ developer had to know and investigate f.ex. our badly written test driver, see section 2.7, - if it were part of the standard Linux kernel - and he had to

introduce Preemption Points in our code. A further problem might be that in autumn 2001 Ingo Molnar's newest ´Low Latency Patch´ we found was for kernel 2.4.0-test7, although the 2.4.10 kernel was out there at that time. Andrew Morton´s Patch [7] follows the same concept, but this patch is rather new and still maintained, the last version is for Linux 2.4.10 at the moment.

2. MontaVista´s Preemption Patch 6 + Andrew Morton´s ´Low Latency Patch´ [12] seems to be an interesting option, at the moment only for x86 systems. This patch hasn´t been tested for a long time yet, and it changes more code of the Linux kernel than Ingo Molnar´s patch does. It´s still under rapid development, at the present stage it also seems to cost performance, but it contains an interesting concept. A preemptable kernel can preempt f.ex. also our badly written test driver, as we have shown in Figure 4, although MontaVista's developers never saw the source code of our test driver.

4 Latencies of the Linux Scheduler

The scheduler - as an important part of every operating system - is invoked very often. We constructed the following measurement situation: On our single processor PC, an AMD K6 with 400 MHz, we ran only one soft realtime process, using the scheduling policy SCHED_FIFO, and many load processes, scheduled using the policy SCHED_OTHER, which is the standard Linux time slice scheduler, normally used for nearly all processes running on a Linux system.
The source code of the SCHED_FIFO process looks like the following:

```
int main(int argc, char ** argv)
{
    int i, j, ret;  double k = 0.0;
    struct sched_param parameter;

    parameter.sched_priority = 53;
    ret = sched_setscheduler(0, SCHED_FIFO,
                                  &parameter);
    if (ret == -1){
      perror("sched_setscheduler"); exit(1);
    }

    for(i=0;i<200;i++)
    {
        // just to run a while
        for(j=0;j<50;j++) { k = sqrt(2.0);}
        sleep(1);  // sleep for one second
    }
    return 0;
}
```

This SCHED_FIFO process is the only soft realtime process in the system, all other processes are scheduled using the standard Linux scheduling policy SCHED_OTHER. In every iteration from 0 to 199 the SCHED_FIFO process sleeps for 1 second. The call of `sleep(1)` invokes the scheduler that chooses one of the SCHED_OTHER load processes to run.
The source code of SCHED_OTHER load processes looks like the following:

```
int main(int argc, char ** argv)
{
      int result;
      while(1) { result = sqrt(400); }
      exit(0);
}
```

Every load process runs in an infinite loop performing a mathematical operation. This way all load processes are always able to run, i.e. their state is always TASK_RUNNING and they are always part of the runqueue, so that they can be choosen by the scheduler to be run next on the processor.
After sleeping for 1 second the SCHED_FIFO process awakens and is queued back into the runqueue. The next time the scheduler is invoked, in the first lines of the kernel function `schedule()` in the file `kernel/sched.c` a time-stamp is taken from the 64 bit Time Stamp Counter (TSC) Register of the AMD K6 400 MHz processor, which counts a clock cycle every 2.5 nanoseconds since the system was booted. In the Linux kernel there are several macros implemented for x86 processors to take a time stamp from the TSC-Register, f.ex. `rdtsc(low, high)`, where `low` and `high` are of the type `unsigned long`. i.e. 32 bit long. Another macro is `rdtscl(low)` to obtain only the lowest 32 bit of the TSC-register.
The function `schedule()` then runs to find out the process of highest priority, which is here the SCHED_FIFO process. Just before the `schedule()` function ends and gives the processor to the SCHED_FIFO process, a second time-stamp is taken from the TSC. The difference of the 2 time-stamps, which represents the duration of the kernel function `schedule()`, is calculated and printed with a `printk()` into the kernel ringbuffer and from there later on into the file `/var/log/messages`.
This measurement is repeated 200 times with a definite number of load processes runnable, as the SCHED_FIFO process sleeps for 1 second for 200 times. Out of this measurement we got one point in Figures 9 and 10. Varying the number of

SCHED_OTHER load processes we could draw the lines shown in both figures.
MontaVista [12] pointed out that the scheduling time increases as a linear function of the number of processes that are ready to run on the processor, even for a soft realtime SCHED_FIFO process.
Our measurements [5] affirmed, that the average scheduling time is a linear function of the number of processes ready to run, see Figure 9 and Table 3. From the theoretical point of view we can suspect that this linear dependance of the scheduling time on the number of processes ready to run is due to a loop through the runqueue, that the scheduler performs in order to find out the process with the highest priority to run next.

```
list_for_each(tmp, &runqueue_head) {
      p = list_entry(tmp, struct task_struct,
                                  run_list);
      if (can_schedule(p, this_cpu)) {
         int weight = goodness(p, this_cpu,
                            prev->active_mm);
         if (weight > c)
            c = weight, next = p;
      }
}
```

`list_for_each` is a macro, defined in /include/linux/list.h, that expands to a loop:

```
#define list_for_each(pos, head) \
      for (pos = (head)->next; pos != (head);
                               pos = pos->next)
```

In the `goodness()` function the priority of the process `p` is calculated, SCHED_FIFO and SCHED_RR processes get a weight of 1000 added to their realtime priority in order to be preferred to all SCHED_OTHER processes.

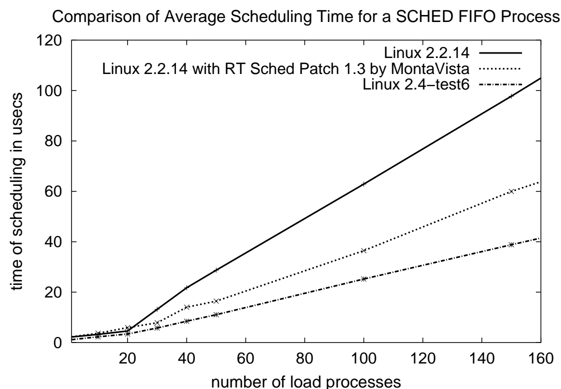

FIGURE 9: *Comparison of average scheduling times measured for a SCHED_FIFO process varying the number of load processes ready to run.*

Measurements show that the Linux 2.4 scheduler works more rapidly than the scheduler of Linux 2.2 (see Figure 9), especially for more than 20 SCHED_OTHER load processes ready to run, which represents already a considerable load.

Comparing the source code of both kernel versions, we noted that in the kernel 2.4 the queuing mechanism has been reimplemented for all queues, also for those in the scheduler. The calculation of the dynamic priorities of SCHED_FIFO processes has been altered, in Linux 2.4 the scheduler task queue does not exist any more, the bottom halves have been endorsed by the concept of Soft IRQ´s in Linux 2.4.

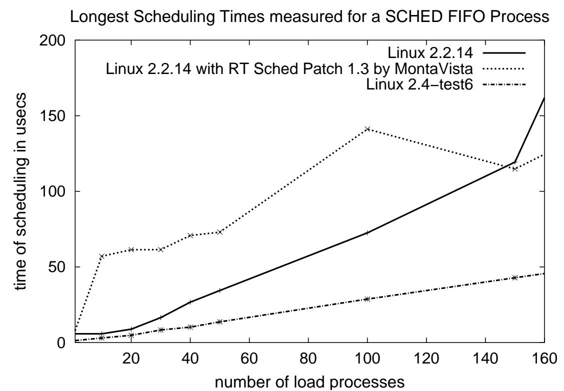

FIGURE 10: *Comparison of longest scheduling times measured for a SCHED_FIFO process varying the number of load processes ready to run.*

We measured only the MontaVista Scheduler Patch for Linux 2.2, because until summer 2001 MontaVista did not provide a version of its scheduler patch newer than for the Linux 2.4-test6 kernel on their ftp-Server, and on our hardware we did not get the patched 2.4-test6 kernel stable. Looking at the source code the scheduler patch was adapted to the 2.4 code and optimizations for SCHED_OTHER processes had been made.

Measuring the MontaVista Scheduler Patch for Linux 2.2, Figure 9 shows that the patch improves the measurements for the average case of Linux 2.2, but we measured some rare ´worst case´ scheduling times during our measurements, when the scheduling time was longer than in standard Linux, see Figure 10.

the Linux kernel code and their mutual interferences.

Using some programs of the Rhealstone Benchmark we tested whether the kernel patches introduce overhead or performance loss into the kernel compared to standard Linux kernels.

The scheduling time even for soft realtime processes has been found to depend on the number of time slice processes ready to run, but Linux 2.4 could lower this effect compared to Linux 2.2 kernels, so that the effect should be important only for computers charged with many load processes, like server systems.

number of load processes	mean	min	max	standard deviation	$\frac{stddev}{mean}$
—	usec	usec	usec	usec	%
0	1.0	1.0	1.1	0.001	0.1
10	2.3	2.0	3.0	0.011	0.5
20	3.4	2.8	4.8	0.035	1.0
30	5.8	4.2	8.3	0.059	1.0
40	8.5	6.2	10.2	0.053	0.6
50	11.1	9.3	13.7	0.071	0.7
100	25.7	23.3	28.7	0.084	0.3
150	38.8	36.3	42.9	0.073	0.2
200	52.1	41.7	56.6	0.108	0.2

TABLE 3: *scheduling time for a SCHED_FIFO process on a Linux 2.4.0-test6 kernel, varying the number of load processes*

5 Summary

Kernel code of the standard Linux kernel is not preemptable in general. This may produce long latencies on the order of hundreds of milliseconds on current hardware. In this paper we tried to line out the pros and cons of 2 general approaches to lower the latencies of the non-preemptable Linux kernel code.

On the first hand it is possible to make the kernel more responsive by introducing Preemption Points into the kernel. Ingo Molnar's and Andrew Morton's 'Low Latency Patches' follow this concept reducing the longest known Linux kernel latencies by about a factor 10 from the order of hundreds of milliseconds to the order of tens of milliseconds on current hardware.

On the other hand using the SMP spinlocks on single processor machines MontaVista presented a kernel Preemption Patch which makes Linux kernel code preemptable, if no spinlock is held. This is an interesting approach, but because there are long held spinlocks in the kernel - long in comparison to 2 context switches - latencies are reduced more efficiently if this patch is combined with f.ex. Andrew Morton's 'Low Latency Patch', as done by MontaVista.

A guarantee about the value of the longest latency in the Linux kernel - with or without the kernel patches - can't be given at the present stage of development in our opinion, because nobody can test all paths of the Linux kernel code and their mutual interferences.

References

[1] R. P. Kar and K. Porter, Feb. 1989, *Rhealstone: A Real-Time Benchmarking Proposal,* Dr. Dobbs Journal, vol. 14, pp. 14–24, `http://www.ddj.com/articles/search/search.cgi?q=Rhealstone`

[2] Kar, R., April 1990, *Implementing the Rhealstone Real-Time Benchmark*, Dr.Dobb´s Journal, `http://www.ddj.com/articles/search/search.cgi?q=Rhealstone`

[3] Digital Equipment Corporation, Maynard, Massachusetts, revised July 1998, *Performance Evaluation: DIGITAL UNIX Real-Time Systems*, `http://citeseer.nj.nec.com/274569.html`

[4] Victor Yodaiken, 2001, *The dangers of priority inheritance*, Draft, `http://www.cs.nmt.edu/~yodaiken/articles/priority.ps`

[5] Alexander Horstkotte, 03/2001, *Reengineering and analysis of the realtime capabilities of the Linux scheduler*, Student Research Project, University of Federal Armed Forces Munich

[6] Ingo Molnar, 2000, *Linux Low Latency Patch for multimedia applications*, `http://people.redhat.com/mingo/lowlatency-patches/`

[7] Andrew Morton, 2001, *Linux scheduling latency* `http://www.uow.edu.au/~andrewm/linux/schedlat.html`

[8] Linux Audio Development Mailing List, 2001, *FAQ regarding LowLatency* `http://www.linuxdj.com/audio/lad/faq.php3#latency`

[9] Linux Audio Development Mailing List, 2000, http://eca.cx/lad/2000/Jul/0153.html

[10] some postings of the Linux Kernel Mailing List concernig MontaVista´s Preemption Patch, Sept. 2000, http://lists.insecure.org/linux-kernel/2000/Sep/2140.html, /2223.html and /2158.html

[11] Yu-Chung Wang and Kwei-Jay Lin, 2000, *Some Discussion on the Low Latency Patch for Linux*, Workshop on Real Time Operating Systems and Applications and second Real Time Linux Workshop, Florida, USA, Download: http://www.thinkingnerds.com/projects/rtos-ws/presentations.html

[12] MontaVista, 2000, *RT-Scheduler and Preemption-Patch for the Linux Kernel* http://www.mvista.com/realtime/ http://www.linuxdevices.com/articles/AT4185744181.html, ftp://ftp.mvista.com/pub/Real-Time

[13] Phil Wilshire, *Real-Time Linux: Testing and Evaluation*, Workshop on Real Time Operating Systems and Applications and second Real Time Linux Workshop, Florida, USA, (2000) http://www.thinkingnerds.com/projects/rtos-ws/presentations.html

[14] S. Heursch, M. Mächtel, 1/2000, *Linux in Eile: Linux im Real-Time-Einsatz*, iX 1/2000, p. 141-145

[15] Sven Heursch, 09/1999, diploma thesis, Universität der Bundeswehr München - ID 07/99, http://bastard.informatik.unibw-muenchen.de/projekte/dipl/heursch/

[16] M. Mächtel, 2000, *Entstehung von Latenzzeiten in Betriebssystemen und Methoden zur meßtechnischen Erfassung*, University of Federal Armed Forces Munich, VDI Verlag, ISBN 3-18-380808-0

[17] M. Mächtel, 9/2000, *Unter der Lupe: Verfahren zum Vergleich von Echtzeitsystemen*, iX 9/2000, p. 120-122

[18] Timerkarte PCI-CTR05 by Measurement Computing, formerly Computerboards, manufacturer, http://www.measurementcomputing.com/

[19] Linux Audio, Latency Graph by Benno Senoner http://www.linuxdj.com/latency-graph/