Evaluation of Interrupt Handling Timeliness in Real-Time Linux Operating Systems

Paul Regnier
Distributed Systems Laboratory (LaSiD)
Computer Science Department (DCC)
Federal University of Bahia
pregnier@ufba.br

George Lima
Distributed Systems Laboratory (LaSiD)
Computer Science Department (DCC)
Federal University of Bahia
gmlima@ufba.br

Luciano Barreto
Distributed Systems Laboratory (LaSiD)
Computer Science Department (DCC)
Federal University of Bahia
lportoba@ufba.br

ABSTRACT

Several real-time Linux extensions are available nowadays. Two of those extensions that have received special attention recently are Preempt-RT and Xenomai. This paper evaluates to what extent they provide deterministic guarantees when reacting to external events, an essential characteristic when it comes to real-time systems. To do so, we define a simple but effective experimental approach. Obtained results indicate that Preempt-RT is more prone to temporal variations than Xenomai when the system is subject to overload scenarios.

1. INTRODUCTION

Real-time systems encompasses a broad range of applications in telecommunications, multimedia, industry, transportation, health etc. In all these scenarios, correctly choosing a Real-Time Operating System (RTOS) is a fundamental design issue. Although technological hardware advances are essential to the development of the IT industry, some of these innovations may introduce undesirable unpredictability to the implementation of a RTOS. For example, cache memories, direct memory access, out-of-order execution and branch prediction units may introduce non-negligible sources of indeterminism [9, 15]. Thus, the construction of a general purpose operating system with a focus on timing predictability remains a challenging research issue.

Although Linux is a popular and widely accepted OS, the standard Linux kernel [4] fails to provide the timing guarantees required by critical real-time systems [10, 1]. To circumvent this problem, several approaches have been developed in order to increase the timing predictability of Linux [7, 13, 5, 22, 6, 3]. The diversity and constant evolution in their design call for comparative studies to assess the determinism degree offered by such platforms. The results of this kind of studies can help real-time systems designers choosing the appropriate solution according to their needs.

This paper presents and compares two RT Linux kernel patches, Preempt-RT (Linux$^{\mathbf{Prt}}$ [7] and Xenomai (Linux$^{\mathbf{Xen}}$) [13], developed to increase the predictability of Linux. The main contributions of this work are: (i) an evaluation procedure based on simple software and hardware COTS, and (ii) a report analysis of Preempt-RT and Xenomai latency performance obtained by our experimental results. Overall, these results show that Linux$^{\mathbf{Xen}}$ provides better timing guarantees than Linux$^{\mathbf{Prt}}$.

The remainder of this paper is structured as follows. Section 2 discusses some factors of unpredictability in Linux and defines the metrics used in our evaluation. Linux$^{\mathbf{Prt}}$ Linux$^{\mathbf{Xen}}$ are described in Sections 3 and 4. We then present our evaluation methodology in Section 5 and experimental results in Section 6. Finally, Section 7 briefly discusses related work and Section 8 concludes the paper.

2. EVALUATION METRICS

The conventional method used to minimize the impact of interruptions on the response time of processes is to divide the implementation of interrupt handlers into two parts. The first part, referred to as the **critical section** of the handler, runs critical operations immediately after its activation, usually with interruptions disabled. One may enable interruptions during some parts of a critical section in order to allow for preemptions. However, such an implementation must rely on locks to ensure controlled access to shared data. The second part of the handler is dedicated to non-critical operations. Its execution can be delayed and normally happens with interruptions enabled. In Linux, this second part of the handlers are called ***softirqs***.

Clearly, the way interrupt handlers are dealt with by an OS kernel interferes in the system timeliness as a whole. In this paper, we consider two metrics to analyze the timing behavior of an OS: interrupt latency and activation latency. These two metrics are explained in the next sections.

2.1 Interrupt latency

An interrupt request, or simply **interrupt**, of the processor is typically asynchronous and can happen at any time during the processor execution cycle. In particular, such requests can occur while the critical section of another interrupt handler is running, with interruptions disabled. This scenario may delay the detection of interrupt requests by the processor in a non-deterministic manner.

The time interval between the instant at which an interrupt request takes place and the starting time of the execution of the associated handler is called **interrupt latency**.

2.2 Activation latency

In a Linux kernel, a *softirq* is able to start just after the associated interrupt handler critical section finishes its execution. However, other interruptions may occur in between, which may postpone the execution of *softirqs*. These possible extra delays have direct impact on real-time systems, where timeout or hardware events are used to trigger tasks, in a similar manner as *softirqs*. For example, a real-time task τ may be suspended while waiting for an event. When such event occurs, the associated interrupt request triggers the corresponding handler which, in turn, wakes up τ. The time interval between the event occurrence instant and the beginning of the execution of τ is called **activation latency**.

As for *softirqs*, the activation latency may be increased by the occurrence of interruptions. Furthermore, the execution of other *softirqs* may be scheduled according to some policy (eg FIFO, fixed priority), which can also generate interference in the activation latency.

3. LINUX PREEMPT-RT

Linux$^{\mathbf{Prt}}$ [11, 18] is a Linux real-time patch originally developed by Ingo Molnar. This patch makes the Linux kernel almost fully preemptible by reengineering the use of locks inside the kernel. As soon as a high priority process is released, it can acquire the processor with low latency, with no need to wait for the end of the execution of a lower priority process, even if such a process is running in kernel mode. Also, in order to limit the unpredictability caused by shared resources, Linux$^{\mathbf{Prt}}$ provides synchronization primitives that are able to use a priority inheritance protocol [19]. Further, a specific implementation of high resolution timers [8] allows the kernel to provide time values in microseconds. For instance, Rosted et al [18, 20] obtained activation latencies of the order of tens of μs.

Linux$^{\mathbf{Prt}}$ creates specific kernel threads to handle both software and hardware interrupts. Upon an interrupt request, the associated handler masks the request, wakes up the associated thread and returns to the interrupted code. This approach greatly reduces the execution latency of the critical part of interrupt handlers. The interrupt thread that has been woken up is eventually scheduled according its priority and then starts executing. Another advantage of Linux$^{\mathbf{Prt}}$ is that several Linux legacy software packages such as C libraries and programming environments can be used.

It is interesting to note that the threaded implementation of interrupt handlers in Linux$^{\mathbf{Prt}}$ may be a source of unpredictability when interrupt threads are delayed by the scheduling policy or by other interrupt requests. Nevertheless, Linux$^{\mathbf{Prt}}$ offers the option `IRQF_NODELAY` which allows for disabling the threaded implementation of a specific interrupt line. When this option is set, interrupts are handled as in standard Linux.

4. LINUX XENOMAI

Xenomai or Linux$^{\mathbf{Xen}}$ is a real-time Linux framework that encompasses an OS kernel, APIs and a set of utilities. It uses an interrupt indirection layer [21], also called nanokernel, to isolate real-time tasks from Linux processes. According to this approach, when an interrupt occurs, the nanokernel forwards the request either to a real-time task or to a conventional Linux process. In the first case, the interrupt handler runs immediately. In the second case, the request is enqueued and further delivered to Linux when there are no more pending real-time tasks. Whenever the Linux kernel requests disabling the interrupts, the nanokernel just makes the Linux kernel believe that interrupts are disabled. The nanokernel keeps intercepting any hardware interrupts. The interrupt requests targeted to Linux are kept enqueued until the Linux kernel requests enabling interrupts.

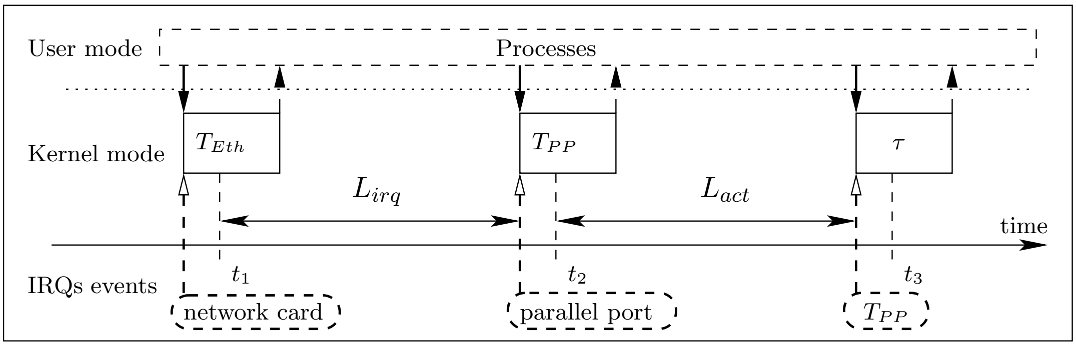
The nanokernel of Linux$^{\mathbf{Xen}}$ is based on a resource virtualization layer called Adeos (Adaptative Domain Environment for Operating Systems) [23]. Adeos eases hardware sharing and provides a small API which is architecture independent. In short, Adeos relies on two basic concepts: domains and hierarchical interrupt pipelines. A domain defines an isolated execution environment, according to which one can run programs or even a complete operating system. The hierarquical interrupt pipeline, called **ipipe**, performs interrupt delivery across different domains. When a domain is registered, it is stored on a specific position in the ipipe according to its timing requirements. The interrupt indirection mechanism handles hierarchical interrupt delivery following the priority associated to each domain.

Real-time services in Linux$^{\mathbf{Xen}}$ correspond to the highest priority domain in the ipipe, which is called the primary domain. The secondary domain refers to the Linux kernel itself, from which common Linux software libraries are available. At this level, however, Linux$^{\mathbf{Xen}}$ offers weaker timing guarantees due to the way user process are mapped into kernel threads.

5. EXPERIMENTAL METHODOLOGY

In general, performing accurate time measurements at the interrupt handler level is not simple and may require the use of external devices such as oscilloscopes or other computers. In fact, the exact instant at which an interrupt request occurs is difficult to be determined since this is an asynchronous event which can be triggered by any hardware device. Nevertheless, since the objective of this work is to characterize and compare the degree of predictability in different operating systems platforms, we adopt a simple experiment setup that is easily reproducible. In other words, we are interested in measuring approximate values of latencies for different real-time OS under similar load scenarios.

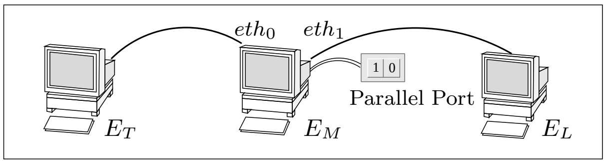
To compare the OS platforms two experiments were set up, both of which use only computer stations connected to each other by standard communication devices. These two experiments, described in Sections 5.1 and 5.2, are to measure interrupt and activation latencies with and without load scenarios. These latencies are denoted L_{irq} and L_{act}, respectively. The procedure to generate load scenarios is given in Section 5.3.



1: Interrupt and activation latencies at station E_M for the first experiment

5.1 First experiment

Figure 2 illustrates how the first experiment was set up. We use three stations, E_T, E_L and E_M and two distinct Ethernet network devices, eth_0 and eth_1, to connect E_T to E_M and E_L to E_M, respectively. The role of E_T is to *trigger* events at the parallel port of E_M. Such events should be timely handled by E_M, the station whose latencies are *measured*. E_L is the *load* station, used to create load scenarios on E_M via eth_1.



2: First experiment setup

The activities handled by E_M are illustrated in Figure 1. The following sequence of events occurs:

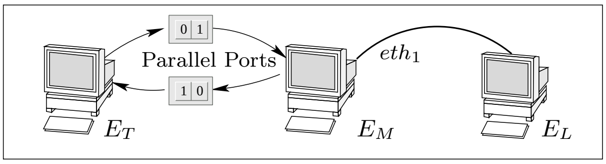
1. E_T sends Ethernet frames to E_M, which are received through its eth_0 device. Upon the receiving of each frame the device eth_0 issues an interrupt request at E_M. In turn, the associated interrupt handler (T_{eth_0}) preempts the application that is executing on E_M.
2. T_{eth_0} sets the Parallel Port Interrupt Request (PP-IRQ) line and saves the instant t_1 in memory. Note that t_1 is the local time on E_M and is read just after the arrival of an Ethernet frame at eth_0.
3. Upon the detection of the PP-IRQ, its handler T_{PP} preempts the application on E_M. Then T_{PP} saves the instant t_2 and wakes up task τ. This second time instant is the local time value at E_M just after the start of T_{PP}.
4. When task τ wakes up, it saves instant t_3 in memory and then is suspended until the next PP interrupt. Thus, t_3 is the time instant at which τ starts executing.

During the experiment runs, the measured values of L_{irq} and L_{act} were transfered from main memory to a file system in E_M by a user process using a FIFO channel. The assigned priority of the user process was lower than the priority of interrupt handlers. Also, this data transferring procedure generated sufficiently rare events (20 per second). This data transferring scheme was to prevent possible interference in the measured values.

In order to compare the behavior of the analyzed platforms, both latencies can be computed by the described procedure as $L_{irq} = t_2 - t_1$ and $L_{act} = t_3 - t_2$, as depicted in Figure 1. However, it is worth noticing that the measurements are realized by the same station that is responsible for managing real-time activities. Indeed, station E_M waits for the asynchronous arrival of an Ethernet frame at eth_0 to trigger the corresponding parallel port interrupt so that measurements can be carried out. This dependence between external and internal events may compromise some measurements. In order to evaluate to what extent such a procedure interfere in the measurements, a second experiment was set up.

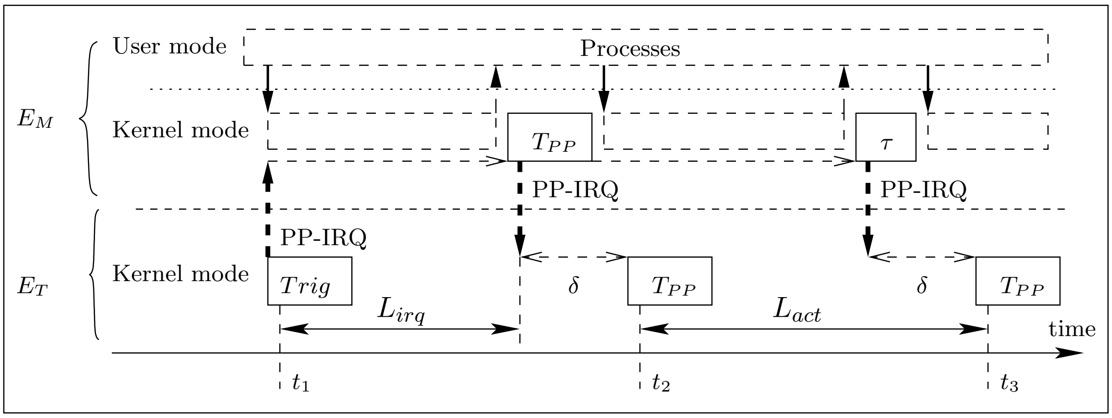
5.2 Second experiment

The same three stations E_M, E_T and E_L are used in this experiment setup. Station E_L is configured as before while the other two stations have different roles, as shown in Figure 4.



4: Second experimental set-up

Similar to the first experiment, the values of L_{irq} and L_{act} correspond to real-time activities executed by E_M. However, the measurements are carried out by E_T instead. The measurement procedure makes use of the parallel port that connects E_T and E_M, as can be seen from the figure. The device eth_0 is no longer necessary. In other words, station E_T triggers PP interrupts at E_M via its parallel port. Station E_M handles such PP interrupt requests, waking up a real-time task τ similarly to the previous experiment. Note that the measurements could not be carried out by E_M in this second experiment unless station local clocks were synchronized to each other. Hence, E_M triggers back interrupts on the E_T IRQ line and the measurements are taken at E_T while these interrupts are handled.



3: Interrupt and activation latencies measurement at station E_M for the second experiment

Figure 3 summarizes the sequence of events that occur in E_M and E_T, which makes up the second measurement procedure:

1. Station E_T triggers an interrupt request on the PP-IRQ line of station E_M and saves instant t_1 in memory. This time instant is the local clock of E_T just after the interrupt is requested.
2. An interrupt is issued at the parallel port of E_M. The handler T_{PP} of E_M is activated, causing the preemption of the application running on E_M.
3. The handler T_{PP} of E_M triggers an interrupt request on the PP-IRQ line of station E_T and wakes up task τ.
4. The PP interrupt handler T_{PP} of E_T saves time t_2 in memory. This time instant corresponds to the value of the local clock of E_T just after the start of its T_{PP}.
5. Task τ wakes up in E_M and triggers a new interrupt request on the PP-IRQ line of station E_T.
6. The handler T_{PP} of E_T is activated to deal with this second interrupt request, saving the current value of its local clock t_3. This instant corresponds to the time at which E_T is informed about the activation of τ.

As can be seen from Figure 3, the described measurement procedure must take into account the interrupt latency δ in E_T. Indeed, differently from the first experiment, now $L_{irq} = t_2 - t_1 - \delta$. On the other hand, some care must be taken in order to measure L_{act} accurately as the interruption issued by τ may take place before or after t_2, introducing an experimental variability not related to the real value of $t_3 - t_2$. This issue will be further discussed when analyzing the obtained experimental results in Section 6.

The value of δ can be estimated by carrying out the first experiment, but without using station E_L. For example, using Linux$^{\mathbf{Xen}}$ running in single mode with minimal load, the estimated value of δ can be taken as the mean value observed in the measurements, $\bar{\delta}$. As will be seen later on, for such a platform $\bar{\delta} = 9\mu s$ with standard deviation $0.1\mu s$. Once this estimation is derived, the operating system patch in E_T must be fixed so that $\bar{\delta}$ can be used as an accurate estimation of δ during the experiments.

During the measurements, the obtained values were transfered from memory to a file system in E_T using the same data transferring scheme used in the first experiment. In order to minimize any possible interference, E_T was run in single user mode with minimal load.

5.3 Load scenarios

The experiments were carried out with and without load scenarios in station E_M. Without any load, E_M was set up with its kernel in single mode and with minimum activities, i.e. both E_L and no other process in E_M generate extra load. As will be seen, in general, the analyzed real-time patches present high levels of predictability under this situation.

The load generated was applied to station E_M, which was stressed by two different types of load, triggered by internal and external events. Both types of load were started a few seconds before the beginning of the measurements. As will be seen, under such load scenarios, it was possible to assess to what extent the analyzed real-time patches can provide predictability.

The internal events that generated CPU and I/O load on E_M were performed by executing the following shell commands:

```
while "true"; do
   dd if=/dev/hda2 of=/dev/null bs=1M count=1000
   find / -name "*.c" | xargs egrep include
   tar -cjf /tmp/root.tbz2 /usr/src/linux-xenomai
   cd /usr/src/linux-preempt; make clean; make
done
```

The external events used to load E_M were due to the arrival of 64 byte UDP packets at eth_1 sent by station E_L. Station E_M was configured as a server and E_L as a client. The packet sending rate was set to $200kHz$, which is the maximum network rate allowed. With this setup, we are able to issue more than $100,000$ interrupt requests per second at eth_1. This device used E_M IRQ line 18, whose priority is lower than the priority of the PP-IRQ line. Thus, in an ideal situation, one would expect that receiving packets from

eth_1 would not interfere in the execution of PP-IRQ related events.

6. EVALUATION RESULTS

The experiments were conducted on three Pentium 4 computers with $2.6GHz$ processors and $512MB$ RAM memory. Three operating system platforms were analyzed, one of which with two configuration options:

- **Linux$^{\mathbf{Std}}$:** Linux standard - kernel version 2.6.23.9 (*low-latency* option);
- **Linux$^{\mathbf{Prt}}$:** Linux with *patch* Preempt-RT (rt12) - kernel version 2.6.23.9;
- **Linux$^{\mathbf{PrtND}}$:** Linux$^{\mathbf{Prt}}$ with option `IRQF_NODELAY` used to initialize the PP-IRQ line;
- **Linux$^{\mathbf{Xen}}$:** Linux with *patch* Xenomai - version 2.4-rc5 - kernel version 2.6.19.7.

Linux$^{\mathbf{Std}}$ was considered for the sake of illustration. Although this general purpose OS is not suitable to deal with real-time applications, it has been used here as a reference, against which one can compare real-time Linux patches. Linux$^{\mathbf{PrtND}}$ corresponds to setting the option `IRQF_NODELAY` at the initialization time of PP-IRQ line. Recall that with this option, interrupt handling of that line is implemented without threads. Latencies L_{irq} and L_{act} were measured using the Time Stamp Counter (TSC), which provided a precision of less than $30ns$ (88 cycles) in our tests. As mentioned earlier, station E_T was used to trigger $20Hz$ events at station E_M. The measured data were the result of running the experiments for ten minutes for each experiment type and platform.

Experimental results are presented through graphs in which the horizontal axes represent the instant at which the latencies were measured, which ranges from 0 to 60 seconds. The vertical axes represent the measured latencies in μs. These values can be multiplied by $2.6 10^3$ to obtain the corresponding number of TSC cycles. Values outside the vertical axis range are represented by a triangle near the maximum value. Below each graph the following values are given: Mean (M), Standard Deviation (SD), minimum (Mn) and maximum (Mx). These numbers were obtained considering the duration of ten minutes of each experiment run. Although each experiment was run for ten minutes, one-minute time window was found sufficient to illustrate the timing behavior of each platform. During this time interval, the total number of events is 1 200 as the arrival frequency of Ethernet frames at the eth_0 network device of station E_M is $20Hz$.

We first present in Section 6.1 the results from the first experiment. Then, in Section 6.2, we analyze the procedure suggested by the second experiment. Section 6.3 discusses the results obtained by these two experiments.

6.1 Results from the First Experiment

For the sake of illustration, we first discuss the results regarding Linux$^{\mathbf{Std}}$. Then, we present the measurements obtained for the other platforms.

6.1.1 Linux$^{\mathbf{Std}}$

As can be seen from Figure 5, the obtained values without load show that interrupt handling in Linux is reasonably efficient. As it will be seen shortly, these values are very close to some RTOS platforms. However, both L_{irq} and L_{act} vary significantly in the presence of load, as expected. In particular, the obtained values of L_{act} in load scenarios confirm that Linux$^{\mathbf{Std}}$ is not suitable to support real-time systems. Indeed, the maximum value of L_{act} was found to be about 17 times the mean value.

6.1.2 Linux$^{\mathbf{Prt}}$, Linux$^{\mathbf{PrtND}}$ and Linux$^{\mathbf{Xen}}$

Figures 6 and 7 plot the values of L_{irq} and L_{act}, respectively. Six graphs are shown in each Figure. The right and left columns show results with and without load, respectively.

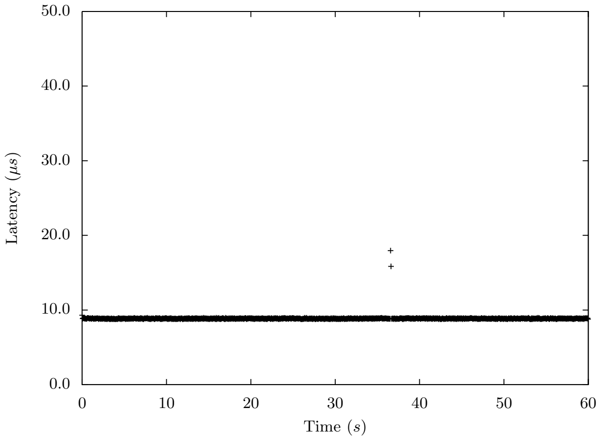
As for the interrupt latencies (see Figure 6), Linux$^{\mathbf{Xen}}$ clearly shows higher predictability when compared to the other platforms. Under load scenarios, this behavior is evident as it can be noticed by the lower mean and standard deviation values. In order to explain the behavior of Linux$^{\mathbf{Prt}}$, some aspects need to be explained. First, when `IRQF_NODELAY` is set, the behavior of Linux$^{\mathbf{Prt}}$ turns to be similar to Linux$^{\mathbf{Std}}$, although Linux$^{\mathbf{Prt}}$ exhibits better results. On the other hand , using threads for interrupt handling increases the interrupt latency due to an extra context-switching overhead. Also, a significantly higher variability on latency values happens when the system is overloaded. This can be explained by the execution delay of the handler. Indeed, between the instant at which T_{PP} issues the interrupt and the instant at which the IRQ thread actually wakes up, several interrupts may occur. In such a scenario, the execution of associated interrupt handlers may delay the execution of T_{PP}.

Figure 7 shows activation latencies with and without load. It is worth noting the behavior of Linux$^{\mathbf{Prt}}$ and Linux$^{\mathbf{Xen}}$ with load. Despite the mean value found for Linux$^{\mathbf{Xen}}$ $(8,7\mu s)$ is greater than the one found for Linux$^{\mathbf{Prt}}$ $(3,8\mu s)$, the standard deviation is significantly lower in favor of Linux$^{\mathbf{Xen}}$. In fact, this is a desirable feature in hard real-time systems. Additionally, for such systems, it is desirable that the worst-case execution time be as close as possible to the average-case execution time.

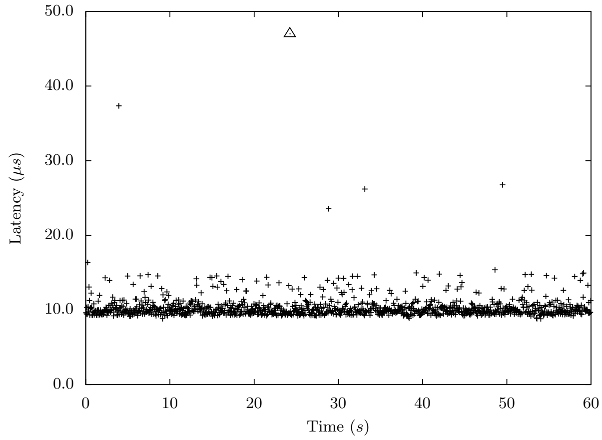
By analyzing the values of activation latencies of Linux$^{\mathbf{PrtND}}$, it can be noticed that those values are acceptable when compared to Linux$^{\mathbf{Prt}}$ in the absence of load. Nonetheless, the results obtained in load scenarios still indicate a slight less predictable behavior than Linux$^{\mathbf{Xen}}$.

6.2 Results from the Second Experiment

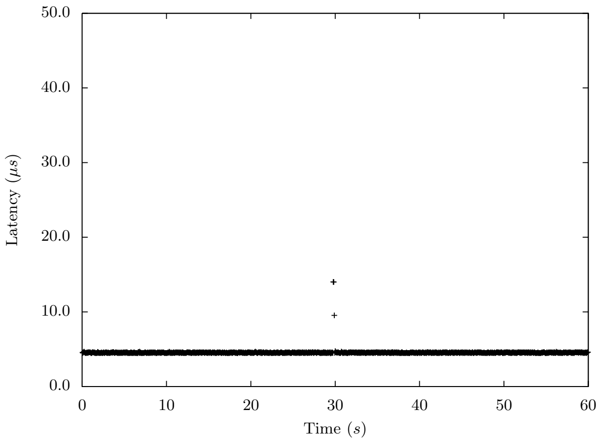
As will be seen, the timing patterns obtained by the second experiment were similar to those described in the previous section. In order to avoid repeating the illustration of such patterns, we summarized these results in Table 1, which is presented in Section 6.3. Before presenting these results, though, we first illustrate the differences between both types of experiments. To do so, we discuss the results regarding Linux$^{\mathbf{Xen}}$ only. This platform serves well for our illustration purposes because: (i) the results of the first experiment have indicated that Linux$^{\mathbf{Xen}}$ is more predictable than the other platforms; (ii) station E_T was configured to use Linux$^{\mathbf{Xen}}$.



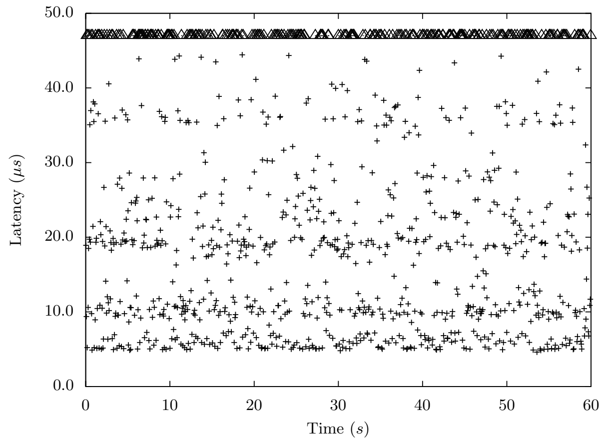
(a) **Linux$^{\mathbf{Std}}$ - Interrupt latency - without load**
M: 8.9, SD: 0.3, Mn: 8.7, Mx: 18.4



(b) **Linux$^{\mathbf{Std}}$ - Interrupt latency - with load**
M: 10.4, SD: 1.9, Mn: 8.8, Mx: 67.7



(c) **Linux$^{\mathbf{Std}}$ - Activation latency - without load**
M: 4.6, SD: 0.4, Mn: 4.4, Mx: 16.2



(d) **Linux$^{\mathbf{Std}}$ - Activation latency - with load**
M: 37.3, SD: 48.2, Mn: 4.6, Mx: 617.5

5: Linux$^{\mathbf{Std}}$ latencies. The eth_0 interrupt handler is triggered by packets arriving at a $20Hz$ frequency.

The results for Linux$^{\mathbf{Xen}}$ are plotted in Figure 8 and will be discussed in Sections 6.2.1 and 6.2.2.

6.2.1 Interrupt latencies in Linux$^{\mathbf{Xen}}$

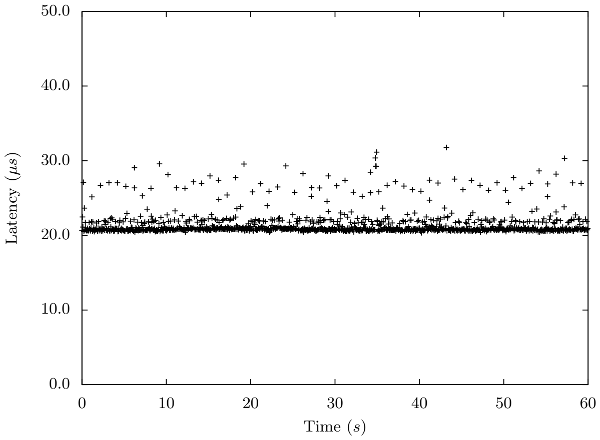
By the experiment setup, the measurements are carried out by station E_T (recall Figure 1). Hence, there is an extra delay δ that must be considered in the measurements. This delay corresponds to the value of L_{irq} in E_T. In other words, since Linux$^{\mathbf{Xen}}$ is being used in both stations, one expects measuring $t_2 - t_1 = 2\delta$ in scenarios without load. From this measurement, δ can be easily derived. Hence, Figure 8a shows the values of L_{irq} minus the mean value of δ, assumed to be $9.0\mu s$.

Once δ is subtracted from the others results plotted in Figure 8b, it can be seen that the system behaves very similarly to the first experiment. A noticeable difference is a significant increase of the standard deviation. This can be explained as follows. In the first experiment, T_{eth_0} issues an interrupt request and then finishes its execution. The pending interrupt is immediately detected and the execution of the associated handler begins with a minimum delay since the indirection scheme of the Adeos nanokernel guarantees that no other interruption can delay the start of T_{PP}. Such scenarios do not occur in the second experiment since the parallel port interrupts are externally triggered by E_T. Therefore, possible interference in the interrupt handler execution can be caused by context-switching overhead. As a result, the second experiment captures actual scenarios more accurately as can be seen from the higher variability of the obtained results.

6.2.2 Activation latencies in Linux$^{\mathbf{Xen}}$

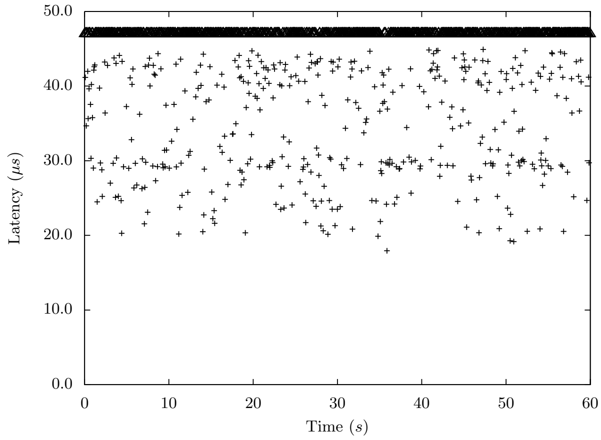
There are two aspects that must be considered when measuring activation latencies by the second experiment (recall Figure 3). Both aspects are dealt with by our experiment set-up.

First, as mentioned earlier, the interrupt request issued by τ may take place before or after t_2, turning the value of $t_3 - t_2$ into an imprecise measurement. For example, if τ issues an interrupt request at E_T before t_2, this request will be triggered before the handling of the pending interrupt requested by T_{PP} at E_T. Thus, these two requests will be handled in



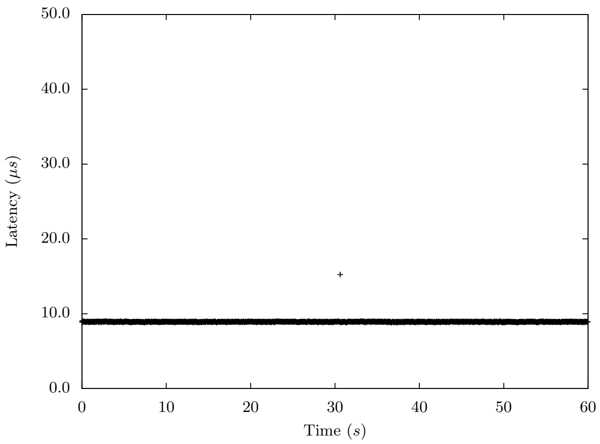
(a) **Linux$^{\mathbf{Prt}}$ - without load**

M: 21.5, SD: 1.7, Mn: 20.3, Mx: 45.1



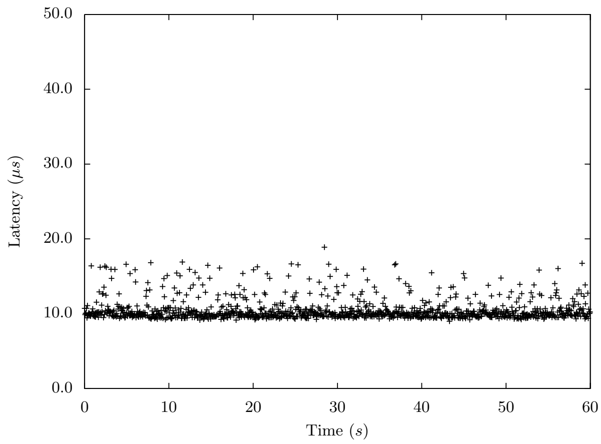
(b) **Linux$^{\mathbf{Prt}}$ - with load**

M: 58.5, SD: 26.4, Mn: 17.2, Mx: 245.9



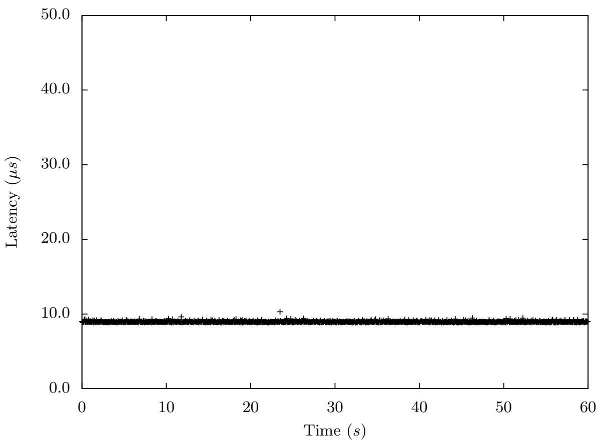
(c) **Linux$^{\mathbf{Prt}}$ - without load, option** `IRQF_NODELAY`

M: 8.9, SD: 0.2, Mn: 8.8, Mx: 16.7



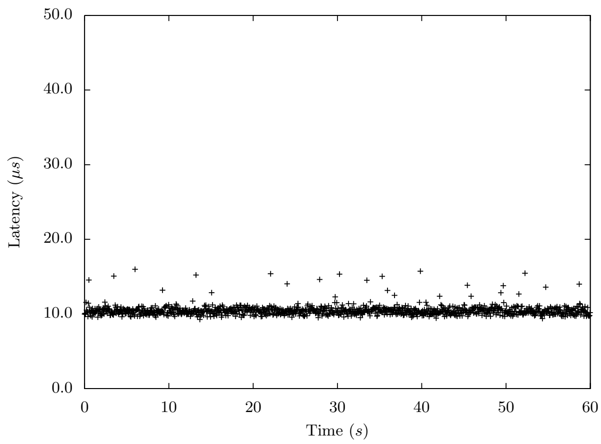
(d) **Linux$^{\mathbf{Prt}}$ - with load, option** `IRQF_NODELAY`

M: 10.6, SD: 1.6, Mn: 8.9, Mx: 35.8



(e) **Linux$^{\mathbf{Xen}}$ - without load**

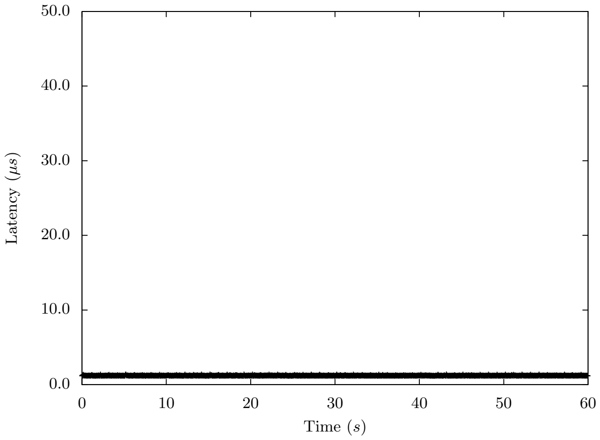
M: 9.0, SD: 0.1, Mn: 8.8, Mx: 11.1



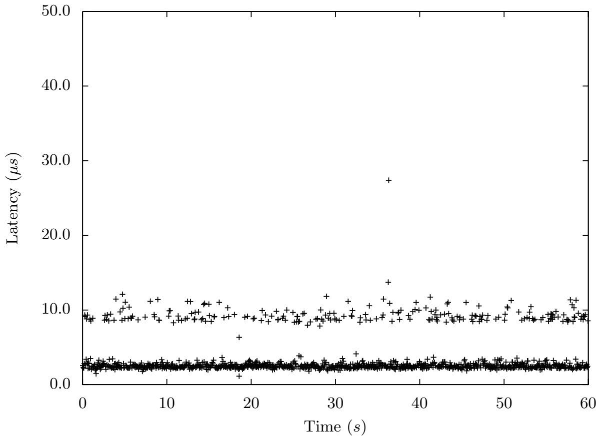
(f) **Linux$^{\mathbf{Xen}}$ - with load**

M: 10.2, SD: 0.1, Mn: 8.8, Mx: 20.8

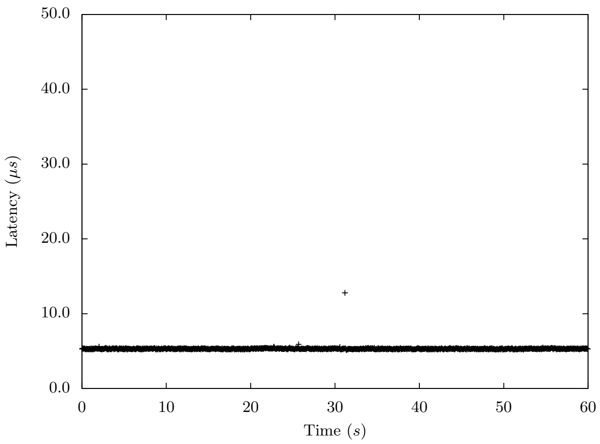
6: Interrupt latencies. The eth_0 interrupt handler is triggered by packets arriving at a $20Hz$ frequency.



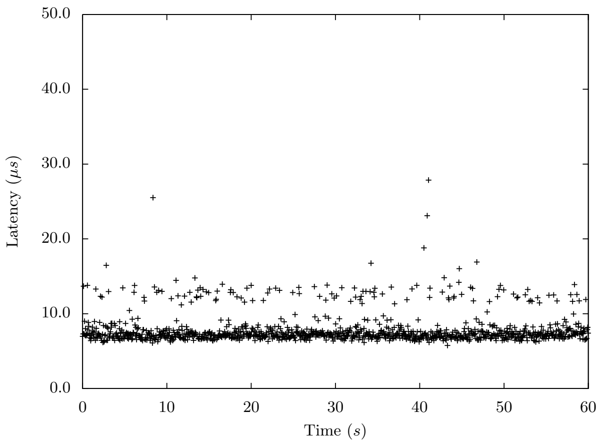
(a) **Linux$^{\mathbf{Prt}}$ - without load**
M: 2.1, SD: 0.2, Mn: 1.2, Mx: 9.4



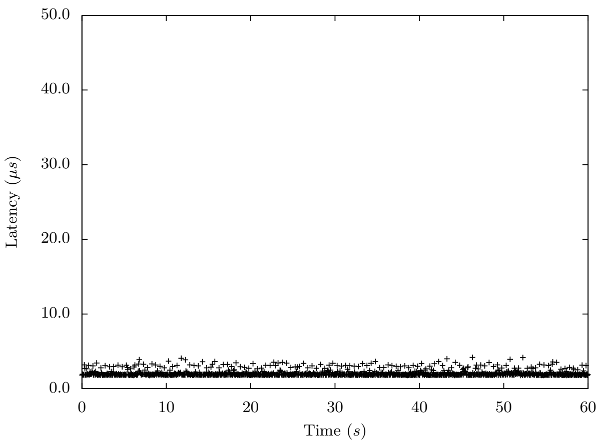
(b) **Linux$^{\mathbf{Prt}}$ - with load**
M: 3.8, SD: 2.8, Mn: 1.1, Mx: 27.4



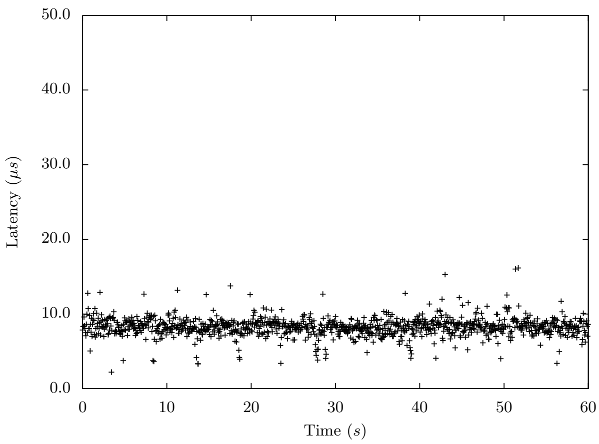
(c) **Linux$^{\mathbf{Prt}}$ - without load, option** `IRQF_NODELAY`
M: 5.3, SD: 0.3, Mn: 5.0, Mx: 13.1



(d) **Linux$^{\mathbf{Prt}}$ - with load, option** `IRQF_NODELAY`
M: 8.0, SD: 2.0, Mn: 5.2, Mx: 31.0



(e) **Linux$^{\mathbf{Xen}}$ - without load**
M: 2.1, SD: 0.5, Mn: 1.8, Mx: 8.4



(f) **Linux$^{\mathbf{Xen}}$ - with load**
M: 8.7, SD: 0.3, Mn: 1.8, Mx: 18.7

7: Activation latencies. The eth_0 interrupt handler is triggered by packets arriving at a $20Hz$ frequency.

1: Latency results in μs for Linux$^{\mathbf{Std}}$, Linux$^{\mathbf{Prt}}$, Linux$^{\mathbf{PrtND}}$ and Linux$^{\mathbf{Xen}}$ using Experiments 1 and 2.

		Linux$^{\mathbf{Std}}$				Linux$^{\mathbf{Prt}}$				Linux$^{\mathbf{PrtND}}$				Linux$^{\mathbf{Xen}}$			
	Load	no		yes		no		yes		no		yes		no		yes	
		$\mathbf{L_{irq}}$	$\mathbf{L_{act}}$	$\mathbf{L_{irq}}$	$\mathbf{L_{act}}$	$\mathbf{L_{irq}}$	$\mathbf{L_{act}}$	$\mathbf{L_{irq}}$	$\mathbf{L_{act}}$	$\mathbf{L_{irq}}$	$\mathbf{L_{act}}$	$\mathbf{L_{irq}}$	$\mathbf{L_{act}}$	$\mathbf{L_{irq}}$	$\mathbf{L_{act}}$	$\mathbf{L_{irq}}$	$\mathbf{L_{act}}$
Exp. 1	Mean	8.9	4.6	10.4	37.3	21.5	2.1	58.5	3.8	8.9	5.3	10.6	8.0	9.0	2.1	10.2	8.7
	SD	0.3	0.4	1.9	48.2	1.7	0.2	26.4	2.8	0.2	0.3	1.6	2.0	0.1	0.5	0.1	0.3
	Min	8.7	4.4	8.8	4.6	20.3	1.2	17.2	1.1	8.8	5.0	8.9	5.2	8.8	1.8	8.8	1.8
	Max	18.4	16.2	67.7	617.5	45.1	9.4	245.9	27.4	16.7	13.1	35.8	31.0	11.1	8.4	20.8	18.7
Exp. 2	Mean	9.0	3.6	12.5	19.9	10.2	3.7	31.2	7.2	9.2	4.6	11.8	14.9	9.1	4.0	11.3	9.8
	SD	0.4	0.6	3.2	17.4	0.5	0.4	19.0	3.1	0.4	0.5	2.3	5.6	0.3	0.3	1.2	2.0
	Min	8.8	-1.3	9.0	2.3	10.0	0.8	10.4	2.2	8.9	-0.3	9.1	4.5	8.8	0.3	9.0	2.7
	Max	18.4	19.0	75.0	428.4	30.8	12.7	203.9	21.2	14.9	14.2	49.2	85.0	13.4	9.6	19.7	11.8

a row, which makes the value of $t_3 - t_2$ too short. On the other hand, if the interrupt request by τ takes place after t_2, as represented in Figure 3, this undesirable interference disappears and $t_3 - t_2$ turns to be an accurate measurement of L_{irq}. In order to circumvent this measurement problem, an extra and constant delay of $\Delta = 10\mu s$ was introduced so that the interrupt request issued by τ always takes place after t_2.

The second aspect is due to the interrupt latency variability at E_T. As this station runs Linux$^{\mathbf{Xen}}$, it was seen in the first experiment that $L_{irq} \in [8.8, 11.1]$ when no load scenarios are considered. This means that when measuring L_{act}, one can obtain values $(t_3 - t_2) \pm 2.3\mu s$ in worst case.

The graphs in Figure 8c show the activation latencies obtained by the described approach. The values are already subtracted by $10\mu s$ and so they correspond to the measured values of L_{act}. The obtained values in the graphs are very close to the ones obtained by the first experiment as can be seen by the small differences between the mean values. Also, as expected, the variability is now higher due to the way the experiment was set up.

6.3 Comparative Analysis

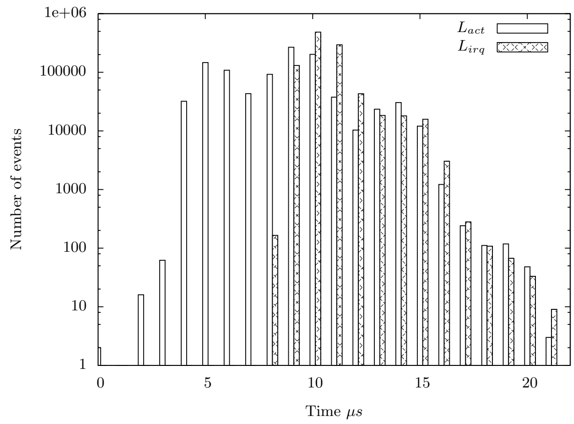
Table 1 summarizes the results regarding all analyzed platforms. Both types of experiments are reported. As can be seen, their results can be used for comparing the platform behaviors using either experiment, as mentioned before.

As expected, the data obtained for Linux$^{\mathbf{Std}}$ indicate that it is not suitable to deal with real-time systems. Load scenarios make the interrupt and activation latencies much larger than the observed mean values.

As observed before, the way Linux$^{\mathbf{Prt}}$ deals with interrupt request may cause excessive delays in interrupt latencies in load scenarios. This behavior is verified in both experiments. When option `IRQF_NODELAY` is used, the obtained values show a behavior similar to Linux$^{\mathbf{Std}}$ in both experiments, although Linux$^{\mathbf{PrtND}}$ seems much efficient.

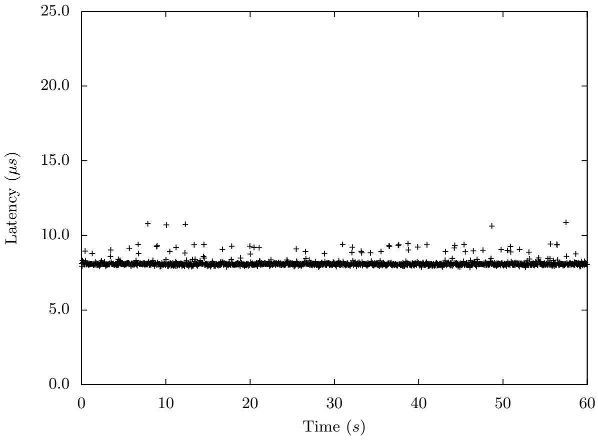
It is interesting to notice that there have been negative values of activation latencies as for the second experiment. This can be explained by the variability of δ at station E_T (recall section 6.2.1). For example, consider that $\delta \in [\delta_{min}, \delta_{max}]$. Also, recall that there is a constant delay of Δ introduced in the measurement. Hence, $t_3 - t_2 - \Delta \in [\delta_{min} - \delta_{max}, \delta_{max} - \delta_{min}]$. Since in our experiments it was observed that $\delta_{min} = 8.8\mu s$ and $\delta_{max} = 11.1\mu s$, a negative value may be found whenever the actual $L_{act} \leq 2.3\mu s$. However, it is important to emphasize that we rarely observed negative values during the experiments, only once in 12 000 measurements.

Among the analyzed platforms Linux$^{\mathbf{Xen}}$ shows higher predictability levels when compared to the other platforms. This characteristic is of paramount importance when it comes to supporting real-time systems. It is worth emphasizing that for such systems predictability is preferable than speed. Thus, although the mean values obtained by Linux$^{\mathbf{Prt}}$ are smaller, Linux$^{\mathbf{Xen}}$ seems a better alternative when predictability is aimed for.

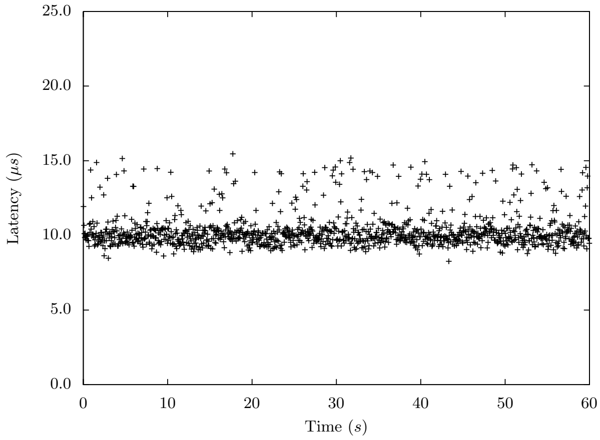


9: 1,000,000 events histogram for Linux$^{\mathbf{Xen}}$. Each histogram, log-scaled, corresponds to the number of events in the $1\mu s$ interval beginning at the corresponding x-value.

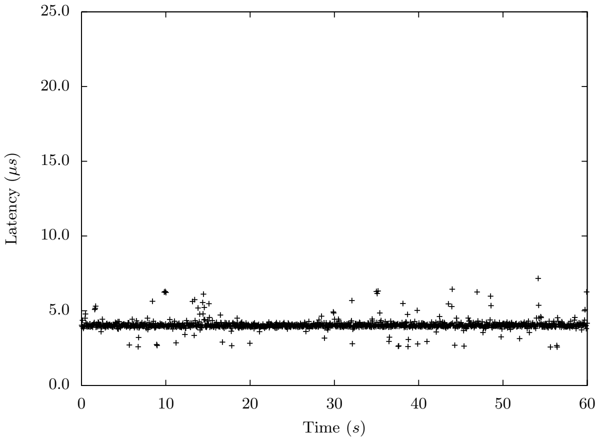
Since Linux$^{\mathbf{Xen}}$ presented the best results in our previous experiments, we decided to run the second experiment during a longer period to see how stable this system would be. Thus, we ran the second experiment for 14 hours with the



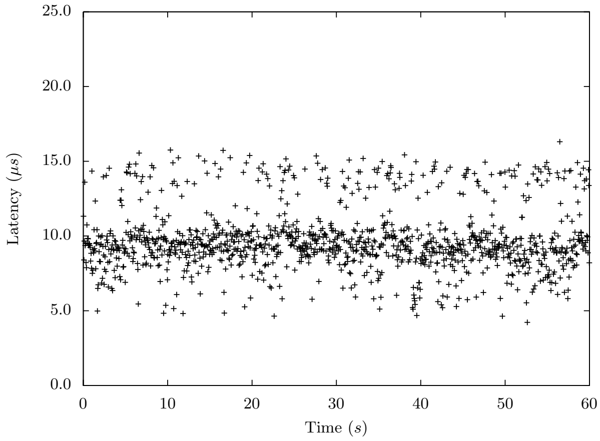
(a) **Linux$^{\mathbf{Xen}}$ - Interrupt latency - without load**
M: 9.1, SD: 0.3, Mn: 9.8, Mx: 13.4



(b) **Linux$^{\mathbf{Xen}}$ - Interrupt latency - with load**
M: 11.3, SD: 1.2, Mn: 9.0, Mx: 19.7



(c) **Linux$^{\mathbf{Xen}}$ - Activation latency - without load**
M: 4.0, SD: 0.3, Mn: 0.3, Mx: 9.6



(d) **Linux$^{\mathbf{Xen}}$ - Activation latency - with load**
M: 9.8, SD: 2.0, Mn: 2.7, Mx: 20.8

8: Linux$^{\mathbf{Xen}}$ latencies. Interrupt requests at E_M are triggered at a $20Hz$ frequency by E_T.

same load scenarios presented earlier. This setup generated more than 1 million events. The histogram of Figure 6.3 presents the number of events per activation and interrupt latencies in 1 μs steps on a logarithmic scale. From this figure, we see that over $100,000$ events had both activation and interrupt latencies within $[10, 11]$ μs. Although some worst-case latencies are greater than those observed in the corresponding 10-minute experiments, these were very rare events.

7. RELATED WORK

Some experimental results comparing Linux$^{\mathbf{Prt}}$ and Linux$^{\mathbf{Std}}$ are presented in [18]. They measured interrupt and scheduling latencies of a periodic task. However, their experiments were conducted without processor load and the methodology used was not precisely described. Siro et al [20] compares Linux$^{\mathbf{Prt}}$, RT-Linux [22] and Linux$^{\mathbf{RTAI}}$ [14] with LMbench [12] by measuring the scheduling deviation of a periodic task. The authors tested the systems with a load overhead, but they did not consider interrupt load. In their website, the developers of the Adeos project [2] present some comparative results for Preempt-RT and Adeos. In their evaluation, they used LMbench [12] to characterize the performance of the two platforms and measured the interrupt latencies gathered from the parallel port.

The interrupt latency results of our work are similar to those obtained by Benoit et al [2] for Linux$^{\mathbf{Xen}}$. However, our results differ from their work for Linux$^{\mathbf{Prt}}$ since we noticed some degradation of time guarantees by this platform, as reported in Sections 6.2.1 and 6.3. Regarding activation latencies under load scenarios, we are not aware of any other comparative work. Experiments similar to those reported here were conducted for Linux$^{\mathbf{RTAI}}$ [16]. As expected, the obtained results are similar to those presented for Linux$^{\mathbf{Xen}}$, since both platforms use Adeos nanokernel. Some of the results presented in this paper have recently been discussed in a local forum [17].

8. CONCLUSION

In this work, we have conducted a comparative evaluation of two Linux-based RTOS. Our comparative methodology has allowed experimental measurements of interrupt and activation latencies in scenarios of variable load. Load of both processing and those due to interrupt handling have been considered. Two experiments have been defined. In the simpler one, the same station that deals with real-time activities is responsible for the measurements. In the second, the measurements are carried out externally, by a different station. Both experiments can be used for comparison purposes although the second one gives the values of interrupt latencies more accurately.

While standard Linux presented latencies in the worst case over $100\mu s$, the platforms Linux$^{\mathbf{Prt}}$ and Linux$^{\mathbf{Xen}}$ managed to provide temporal guarantees with a precision below $20\mu s$. However, in order to achieve this behavior with Linux$^{\mathbf{Prt}}$, it was necessary to disable the interruption threading for the parallel port IRQ line, making the system less flexible. With a threaded implementation, the behavior of Linux$^{\mathbf{Prt}}$ suffers considerable deterioration of its temporal predictability. Linux$^{\mathbf{Xen}}$ was found more appropriate since offers a user-mode programming environment as well as better temporal predictability, a desirable characteritic for supporting real-time systems.

9. REFERENCES

[1] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of the linux kernel. In *Proc. of the Real-Time Technology and Applications Symposium (RTAS02)*, pages 1–4, 2002.

[2] K. Benoit and K. Yaghmour. Preempt-RT and I-pipe: the numbers. http://marc.info/?l=linux-kernel&m=112086443319815&w=2, 2005. Last access 03/08.

[3] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–126, 2006.

[4] M. C. D. P. Bovet. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, 2005.

[5] L. Dozio and P. Mantegazza. Linux real time application interface (RTAI) in low cost high performance motion control. In *Proceedings of the conference of ANIPLA, Associazione Nazionale Italiana per l'Automazione*, 2003.

[6] G. Fry and R. West. On the integration of real-time asynchronous event handling mechanisms with existing operating system services. In *Proceedings of the International Conference on Embedded Systems and Applications (ESA'07)*, 2007.

[7] I. Molnar et al. PreemptRT. http://rt.wiki.kernel.org - Last access jan. 08, 2008.

[8] L. Torvalds et al. Kernel. http://www.kernel.org - Last access jan. 08, 2008.

[9] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[10] M. Marchesotti, M. Migliardi, and R. Podestà. A measurement-based analysis of the responsiveness of the Linux kernel. In *Proc. of the 13th Int. Symposium and Workshop on Engineering of Computer Based Systems*, volume 0, pages 397–408. IEEE Computer Society, June 2006.

[11] P. McKenney. A realtime preemption overview. http://lwn.net/Articles/146861/ - Last access jun. 08, 2005.

[12] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[13] P. Gerum et al. Xenomai. http://www.xenomai.org - Last access jan. 08, 2008.

[14] P. Mantegazza et al. RTAI. http://www.rtai.org - Last access jan. 08, 2008.

[15] S. L. Pratt and D. A. Heger. Workload dependent performance evaluation of the linux 2.6 i/o schedulers. In *Proceedings of the Linux Symposium*, pages 425–448, 2004.

[16] P. Regnier. Especificação formal, verificação e implementação de um protocolo de comunicação determinista, baseado em ethernet. Master's thesis, Federal University of Bahia, Salvador, BR, 2008.

[17] P. Regnier, G. Lima, and L. Barreto. Avaliação do determinismo temporal no tratamento de interrupções em plataformas de tempo real linux. In *Proc. of Workshop of Sistemas Operacionais*, Belém, PA, Julho 2008.

[18] S. Rostedt and D. V. Hart. Internals of the rt patch. In *Proceedings of the Linux Symposium*, pages 161–172, 2007.

[19] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An approach to real-time synchronisation. *IEEE Transaction on Computers*, 39(9):1175–1185, 1990.

[20] A. Siro, 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 *Proceedings of the 9th Real-Time Linux Workshop*, 2007.

[21] D. Stodolsky, J. Chen, and B. Bershad. Fast interrupt priority management in operating systems. In *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 105–110, Sept. 1993.

[22] V. Yodaiken et al. RT-Linux. http://www.rtlinuxfree.com - Last access jan. 08, 2008.

[23] K. Yaghmour. The real-time application interface. In *Proceedings of the Linux Symposium*, July 2001.