Rapid Reaction Linux : Linux with low latency and high timing accuracy

combining the ´Low Latency Patch´ with the ´UTIME Patch´, adding some ideas

Arnd C. Heursch 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,

http://inf33-www.informatik.unibw-muenchen.de

Abstract

Rapid Reaction Linux has been created at the University of German Federal Armed Forces in order to enhance realtime capabilities of the standard Linux kernel. Rapid Reaction Linux combines two well known patches to achieve this goal on the Intel x86 architecture using processors like Intel Pentium or any newer descendant.

Rapid Reaction Linux combines the LOW LATENCY Patch [Molnar00], provided by Ingo Molnar, which has been found to reduce long latencies in the Linux kernel, with the UTIME Patch [Kansas97] of Kansas University that improves the precision of standard Linux timing services. Rapid Reaction Linux is not related to the well known KURT Linux [KURT98], except for the UTIME patch, both systems are relying on.

1 Introduction

In the past year, 2000, we have seen two main approaches to lower the latencies of the Linux kernel in order to make Linux more responsive and suitable for time-critical applications:

- The introduction of Preemption Points into the Linux kernel [Molnar00, Morton01].
- The approach to make all the Linux system calls and all other kernel code preemptable, if it does not contain locks [MontaVista00].

As the project, that started first, the introduction of Preemption Points into the kernel, f.ex. by the "Low Latency Patch" [Molnar00], has already shown good results to lower many long latencies [Wilshire00, Wang00].

Since time-critical tasks also need a precise time base, it would make sense to improve the accuracy of the Linux timing base combined with low latencies. Instead of simply decreasing the period of the timer interrupt, which would add the overhead of the timing routines, we chose the UTIME Patch [Kansas97] which reprograms the timer chip to the next foreseeable event, ported it to Linux 2.4 and combined it with one of the low latency patches. After adding some ideas and lines of code, we called the result "Rapid Reaction Linux".

In the following sections we present the measurements we made with Linux and Linux realtime enhancements to show which advantages "Rapid Reaction Linux" can provide. Then we speak about the code changes, we made in the "Rapid Reaction patch" and we close examining possible performance changes of our kernel patch.

2 Testing Linux and Linux realtime enhancements

The test program we use has been presented by Phil Wilshire at the second Real Time Linux Workshop [Wilshire00]. A task executes f.ex. 1000 times a `nanosleep(50 ms)` to sleep every time for a period of 50 ms, Pseudo code:

```
for(int i = 0; i< 1000; i++)
{
        get_time_stamp_from_TSC(t1);
        nanosleep(50 ms)
        get_time_stamp_from_TSC(t2);
        time_slept = t2-t1;
        delay = time_slept - 50 ms;
        /* results, see Tables 1,2 */
}
```

The time `t2-t1` is determined using the 64 bit Time Stamp Register (TSC) of the x86 processor. This test reveals `3 problems` of Standard Linux:

1. Linux has a standard lag on `nanosleep()` of 10 ms, i.e. 1 jiffie, when `nanosleep()` is called periodically, i.e. instead of a period of 50 ms, Linux sleeps about 60 ms in the sample above. This can be seen from our measurements, regarding column ´mean delay´ in Table 2, best to be seen in lines A,C,D and F because of short latencies, and from the variable ´rawdata´ in the Linux test program given in [Wilshire00].

2. At the first time `nanosleep(50 ms)` is executed, the lag is often much shorter than the 'mean delay' of 10 milliseconds (ms), because the start of the first period is not synchronized to the timer interrupt, reawakening the sleeping process. This problem can be observed at the column 'min delay' in the lines A,B,C,D and F in Table 2. The test program given in [Wilshire00] has been changed to evaluate also this first measured value.

3. When running a `dd` job and writing the buffers to the hard disk using `sync` in background, causing heavy hard disk activity - as proposed in [Wilshire00] -, Linux shows long worst case latencies on the order of hundreds of milliseconds, see lines A,B,C and F of Table 1.

Rapid Reaction Linux improves all 3 problems, see line E of Tables 1 and 2.

In Table 1 and 2, a periodical task with a period of 50 ms shall be executed. Instead of the desired period of 50 ms, all standard Linux kernels execute this task normally with an delay of 10 ms, i.e. all 60 ms. Only the original UTIME Patch [Kansas97] for Linux 2.2.13 and “Rapid Reaction Linux” normally execute the periodical task with the desired period, i.e a mean delay of nearly 0 milliseconds, see Table 2. The 31 microseconds mean $delay$ we see in line E of Table 2 might be partly due to the Linux scheduler, that selects the SCHED_FIFO process among all other process. Except for Rapid Reaction Linux the first period after starting the measurement normally produces the minimum value, that can differ up to 10 ms from the mean value, because the start of the measurements and the first `nanosleep(50 ms)` is not synchronized to the timer interrupt. The “ping” load in Table 2 isn't a heavy load for the system, so the difference in between the 'maximum $delay$' and the 'mean $delay$' in Table 2 is only in between 100 and 300 microseconds. But when - as shown in Table 1 - a heavy disk load is executed on the system as background load, for the most Linux versions, the maximum delay measured differs from the mean delay about 100 ms. This is due to latencies caused by non-interruptible system calls invoked by the disk load program, probably to write the buffers to disk (sync). Only the “Low Latency Patch”, Line D, and Rapid Reaction Linux, Line E, that incorporates it, can reduce these latencies to the order of 5 ms on our hardware. As “Rapid Reaction Linux” combines the UTIME Patch, ported to 2.4, with the “Low Latency Patch” and because it applies some changes described in section 8.2 and 8.3, “Rapid Reaction Linux” can provide the lowest (maximum - minimum) $delay$ values with a very good mean $delay$ near 0 ms in both Tables 1 and 2.

3 Periodical Tasks in Rapid Reaction Linux

In standard Linux the period of a periodical task has to be a multiple of 10 milliseconds. In Table 2 we see, that standard Linux adds 10 ms by default to the period and in Table 4 we see - if the period is not a multiple of 10 ms - Linux rounds it up to the next 10 ms boundary. So a periodical task is scheduled with a real period:

$$real\_period = desired\_period + standard\_delay$$

where $standard\_delay$ normally is a fixed value in between 10 up to 20 milliseconds, when there is no heavy load on the system. In standard Linux it is not possible to run a task with a period less than 20 ms, if you don't want to perform 'busy waits' (see Table 4). A 'busy wait' is performed in Standard Linux and in Rapid Reaction Linux, if the sleeping period is 2 milliseconds and beyond and if the process is scheduled by a soft-realtime policy, i.e. by the scheduling policies SCHED_FIFO or SCHED_RR.

In Rapid Reaction Linux the period hasn't to be a mul-

	$delay$ in ms **load: disk stress**	Min $delay$ ms	Mean $delay$ ms	Max $delay$ ms
A	Linux 2.2.13	0.2	11.2	119.6
B	A+UTIME Patch	-9.8	1.7	125.2
C	Linux 2.4.0-test6	0.2	10.9	103.5
D	C+LowLatency Patch C4	3.1	9.9	11.9
E	C + Rapid Reaction Patch	0.019	0.341	5.2
F	C+Preemption Patch 1.5	0.8	12.2	119.9

Table 1: Results of a task calling `nanosleep()` for a period of 50 ms periodically on an AMD K6 with 400 MHz. The columns {Min, Max, Mean} mean the $delay$, i.e. the time interval the kernel variant caused the soft-Realtime Process to sleep longer than the period of 50 ms (milliseconds). Ideally $delay$ should be 0 ms for {Min, Max, Mean}. Shown are the minimum and maximum times measured {Min $delay$, Max $delay$ } and the mean $delay$ measured {Mean $delay$ }. The test programm ran over 1000 periods in init 5. The IDE hard disks were not tuned using the program "hdparm". During all these measurements a program performed hard disk stress [Wilshire00] in the background.

	$delay$ in ms **ping load**	Min $delay$ ms	Mean $delay$ ms	Max $delay$ ms
A	Linux 2.2.13	1.7	10.0	10.1
B	A+UTIME Patch	-0.7	0.012	0.3
C	Linux 2.4.0-test6	4.6	10.0	10.2
D	C+LowLatency Patch C4	8.4	10.0	10.1
E	C + Rapid Reaction Patch	0.020	0.031	0.141
F	C+Preemption Patch 1.5	5.3	9.99	10.2

Table 2: Results of the same measurement program as used in Table 1. The only difference to Table 1 is that this time the background load consists of a program generating net stress by pinging another host [Wilshire00] during the measurements. As we can see from the fact that the max $delays$ do not differ much from the mean $delays$, the "ping load" is no serious load compared to the "disk load" of Table 1

tiple of 10 ms, although the period of the Linux timer interrupt remained unchanged at 10 milliseconds. Table 3 shows that in Rapid Reaction Linux it is possible to choose other periods. It is possible to schedule tasks with a period of 3.250 ms, 4 ms, 13.350 ms, ... and the standard $delay$ is only in between 20 to 30 microseconds. The maximum $delay$ measured was about 400 microseconds, while there was no heavy load on the system. On our AMD K6, 400 MHz, only a X as graphical user interface with some terminals was running.

Table 5 shows once more the difference between Standard Linux and Rapid Reaction Linux, but this time with a heavy disk load, produced by the program `dd` and `sync` operations, that write the buffers to the IDE hard disk [Wilshire00]. Standard Linux again shows its standard delay of 10 ms as a period of 50 ms is a multiple of 10 ms. The heavy disk load produces some very long latencies up to 110 ms in Standard Linux.
Rapid Reaction Linux can provide a mean delay of only 300 microseconds to the periodical task. The maximum delay measured here was about 3.3 milliseconds The minimum delay is again at 20 microseconds, as in the case without load for Rapid Reaction Linux (see Table 3).
The mean delay increases slightly for Standard Linux from about 10 to 10.9 ms and for Rapid Reaction Linux from near 0 to 0.3 ms, because the increase of the maximum delays increases the mean, too.

lines in Tables 1 and 2	provider of kernels and kernel patches
A,C:	Standard Linux kernels
B:	UTIME Patch by [Kansas97]
D:	Low Latency Patch C4 by [Molnar00]
E:	Rapid Reaction Patch by the authors
F:	Preemption Patch 1.5 by [MontaVista00]

4 Latencies in the Standard Linux kernel

4.1 the Standard Linux Scheduler

The standard Linux scheduling algorithm, named SCHED_OTHER, is a Round Robin scheduler. Time-critical tasks instead should be scheduled using the SCHED_FIFO or SCHED_RR policies, that provide fixed priorities and are preferred to any SCHED_OTHER process. Those processes are named Soft-Realtime Processes hereafter.

delay in ms E: **Rapid Reaction Linux, no load**	period ms	min *delay* ms	mean *delay* ms	max *delay* ms
Busy Wait →	2.000	-0.001	0.00086	0.080
	3.250	0.018	0.020	0.110
	4.000	0.017	0.020	0.126
	13.350	0.017	0.031	0.208
	27.500	0.009	0.021	0.112
	38750	0.010	0.021	0.156
	50000	0.013	0.021	0.078
	53.350	0.010	0.025	0.393

Table 3: This table shows other periods of a periodical task, that are possible in Rapid Reaction Linux, here based on the kernel Linux 2.4.0-test6, if there is no heavy disk load on the system. The test program ran on an AMD K6 400 Mhz in runlevel init 5 for 1000 periods for each measurement, without any additional load program. For SCHED_FIFO tasks with a period of 2 ms and beyond, Standard Linux and Rapid Reaction Linux perform a Busy Wait, which produces highest accuracy, but no other process can be executed in these 2 milliseconds.

4.2 Reasons for Latencies in the Linux kernel

Linux system calls as well as Linux kernel daemons both execute in kernel mode and cannot be preempted. So another process ready to run has to wait until the system call has been finished. This causes worst case latencies on the order of hundreds of milliseconds or even more on current Intel PC's (see lines A,B,C of Table 1). We regard an interrupt that awakens a soft realtime process out of its interrupt service routine (ISR). In the ISR the variable `current→need_resched` is set to 1. If the interrupt occured while the processor was in user mode, the scheduler will be started immediately after the ISR has been finished. If the processor was in kernel mode, the system call is finished first, causing possibly a long latency.

5 Low Latency Patch

Ingo Molnar [Molnar00] 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

delay in ms C: **Linux 2.4.0-test6 no load**	period ms	min *delay* ms	mean *delay* ms	max *delay* ms
Busy Wait →	2.000	-0.005	-0.0046	0.003
	3.250	11.3	16.7	16.8
	4.000	11.6	16.0	16.0
	13.350	15.5	16.6	16.7
	27.500	6.7	12.5	16.7
	38.750	7.2	11.2	11.3
	50.000	6.4	10.0	10.05
	53.350	8.9	16.6	16.7

Table 4: This table shows other periods of a periodical task, on the standard Linux 2.4.0-test6 kernel, if there is no heavy disk load on the system. In standard Linux it is not possible to run a task with a period less than 20 ms, except you use Busy Waiting for SCHED_FIFO processes that only have to wait for 2 ms or less. The standard lag of standard Linux is 10 ms. Tasks with a period, thats not a multiple of 10 ms are delayed to the next bigger period that is a multiple of 10 ms, plus 10 ms. F.ex a task with a period of 50 ms is scheduled with a mean delay of 10 ms, so it is in fact scheduled with an period of 60 ms. A task with a period of 53.350 has a mean delay of 16.6, so its in fact scheduled with a period of about 70 ms. So, in standard Linux it is not possible to run a task with a period less than 20 ms. The test program ran on an AMD K6, 400 MHz, in runlevel init 5 for 1000 periods for each measurement,without any additional load program running.

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

A possibility to reduce such kernel latencies is to introduces Preemption Points into the system calls of the kernel:

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

The standard Linux kernel 2.4.3, f.ex. contains already 34 conditional preemption points for the Intel x86 architecture and 53 Preemption Points for all other architectures together.

$delay$ in ms **load: disk stress**	period ms	min $delay$ ms	mean $delay$ ms	max $delay$ ms
C: Linux 2.4.0-test6	50.000	0.242	10.9	110
E: Rapid Reaction Linux	50.000	0.018	0.3	2.9
E: Rapid Reaction Linux	53.530	0.020	0.3	3.3

Table 5: This table compares Rapid Reaction Linux to standard Linux once more: In Rapid Reaction Linux it is due to the UTIME patch possible to choose a frequency which isnt a whole number. In standard Linux this is not possible. Due to the Low Latency Patch latencies are greatly reduced in Rapid Reaction Linux. All measurements in this table have been made while Phil Wilshires program generated heavy disk load. This explains why the max delay is higher as in Table 3, where there is no load on the system. At frequencies beyond 50 ms it has not been possible to execute a sync of our IDE-hard disk in between.

The 'Low Latency' kernel Patch created by Ingo Molnar [Molnar00] introduces about 50 additional Preemption Points into the standard Linux kernel code at positions where long latencies, f.ex. caused by long non-interruptible system calls, occur in standard Linux as shown above. These Points are called ´Conditional Preemption Points´ since the kernel is preempted there only if the variable `current→need_resched` is set to 1. Then in the system call at the Preemption Point the scheduler is invoked and f.ex. a time-critical process can get the processor. As stated in literature [Wang00] Ingo Molnar managed to place his Preemption Points without affecting the stability of the Linux kernel. He successfully reduced many special long latencies to the order of 5 to 10 ms [LinAudio01]. The value of (max $delay$ - mean $delay$) in line D of Table 1 confirms this testimony.

6 Time base of the standard Linux kernel

6.1 the timer interrupt - the heart beat of a linux system

The standard Linux Kernel programs the timer chip of the PC to generate a timer interrupt all 10 ms. The timer Interrupt Service Routine increments the global kernel variable `unsigned long volatile jiffies` by 1 every 10 ms. All timing services of the standard linux kernel are calculated on the basis of `jiffies`. So no timing service has a resolution higher than 10 ms. An exception to that rule are a few busy waits for soft real-time processes.

6.2 How do timing services work in standard Linux ?

Let´s assume a user process wants to sleep for a period of 50 ms, therefore it invokes the system call `nanosleep()` with the right parameter. As always when a system function requests a timing service in Linux, the struct of a linux kernel timer is filled with the appropriate values and put into a list of kernel timers, i.e. a list of all events in the system to be scheduled at a fixed time in future.

This is the kernel timer structure of the Linux 2.4.4 kernel:

```
struct timer_list {
        struct list_head list;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
};
```

The field ´expires´ gets a jiffie value in the future when the timer shall expire. All 10 ms, when a timer interrupt occurs in standard Linux, the timer ISR activates the timer bottom half, which is executed by the scheduler. In the bottom half the kernel timers in the kernel timer list are checked whether the value of its `expire`-field is less than the actual `jiffies` value. If so, a certain timer has expired and the bottom half now executes the function the pointer of the function field points to. The data field serves as parameter given to this function. That way, the process that called `nanosleep(..)` before, is awakened after sleeping for the predefined time and some $delay$.

The standard lag on `nanosleep()` in the standard Linux kernel, even for a soft realtime process, is 10 ms. Furthermore a worst case latency on the order of hundreds of milliseconds on current hardware can occur, as show also our measurements in lines A,B,C of Table 1.

7 the UTIME Patch

Now we look at a mechanism to improve the timing resolution of the standard Linux kernel:

By changing the value of the global kernel variable HZ it would be possible to generate timer interrupts at a higher rate, f.ex. all 1 ms. But this would lead to a very high timing overhead, executing timing service routines much more often.

The UTIME Patch [Kansas97], developed at the University of Kansas, takes advantage of the fact that soft realtime tasks may have stringent timing requirements on the order of 100 microseconds (usec), but it is very unusual that there is such an requirement every 100 usec. The solution provided by UTIME is to reprogram the timer chip after every timer interrupt to the next foreseeable event. If there is any task, f.ex. requiring scheduling in 7 milliseconds (ms), the timer chip is reprogrammed to generate an interrupt at exactly that time. If there is not such a timing event the timer chip is reprogrammed to generate the next timer interrupt just in 10 ms. So it´s guaranteed that the interval in between two timer interrupts is never longer than 10 ms, the period of the timer interrupt in standard linux.

UTIME extends the global kernel variable `unsigned long volatile jiffies` by a variable called `jiffies_u`, that is adjusted at every timer interrupt to the appropriate value in between 0 and 9999 microseconds (usec) to indicate how much time there is until the variable jiffies has to be increased again. The linux time base does not suffer from the jitter of the 8054 timer chip on the order of microseconds, because `jiffies_u` is set according to the actual value of the Time Stamp Clock Register (TSC), which is a 64 bit processor register in the Intel Pentium and all its descendants, also those x86 processors made by AMD, f.ex. This register is updated every clock cycle by the processor.

7.1 Porting the UTIME Patch from Linux 2.2.13 to 2.4

Since there is only a version of UTIME for Linux 2.2.13 available to download [Kansas97] we ported the patch to the Linux 2.4.0-test6 kernel on our own.

We had to adapt the UTIME patch to the Linux 2.4 kernel, some of the changes we mention here:

- Linux 2.2 and Linux 2.4 contain a complex management for time-critical tasks, containing a vector with six lists of events to execute at specified times. This structure is probably faster, but possibly not realtime-compliant [KURT98], so we used the older but realtime compliant "old timer list" used in Linux 2.0 and 2.2.
- Linux 2.2 contains a kernel variable "lost_ticks", which serves to store the jiffies for the time in between the timer interrupt occurs and the timer bottom half updates the time basis. This variable has been erased for the x86 platform in Linux 2.4 and replaced by the difference "jiffies-wall_jiffies", containing the new variable "wall_jiffies". We adapted the UTIME patch to these changes.
- Many other slight changes were necessary.

8 Rapid Reaction Linux

8.1 UTIME and Low Latency patch forming Rapid Reaction Linux

Rapid Reaction Linux combines the 'Low Latency Patch' with the 'UTIME Patch' to be able to serve time-critical soft realtime tasks in a better way.

Why did we decide to combine just these 2 patches ?

- Due to our measurements - compare line A,C,D and F of Table 1 - and the study of literature [Wilshire00, LinAudio01] we got the impression that Ingo Molnars 'Low Latency Patch' reduces kernel latencies best at the moment. Incorporating the ´Low Latency Patch´ solves problem 3) of section 2, i.e. reduces many long latencies.
- Since the Low Latency Patch seems to be able to reduce many known worst case latencies to the order of tens of milliseconds on current PC systems, it is worth to improve the resolution of Linux timing services beyond 10 milliseconds, too. That´s what the UTIME patch does, without generating much overhead, when there are no time-critical tasks to process. So we ported the UTIME patch from the Linux 2.2 kernel to Linux 2.4
- The code changes of the ´Low Latency´ Patch are somehow orthogonal to the changes the UTIME patch mades, because both patches aim at different goals. So the two patches don´t interfere.

There are two further changes we made to the kernel code, worth mentioning, presented in the two following subsections. These changes also lead to the better results of Rapid Reaction Linux in line E of Table 1 and 2, compared to a combination of the best values of line B, the UTIME Patch, and line D, the Low Latency Patch.

8.2 improving UTIME kernel timers oneshot behaviour

Since Rapid Reaction Linux incorporates the UTIME Patch, `nanosleep()` has no standard lag of 10 ms, i.e. problem 1) of section 2 is solved. But we had still to solve problem 2), that the first `nanosleep(50 ms)` often ends up to 10 ms too soon, because of the outdated system time. To solve this problem it is necessary to have the linux time base actually updated before adding the period, the process wants to sleep, to the actual time. Regarding `nanosleep()` we called the UTIME function `update_jiffies_u()` to update the system time before starting the sleeping period. `update_jiffies_u()` uses the TSC register of the processor. That way we could avoid also the first jitter of `nanosleep()`, see the ´0.019 ms´ as minimum delay and the mean delay of ´0.341 ms´ of Rapid Reaction Linux in line E of Table 1,f.ex. compared to '-9.8 ms' of the UTIME Patch in line B.

The following code shows the changes made to the code of the UTIME function `schedule_timeout_utime(..)`, called by `nanosleep()`:

```
/* Rapid Reaction Linux begin
   update jiffies_u and jiffies
     to the actual time */
update_jiffies_u();
/* Rapid Reaction Linux end */

expire = *timeout + jiffies;
expire_u = *timeout_u + jiffies_u;

/* UTIME: */
expire += expire_u/USEC_PER_JIFFIES;
expire_u = expire_u%USEC_PER_JIFFIES;

init_timer(&timer);
timer.expires = expire;
timer.usec = expire_u;        // UTIME
timer.data = (unsigned long) current;
timer.function = process_timeout;

add_timer(&timer);
schedule();
del_timer(&timer);
```

`process_timeout` is a function, that is started in the timer bottom-half, that has been triggered by a timer interrupt. `process_timeout` serves to awake the process, put to sleep before by calling `nanosleep()`, after the timer interrupt has come at the predefined time. The data of the process to awake has been stored in the field `timer.data`, see code above.

8.3 introduction of soft realtime timers

Being able to program timer interrupts more precise than 10 ms to trigger a time-critical event, f.ex. to awake a soft realtime process, it is reasonable that the scheduler is started as soon as possible after the Interrupt service routine (ISR) is done. To achieve this goal we added the field '`timer.need_resched`' to the kernel timer structure. In Rapid Reaction Linux this field will be set to 1, every time a Soft-Realtime process wants to sleep for a precise time interval. When the affiliated timer interrupt occurs, the variable `current→need_resched` is set to 1 in the ISR to reschedule as soon as possible making possibly use of a Conditional Preemption Point.

In the Linux kernel function `add_timer()` we added support for the Soft Realtime processes and threads. If `add_timer()` is not called out of an interrupt service routine or out of a bottom half - in these two cases `in_interrupt()` would be 1 -, then it is checked whether the calling process is running with Soft-Realtime priority. If so, we set the field `timer→need_resched` to 1. We added this field to the timer structure to be able to express that a special timer interrupt is generated to schedule a time-critical process.

```
void add_timer(struct timer_list * timer)
{
  unsigned long flags;
  struct timer_list *p;

  /* Rapid Reaction Linux begin */
  /* initialize */
  timer->need_resched = 0;
  if(!in_interrupt())
  {
    if(current->policy &
         (SCHED_RR | SCHED_FIFO))
    {
```

```
        timer->need_resched = 1;
    }
  }
  /* Rapid Reaction Linux end */

  p = &timer_head;
  spin_lock_irqsave(&timerlist_lock, flags);
  do
  {
     p = p->next;
  } while ((timer->expires>p->expires) ||
     ((timer->expires==p->expires) &&
       (timer->usec>p->usec)));
  timer->next = p;
  timer->prev = p->prev;
  p->prev = timer;
  timer->prev->next = timer;
  spin_unlock_irqrestore(
                  &timerlist_lock, flags);
  if (timer->prev==&timer_head)
  {
      reload_timer(timer->expires,
                   timer->usec);
  }
}
```

The UTIME Patch uses directly after a timer interrupt the following function to find out, how many microseconds later from now on - maximal 10000 usecs later of course - the next timer interrupt shall come. Here we introduced a global kernel variable named `need_resched_next_timer_isr`, that is set to 1, if the next timer interrupt serves to schedule a soft real-time process or thread, i.e if `timer→need_resched` equals 1. The reason why this global variable is needed is that we don´t know at the interrupt service routine the timer structure that triggered this timer interrupt. But it is important to know in the ISR, that the actual timer interrupt has been triggered to f.ex. awake a soft realtime process.

```
extern inline unsigned long
       time2next_event(void)
{
  unsigned long timer_jiffies;
  unsigned long timer_jiffies_u;
  struct timer_list *next_timer;

  /* skip all the expired timers...*/
  /* if get_unexpired_timer returns null
  * then the next timer boundary
  * is a jiffies interrupt */
  need_resched_next_timer_isr = 0;
  next_timer = get_unexpired_timer();
  if (!next_timer)
  {
     timer_jiffies = jiffies + 1;
     timer_jiffies_u = 0;
  }
  else
  {
     timer_jiffies = next_timer->expires;
     timer_jiffies_u = next_timer->usec;
     /* Rapid Reaction Linux begin
     need_resched_next_timer_isr:
     global kernel variable*/
     need_resched_next_timer_isr =
          next_timer->need_resched;
     /* Rapid Reaction Linux end */
  }

  if (timer_jiffies == jiffies)
  {
       return (timer_jiffies_u -
jiffies_u);
  }
  else
  {
       /* aim for the HZ boundary */
       /* Rapid Reaction Linux begin */
       need_resched_next_timer_isr = 0;
       /* Rapid Reaction Linux end */
       return (USEC_PER_JIFFIES -
jiffies_u);
  }
}
```

During the Interrupt Service Routine of the timer interrupt we find out - in cases the timer interrupt serves a soft real time task - that the variable `need_resched_next_timer_isr` equals 1. We set it back to 0 and we set `current→need_resched = 1` to invoke the scheduler as soon as possible. That´s what all our changes aimed at, because having set `current→need_resched = 1` in the timer ISR we are able to use all the conditional preemption points, the 'Low Latency Patch' introduced into Linux, to reduce kernel latency times, caused by long term system calls that normally cannot be preempted.

```
void utime_do_timer_oneshot
               (struct pt_regs *regs)
{
update_jiffies_u();

/* Rapid Reaction Linux begin */
if(need_resched_next_timer_isr)
{
  need_resched_next_timer_isr = 0;
  current->need_resched = 1;
}
/* Rapid Reaction Linux end */
```

```
load_timer();
mark_bh(TIMER_BH);
if (jiffies_intr) {
  orig_do_timer(regs);
  }
}
```

9 Examining performance changes

9.1 The Rhealstone Real-Time Benchmark on Linux 2.4

The Rhealstone Benchmark is a well known benchmark for Real-Time operating systems. It has been developed in 1989 [Kar89, Kar90]. 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 Rapid Reaction Patch or its components affect the performance of the standard Linux kernel.

To set up the Rhealstone Benchmark we looked at [Kar89, Kar90] to implement the programs in Linux. Later on we compared our programs and results to [DEC98]. 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, 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. For the same reason it did not make sense to measure the ´Task or Context Switch Time´ We did not measure the ´Deadlock Breaking Time´ either, because Standard Linux does not possess an implementation of semaphors with ´priority inheritance´ [Yod01], 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 [Kar90], 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 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 6 and 7 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 uses 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.

- **Semaphor Shuffle Time (System V)**
  This means the average time in between a process requests a semaphor, that is actually helt by another process of lower priority, until it obtains the semaphor. The time the process of lower priority runs until it releases the semaphor is not included in the measurement. So here the implementation of the semaphors 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)**
  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 not only affected by the operating system, but also 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 first 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. Nevertheless the values of the IRT shown in Table 7 are reasonable and similar to those of other measurements we made.

	Version	Preemption Time [usec]	Intertask Message Passing Time [usec]
A	Linux 2.2.13	1.5	3.1
B	A+UTIME Patch	1.6	3.4
C	Linux 2.4.0-test6	1.1	3.4
G	Linux 2.4.9	1.1	3.4
D	C+Low Latency Patch C4	1.1	3.5
E	C+Rapid Reaction Patch	1.2	3.6

Table 6: Results of some of the RHEALSTONE Benchmark programs, measured on an AMD K6, 400 MHz processor on different versions of the Linux kernel, measured in the modus ´init 1´, linked with the compiler option -O2 to optimize. Every measurement has been repeated for 7 million times.

	Version	Semaphor Shuffle Time [usec]	Interrupt Response Time [usec]
A	Linux 2.2.13	5.6	3.5
B	A+UTIME Patch	5.2	3.5
C	Linux 2.4.0-test6	6.8	3.5
G	Linux 2.4.9	5.3	3.5
D	C+Low Latency Patch C4	6.8	3.6
E	C+Rapid Reaction Patch	5.8	3.6

Table 7: Results of some of the RHEALSTONE Benchmark programs, the measurement conditions were the same as specified in the caption of Table 6

9.2 RHEALSTONE Benchmark Measurements, interpretation of the results

We obtained the results shown in Tables 6 and 7 from the measurements of our Rhealstone benchmark programs measured at the runlevel 'init 1' (all times in microseconds)

Of course, these benchmark programs only measure a few often used execution paths in the kernel.

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. In Table 6 we can see, that "Rapid Reaction Linux" has an overhead of about 0.05 to 0.1 microseconds at the Preemption Time, but the value is still smaller than the values of the Linux 2.2 kernels. For the Intertask Message Passing Time it shows up to 0.2 microseconds of overhead. In Table 7 the "semaphor shuffle time" of Rapid Reaction Linux is nearly the same as for Linux 2.2.13, and better than Linux 2.4.0-test6. As Rapid Reaction Linux in line F is based on Linux 2.4.0-test6, we can say it decreases the Semaphor Shuffle Time, although its value is higher than the one of Linux 2.4.9. When we port "Rapid Reaction Linux" to Linux 2.4.9 in future, we will see whether it will decrease the value of 2.4.9, too. In our opinion - although the values of Rapid Reaction Linux tend to be a little bit higher than in Linux 2.4, - they can be compared to those of Linux 2.2 and they are no reason to fear a major performance loss in "Rapid Reaction Linux".

10 Availability

We will continue to develop Rapid Reaction Linux to perform time-critical operations more promptly and efficiently.

The kernel patch for Rapid Reaction Linux shall be licensed under the terms of GPL. For the standard Linux kernel 2.4.0-test6 the patch is available for download on our homepage:

`http://inf33-www.informatik.unibw-muenchen.de/research/rrlinux/rapid.html`

11 Summary

Our measurements and the measurements of many others show that the “Low Latency Patch” [Molnar00] can reduce many long latencies from the order of 100 ms to the order of 5 to 10 ms. Having a Linux kernel with reduced latencies it is even more interesting to have also an accurate time basis.

Therefore we ported the UTIME Patch [Kansas97] from Linux 2.2 to Linux 2.4 and combined it with the “Low Latency Patch”. We improved the accuracy of the UTIME timers and introduced “soft realtime timers”. The resulting Linux kernel patch we called Rapid Reaction Linux. In our measurements it shows a very good timing accuracy when executing periodical tasks.

Standard Linux shows a standard lag of 10 ms for periodical tasks, Rapid Reaction Linux does not. In standard Linux the first period of the task is often shorter than the following periods, in Rapid Reaction Linux the first period has the correct length like all the following periods. Standard Linux can’t schedule periods in between 2 and 20 ms, Rapid Reaction Linux can.

Standard Linux schedules tasks with periods which are not a multiple of 10 ms, - the period of the timer interrupt -, with a delay in between 10 and 20 ms - not including possible latencies - to round up their period to a 10 ms boundary. Rapid Reaction Linux can schedule these periods - without rounding their period - , with a mean delay on the order of tens of microseconds. So Rapid Reaction Linux is best suited for periodical tasks and/or for waiting only once an desired amount of time.

A guarantee about the value of the longest latency in the Linux kernel - with or without the kernel patch of Rapid Reaction Linux - 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.

Concerning the latencies the “Low Latency Patch” incorporated in Rapid Reaction Linux reduces many long latencies from the order of 100 ms to the order of 5 ms to 10 ms on current hardware. So the higher timing accuracy is disturbed much less by latencies induced by Linux system calls or kernel threads. This is an important reason for the higher timing accuracy to make sense.

We will continue to develop Rapid Reaction Linux to better the services it can provide for time-critical tasks.

References

[Wilshire00] 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`

[Wang00] Yu-Chung Wang and Kwei-Jay Lin, *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: see [Wilshire00], (2000)

[Molnar00] Ingo Molnar, Linux Low Latency Patch for multimedia applications, `http://people.redhat.com/mingo/lowlatency-patches/`

[Kansas97] Kansas University, UTIME Patch - *Micro-Second Resolution Timers for Linux*, `http://www.ittc.ukans.edu/utime/`

[KURT98] KURT-Linux, Kansas University Real-Time Linux, `http://www.ittc.ukans.edu/kurt/`

[MontaVista00] Montavista, Preemption-Patch for the Linux Kernel `http://www.linuxdevices.com/articles/AT4185744181.html`, `ftp://ftp.mvista.com/pub/Real-Time`

[Srinivasan98] Balaji Srinivasan, *A Firm Real-Time System Implementaion using Commercial Off-The-Shelf Hardware and Free Software*, Master’s thesis, University of Kansas (1998), `http://www.ittc.ukans.edu/kurt/`

[Kar89] R. P. Kar and K. Porter, ”Rhealstone: A Real-Time Benchmarking Proposal,” Dr. Dobbs Journal, vol. 14, pp. 14–24, Feb. 1989, `http://www.ddj.com/articles/search/search.cgi?q=Rhealstone`

[Kar90] Kar, R., Implementing the Rhealstone Real-Time Benchmark, Dr.Dobb´s Journal, April, 1990

[DEC98] Digital Equipment Corporation, Maynard, Massachusetts, Performance Evaluation: DIGITAL UNIX Real-Time Systems, revised July 1998, `http://citeseer.nj.nec.com/274569.html`

[Yod01] Victor Yodaiken, The dangers of priority inheritance, Draft http://www.cs.nmt.edu/~yodaiken/articles/priority.ps , 2001

[Morton01] Andrew Mortons Low Latency Patches, University of Wollongong, Australia, 2001, http://www.uow.edu.au/~andrewm/linux/schedlat.html

[LinAudio01] The Home of the Linux Audio Development Mailing List, FAQ, http://www.linuxdj.com/audio/lad/faq.php3#latency