Linux PREEMPT-RT v2.6.33 versus v3.6.6: Better or worse for real-time applications?

Hasan Fayyad-Kazan
Electronics and Informatics Department
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels-Belgium
hafayyad@vub.ac.be

Luc Perneel
Electronics and Informatics Department
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels-Belgium
luc.perneel@vub.ac.be

Martin Timmerman
Electronics and Informatics Department
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels-Belgium
martin.timmerman@vub.ac.be

ABSTRACT

Linux was originally designed as a general purpose operating system without consideration for real-time applications. Recently, it became a more reliable candidate in the real-time field due to its daily improvements, both for general purpose and real-time usages. In this research, we test two Linux PREEMPT-RT versions (v3.6.6 and v2.6.33.7) in the aim of benchmarking its performance and behaviour to give an insight whether the enhancements in its kernel are improving the determinism of the operating system. Our benchmark will be based on the following experimental measurements' metrics: thread switch latency, interrupt latency, sustained interrupt frequency, mutex and semaphore acquisition and release durations, and finally the locking behaviour of mutex. These measurements are executed for each Linux version, on the same x86 platform (ATOM processor) using the same test framework and measurement equipment. Comparing the results show that Linux v3.6.6 has significantly better worst case results which makes the actual Linux PREEMPT-RT version a better candidate for RT-applications. Suggestions are made for further improvements.

Keywords

Real-time, Linux, PREEMPT-RT

1. INTRODUCTION

Because of its free open source advantage, stability and supporting multi-processor architecture, Linux operating system (OS) stands high in many (embedded) commercial product developers' favour and becomes one of the fastest-growing (embedded) operation systems [1]. Also, with the development of open-source projects, embedded Linux provides many opportunities for developing customized operation system. Moreover, its reliability and robustness made it widely used even in safety and mission critical systems.

In these contexts, time is extremely important; the system in which its correctness depends not only on the logical results of computations, but also on the time at which the results are produced is called real-time system [2].

EWiLi'13, August 26–27, 2013, Toulouse, FRANCE

Although Linux is a popular and widely used OS, the standard Linux kernel fails to provide the timing guarantees required by critical real-time systems [3]. To circumvent this problem, academic research and industrial efforts have created several real-time Linux implementations [4]. The most adopted solutions are RTLinux/RTCore [5], RTAI [6], Xenomai [7] and the PREEMPT-RT [8] patch. Each one of these real-time enhanced kernels has its internal architecture, its strength and weaknesses [9]. All these approaches operate around the periphery of the kernel, except PREEMPT-RT patch which is mainlined in the current kernel and used by great actors such as WindRiver in their Linux4 [10] solution.

In this research, we evaluate two different versions of Linux PREEMPT-RT, an old version (2.6.33.7) and a newer version (3.6.6). The reason for choosing these kernel versions is that since Linux version 2.6 it started to be possible to get soft real-time performance through a simple kernel configuration to make the kernel fully preemptable, while Linux 3.6 is chosen as it is the latest version at the time of doing the tests. The aim of this evaluation is to have benchmark information that can show whether Linux real-time enhancements and changes since version 2.6 are making it a more reliable real-time operating system or not. For this evaluation, a testing suite of five performance tests and one behaviour test are used. The performance tests are: thread switch latency, interrupt latency, sustained interrupt frequency, and semaphore and mutex acquire-release timing in contention case, while the behaviour test checks the mutex locking behaviour.

2. LINUX PREEMPT-RT

Linux PREEMPT-RT (LinuxPrt) [11, 12] is a Linux real-time patch maintained by Ingo Molnar and Thomas Gleixner [15]. This patch is the most successful Linux modification that transforms the Linux into a fully preemptible kernel without the help of microkernel (the architecture implemented in RTAI or RTLinux) [13]. It allows almost the whole kernel to be preempted, except for a few very small regions of code ("raw_spinlock critical regions"). This is done by replacing most kernel spinlocks with mutexes that support priority inheritance and are preemptive, as well as moving all interrupts to kernel threads [9, 14].

Also, this patch presents new operating system enrichments to reduce both maximum and average response time of the Linux kernel [9]. These enhancements were progressively added to the Linux kernel to offer real-time capabilities. The most important enhancements are: High resolution timers (a patch set, which is independently maintained by Thomas Gliexner [15], which allows precise timed scheduling and removes the dependency of timers

on the periodic scheduler tick [16]), complete kernel preemption, interrupts management as threads, hard and soft IRQ as threads, and priority inheritance mechanism. Some of these new features like threaded IRQ are currently pushed to the mainline kernel by the patch maintainers [9] showing that RT-enhancements also benefit non-RT applications.

3. EXPERIMENTAL SETUP

In order to have maximum control on the Linux behaviour, we decided to build the kernels ourselves using the `buildroot` tool. For Linux 2.6.33.7, we use the real-time patch version 30, while the real-time patch version 17 for the Linux 3.6.6 version is used being the latest available version at the moment of our evaluation tests. Building such Linux kernels where real-time is the main goal requires some precautions. Here is a highlight on some of the configuration options that need particular attention when building the RT kernels to make them more predictable. Those configuration options are the following:

- Enable the "*Optimize for size*" option which can increase cache hits and thus performance as well. This option is typically used in embedded systems.
- Disable the "*Tickless system*" option which generates an operating system clock tick only when a thread wants to be waken-up instead of a periodic one. This improves power consumption and CPU usage. However, the clock tick becomes more complex with this option, and this option does not avoid clock ticks. In the end, one has less but longer ticks which is not good for real-time performance.
- Disable the "*power management*" option which puts the CPU in a lower power state which in turn can take some time to throttle back towards full CPU speed. Again this can impact latency on critical moments and is thus bad for real-time purposes.
- Compiling just the minimal modules because the less code included in the kernel, the less that can jeopardize the real-time behaviour.
- Disable "*memory swapping*" because when memory is swapped away onto some storage medium, the time to retrieve it when needed is a huge, unpredictable and a serious factor slower than when loaded from RAM.

Evaluating these kernel versions is performed using several performance and behaviour tests. The library used between the testing applications and the kernel is the `µClibc` version 0.9.33. Another solution could be `glibc`. However, in smaller embedded systems, `µClibc` can be a better option due to its lower footprint. As `µClibc` only recently introduced NPTL support (from 0.9.32), we decided to use it in our evaluation tests to check its behaviour as well. Before version 0.9.32, `µClibc` could not be used for time-critical systems due to the lack of priority inheritance locking support. This interfacing library is important because user applications (when using POSIX calls) can access the real-time features of the kernel only if the interfacing C library supports them; Otherwise, direct system calls in kernel space are needed, making the application code not portable.

The test application uses `mlockall()` to assure that all test programs are locked into memory. Further, the application is statically linked and started from a RAM disk (`tmpfs`) to avoid swapping out read-only code pages. Finally the real-time run away protection is disabled by setting the kernel configuration parameter `(/proc/sys/kernel/sched_rt_runtime_us)` to (-1). All these precautions improve the determinism of the real-time application and operating system.

We conducted our tests on an ATOM-based platform (Advantech SOM-6760) with the following characteristics:

- CPU: Intel Atom Z530 running at 1.6 GHz
- 32 Kbytes L1 instruction cache
- 24 Kbytes L1 write back data cache
- 512 Kbytes 8-way L2 cache
- 1 core with disabled hyper-threading support
- 512 MB DDR2 RAM
- VMETRO P-Drive PCI exerciser (PCI interrupt D, local bus interrupt level 10)
- VMETRO PBT-315 PCI analyser

4. TESTING PROCEDUERS AND RESULTS

4.1 Measuring Process

A PCI device-simulator (VMETRO P-Drive) is inserted as a peer for the OS and as such, generating measurement samples by device access. It also generates interrupts while doing the interrupts tests, in an independent non-synchronized way with the Platform Under Test (PUT). The test software generates start and stop events by writing a 32-bit word on the PCI bus towards the P-Drive. The traffic on the PCI bus is then captured using PCI-Analyzer (VMETRO PBT-315) which in turn gives us the timing information on the event durations in the system. The major advantage of this system is that the interrupt latency (from hardware interrupt line being toggled to the interrupt handler) can easily be measured and taken into account.

Note that in the tests' comparison figures below, the lower values mean better quality, and all the values in the figures are in µs. As we are interested in real-time behaviour, our focus is mainly on the worst case latency results.

4.2 Performance Metrics

A quick online survey of RTOS metrics maintained by third party consultants, students, researchers, and official records (of distributor companies) reveals that the following three characteristics are used to evaluate a RTOS solution [12]:

- Memory footprint
- Latency
- Services performance

Among these measurements, footprint provides an estimated usage of memory by a RTOS on an embedded platform. The other two characteristics measure various types of RTOS overhead or runtime performance. Latency is reported in two different ways: interrupt and scheduling, and services performance is the minimum time taken by the RTOS interface to complete a system call [12].

In this paper, we test latency and services' performance metrics. Before presenting the evaluation tests and the obtained results, we always perform two preliminary tests (the first 2 tests below) to assure the accuracy and precision of the tests.

4.2.1 Tracing Overhead

This test calibrates the tracing system overhead which is fundamentally hardware related. The results presented in this paper are corrected from this constant bias.

Tracing accuracy depends on the PCI clock (33MHz), as this is the minimum time frame that can be detected. As a consequence, the results in this paper are correct to +/- 0.2 µseconds and are therefore rounded to 0.1 µseconds. The Y-Axis's in the charts are all in µseconds.

4.2.2 Clock Tick Processing Duration

This test examines the clock tick processing duration in the kernel. The results of this test are extremely important as the clock interrupt might disturb all other performed measurements just like it will disturb latencies in real-time applications as well if one uses the same type of hardware platform. This is due to the fact that on the PUT, being a commercial PC motherboard, the tick timer has the highest priority. Using a tickless kernel will not even prevent this from happening (it will only lower the number of occurrences). The tested OS kernels were not using the tickless timer option.

Figure 1 shows a comparison of the average and maximum clock processing durations of each Linux version:

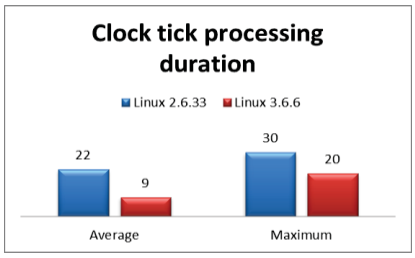

Figure 1: Average and maximum clock tick durations' comparison

As we are focusing the testing on the real-time behaviour and performance of these two Linux versions, the maximum values out of the samples are our concern rather than the average ones. However, for the sake of completeness and comparison, we also publish the average values. Figure 1 shows that Linux 3.6.6 values are better than version 2.6.33.

Before going into the results' details, we give a brief explanation on how the tests are done and the raw measurements are corrected. Figure 2 is an example showing the results over time of the "switch latency between two threads" test performed on Linux 3.6.6. The X-Axis indicates the time when a measurement is performed with reference to the start of the test. The Y-Axis indicates the duration of the measured event.

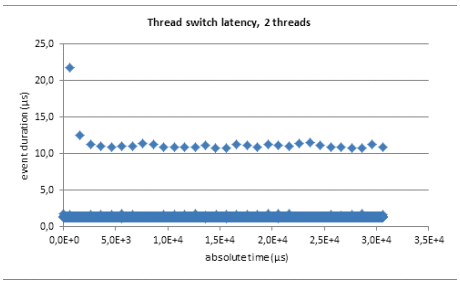

Figure 2: Thread switch latency between 2 threads, on Linux 3.6.6

Figure 2 shows that the average latency is 1.3 µs while the maximum latency is 21.8 µs, which is in this case at the beginning of the test. These values are actually the ones that are used in the comparison figures. The latencies introduced by the operating system clock tick processing can clearly be identified.

4.2.3 Thread Switch Latency between Threads of Same Priority

This test measures the time needed to switch between threads having the same priority. For this test, threads must voluntarily yield the processor for other threads, so `SCHED_FIFO` scheduling policy is used. If we wouldn't use the FIFO policy, a round-robin clock event could occur between the yield and the trace and then the thread activation is not seen in the test trace. The test looks for worst-case behaviour and therefore it is done with an increasing number of threads, starting with 2 and going up to 1000. As we increase the number of active threads, the caching effect becomes visible (the 1000 threads scenario in figure 3a) as the thread context will no longer be able to reside in the cache with higher number of threads used (on this platform the L1 caches are 32KB and 24 KB, both for the instruction and the data caches respectively).Further, one will clearly see the influence of clock interrupts (Figure 3b).

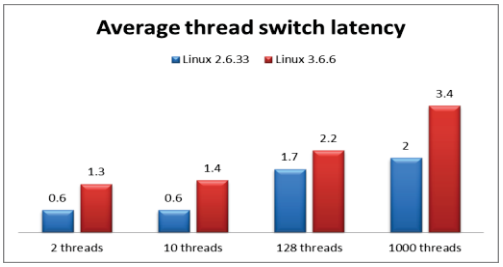

Figure 3a: Average "thread switch latency" tests with different scenarios

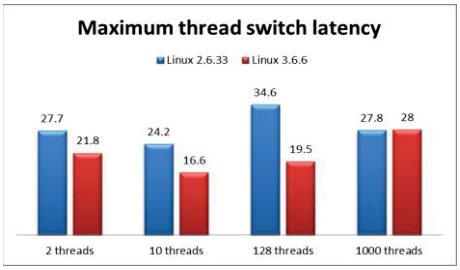

Figure 3b: Maximum "thread switch latency" tests with different scenarios

Analysing the above figures show that the average switch latency (figure 3a) for Linux 3.6.6 is larger than the ones of 2.6.33, while the maximum values (figure 3b) are better than the ones of 2.6.33. This is because the maximum thread latency depends also, in our configuration, on the clock tick interrupts which adds up to the bare maximum latency. A good RT hardware design should put the clock ticker on a minimum interrupt level to enhance these values.

4.2.4 Interrupt Latency

This test measures the time required to switch from a running thread to an interrupt handler. It measures the time from the PCI interrupt line going logical high up to the beginning of the interrupt handler which clears the interrupt condition by accessing the PCI device. Figure 4 shows a comparison between the average

and maximum interrupt latencies for the Linux versions under test.

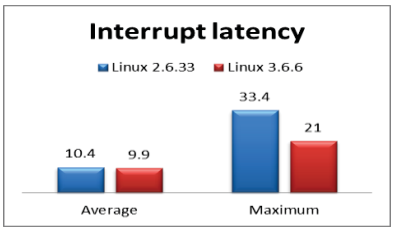

Figure 4: Average and maximum interrupt latencies comparison

For real-time systems, the maximum interrupt latency (worst case duration) needs to be considered. Although one is never sure of obtaining the worst case duration via measurements, dramatically increasing the number of samples may help to get closer to the real worst case value. That's the reason why the long duration interrupt test (test metric 5) is catching a billion of interrupts. The following test is therefore considered of great importance and provides a simple RT-metric for comparison.

4.2.5 Maximum Sustained Interrupt Frequency

This test detects when an interrupt cannot be handled anymore due to the interrupt overload. In other words, it shows a system limit depending on, for example, how long interrupts are masked, how long higher priority interrupts (the clock tick or other) take, and how well the interrupt handling is designed.

This test gives a very optimistic worst case value due to the fact that because of the high rate of interrupts, the amount of spare CPU cycles between the interrupts is limited or nil. Also, depending on the length of the interrupt handler, it might mostly be present in the caches. In a real world environment, the worst case will be greater.

Unless the above considerations, this is a very popular test to compare RTOSs and see where the interrupt handling limit of this RTOSs is.

In this test, 1 billion interrupts are generated at specific interval rates. Our test suite measures whether the system under test misses any of the generated interrupts. The test is repeated with smaller and smaller intervals until the system under test is no longer capable of handling this extreme interrupt load.

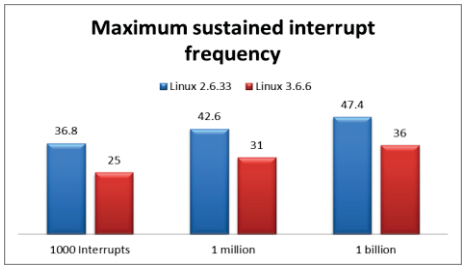

Figure 5: Maximum "sustained Interrupt frequency" comparison

Figure 5 shows that Linux 2.6.33 can handle all the 1000 generated interrupts without missing any one only if the duration between the generated interrupts is 36.8 µs which is 25 µs for Linux 3.6.6. Below these values, both Linux versions start to miss some interrupts.

Further on, the systems were tested by generating bursts of higher number of interrupts (1 million and 1 billion interrupts), which on the long run shows that the guaranteed interrupt duration for Linux 2.6.33 is 47.4 µs (1 billion interrupts scenario) while it is 36 µs for Linux 3.6.6 . This shows that Linux 3.6.6 fares better in handling the interrupts.

4.2.6 Semaphore Acquire-Release Timings in the Contention Case

This test checks the time needed to acquire and release a semaphore, depending on the number of threads pending on the semaphore. In other words, it measures the time in the contention case when the acquisition and release system call causes a rescheduling to occur.

The purpose of this test is to see if the number of pending threads has an impact on the durations needed to acquire and release a semaphore. It attempts to answer the question: "How much time does the OS needs to find out which thread should be scheduled first and is this a constant time?"

In this test, as each thread has a different priority, the question is how the OS handles these pending thread priorities on a semaphore.

Here is the test scenario: 128 threads with different priorities are created (only 90 in Linux OS as it does not support 128 different real-time priorities). The creating thread has a lower priority than the created threads. When the created thread starts execution, it tries to acquire the semaphore; but as this semaphore is taken by the creating thread, the created thread blocks, and the kernel switches back to the creating thread. The time from the acquisition attempt (which fails) to the moment the creating thread is activated again is called here the "acquisition time". This time includes the thread switch time.

After the last thread is created and pending on the semaphore, the creating thread starts to release the semaphore repeating this action the same number of times as the number of pending threads on the semaphore. The moment the semaphore is released, the "release duration" time is started. The highest priority thread that was pending on the semaphore will become active and it will stop the "release duration" time for the current pending thread. The "release duration" also includes the thread switch duration.

The testing results show that the number of threads pending on a semaphore has NO impact on the release time periods, which is a good result for both Linux versions as this means that they behave independently from the number of queued threads and as such keep a predictable response.

Figures 6a and 6b show a comparison between the average and maximum acquisition and release durations for both Linux versions.

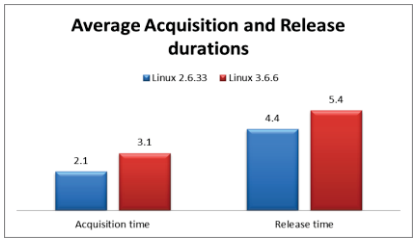

Figure 6a: Semaphore average "acquisition and release durations" comparison

Figure 6a shows that the average acquisition and release durations of Linux 3.6.6 are higher than the ones of Linux 2.6.33. This is because the acquiring and releasing durations include the "thread switch duration" in their measurements, which confirms the other results that indicate an increase in thread switch latency in the newer kernel.

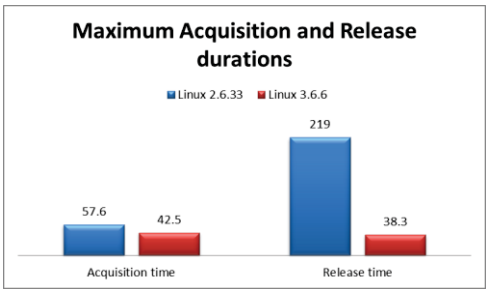

Figure 6b: Semaphore maximum "Semaphore acquisition and release durations" comparison

We noticed in Linux version 2.6.33 that for some reason, worst case behaviour was very bad in this test (219 µs). As we do black box testing, the reason for this behaviour was unclear. It was not related to the number of threads pending on a semaphore.

4.2.7 Mutex Locking Behaviour

This test checks the behaviour of the mutex locking primitive using the `pthread_mutex_lock` and related POSIX calls. This test creates three threads. The low priority thread starts first. It creates a semaphore with count zero and starts the medium priority thread which activates immediately. The medium priority thread tries to acquire the semaphore, but as the semaphore count is zero, it blocks on the semaphore. The low priority thread continues execution; it creates a mutex for which it takes ownership and starts now a high priority thread which activates immediately. The high priority thread tries to acquire the mutex owned by the low priority thread and blocks also. The low priority thread resumes operation. Now the low priority thread releases first the semaphore and then the mutex. Remark that all threads of the described test are locked to the same processor avoiding parallel execution on a multi-processor system.

In case the mutex supports priority inheritance, whenever the high priority thread requests the mutex, the low priority thread that has the ownership of the mutex will inherit the priority of the high priority thread that requested for the mutex ownership, and will do its job at this high priority. As a result, the low priority thread will first release the semaphore which will not wake-up the medium priority thread because the low priority thread is still running at high priority level and the low priority thread will continue its execution until it releases the mutex that was requested by the high priority thread. After releasing the mutex, the high priority thread will unblock and the low priority thread goes back to its normal priority level execution state (low priority level). Now, the high priority thread becomes active, preempts the low priority thread and executes until it finishes its job. In the non-priority inheritance case, the medium priority thread will start upon the semaphore release and actually block the high priority thread for a long time, which results in priority inversion.

We do not deal with the priority ceiling mechanism in this paper because Linux limit its support to the priority inheritance mechanism.

Priority inversion avoidance mechanism was one of the first PREEMPT-RT achievements that were done in the mainstream kernel, so we did not expect any problems. Priority inversion behaves as expected. This also proves that the `uClibc` NPTL implementation provided kernel priority inheritance. `uClibc` versions before version 0.9.32 did not had this support.

4.2.8 Mutex Acquire-Release Timings in the Contention Case

This test is comparable to the previous test metric 7, but performed in a loop. Here we measure the time needed to acquire and release the mutex in the priority inversion case. This test is designed in order to enforce a thread switch during the acquisition:

- The acquiring thread is blocked
- The thread with the lock is released.

The acquisition time starts from the moment a mutex acquisition is requested by a thread until the activation moment of the lower priority thread with the lock.

Note that before the release, an intermediate priority level thread is activated (between the low priority one owning the lock and the high priority one asking the locked resource). Due to the priority inheritance, this thread does not start running (the low priority thread owning the lock, inherited the high priority of the thread requesting the locked resource).

The release time is measured from the moment of the release call until the moment the thread requesting the mutex is activated. This measurement also includes a thread switch.

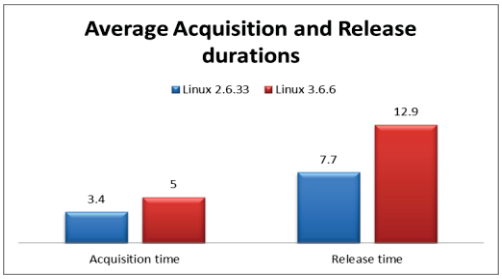

Figure 7a: Average "Mutex acquisition and release durations" comparison

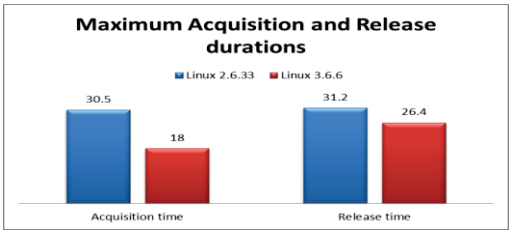

Figure 7b: Maximum "Mutex acquisition and release durations" comparison

Both Linux versions behave strangely here in a way that a mutex release seems to cause a double thread switch (time take even more than double as long). As the required time for a thread switch latency has increased in Linux 3.6.6, it is normal that the average durations (Figure 7a) are increased. The maximum results (Figure 7b) are of more importance than the average results. We

see that Linux 3.6.6 acquisition and release durations are better than the ones of Linux 2.6.33.

5. CONCLUSION

This paper showed a comparison between the average values and the maximum ones between two Linux versions (Y-Axis values are in µs). As the aim of this paper is testing the real time behaviour and performance, the maximum values are our concern. Here is a comparison summary of all the performed tests together with their measured worst-case maximum values.

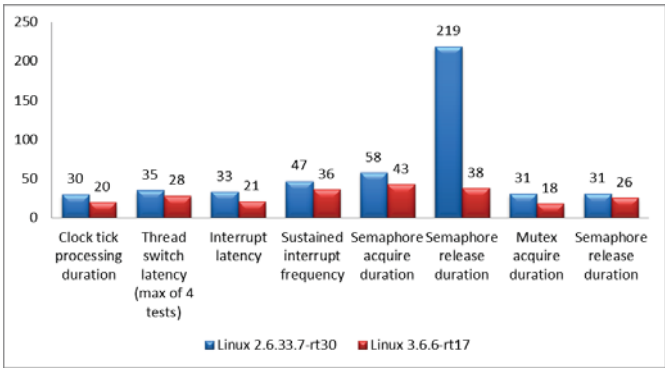

Figure 7: Comparison of all the performed tests together with their measured worst-case values.

This figure clearly shows that Linux 3.6.6 has improved in reference to Linux 2.6.3. Calculating the average percentage of all these improvements showed that Linux 3.6.6 real time performance and behaviour has improved by 35 %, which is a good step forward for better real-time Linux.

Although the average thread switch latency and the double thread context switch during the priority inversion case still need to be worked on, the version 3.6.6 is now a candidate to be considered in a new RT design.

6. REFERENCES

[1] K. Song and L. Yan, "Improvement of Real-Time Performance of Linux 2.6 Kernel for Embedded Application," in *International Forum on Computer Science-Technology and Applications*, Chongqing, 2009.

[2] J. Stankovic and K. Ramamritham, "What is predictability for real-time systems?," *Real-Time Systems,* vol. 2, no. 4, pp. 247-254, 1990.

[3] P. Regnier, G. Lima and L. Barreto, "Evaluation of Interrupt Handling Timeliness in Real-Time Linux Operating Systems," *ACM SIGOPS Operating Systems Review,* vol. 42, no. 6, pp. 52-63, 2008.

[4] C. Zujue, L. Xing and Z. Zhixiong, "Research Reform on Embedded Linux's Hard Real-Time Capability in Application," *International Conference on Embedded Software and Systems Symposia,* pp. 146-151, 29 July 2008.

[5] FSMLabs, "High Performance and Deterministic System Software-FSM Labs," [Online]. Available: http://www.fsmlabs.com/.

[6] P. d. M. -. D. d. I. Aerospaziale, "RTAI-Official website," [Online]. Available: https://www.rtai.org/.

[7] Xenomai, "Xenomai: Real-Time Framework for Linux," [Online]. Available: http://www.xenomai.org/.

[8] M. Mossige, P. Sampath and R. Rao, "Evaluation of Linux rt-preempt for embedded industrial devices for Automation and Power Technologies," in *Proceedings of the Ninth Real-Time Linux Workshop*, 2007.

[9] N. Litayem and S. Ben Souad, "Impact of the Linux Real-time Enhancements on the System Performances for Multi-core Intel Architectures," *International Journal of Computer Applications,* vol. 17, no. 3, 2011.

[10] WindRiver, "The First with the Latest: Wind River Linux 4," [Online]. Available: http://www.windriver.com/announces/linux4/.

[11] P. McKenney, "A realtime preemption overview," [Online]. Available: http://lwn.net/Articles/146861/.

[12] S. Rostedt and D. V.Hart, "Internals of the RT Patch," in *Proceedings of the Linux Symposium*, 2007.

[13] K. Dongwook, L. Woojoong and P. Chanik, "Kernel Thread Scheduling in Real-Time Linux for Wearable Computers," *ETRI Journal,* vol. 29, no. 3, pp. 270-280, 2007.

[14] S. Arthur, C. Emde and N. McGuire, "Assessment of the Realtime Preemption Patches (RT-Preempt) and their impact on the general purpose performance of the system," in *Real-Time Linux Workshop*, Linz-Austria, 2007.

[15] RTwiki, "CONFIG PREEMPT RT Patch," [Online]. Available: https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch

[16] eLinux.org, "High Resolution Timers," [Online]. Available: http://elinux.org/High_Resolution_Timers.