Medicon Engineering Themes
Volume 5 Issue 6 December 2023
Article Type: Research Article
ISSN: 2834-7218

Measurement of Kernel Preemption Time of Real Time Operating Systems

Roshan Chitrakar[1]* and Arun Timalsina[2]

[1]*Department of Graduate Studies, Nepal College of Information Technology, Nepal*

[2]*Department of Computer and Electronics Engineering, Pulchowk Campus, IoE, Tribhuvan University, Nepal*

***Corresponding Author:** Roshan Chitrakar, Department of Graduate Studies, Nepal College of Information Technology, Balkumari-8, Lalitpur, Postal Code 44600, Nepal.

Received: November 08, 2023; **Published:** November 30, 2023

DOI: 10.55162/MCET.05.177

Abstract

In a Real Time System, when a high priority task is scheduled, it preempts ongoing lower priority tasks and sometimes it pre-empts even the kernel too. In normal cases, the time needed for the preemption is too small and hence is not considered in task scheduling. But, in some mission critical hard real time systems, even a small fraction of a second should be taken into account as it could affect the deadline. This research work proposes two models - embedded model and software model, among which the latter has been used for the experiment that calculates the kernel preemption time. The hardware model consists of a triangular wave generator called TRIANG that takes timer values from real time tasks and sends outputs to a comparator module. Finally, the comparator compares the time differences, and hence, calculates the kernel preemption time. Whereas the software model uses the RT-Linux architecture in which device drivers are used instead of hardware components. A high precision timer clock is also used to measure the task release times.

The results from the experiments show that the release time jitter due to the kernel preemption is found to be fluctuating in the range of -94 and +93 microseconds in 3% of the cases. So, the kernel preemption time for the hard and mission critical real time systems is taken to be less than 99 microseconds. Whereas 85% of the cases have release time jitters of single digit only. So, the kernel preemption time in average case is considered to be less than 9 microseconds.

Keywords: Real Time System; Triangular Wave Generator; Task Scheduling; Kernel Preemption Time

Abbreviation

Acronym	*Full form*
API	Application Programming Interface
GPES	GPGPU Preemptive Execution System
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HRT	Higher Resolution Timer
IETS	Instruction Execution Time Simulator
KPT	Kernel Pre-emption Time

OS	Operating System
RT	Real Time
RTOS	Real Time Operating System
RTS	Real Time System
TMT	Time Measuring Tool
TRIANG	Triangular Wave Generator
WCET	Worst-Case Execution Time

Introduction

A Real Time System is in general categorized into three types- Hard, Soft and Firm Real Time System according to the usefulness function on the basis of their timing constraints i.e. deadline [1]. Hard RTS is subject to crash or may cause physical hazards upon not meeting the deadline. A mission critical system, for example, always has a hard deadline [2]. In order to schedule the tasks of the system, all parameters including temporal and resource parameters for task preemption should be known well in advance [3]. The correctness of the timing measurements is vital in this regard because task should be executed exactly at the one time (considering the task execution time and delays due to other overheads) in order to meet the deadline at the given time [4].

A few studies on measuring the time of task switching, mode changes and context switching have shown that schedulability test of a real time task with a hard deadline is affected by these system overheads [5-7]. So, the question is obviously valid as to whether a kernel preemption time in a real time or embedded system can affect the schedulability test [8]. The failure and tragic-crash of some satellites and spacecrafts have widened the scope of the study like this one [9-11].

Real time tasks often require task priority changes and preemption of tasks for the successful execution i.e. meeting deadlines effectively [12]. So it is recommended that an operating system with preemptible kernel be used for any real time system. The capability of kernel preemption gives rise to efficiency in scheduling real time tasks and successfully completing them in the deadline [13].

Preemptible kernels provide for one user process to be preempted in the midst of a system call so that a newly awaken higher-priority process can run [14]. This preemption cannot be done safely at arbitrary places in the kernel code. One section of code where this may not be safe is within a critical section, which must not be executed by more than one process at the same time [6]. In some OSes, critical sections are protected by spin locks and the preemptible kernel alters the call to spin locks by preventing the preemption, and allows preemption in other sections [15]. When a higher-priority process awakens, the scheduler will preempt a lower-priority process in a system call if the system call code has not indicated, via the modified spin-lock code, that preemption is not possible [16].

In addition, with a preemptible kernel, breaking locks to allow rescheduling is simpler than with the preemption (low-latency) patches [17]. If the kernel releases a lock and then re-acquires it, when the lock is released, preemption will be checked for. When a spin lock is acquired, the counter is incremented. When a high-priority process awakens, the scheduler checks to see whether the preemption counter indicates, by having the value zero, that preemption is allowed [18]. By employing a counter, the mechanism works when locks are nested. With this mechanism, however, any spin-lock-held critical section prevents preemption, even if the lock is for an unrelated resource [19].

A typical real-time system has tasks that must be executed in a deterministic and real-time manner [20]. An idea is to design a sub-kernel in such a way that the real-time portion of the application is handled separately by this portion [21]. Thus, these real-time sub-kernels deliver an API for tasking, interrupt handling and communication with processes. The main operating system is suspended while the sub-kernel's tasks run i.e. dealing with an interrupt - specially hardware interrupt, by not allowing to disable the interrupts [22].

Therefore, this work is focused in designing a method of measuring kernel pre-emption time of a real time operating system by utilizing a real time sub-kernel of RTLinux. The main contribution of the research is proposition models that are used to measure the Kernel Preemption Time of RTOS.

Related Work

Many research works have been done regarding measurement and time analysis of different events or processes of real time operating systems. Faller, Newton has proposed a method that does not require an external measuring tool for collecting information about context-switching and interrupt-acknowledge times in UNIX-like operating systems. This paper has discussed the probable sources of latency and illustrated with actual results obtained by the application of the method to TROPIX, a real-time UNIX-like operating system, running on a Motorola 68010-based computer [23]. In another research work, Holenderski et al. have presented a tool called "Grasp" that traces, visualizes and measures the behavior of real-time systems by also providing a simple plugin infrastructure for customizing them. The Grasp has been demonstrated based on experiences during the development of various real-time extensions for the commercially available µC/OS-II real-time operating system [24]. Zhou et al. have presented an efficient GPGPU preemptive execution system (GPES), which combines user-level and driver level runtime engines to reduce the pending time of high-priority GPGPU tasks that may be blocked by long-freezing low-priority competing workloads. They have demonstrated that GPES is able to reduce the pending time of high-priority tasks in a multitasking environment by up to 90% over the existing GPU driver solutions, while introducing small overheads [25]. Marchesotti et al. has studied a major sources of unresponsiveness in the kernel, namely the presence of non-preemptible code sections, and have quantified its effects with a series of micro-benchmarks. They have identified which Linux kernel's modules can cause erratic behavior with respect to the responsiveness of the Linux kernel to hardware interrupts [26].

Stärner and Lars has presented a simulation framework suitable for examining the cache interference cost in preemptive real-time systems. They have shown a significant performance gap between the best- and worst-case execution times and that the worst-case performance of some software modules was found to be more or less independent of the cache configuration [27]. Similarly, Blackham et al. have presented a WCET analysis of seL4, world's first formally-verified operating-system kernel. They concluded that by featuring machine-checked correctness proofs of its complete functionality, seL4 is an ideal platform for safety-critical and time-critical systems. They also concluded that sel4 has created a foundation for integrating hard real-time systems with less critical time-sharing components on the same processor, supporting enhanced functionality while keeping hardware and development costs low [28]. In a similar work, De Oliveira et al. have traced a parallel between the theory of response-time analysis and the abstractions used in the Linux kernel. They have first identified the PREEMPT_RT Linux kernel mechanisms that impact the timing of real-time tasks and map these impacts to the main abstractions used by the real-time scheduling theory. Then, they have described a customized trace tool, based on the existing trace infrastructure of the Linux kernel, that allows the measurement of the delays associated with the main abstractions of the real-time scheduling theory. Finally, they have used this customized trace tool to characterize the timing lines resulting from the behavior of the PREEMPT_ RT Linux kernel [15].

Efforts have already been made in the area of measuring the kernel preemption time of real time operating systems. In this course, Adam, George K. has performed real-time measurements of Linux kernels with real-time support provided by the PREEMPT_RT patch on embedded development devices such as BeagleBoard and Raspberry Pi and proposed a new measurements software. He demonstrated that the PREEMPT_RT patch overall improves the Linux kernel real-time performance compared to the standard one. The reduced worst-case latencies on such devices running Linux with real-time support could make them potentially more suitable for real-time applications as long as a latency value of about 160 µs, as an upper bound, is an acceptable safety margin [29].

But, the problem is still there in recording the exact amount of time involved in just preempting the task or the kernel itself that could have been used in executing the real time task otherwise. During the schedulability analysis, more accurate timing calculations could be done if all of these timing requirements are known a-priori. Therefore, this study work is focused in designing a technique to measure the time spent on non-RT tasks including preemption, task switching and context switching etc and this study also carries an experimentation to measure the time spent in kernel preemption by the system.

This research work makes use of commonly available RTOS that support kernel preemption and are used in uni-processor environment. It makes use of timers and clocks that are used by a Real Time scheduler through number of system calls. It also takes into account every time duration spent by a real time operating system in the course of scheduling and preempting a real time task, stealing resources from tasks and the kernel preemption time as well. The result of the study could be used in designing any real time or embedded systems - specially in the mission critical systems or any system that has hard deadline.

Measuring the time of any CPU activity or system activity needs more accurate methods - sometimes may require designing a device as well. The core idea of this study work is to suggest a technique as to how the time could be measured accurately in terms of the smallest clock level.

Proposed Models

The conceptual model is based on a uni-processor real time operating system environment. It contains a collection of processes running, the interrupt activity and the activity of hardware devices. The environment is assumed so robust that any number of any kinds of applications can be run concurrently and still performs acceptably.

This study proposes two conceptual models:

1. Embedded model.
2. Software model.

Model 1: Embedded Model

There is a device or a software simulator called the TRIANG that generates a triangular sweep wave [30]. Besides, 'n' numbers of tasks with real time priorities are scheduled. Both the waves and the jobs share the central memory as shown in Figure 1. Obviously, the timer is associated with all the tasks and also to the TRIANG. The system calls provided by the RTOS are used to activate timer functions.

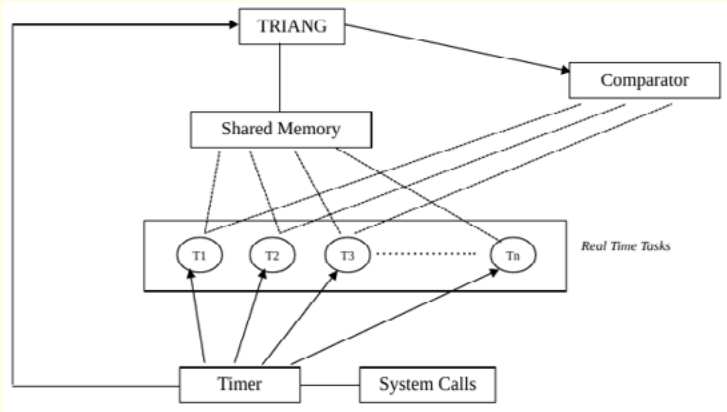

Figure 1: Embedded Model of the Kernel Preemption Measurement.

Being the main role of the model, the comparator (either a hardware device or a piece of software) compares real-time task parameters and calculates the kernel preemption time as well.

Model 2: Software Model

This model suggests the same experiment to be carried out twice on the same Operating System - once without the preemption, and

the next with the preemption enabled [31]. The main goal of the experiment is to record temporal parameters (e.g. execution time, release time etc.) of the jobs in both the cases. The preemption time is calculated as the delay in releasing the jobs in the preemptible environment.

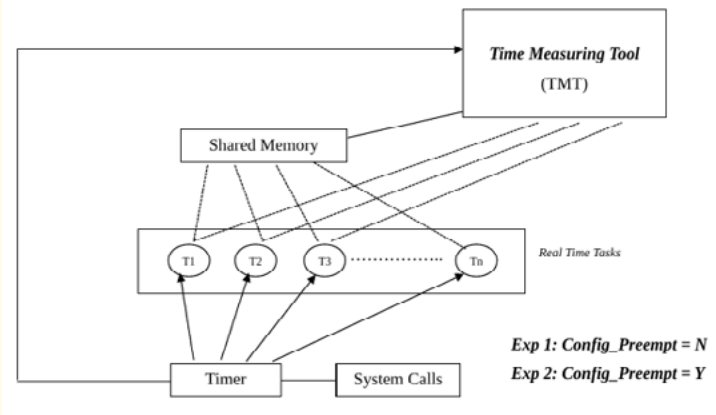

Figure 2: Software Model of Kernel Preemption Measurement.

The Embedded model is altered a bit in order to obtain the Software model as show in Figure 2. The triangular wave generator is absent here and the comparator is replaced with a software tool to which I have given the name "*Time Measuring Tool*" that could be one of the following:

1. Instruction Execution Time Simulator (IETS).
2. Benchmarking Tool.
3. A Program code written manually.

Instruction Execution Time Simulator

The calculation of instruction times requires more than just the information supplied in the CPU manufacturer's data books. It is also dependent on memory access times and wait states that are determined by the overall system design. Some companies that frequently design real-time systems on a variety of platforms use simulations programs to predict instructions execution time and CPU throughput. Then engineers can input the CPU type, memory speeds, and an instruction mix, and calculate total instruction times and throughput.

Benchmarking Tool

A readymade tool may be obtained from software houses or a company that is working on time measuring projects. Considerations must be taken of different issues like the time taken by the tool itself to activate measuring the time (latency) or the blocking of the tool by OS kernel or extra operating overhead plus real time considerations and so on.

A Program Code Written Manually

Short sections of code can be written to measure timing parameters by reading the system clock before and after the execution of the task code. The time difference can then be measured to determine the actual time of execution. Of course, if the code normally takes only a few microseconds, it is better to execute several thousand times and then divide the total execution time by the time spent. This will help remove any inaccuracy introduced by the granularity of the clock.

Experimental Model

Between the two models, the software model was chosen for the experiment while another model is left as a separate research work. RTLinux was selected as the OS kernel due to its superiority among others [14].

The model makes use of virtual devices that are served by device drivers. Unlike any other user given tasks, device drivers are not preemptible and therefore any request to the device runs to the completion. One virtual device generates busy looping inside the kernel and the other regulates a periodic task to run inside the kernel [32]. In the course of the completion of the operations in the devices through the drivers, it surely preempts the kernel in case of the preemptible kernels. And, in case of non-preemptible kernel, it blocks the task in the worst case. Instead of generating enough interrupts from any hardware, this model shown in Figure 3 makes sure that a tremendous amount of software interrupts is generated which is sufficient enough to preempt the kernel.

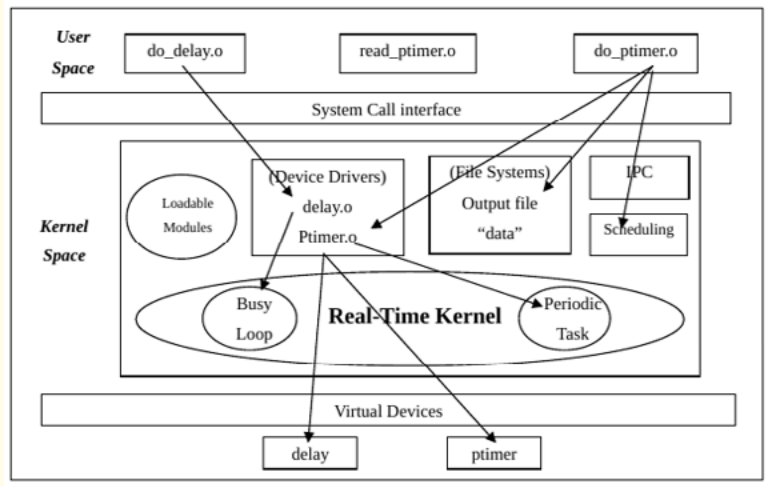

Figure 3: Experimental Model based on RTLinux Architecture.

The *Time Measuring Tool* is simulated by the user programs themselves. These programs do specific operations to the devices through device drivers. A real time periodic task is also defined, regulated and controlled by these programs. Various system calls are used to pass information to the Kernel space and real time parameters to the scheduler.

The key concept of the model is that it measures the intervals of two consecutive periodic releases of the real time task - giving rise to the measurement of the release time jitters due to the kernel preemption delay. Essentially, the measurement is done for a several thousand releases of the task. The worst time is considered to be the required measurement.

The model suggests the same experiment to be carried out once again without the preemption enabled in order to show delays in release time due to the interrupt blocking and thus by supporting the previous results [14].

The Higher Resolution Timer in Linux

The Linux kernel time sub system is built on top of jiffies - the time resolution ranging from 1 ms to 10 ms. An "Open Source Project" has tried to improve the resolution by introducing sub jiffies in the jiffy-based clock. In this attempt, a time instance is represented as (jiffy + sub jiffy) to give higher resolution in the clock. Jiffies are managed by the common time code, while sub-jiffies are managed by arch-specific code as and when necessary. This split in ownership and management causes problems - mainly resulting in convoluted code and tricky bugs. There are some other issues like aliasing of the same time instance etc.

Jun Sun has proposed a new idea. His idea is to completely revamp the time system with native high resolution from hardware to all the way up [33]. The existing jiffy time subsystem can be initially running on top of an emulation layer and later completely removed when users of time subsystem move to the new interface. The key of the new idea is sometime called monotonic time. The monotonic time is built on top of an abstraction of hardware clock, called tock. At last and the on top of all the clocks is the Wall time, or calendar time, is built on top of monotonic time. (See appendices for the code of the timer files "tock.h", "mtime.h", and "wtime.h"). The three-layered notion about time, i.e., tock, mtime and wtime, are simple and yet technically sound. They should satisfy all potential time and timer need. It is simple way to provide high resolution. It transforms Linux to a tick-less operating system, making power management easier too. The transition path from jiffies to jiffy-less is clear and easy.

However, there are some disadvantages in this HRT also. 64-bit manipulation is inefficient on 32-bit machines. A couple of tactics should lessen this effect, but won't eliminate it. Therefore, 64-bit operation abstraction is to be used so that those operations can be efficient on 64-bit machines. Time should be dealt with latch-and-use style to reduce locking need, i.e., transient 64-bit variables should be latched into local 64-bit variables and then operations are done to local 64-bit variables.

Jiffy-subsystem gets inefficient on the emulation layer. The overhead mainly comes from re-inserting the timer back to the list to emulate jiffy interrupts. First of all this overhead is not big. Secondly a couple of things that could help this: make use of tock's periodic alarm, and/or introduce periodic timers in addition to one-shot timers.

With the new high-resolution interface it is conceivable system will have more overhead for a lot of timers expiring in a short time of period. One possible counter-measure is to allow timer users to specify the resolution they want. A low resolution can be set as default system-wide. Timers with lower resolution are coerced to align along their resolution boundaries.

Timer in the Experiment

It would be excellent to have the experiment carried out with utilizing the jiffy-less clock as suggested by Jun Sun but this is left for another piece of work, as the 64-bit timer has not been tested thoroughly in 32-bit operating systems. This experiment is based on "time_bh" and the default tick-based jiffy clock is split into greater resolution that is able to measure the time in terms of microseconds.

Experimental Detail

The experiment is designed to measure jitters in release time of a real time periodic task while two virtual device drivers are generating interrupts inside the kernel [34]. The task has a static priority and is supposed to be released at every ***n*** microseconds, where ***n*** can be set manually in the program code. The hardware is simulated with the help of virtual device drivers. The experiment suite collects intervals between two consecutive instance release times. Additional workload to the system may be working during the testing period e.g. MP3 video player, CD burning or CPU intensive processes etc.

The experiment is carried out once each in two different kernel configurations - the first with the preemption disabled and the next with enabled. This is done just to distinguish the delay due to kernel preemption from the time of blocking of the task. It also makes sure that the timing difference is nothing but just because of the kernel preemption in the latter case. The task period is set to have 10000 (ten thousand) micro seconds; the no. of periods is set to be 5000 (five thousand) times with the fixed priority of 10.

The experiment is carried out in an IBM compatible computer with the following specifications: - Intel Pentium 1.7 GHz of CPU, 256 MB of RAM. RedHat Linux 9.0 (Shrike) with Kernel 2.4.20-8 with the real time kernel RTLinux 3.2 Pre 2 was chosen as the operating system.

Findings of the Experiment

The result showed that the periodic real time task with kernel preemption enabled has very less jitters compared to its defined period. Among 5000 continuous periods of the task, the differences of two consecutive release times are reported to have less, equal and more than the defined period as shown by Table 1:

<10000 sec	*10000 sec*	*>10000 sec*	*Total*
2853	568	1579	5000
57.06%	11.36%	31.58%	100.00%

Table 1: Difference of the Consecutive Release Times.

The earliest release and the latest release times with the jitters are shown in Table 2.

	Release Time Difference	*Defined Period*	*Jitters*	*Percentage*
Min	9906	10000	94	0.94%
Max	10093	10000	93	0.93%

Table 2: Min and Max data observed from the experiment.

It is interesting observation that the occurrence of the min and max jitters are only once each - contributing only 0.04% of the total.

Jitters	*Count*	*Percentage*
0	568	11.36%
0-9	4270	85.40%
10-99	162	3.24%
Total	5000	100.00%

Table 3: Categorical distribution of release time jitters.

Hence, it would be worth looking at the categorical distribution of the jitters measured. All jitters are grouped into 3 (three) categories: - No jitters at all (0), One digit number (0-9) and two-digit number (10-99). Here is Table 3 showing the occurrences of jitters and their percentage with respect the data collected.

Jitters	*Count*	*Percentage*
0	568	11.36%
0-9	4270	85.40%
10-19	94	1.88%
20-29	25	0.50%
30-39	15	0.30%
40-49	14	0.28%
50-59	8	0.16%
60-69	3	0.06%
70-79	0	0.00%
80-89	1	0.02%
90-99	2	0.04%
Total	5000	100.00%

Table 4: Release Time Jitters categorically broken-down.

We can also look at the more categorical breakdowns as shown in Table 4. We can see in the table that most of the jitters are fluctuating between 10 microseconds whereas very few periods have release time jitters of 50 and more. We also notice that only one or two periods are found to have release time jitters of 60 or more.

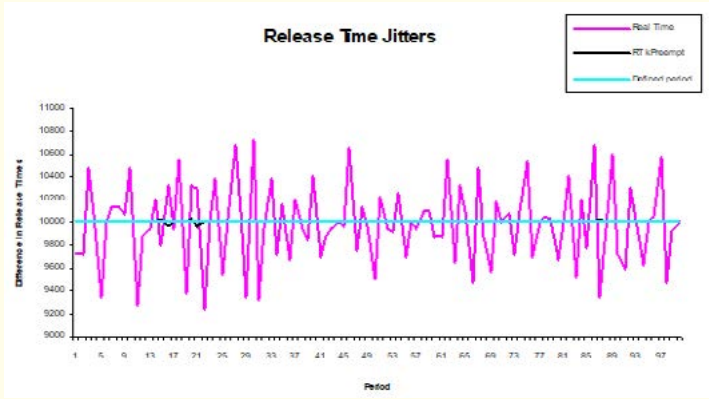

Figure 4: Release Time Jitters of 100 Periods.

All the collected data are shown with the help of a line chart. A number of line charts are prepared with increasing number of data collected. All the line charts have data series of "preemption disabled" case also because of the intent of showing the interrupt blocking time as well.

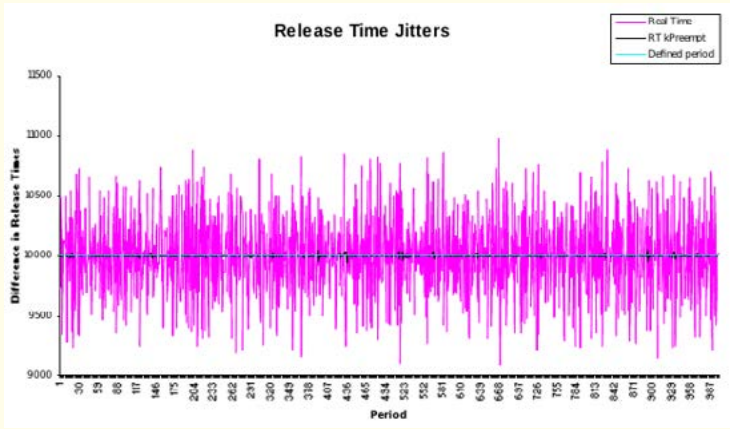

Figure 5: Release Time Jitters of the First 1000 Periods.

The central ideal straight line is the defined period showing the value of 10000 microseconds at each period. The dark line touching above the ideal line is the preemptible case and the most curved line represents the non-preemptible case. The graph shown by Figure 1 shows the 100 collected data. In here, the preemptible case has a uniformity and very less fluctuation at the ideal line at 10000.

The two graphs Figure 5 and Figure 6 show 1000 (one thousand) and 5000 (five thousand) data respectively. The idea of plotting such a successive line graph is to see whether any greater fluctuation is observed gradually at any point. Now, there are a few countable amounts of jitters seen during this long period. The dark vertical lines coming up and going down are the points of remarkable jitters.

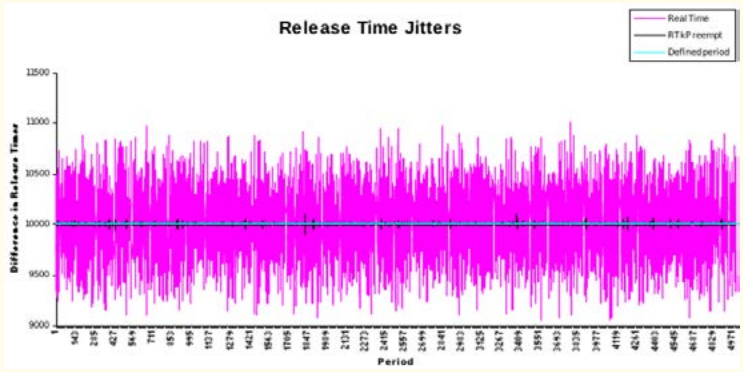

Figure 6: Release Time Jitters of all 5000 Periods.

Discussion

From the experiment carried out, it is seen that the release time jitter due to the kernel preemption is found to be fluctuating in the range of -94 and +93 microseconds - the minus sign (-) meaning the earlier release and the plus sign (+) the later release. Unlike in a non real time system where average case is calculated, in any real time case, we always consider the worst case.

Therefore, the kernel preemption time is accepted to be 94 microseconds. But, from the statistics shown above implies that 85% of periods have release time jitters only of less than two digits. So, the kernel preemption time in average case is considered to be less than 10 microseconds. However, in mission critical hard real time systems, we cannot ignore even the 3% occurrence of less than 100 microseconds' delay.

In the non-preemptible case, the greater time difference found is due to interrupt blocking of the task by the kernel. This experiment distinguishes the delay due to the blocking from the kernel preemption time. The non-preemptible case is only the supporting experiment to the preemptible case.

Earlier works that used similar models have produced the average Kernel Preemption Time of a few milliseconds only, whereas the model proposed in this work has been able to measure the average Kernel Preemption Time of microseconds level. Therefore, the novelty of the proposed model has been justified by reducing the Kernel Preemption Time by 10 times.

Conclusion and Future Work

This research has proposed two models for measuring Kernel Preemption Time of Real Time Operating Systems. An experimentation has also been carried out using the software model. As a result from the experiment, the KPT for general RT tasks is measured to be 9 (nine) microseconds. Whereas, KPT for mission critical RT tasks, it is measured to be 99 (ninety nine) microseconds.

Here are two further works that can be done as extension of this work viz. (1) Revise and fine-tune the experiment suite to get rid of a few cases of more 50 microseconds kernel preemption delay i.e. 14 occurrences. Look at the possibility of reducing the delay of more than 10 microseconds; and (2) Carry out a separate experiment as proposed in the Embedded Model.

References

1. M Blej and M Azizi. "Comparison of Mamdani-type and Sugeno-type fuzzy inference systems for fuzzy real time scheduling". International Journal of Applied Engineering Research 11.22 (2016): 11071-11075.

2. R Luna and SA Islam. "Security and reliability of safety-critical rtos". SN Computer Science 2 (2021): 1-14.
3. IB Hafaiedh and MB Slimane. "A parameterized formal model for the analysis of preemption-threshold scheduling in real-time systems". IEEE Access 8 (2020): 58180-58193.
4. P Pazzaglia and M Maggio. "Characterizing the Effect of Deadline Misses on Time-Triggered Task Chains". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41.11 (2022): 3957-3968.
5. X Jiang., et al. "Real-time scheduling of parallel tasks with tight deadlines". Journal of Systems Architecture 108 (2020): 101742.
6. BB Brandenburg. "Multiprocessor real-time locking protocols: A systematic review". arXiv preprint (2019): arXiv:1909.09600.
7. H Choi, H Kim and Q Zhu. "Toward practical weakly hard real-time systems: A job-class-level scheduling approach". IEEE Internet of Things Journal 8.8 (2021): 6692-6708.
8. J Qiao, H Wang and F Guan. "A multiprocessor real-time scheduling embedded testbed based on Linux". International Journal of Embedded Systems 14.5 (2021): 451-464.
9. TJ Sena. "Providing Clarity for Fault-Based Liability in International Space Law: A Practical Approach through Principles of General International Law". J. Space L 46 (2022): 1.
10. DTC HAS. "Dr. Connolly named chancellor again".
11. C Rockwell. Treachery in Outer Space. Good Press (2023).
12. J Lee., et al. "Optimal priority assignment for real-time systems: a coevolution-based approach". Empirical Software Engineering 27.6 (2022): 142.
13. F Marković. "Preemption-Delay Aware Schedulability Analysis of Real-Time Systems". PhD Thesis, Mälardalen University (2020).
14. F Reghenzani, G Massari and W Fornaciari. "The real-time linux kernel: A survey on preempt_rt". ACM Computing Surveys, CSUR 52.1 (2019): 1-36.
15. DB de Oliveira., et al. "Automata-based formal analysis and verification of the real-time Linux kernel" (2020).
16. BB Brandenburg, JM Calandrino and JH Anderson. "On the scalability of real-time scheduling algorithms on multicore platforms: A case study". in 2008 Real-Time Systems Symposium, IEEE (2008).
17. G Gala, I Kadusale and G Fohler. "Joint time-and event-triggered scheduling in the linux kernel". arXiv preprint (2023): arXiv:2306.16271.
18. K Murti and K Murti. "Real-Time Operating Systems (RTOS)". Design Principles for Embedded Systems (2022): 189-224.
19. A Züpke. "Efficient and predictable thread synchronization mechanisms for mixed-criticality systems on shared-memory multi-processor platforms" (2021).
20. Y Peng, A Jolfaei and K Yu. "A novel real-time deterministic scheduling mechanism in industrial cyber-physical systems for energy internet". IEEE Transactions on Industrial Informatics 18.8 (2021): 5670-5680.
21. C Wang., et al. "Improving real time performance of Linux System using RT-Linux". in Journal of Physics: Conference Series, IOP Publishing (2019): 052017.
22. H Lee., et al. "Idempotence-based preemptive GPU kernel scheduling for embedded systems". IEEE Transactions on Computers 70.3 (2020): 332-346.
23. N Faller. "Measuring the latency time of real-time Unix-like operating systems". International Computer Science Institute, Berkeley, California, Technical Report TR-92-037 (1992).
24. M Holenderski. "Grasp: Tracing, visualizing and measuring the behavior of real-time systems". in International workshop on analysis tools and methodologies for embedded and real-time systems, WATERS (2010).
25. H Zhou, G Tong and C Liu. "GPES: A preemptive execution system for GPGPU computing". in 21st IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE (2015).
26. M Marchesotti, M Migliardi and R Podestà. "A measurement-based analysis of the responsiveness of the Linux kernel". in 13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems, ECBS'06, IEEE (2006).
27. J Stärner and L Asplund. "Measuring the cache interference cost in preemptive real-time systems". in Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (2004).

28. B Blackham. "Timing analysis of a protected operating system kernel". 2011 IEEE 32nd Real-Time Systems Symposium, IEEE (2011).
29. GK Adam. "Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers". Computers 10.5 (2021): 64.
30. DR Mall. "Real Time Operating Systems". in Lecture delivered at NCIT, Kathmandu, Nepal (2003).
31. A Morton. "Linux: Kernel Preemption, to enable or not to enable" (2004). [Online].
32. C Niyizamwiyitira and L Lundberg. "A utilization-based schedulability test of real-time systems running on a multiprocessor virtual machine". The Computer Journal 62.6 (2019): 884-904.
33. J Sun. "Revamp Linux Time Subsystem" (2004). [Online].
34. J Sun. "Preemption Latency Test Utilities" (2004). [Online].

Volume 5 Issue 6 December 2023