Long-term monitoring of apparent latency in PREEMPT_RT Linux real-time systems

Carsten Emde
Open Source Automation Development Lab (OSADL) eG
Aichhalder Str. 39, 78713 Schramberg, Germany
C.Emde@osadl.org

Abstract

The real-time wakeup latency tracer (wakeup_rt) in conjunction with the wakeup latency histogram (CONFIG_WAKEUP_LATENCY_HIST) is part of the Linux kernel trace subsystem. It monitors processes that have the highest priority of the system throughout their entire wakeup processing and do not share this priority with any other concurrently running or scheduled process. Thus, this tracer should be able to continuously monitor a system's worst-case latency. However, in a situation where interrupts are disabled, the timer interrupt to enqueue a process is already late, but the latency between enqueuing and wakeup still may be normal. In this case, an erroneously low latency would be recorded.

To overcome this limitation and to create a true recording of the apparent latency of every single wakeup procedure, an additional tracer was implemented. This tracer determines the offset between the intended timeout and the real timeout of the timer interrupts; the CONFIG_MISSED_TIMER_OFFSETS_HIST configuration item is used to enable this tracer.

When implementing the two tracers, care was taken to minimize the introduced performance penalty.

The combination of the real-time wakeup latency tracer with the timer offset tracer allows to continuously monitor a real-time system during its entire life-time in such a way that every single violation of the scheduling deadline would be registered. This makes it possible to ensure real-time capabilities even in systems where path analysis is no longer feasible.

1 Path analysis vs. latency monitoring

There is no doubt that path analysis is the "Gold Standard" of determining a system's worst case latency and, thus, is providing the most important characteristic of a real-time operating system. The procedure appears to be simple: Find the longest code path during which the system is irresponsive to asynchronous external events, lookup the number of processor cycles of every assembly instruction of this path, calculate their sum and divide the result by the clock frequency. State-of-the-art processors of the year 2010, however, are less well suitable for path analysis, since a particular assembly instruction may take any number of cycles. This is due to

- BIOS interference through system management interrupts (SMIs)
- instruction and data caches that may need to be flushed and filled before an instruction can be fetched or data can be loaded
- microcode patches that may modify the duration of a particular assembly instruction

In addition, path analysis is virtually impossible in general-purpose operating systems with a large code basis such as Linux.

As an alternative, the apparent worst-case latency of a system can be determined empirically. This is often done independently from a given project, e.g. in the lab under artificial load conditions. A signal generator is connected to a digital input line to trigger repeated interrupts, and a user space application with real-time priority is wait-

ing for this input line and signals the beginning of program execution via a (preferably non-interrupt driven) digital output line. The time difference between sending the input trigger and receiving the output signal reflects an individual latency value. Such measurements are typically performed repeatedly during a day or even a week, and the longest observed latency is taken as the result of the measurement. If the hardware-induced latency is known and the configuration of the system does not change, a somewhat simpler software procedure can be employed by measuring the latency of cyclic timer interrupts. This is best done using the *cyclictest* [1] program that was developed for this purpose and is part of the popular *rt-tests* suite [2]. Command line options of *cyclictest* allow to pin a measurement thread to a particular core of multi-core processors and to run one thread per core.

However, there are a number of shortcomings of such recordings:

- The load condition may not reflect the actual load of the production system
- The system configuration may not reflect that of the production system
- Rarely occuring latencies are not detected
- Other input controllers may suffer from longer latencies

It would, therefore, be preferable to continuously monitor each and every wakeup sequence of all real-time tasks in the running system under production conditions. This measurement may even be enabled during the entire life-time of the system. If, for example, during a 10-year recording period, no wakeup sequence was detected that exceeded the specified deadline, the system can truly be considered, at least retrospectively, as real-time compliant.

2 Equipping the Linux kernel with a continuous latency monitoring subsystem

Version 2.6.33-rt29 of the Linux PREEMPT_RT real-time patches already contains two important components that are needed for the continuous monitoring of latencies: Wakeup latency (CONFIG_WAKEUP_LATENCY_HIST) and missed timer offsets (CONFIG_MISSED_TIMER_OFFSETS_HIST):

1. **Wakeup latency:** To determine the effective wakeup latency, the kernel stores the time stamp when a process is enqueued, and takes a second time stamp shortly before control is passed to this process. The difference between these two time stamps is then taken as wakeup latency and entered into a histogram with a granularity of 1 μs and a range from 0 to 10.24 ms. Such latencies may not necessarily reflect a system's worst case latency, since the process in question may be preempted by another one with a higher priority or may fail to be scheduled in time, since another process with the same priority is already running. Therefore, two different variables are created, **latencies of real-time processes** that had the highest priority of the system throughout the entire wakeup procedure and **latencies of other processes** that may have been interrupted during scheduling. The latter is called "sharedprio" wakeup latency.

2. **Missed timer offsets:** Whenever a timer is expiring, the difference between the intended expiration time and the actual time is calculated and entered into another histogram with the same granularity and range as above. This variable is called "missed_timer_offsets". The Linux kernel stores this value along with other task variables for later use, if both CONFIG_WAKEUP_LATENCY_HIST and CONFIG_MISSED_TIMER_OFFSETS_HIST are configured and enabled.

2.1 Timer and wakeup latency

Simply measuring the interval between enqueueing and wakeup, however, may not be appropriate in cases when a process is scheduled as a result of a timer expiration, since the timer already may have missed its deadline. This may happen in a situation when interrupts are disabled temporarily. After the interrupts are re-enabled, enqueing takes place and may result in an apparently normal wakeup latency. To also consider such latencies that are based on a delayed timer expiration, a new latency variable was added: **Timer and wakeup latency**. For this purpose, a third histogram, again with the same granularity and range as above, is made available. Whenever a wakeup was the result of a timer expiration, the timer offset is added to the wakeup latency and entered into this histogram. There is no separate configuration item, since this

histogram is configured implicitly when both CONFIG_WAKEUP_LATENCY_HIST and CONFIG_MISSED_TIMER_OFFSETS_HIST are enabled in the Linux kernel. As of October 2010, this histogram is not yet available in the PREEMPT_RT real-time patches but scheduled for inclusion in one of the next releases. The variable is called "timerandwakeup".

2.2 Configuraton and activation of latency recording

By default, any histogram recording is disabled at boot time. Therefore, the two items CONFIG_WAKEUP_LATENCY_HIST and CONFIG_MISSED_TIMER_OFFSETS_HIST may be configured without any relevant performace penalty. To enable them at runtime, a non-zero value must be written to the related variables in the "enable" subdirectory of the latency histograms, e.g.

```
cd /sys/kernel/debug/tracing/latency_hist
echo 1 >enable/wakeup
echo 1 >enable/missed_timer_offsets
echo 1 >enable/timerandwakeup
```

2.3 Tracing of affected processes

In order to provide some insight into the possible impact of a prolonged latency, the process ID and name of the affected process, its priority and latency are recorded in the variable "max_latency-CPUx" of the histogram subdirectory.

2.4 Resetting the latency histograms

To clear the histograms, the reset variable is provided. If, for example, it is desired to clear all histogram data, the following procedure can be used:

```
cd /sys/kernel/debug/tracing/latency_hist
for i in ‘find . | grep /reset$‘
do
  echo 1 >$i
done
```

3 Continuous recording and graphical display

The Munin monitoring framework [3] that is based on Tobias Oetiker's Round Robin Database Tool [4] was used to visualize the continuous recording of a system's worst-case latency. By default, a sample is taken every five minutes and the highest recorded latency is entered into the database. Thereafter, the histogram is reset. The following example recordings have been obtained on a uniprocessor and a 4-core hyperthreaded system that underwent regular system load generation and stimulated latency recordings. More specifically, *cyclictest* runs were started at 7 a.m. and 7 p.m. and lasted for about six hours. Load generation was started at 9 a.m. and 9 p.m. and was only halted when the latency recording was finished. The uniprocessor test board was equipped with an AMD Athlon 64 processor 2800+ [5], the multi-core board was running an Intel i7 Nehalem 975 @3333 MHz [6]. The Linux kernel version was 2.6.33.7-rt29.

3.1 Wakeup latency

The recording in Figure 1 was obtained in a uniprocessor system and represents consecutive wakeup latency maxima of 5-min intervals.

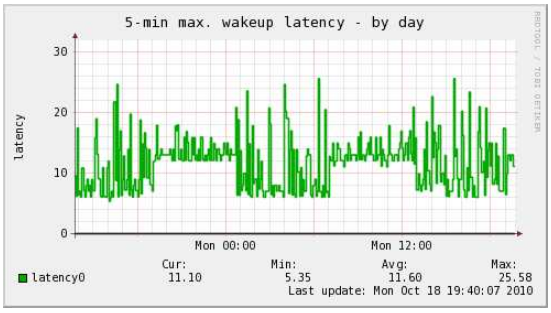

FIGURE 1: *Example of a 30-hour recording of the wakeup latency*

Since no raw data are used for input but latency maxima of a given time frame had been calculated, the derived results in the columns labeled "Min:" and "Avg:" are irrelevant; the only relevant result is the maximum of consecutive 5-min maxima at the rightmost column labeled "Max:". Under idle conditions, the latency of the examined processor has a somewhat larger variability than under load; its maximum is 25.58 μs.

3.2 Missed timer offsets

Figure 2 displays continuously recorded offsets of expired timers (same time interval as in Figure 1).

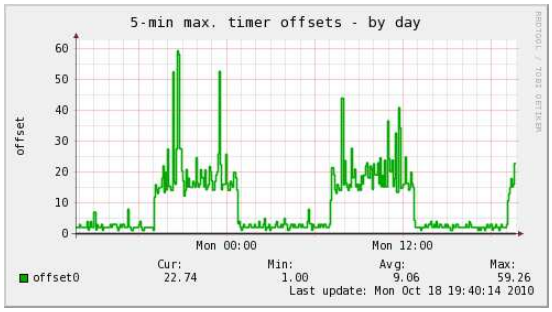

FIGURE 2: *Example of a 30-hour recording of missed timer offsets*

When the system is idle, the timer interrupts are executed nearly always in time. Under load, the timer deadline is missed by up to 59.26 μs. However, this delay does not necessarily represent a relevant latency, since the particular process may not be the one and only real-time process.

3.3 Timer and wakeup latency

The newly provided combined timer and wakeup latency recording is presented in Figure 3 (again same time interval as above).

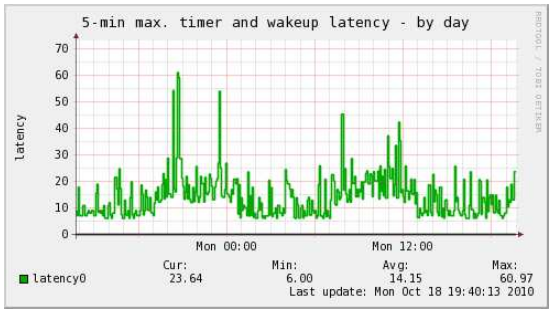

FIGURE 3: *Example of a 30-hour recording of timer and wakeup latency*

Under idle conditions, the 5-min latency maxima of the examined processor do not exceed 30 μs. During load period #1 and #2, the largest recorded latencies increase to about 60 and 45 μs, respectively.

3.4 Timer and wakeup latency of a multi-core processor

Figure 4 contains the newly provided combined timer and wakeup latency recording (same as in Figure 3) in an 8-way (hyperthreaded 4-core) processor.

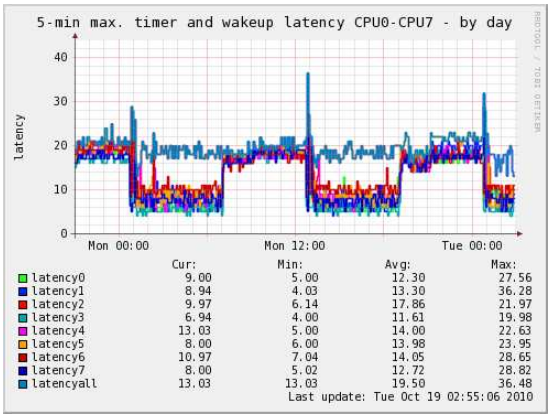

FIGURE 4: *Example of a 30-hour recording of timer and wakeup latency of a multi-core processor*

As the latency is recorded separately per core, specific differences between the various cores can be detected. They are mostly related to the various interrupts that are assigned and pinned to a particular core. Such recordings may help to further increase the real-time performance of the system by selecting the core with the lowest latency for the most demanding real-time task. In the above example, core #1 reveals the highest (36.28 μs) and core #3 the lowest (19.98 μs) latency. The additional variable "latencyall" records the maximum latency of the various cores at every latency sample and is used to compare various processors to each other. It is determined by the *Munin* plugin and unrelated to the kernel functionality.

3.5 User-space latency of a uniprocessor determined by *cyclictest*

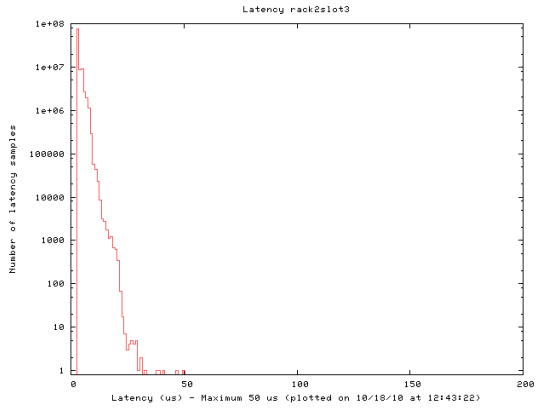

FIGURE 5: *Example of a 6-hour unicprocessor latency plot*

The latency histogram in Figure 5 was recorded during the second load period of Figure 1 to 3. It is based on a total of 100,000,000 wakeup sequences generated with the *cyclictest* call:

```
 cyclictest
-l100000000 -m -a0 -t1 -n \
           -p99 -i200 -h200 -q
```

The maximum detected latency amounts to 50 μs which is in good accordance to the latency of 45 μs obtained using continuous latency monitoring (second load period in Figure 3).

3.6 User-space latency of a multicore processor determined by *cyclictest*

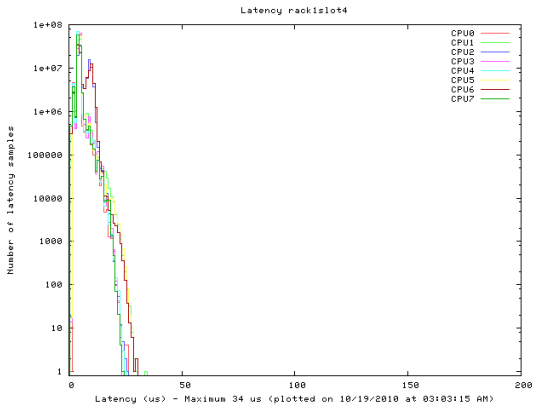

FIGURE 6: *Example of a 6-hour multicore processor latency plot*

The latency histogram in Figure 6 was also recorded during the second load period as depicted in Figure 1 to 3. It is also based on a total of 100,000,000 wakeup sequences; this time, the following *cyclictest* call was used:

```
cyclictest -l100000000 -m -Sp99 -d0 \
           -i200 -h200 -q
```

The maximum detected latency amounts to 34 μs which is in good accordance to the latency of 32 μs obtained using continuous latency monitoring (second load period in Figure 3). The responsible CPU core #1 was detected accordingly as the source of this latency.

4 Testing the reliability of the continuous latency recording

Before the results of the continuous latency recording can be used and trusted, they must be verified against an established procedure. Therefore, continuous monitoring was installed and enabled on a wide variety of systems and the results compared with latency measurements obtained from repeated runs of the *cyclictest* utility on the same system. The continuously recorded timer and wakeup latency values and the latency measured with *cyclictest* were plotted against each other and evaluated using linear regression; an example is given in Figure 7. Latency injection using artifical interrupt and preemption blocking was used to generate a sufficiently large range of system latencies.

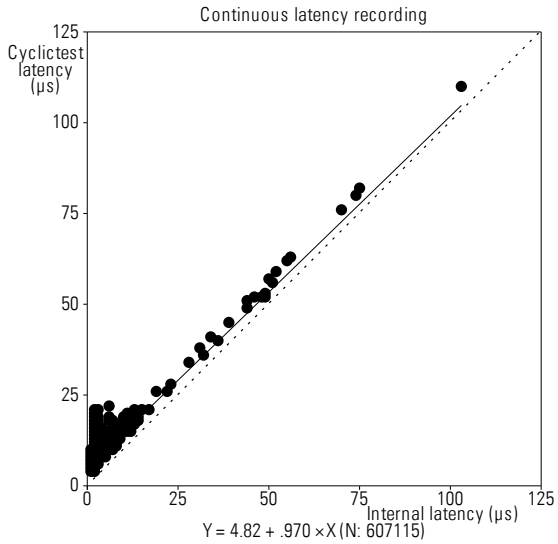

FIGURE 7: *Comparison of continuously monitored latency with cyclictest latency*

This plot shows a typical correlation of the two methods to determine a system's latency. It is based on a total of 607,115 data pairs. The regression line

$$Y = 4.82 + .970 * X$$

yields a small intercept which indicates that the continuously monitored latency underestimates the *cyclictest* results on average by 4.82 μs. This is due to the context switch from system to user space that the continuous recording

cannot assess. In addition, continuous monitoring tends to underestiate small latencies. Large latencies, however, are relatively well correlated.

5 OSADL quality assurance of the PREEMPT_RT Linux kernel patches

The continuous latency recordings and the *cyclictest* histograms are part of the OSADL quality assurance program ot the PREEMPT_RT Linux kernel and are publicly available on the Internet [7]. The systems under test are hosted in special OSADL test racks and form the OSADL QA farm. The above demonstrated uniprocessor and multi-core processors are located in rack #2/slot #3 and rack #1/slot #4, respectively. The other rack slots host many more popular systems that are also continuously tested to ensure production quality of the PREEMPT_RT Linux real-time kernel patches.

6 Conclusion

A method was developed to continuously monitor the apparent worst-case latency of a Linux PREEMPT_RT real-time system. It is based on the existing monitoring of timer expiration offsets and wakeup latencies. The two variables are combined and entered into another latency histogram called "timerandwakeup". The newly created variable was compared with the existing *cyclictest* method. The observed slight underestimation of the latency was expected, since the latency recording from kernel space is unable to determine the time to finish the context switch and start execution is user space. Besides this, there was a good correlation between the two methods. Since continuous recording of the apparent latency does not create any major impact on the system performance, it is recommeded to enable this newly available histogram at least during development. In addition, it is even possible to leave continuous latency monitoring enabled under production conditions. This may help to document the system's long-term performance and also to trace a possible misbehavior, if a violation of the real-time capabilities is suspected.

References

[1] *https://rt.wiki.kernel.org/index.php/Cyclictest*

[2] *http://git.kernel.org/?p=linux/kernel/git/clrkwllms/rt-tests.git;a=summary*

[3] *http://munin-monitoring.org/*

[4] *http://www.mrtg.org/rrdtool/index.en.html*

[5] *http://www.amd.com/us/products/desktop/processors/athlon/Pages/AMD-athlon-processor-for-desktop.aspx*

[6] *http://ark.intel.com/Product.aspx?id=37153*

[7] *https://www.osadl.org/QA/*