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 can be found nowadays. Two of them have received special attention recently, the patches 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 that 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 include various applications related to areas of telecommunications, multimedia, industry, transport, medicine, etc. For such systems, the choice of Real-Time Operating Systems (RTOS) is a fundamental aspect of design. Despite the importance of technological hardware developments, some innovations may introduce obstacles to the construction of RTOS. For example, the cache memory access, direct memory access (DMA), co-processing, prediction of instructions, multi-core units, pipelines and executions out of order are non-negligible sources of indeterminism [9, 15]. Thus, the construction of a general purpose operating system with a focus on timing predictability remains a challenge of current research.

Despite its spread and popularity, the *kernel* standard Linux [4] fails in the provision of timing guarantees typical of real-time critical systems [10, 1]. To circumvent this problem, several approaches have been developed in order to increase the timing predictability of Linux [7, 13, 5, 21, 6, 3]. The rapid evolution and diversity of solutions available calls for comparative studies to assess the determinism offered by each platform in order to help the designer of real time systems in choosing the appropriate solution to be used.

This paper aims to present and compare the *kernel* Linux patches Preempt-RT (Linux$^{\mathbf{Prt}}$ [7] and Xenomai (Linux$^{\mathbf{Xen}}$) [13], developed to increase the predictability of Linux. The main contribution of this work are: (i) the proposal of an evaluation methodology, based on simple software and hardware cots, which uses an overload of the processor load through I/O processing and interruption, (ii) the assessments of results that confirm the ability of Linux$^{\mathbf{Xen}}$ Xen to offer critical time guarantees, and (iii) the confirmation that, in situations of intense load, Linux$^{\mathbf{Prt}}$ does not offer timing guarantees as predictable as Linux$^{\mathbf{Xen}}$.

Section 2 describes some factors of unpredictability of Linux and defines the metrics adopted for the comparison of Linux$^{\mathbf{Prt}}$ and Linux$^{\mathbf{Xen}}$. These two platforms then are described in Sections 3 and 4. Thereafter, the description of the methodology of experimentation in Section 5 precedes the presentation of experimental results in Section 6. Finally, Section 7 briefly introduced the related work and Section 8 concludes this work.

2. COMPARISON METRICS

The conventional method used to minimize the impact of interruptions on the implementation of processes is to divide the implementation of the interruption handler into two parts. The first part, refered to as the **critical section** of the handler, run critical operations immediatly, with interruptions disabled. Eventually, one can enable interruptions during par of the critical section in order to allow preemption. However, such implementation must rely on locks to ensure a 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 the interruptions enabled. In Linux, such delayed executions are called ***softirqs***.

2.1 Interrupt latency

An interrupt request, or simply **interruption**, of the processor by a device is tipically asynchronous and can happen at any time during the processor execution cycle. In particular, such a request can occur while the critical section of another interrupt handler is running, with interruptions disabled. This scenario may cause a non deterministic latency for the detection of interrupt request by the processor.

The time interval between the instant in which an inter-

rupt request happens and the beginning of its associated handler execution is called **interrupt latency**. As it characterizes the system's capability to react to external events, this quantity was regarded as a metric for the purpose of the comparison of the studied platforms .

2.2 Activation latency

In the *kernel* Linux, just after the end of the interrupt handler critical section, the associated *softirq* becomes able to perform. However, between the instant in which the critical section execution terminates and the instant in which deferred *softirqs* begins to execute, others interruptions may occur, causing a possible delay in the *softirqs* execution.

In real-time platforms, timeouts events or hardware events are used to trigger tasks, in a similar manner as for *softirqs*. Such a task, often periodic, is suspended while waiting for some event. When this event occurs, the associated interrupt request triggers the corresponding handler which, in turn, wakes the task up. The time interval between the instant when the event occur and the beginning of the execution of the associated task is called **activation latency**. As for the *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. Like the interrupt latency, the activation latency characterizes the capability of a system to react to external events. Thus, this quantity was also regarded as a metric for the purpose of the comparison of the studied platforms.

3. LINUX PREEMPT-RT

To provide accurate time measurement, Linux$^{\mathbf{Prt}}$ [11, 17] uses a new implementation of high resolution timers developed by Thomas Gleixner [8]. Either based on the value of the Time Stamp Counter (TSC) register of the Intel architecture or on high resolution clocks, the implementation offers an API which allows for micro-seconds time resolution measurement. According to results presented [17, 19], the activation latencies obtained using this API are of the order of some tens of μs in current computers.

Linux$^{\mathbf{Prt}}$ includes several changes that make the *kernel* totally preemptable. Thus, as soon as a process of highest priority is released, it can acquire the processor with minimal latency, with no need to wait for the end of the execution of a process of lower priority, even if such a process is running in *kernel* mode. For instance, in order to limit the effects of unpredictability caused by shared resources, Linux$^{\mathbf{Prt}}$ modify the primitives of synchronization to enable the implementation of a protocol based on inheritance of priority [18].

As for the interrupt latency, Linux$^{\mathbf{Prt}}$ uses threads of interruptions. When an IRQ (Interrupt Request) line is initialized, a thread is created to manage the interrupt requests associated with this line. When a request occur, the associated handler masks the request, wakes the associated thread up and returns to the interrupted code. Thus, the critical part of the interrupt handler is reduced to its minimum and the latency caused by its execution is both short and deterministic. Eventually, the interrupt thread is scheduled, according to its priority, giving room for unpredictability, which will be object of this study.

Using programs correctly written, respecting the rules of Linux$^{\mathbf{Prt}}$ programming and allocating resources in accordance with the time requirements, the Linux$^{\mathbf{Prt}}$ platform has the advantage to offer the programming environment of the Linux operating system, giving access to C libraries and to the many available softwares for this system.

4. LINUX XENOMAI

Unlike Linux$^{\mathbf{Prt}}$, the Linux$^{\mathbf{Xen}}$ platform uses an approach based on the interrupt indirection mechanism introduced in the technique of "optimistic interrupt protection " [20]. When an interrupt request happens, the indirection layer, also called the *nanokernel*, determines if the request is destinated to a real time task or if is is directed to a Linux process. In the first case, the interrupt handler is executed immediately. Otherwise, the request is enqueued and eventually delivered to Linux when no real time task is pending. Whenever the Linux *kernel* must disable interruptions, the *nanokernel* just let Linux believe that the interruptions are disabled. However, the *nanokernel* continues to intercept any interruption of hardware. In this case, the interruption is treated immediately if it is directed to a real time task. Otherwise, the interruption is enqueued until the *kernel* Linux get back to enable interruptions.

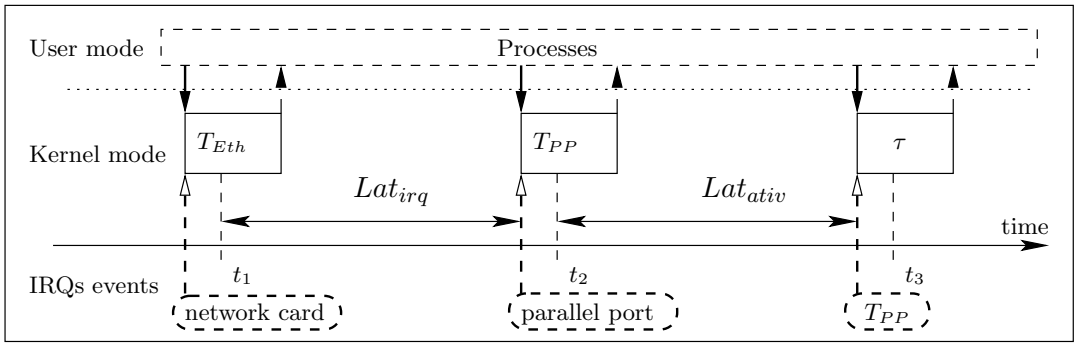
For the implementation of the *nanokernel*, Linux$^{\mathbf{Xen}}$ uses a layer of virtualization of resources called Adeos (Adaptative Domain Environment for Operating Systems) [22]. The patch Adeos facilitates the sharing and use of hardware and provides an integrated programming simple and independent of architecture. In short, Adeos is based on the concepts of domain and hierarchical interrupt pipeline. A domain features an isolated environment for execution, in which one can run programs or even a complete operating systems. The hierarquical interrupt pipeline, called **ipipe**, serves to prioritize the delivery of interruptions between domains. When a domain is registered in Adeos, it is allocated a position in the ipipe according to its time requirements. Adeos then uses the mechanism of interrupt indirection to organize the delivery of interrupts hierarchically, according to the domains priorities.

The real-time services of Linux$^{\mathbf{Xen}}$ correspond to the domain of highest priority in the ipipe, called "primary" domain. This domain corresponds therefore to the real time nucleus in which tasks are executed in protected mode.

The "secondary domain", in turn, refers to the *kernel* Linux, in which the set of usual Linux software libraries is available. However, the timing guarantees are weaker, given that the code can use the blocking system calls of Linux.

5. EXPERIMENTAL METHODOLOGY

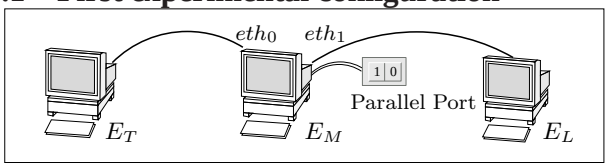
In general, perform accurate time measurements at the level of the interrupt handler may not be as simple. In fact, the exact instant in which a interrupt request happens is difficult to measure, as this event is asynchronous and can be caused by any hardware device. To obtain reliable interrupt and activation latency measurements with a high degree of accuracy, external devices such as scopes or other



1: Interrupt and activation latencies measurement at station E_M

computers, are needed. Since the objective of this work was to characterize and compare the degree of determinism of operational platforms studied, a simple and effective experimental methodology was first adopted, which can be easily replicated in other contexts. A second experiment, based on an more elaborated measurement strategy was used to assess the validity of obtained results.

5.1 First experimental configuration



2: First experimental configuration

The first experiment use three stations:

1. The triggering station E_T is used to send Ethernet packets with a fixed frequency to station E_M;
2. The measurement station E_M where latencies are measured and where a real time task τ is waiting for external events;
3. The loader station E_L is used to create an interrupt request load at station E_M.

The E_T and E_L stations are connected to the E_M station by two separate Ethernet network, using two Ethernet device (eth_0 and eth_1), as illustrated by Figure 2.

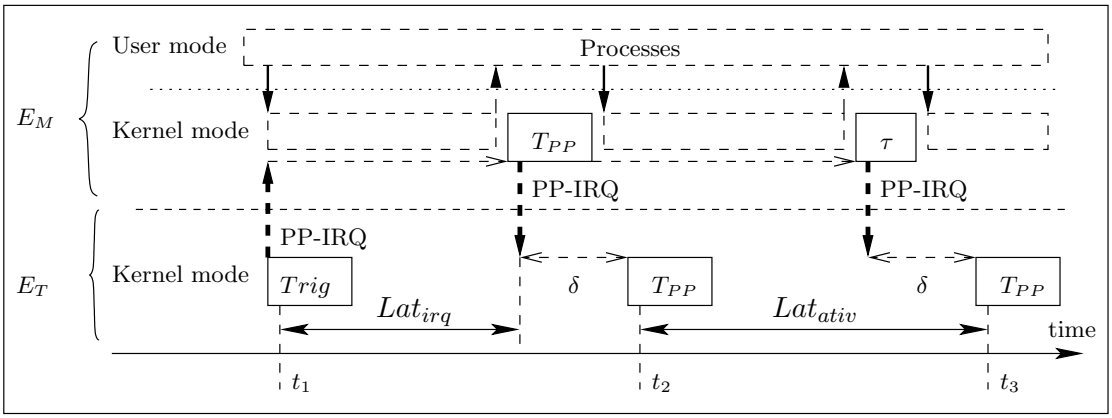
The arrivals of packages sent by station E_T serve to trigger a cascade of events at station E_M, allowing the simulation of external events via its parallel port (PP). More explicitly, each arrival of a package at device eth_0 is used to trigger an interrupt request to the PP, setting its IRQ line (see Figure 1): in the interrupt handler of the eth_0 network device, the IRQ line of the parallel port is set and the corresponding instant t_1 of the interrupt request is memorized. Thereafter, this initial event starts a sequence of two events used to measure the latency of (Lat_{irq}) and activation (Lat_{ativ}) activate). It is worth noting that there is no relationship between the arrival of packages at the eth_0 device and the E_M processing activity.

The measures were realized according to the following roadmap, as illustrated in Figure 1.

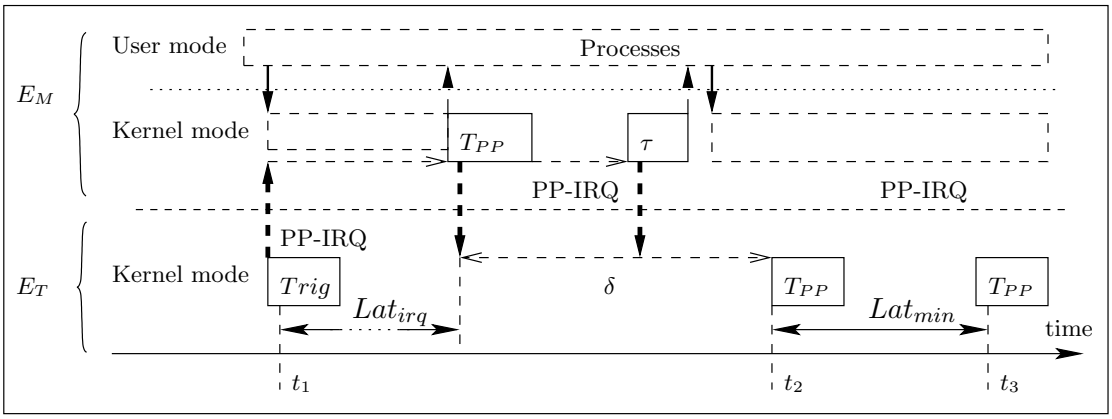
- The E_T station sends Ethernet frame to the eth_0 device of the E_M station, causing interrupt asynchronously in regard to applications executing on E_M.
- The interruption associated with the arrival of a package causes preemption of the application executing on the processor by the interrupt handler (T_{eth_0}) of the network device eth_0.
- The interrupt handler (T_{eth_0}) is reduced to its minimum: it only sets the Parallel Port Interrupt Request (PP-IRQ) line and stores the instant t_1 in memory. Hence, t_1 is the value read on the E_M local clock at the setting instant of the PP-IRQ line, just after the arrival of an Ethernet frame.
- The interrupt request associated with the setting of the PP-IRQ line causes the preemption of the application running on the processor by the PP interrupt handler T_{PP}.
- T_{PP} records the instant t_2 and wakes up task τ. This second instant t_2 corresponds to the value of the E_M local clock, just after the start of T_{PP}.
- When that task τ wakes up, it records the instant t_3 and is suspended until the next PP interruption. So, t_3 is the instant when task τ begins to execute at the end of the cascade of events caused by the arrival of a package in the network card.

As depicted in Figure 1, Lat_{irq} corresponds to the difference $t_2 - t_1$ and Lat_{ativ} to the difference $t_3 - t_2$. During the experiment, the transfer of measurements of memory to the file system was performed by a FIFO channel read by a user process in order to prevent any interference between the data acquisition and its storage in the file system. Such isolation was guaranteed by two facts: first the priority of the user process was smaller than the priority of interrupts handlers executed in *kernel* mode and second, data transfers were sufficiently rare events (20 per second) not to interfere in the measures realized.

One of the main drawbacks of the experimental setting describes in this Section is that measurements are realized by



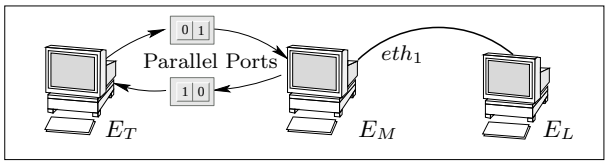
3: Interrupt and activation latencies measurement at station E_M



4: Minimal activation latency - The first and second interrupt trigerred at E_T by E_M occurs in a time interval leather than δ.

the station which is trigerred. To do so, station E_M waits for the asynchronous arrival of an Ethernet frame at eth_0 to launch the cascade events which leads to local measurements. The consequence of such approach is that the parallel port interrupt is always requested on behalf of the eth_0 interrupt handler, thus compromising the validity of the interrupt latency measure. In order to avaliate the impact of such bias, a second experiment was implemented, following suggestion of a revisor of a previous version of this work.

5.2 Second experimental configuration



5: Second experimental configuration

The second experiment use the same three stations, but, while station E_L remains configured as before, stations E_T and E_M were used in different modes, as illustrated by Figure 5:

1. The triggering station E_T is now connected to the E_M station via the parallel port and directly triggers PP interrupt request at a fixed frequency of $20Hz$;
2. The measurement station E_M handles the PP-IRQ and wakes up the real time task τ which is waiting for external events;

In this new configuration, time measurements were not be realized at E_M, because the instant of the initial PP-IRQ was not known by E_M, unless clocks would be synchronized. To avoid the need for such synchronization, time measurements were only carried by E_T. To do so, the parallel port was used by E_M to trigger an interrupt request at E_T, as illustrated in Figure 3. Whenever such request happened, the PP-IRQ handler of E_T stored the value of the TSC. Such approach increased the timestamp value of the interrupt event produced by E_M by the interrupt latency of station E_T. However, as this station used Linux$^{\mathbf{Xen}}$ and was running in single mode with minimal load, this extra latency, denoted δ in Figure 3, was bounded and deterministic. An estimate of δ can be deduced dividing the total latency obtained with E_M using Linux$^{\mathbf{Xen}}$ with minimal load. Such estimates gives $\delta \approx 9\mu s$. Using this value, the desired measurements of Lat_{irq} and Lat_{ative} can be deduced with a good accuracy.

The measures were realized according to the following roadmap, as illustrated in Figure 3.

- The E_T station triggers an interrupt request on the PP-IRQ line of station E_M and stores the instant t_1 in memory. Hence, t_1 is the value read on the E_T local clock at the setting instant of the PP-IRQ line.
- The interrupt request triggered on the PP-IRQ line of station E_M causes the preemption of the application running on the E_M processor by the PP interrupt handler $T_{PP}(E_M)$.
- $T_{PP}(E_M)$ triggers an interrupt request on the PP-IRQ line of station E_T and wakes up task τ.
- The interrupt request triggered on the PP-IRQ line of station E_T causes the execution of the PP interrupt handler ($T_{PP}(E_T)$) which stores the t_2 instant value. This second instant t_2 corresponds to the value of the E_T local clock, just after the start of $T_{PP}(E_T)$.
- When that task τ wakes up, it triggers an interrupt request on the PP-IRQ line of station E_T.
- Finally, the interrupt request triggered on the PP-IRQ line of station E_T causes the execution of the PP interrupt handler ($T_{PP}(E_T)$) which stores the t_3 instant value. So, t_3 is the instant when task τ begins to execute increased by the E_T interrupt latency.

As depicted in Figure 3, Lat_{irq} corresponds to the difference $t_2 - t_1 - \delta$ and Lat_{ativ} to the difference $t_3 - t_2 - \delta$.

As in the first experiment, the transfer of measurements of memory to the E_T file system was performed by a FIFO channel read by a user process in order to prevent any interference between the data acquisition and its storage in the file system.

5.3 I/O, processing and interrupt loads

Initially, we realized experiments with a minimum load on the processor of the E_M station (*kernel* mode single). Thus, the temporal behavior of the three platforms in favourable situation was observed. Then, two different types of loads were used simultaneously to overload the E_M station. Such overloads were meant to assess the capability of each platform to ensure deterministic latencies in the treatment of interruptions and activation of real-time tasks, despite the existence of other non-critical activities. The loads of I/O and processing were realized executing the following instructions at E_M station:

```
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
```

Another stress of interruption was created using a UDP communication between the E_M station configured as a server and the *Load* station configured as a client. To isolate the communication between the E_M and *Load* stations, the second network device (eth_1) of the E_M station has been used, as well as illustrated by Figure 1. During the experiment, the process client hosted by the *Load* station sent small packages of 64 bytes at the maximum frequency ($200kHz$) allowed by the bandwidth network. Thus, more than $100,000$ per second interruptions were generated at the eth_1 E_M card. This device made use of the E_M IRQ line 18, whose priority is lower than the priority of the PP-IRQ line. Such a low priority was choosen in order to test the efficiency of the sudied platform, as one should expect that latencies would not be affected by the load.

In experiments with overload, the two types of loads were applied simultaneously and the measurements were only started a few seconds later.

6. LINUX^{PRT} AND LINUX^{XEN} ASSESSMENT

6.1 Configuration

The experiments were conducted on three Pentium 4 computers with $2.6GHz$ processors and $512MB$ RAM memory, in order to illustrate the temporal behavior of the three following platforms:

- **Linux^{Std}:** Linux standard - *kernel* version 2.6.23.9 (*low-latency* option);
- **Linux^{Prt}:** Linux with *patch* Preempt-RT (rt12) - *kernel* version 2.6.23.9;
- **Linux^{Xen}:** Linux with *patch* Xenomai - version 2.4-rc5 - *kernel* version 2.6.19.7.

The Linux^{Std} configuration was used to perform reference experiments in order to allow for the comparison with the two real-time platforms Linux^{Prt} and Linux^{Xen}. The stable version of *kernel* 2.6.23.9, released in December 2007, was chosen for the study of Linux^{Prt} because this patch has evolved rapidly since its first stable version released two years ago. However, the *kernel* version of 2.6.19.7 was used for the study of version 2.4-rc5 of Xenomai. In fact, it was considered unnecessary to upgrade the *kernel*version because Xenomai is based on Adeos (see Section 4) and therefore, the temporal guarantees offered to applications executing in the first domain only depend on the Xenomai version and its associated patch Adeos, and not on the Linux *kernel* version.

The interrupt and activation latency measurements were made by reading the Time Stamp Counter (TSC), allowing a precision of less than $30ns$ (88 cycles), experimentally verified. In both experiments, a $20Hz$ frequency was used by station E_T to trigger events at station E_M. For each platform and configuration, two experiments were conducted of 10 minutes each. The first was carried on without any load of the system and the second was carried on applying the loads presented in Section 5.3.

Experimental results are presented through Figures in which the horizontal axis represents the instant of observation ranging from 0 to 60 seconds and the vertical axis represents the

latencies measures in μs (such values should be multiplied by 2.610^3 to obtain the number of TSC cycles). Although each experiment was run for ten minutes, Figures are presented for a range of $60s$, as such interval is sufficient to observe the pattern of behavior of each platform. During this one minute interval, the total number of events is 1200, as the arrival frequency of Ethernet frame at the eth_0 network device of station E_M is $20Hz$.

Below each figure, the following values are given: Mean (Mean), Standard Deviation (SD), minimum (Mn) and maximum (Mx). These numbers were obtained considering the duration of ten minutes of each experiment. As much as possible, the same vertical scale was used for all graphics. As a result, high values may have layed outside the graphics. Such occurrences was represented by a triangle near the maximum value of the vertical axis.

6.2 First configuration experimental results

The experimental results for the first configuration (see Section 5.1) are shown in Figure 6 and7.

6.2.1 Interrupt latencies

Figure 6 shows the interrupt latencies measured, with and without load of the system. As can be seen, without load, Linux$^{\mathbf{Std}}$ and Linux$^{\mathbf{Xen}}$ has similar behaviour. With loads, there is a significant variation of Linux$^{\mathbf{Std}}$, as expected.

With respect to Linux$^{\mathbf{Prt}}$, two results draw attention. First, in absence of load, the system displays latencies of around $20\mu s$. This is due to the threaded implementation of interruption seen in Section 3. Second, contradicting the expectations, the application of load had a significant impact, causing a high variability of latency. In fact, between the instant in which the T_{PP} handler requires the interruption of the processor and the instant in which the IRQ thread actually wakes up, one or more interruptions may occur. In such a case, the execution of the associated handlers may delay the execution of T_{PP}.

To cancel this undesirable variability, one can use Linux$^{\mathbf{Prt}}$ without using the implementation of threads of interruption. For this, one use the option `IRQF_NODELAY` at initialization time of the IRQ line. As can be observed, using this option in requesting the PP-IRQ line, the behavior of Linux$^{\mathbf{Prt}}$ turns to be similar to Linux$^{\mathbf{Std}}$.

6.2.2 Activation latencies

Figure 7 shows the results for the activation latencies without and with load of the processor. As can be seen, the behavior of Linux$^{\mathbf{Std}}$ is inadequate to meet the real-time requirements. On the other hand, Linux$^{\mathbf{Prt}}$ and Linux$^{\mathbf{Xen}}$ have values of latencies within the expected standard. It is worth noting the behavior of these systems with load. Although the average value found for Linux$^{\mathbf{Xen}}$ $(8,7\mu s)$ is superior to the Linux$^{\mathbf{Prt}}$ $(3,8\mu s)$, the standard deviation is significantly lower in favor of Linux$^{\mathbf{Xen}}$, desirable feature in real-time critical systems. In fact, for such systems, hopes that the worst case is close to the average case.

It is also interesting to compare the behavior of Linux$^{\mathbf{Prt}}$ without using the implementation of interuption thread , that is, with the option `IRQF_NODELAY`, commented earlier. As can be seen in Figure7, despite the latencies of activation without load apresenting good results in comparison to Linux$^{\mathbf{Prt}}$, its values with load indicate a slightly less predictable behavior than Linux$^{\mathbf{Xen}}$.

6.3 Second configuration experimental results

The experimental results for the second experimental configuration (see Section 5.2) are shown in Figure 8 and 9.

6.3.1 Interrupt latencies

Figure 8 shows the interrupt latencies measured, with and without load of the system. As expected, an extra latency of 9 to $10\mu s$ is observed for the interrupt latencies measurements of all platforms. As explained in Section 5.2, this extra latency corresponds to the interrupt latency of E_T which was denoted δ in Figure 3.

The only other noticeable difference is related to Linux $^{\mathbf{Prt}}$ whose interrupt latencies values are somehow smaller in the second experiments. However, this is due to the thread implementation, which reacts faster to an hardware interrupt request than to a software interrupt request. As in the first experiment, using the option `IRQF_NODELAY` to disable the parallel port IRQ threading, the behavior of Linux$^{\mathbf{Prt}}$ turns to be similar to Linux$^{\mathbf{Std}}$.

As a first conclusion, we emphasize that results obtained with the second configuration are similar to those obtained with the first experimental configuration, thus motivating the use of the first experiment configuration for future comparison works.

6.3.2 Activation latencies

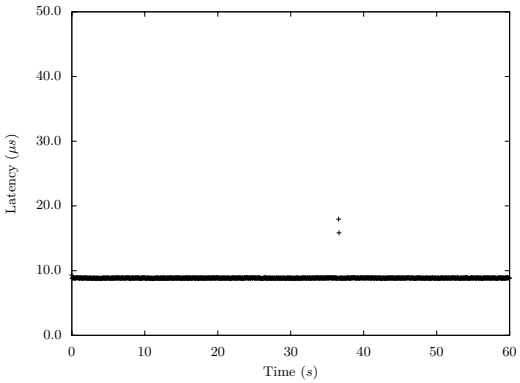
Figure 9 shows the activation latencies measured, with and without load of the system.

Somehow surprisingly, the δ extra latency observed in the interrupt latencies measurements is much smaller ($\delta \approx 2.5\mu s$) than the same latency observed after the first PP-IRQ issued by E_M. This can be explained as follow. When station E_M triggers the activation PP-IRQ, station E_T is just returning from the interrupt context of the interrupt PP-IRQ previously triggered by E_M. As a consequence, the E_M interrupt latency is significantly reduced. It is worth recalling that station E_T executes the Xenomai platform with minimal load, thus between the return of the first and the second interrupt triggered by E_M, no other interrupt can happen. However, the standard deviation is significantly increased, as the response time of the interrupt device (APIC) is source of temporal variation.

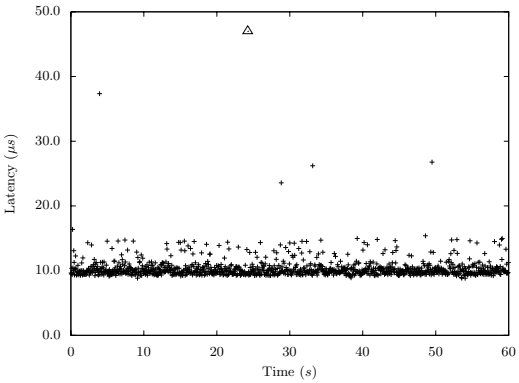
Except for this increase of the latencies by the δ interrupt latency at station E_T, and its related standard deviation, results obtained with the second experimental configuration are similar to those obtained with the first experimental configuration.

7. RELATED WORK

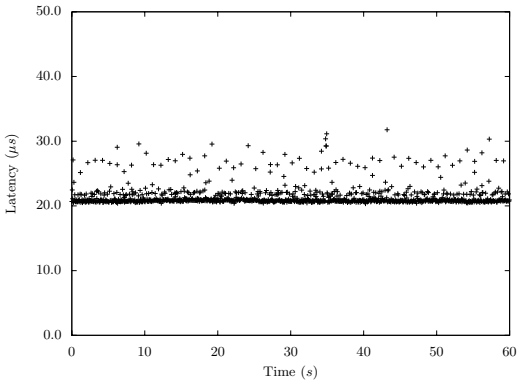
Some experimental results comparing Linux$^{\mathbf{Prt}}$ with Linux$^{\mathbf{Std}}$ are presented in [17]. Two metrics are used, interrupt and scheduling latencies, related to the schedule of a periodic



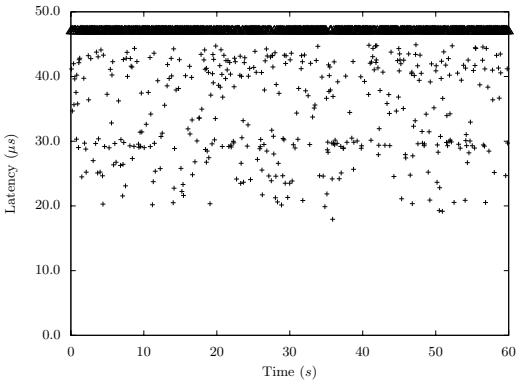
(a) **Linux$^{\mathbf{Std}}$ - without load**
M: 8.9, SD: 0.3, Mn: 8.7, Mx: 18.4



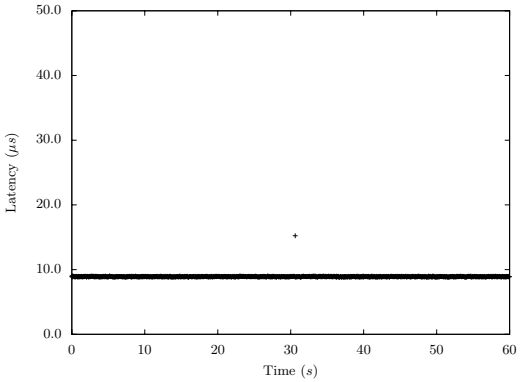
(b) **Linux$^{\mathbf{Std}}$ - with loads**
M: 10.4, SD: 1.9, Mn: 8.8, Mx: 67.7



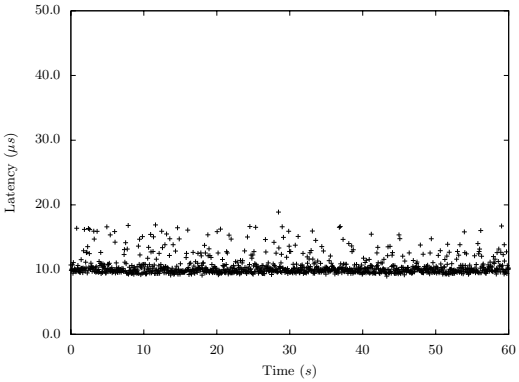
(c) **Linux$^{\mathbf{Prt}}$ - without load**
M: 21.5, SD: 1.7, Mn: 20.3, Mx: 45.1



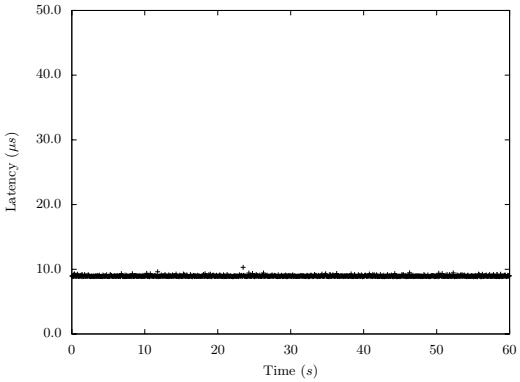
(d) **Linux$^{\mathbf{Prt}}$ - with loads**
M: 58.5, SD: 26.4, Mn: 17.2, Mx: 245.9



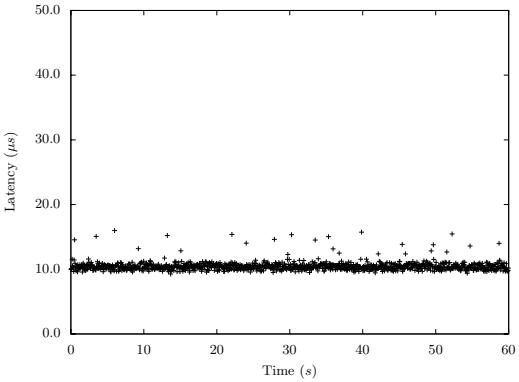
(e) **Linux$^{\mathbf{Prt}}$ - without load, option** `IRQF_NODELAY`
M: 8.9, SD: 0.2, Mn: 8.8, Mx: 16.7



(f) **Linux$^{\mathbf{Prt}}$ - with loads, option** `IRQF_NODELAY`
M: 10.6, SD: 1.6, Mn: 8.9, Mx: 35.8

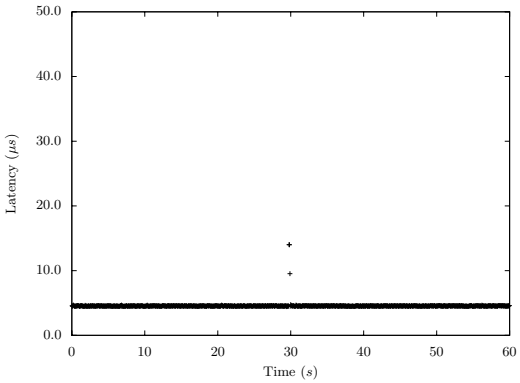


(g) **Linux$^{\mathbf{Xen}}$ - without load**
M: 9.0, SD: 0.1, Mn: 8.8, Mx: 11.1

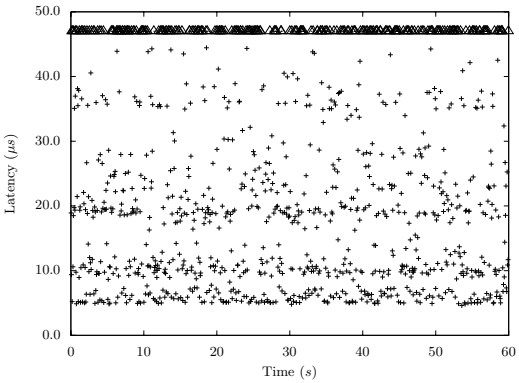


(h) **Linux$^{\mathbf{Xen}}$ - with loads**
M: 10.2, SD: 0.1, Mn: 8.8, Mx: 20.8

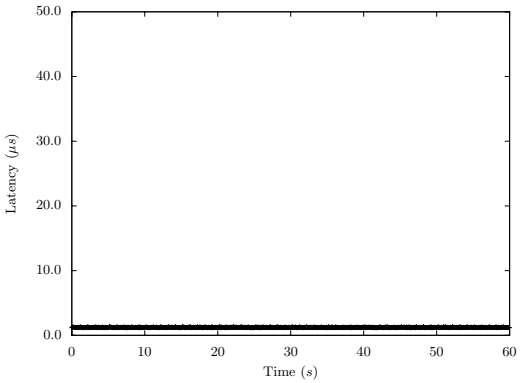
6: Interrupt latencies with PP-IRQ triggered by the eth_0 interrupt handler with a $20Hz$ frequency.



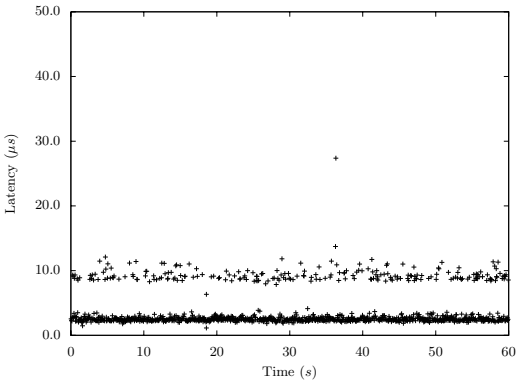
(a) **Linux$^{\mathbf{Std}}$ - without load**
M: 4.6, SD: 0.4, Mn: 4.4, Mx: 16.2



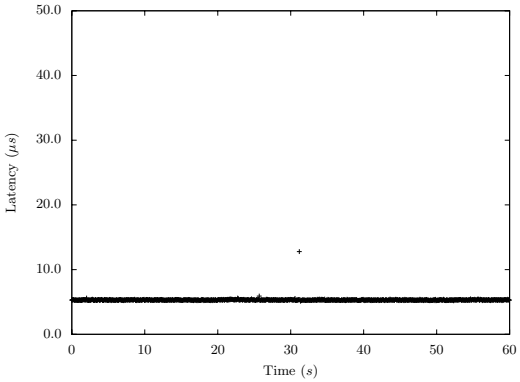
(b) **Linux$^{\mathbf{Std}}$ - with loads**
M: 37.3, SD: 48.2, Mn: 4.6, Mx: 617.5



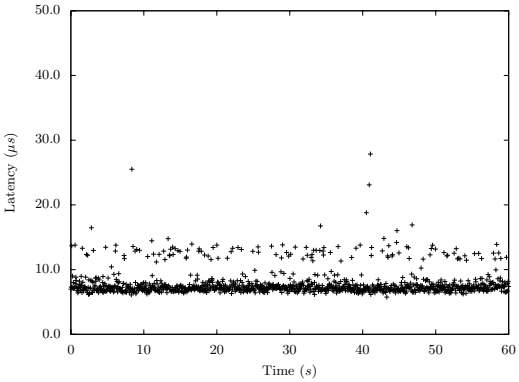
(c) **Linux$^{\mathbf{Prt}}$ - without load**
M: 2.1, SD: 0.2, Mn: 1.2, Mx: 9.4



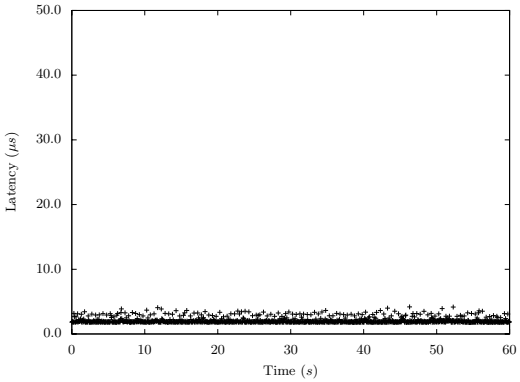
(d) **Linux$^{\mathbf{Prt}}$ - with loads**
M: 3.8, SD: 2.8, Mn: 1.1, Mx: 27.4



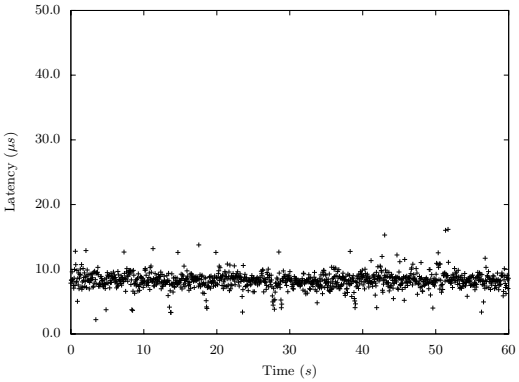
(e) **Linux$^{\mathbf{Prt}}$ - without load, option** `IRQF_NODELAY`
M: 5.3, SD: 0.3, Mn: 5.0, Mx: 13.1



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

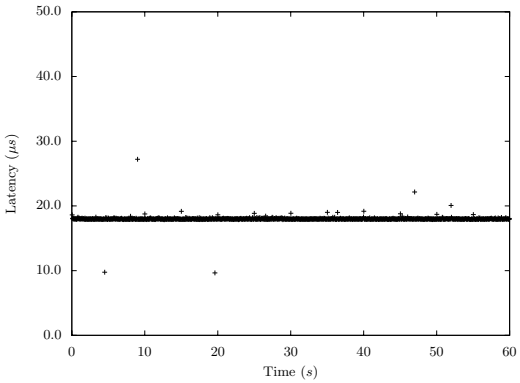


(g) **Linux$^{\mathbf{Xen}}$ - without load**
M: 2.1, SD: 0.5, Mn: 1.8, Mx: 8.4



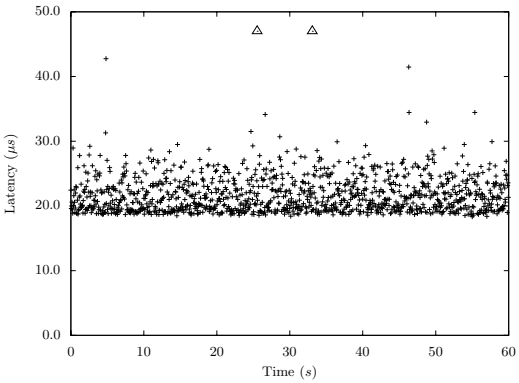
(h) **Linux$^{\mathbf{Xen}}$ - with loads**
M: 8.7, SD: 0.3, Mn: 1.8, Mx: 18.7

7: Activation latencies with PP-IRQ triggered by the eth_0 interrupt handler with a $20Hz$ frequency.



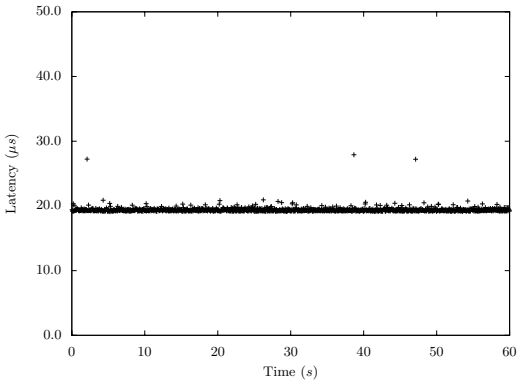
(a) **Linux$^{\mathbf{Std}}$ - without load**

M: 18.0, SD: 0.3, Mn: 9.6, Mx: 27.3



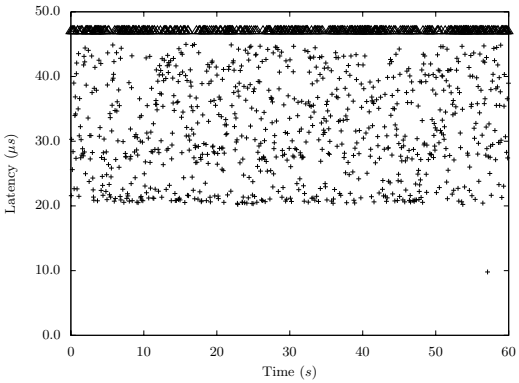
(b) **Linux$^{\mathbf{Std}}$ - with loads**

M: 21.8, SD: 3.2, Mn: 9.8, Mx: 80.6



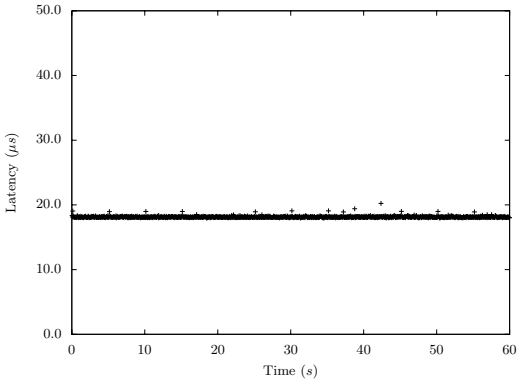
(c) **Linux$^{\mathbf{Prt}}$ - without load**

M: 19.4, SD: 0.5, Mn: 9.8, Mx: 42.2



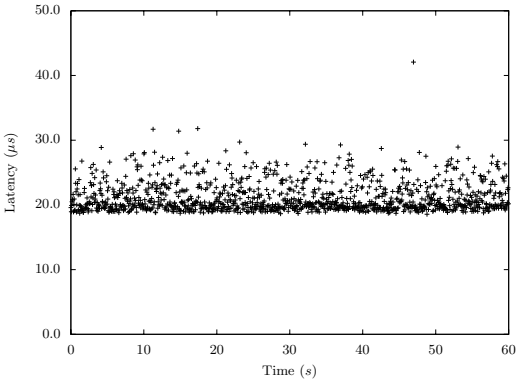
(d) **Linux$^{\mathbf{Prt}}$ - with loads**

M: 41.9, SD: 18.9, Mn: 9.7, Mx: 324.4



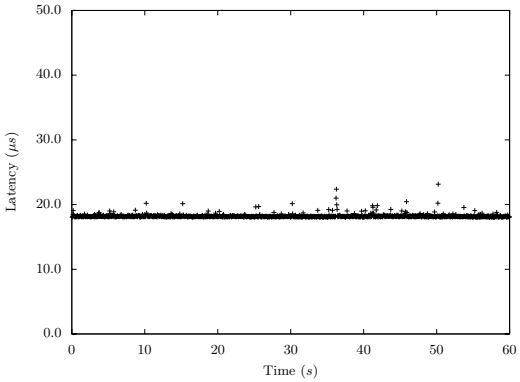
(e) **Linux$^{\mathbf{Prt}}$ - without load, option** `IRQF_NODELAY`

M: 18.1, SD: 0.3, Mn: 9.1, Mx: 25.5



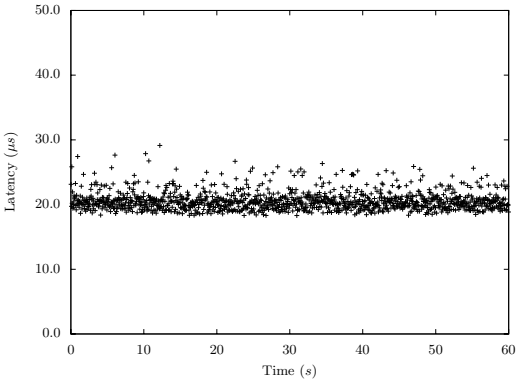
(f) **Linux$^{\mathbf{Prt}}$ - with loads, option** `IRQF_NODELAY`

M: 21.2, SD: 2.4, Mn: 9.1, Mx: 50.5



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

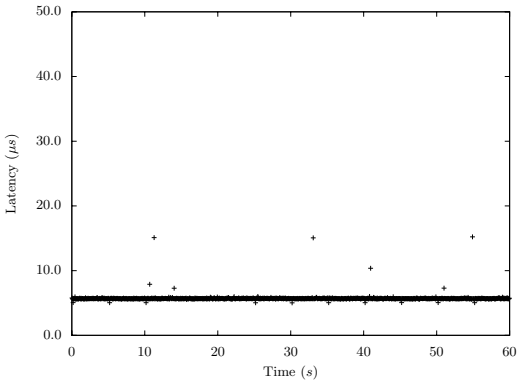
M: 18.2, SD: 0.3, Mn: 9.1, Mx: 23.1



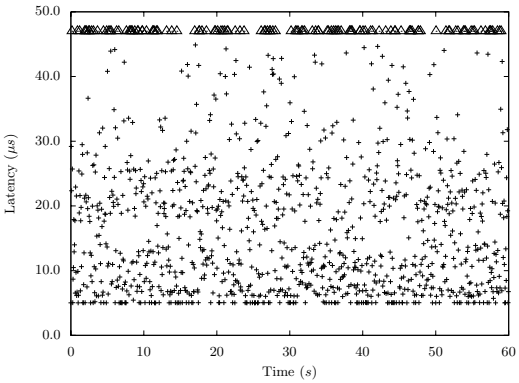
(h) **Linux$^{\mathbf{Xen}}$ - with loads**

M: 20.7, SD: 1.5, Mn: 9.1, Mx: 30.0

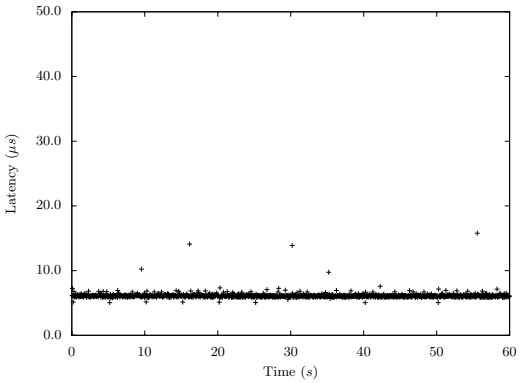
8: Interrupt latencies with PP-IRQ triggered on E_M by E_T at a $20Hz$ frequency.



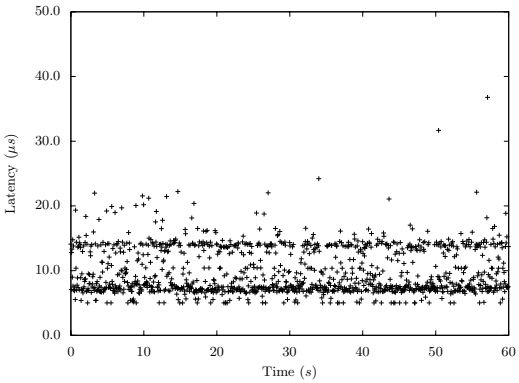
(a) **Linux$^{\mathbf{Std}}$ - without load**
M: 5.7, SD: 0.6, Mn: 5.0, Mx: 15.6



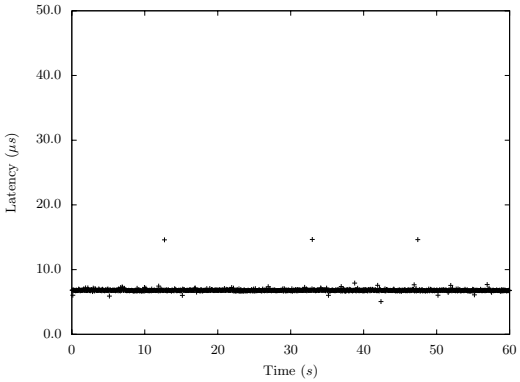
(b) **Linux$^{\mathbf{Std}}$ - with loads**
M: 23.3, SD: 28.5, Mn: 5.0, Mx: 623.3



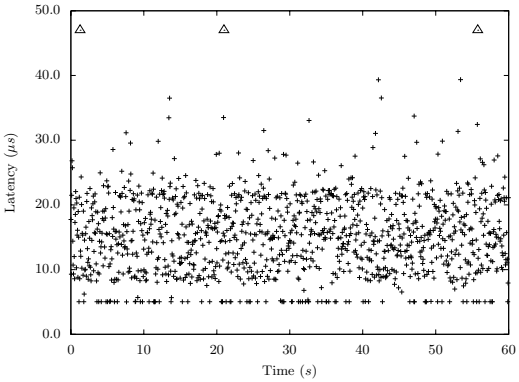
(c) **Linux$^{\mathbf{Prt}}$ - without load**
M: 6.1, SD: 0.6, Mn: 5.0, Mx: 17.3



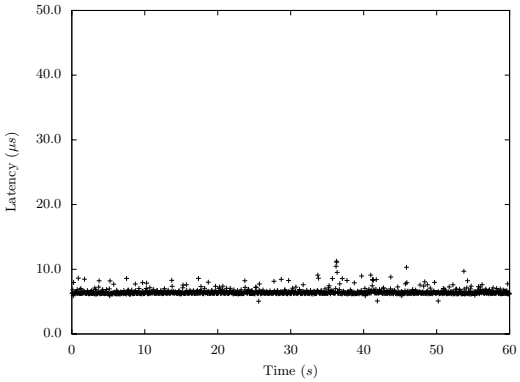
(d) **Linux$^{\mathbf{Prt}}$ - with loads**
M: 10.1, SD: 3.7, Mn: 5.0, Mx: 70.5



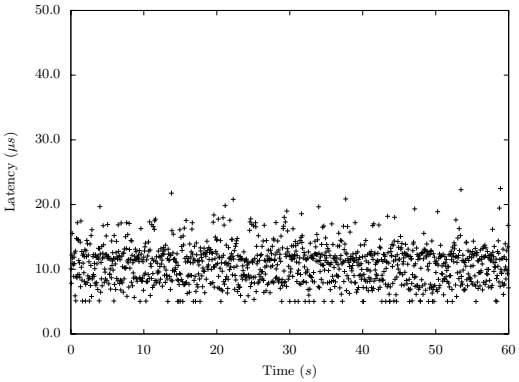
(e) **Linux$^{\mathbf{Prt}}$ - without load, option** `IRQF_NODELAY`
M: 6.8, SD: 0.4, Mn: 5.1, Mx: 15.4



(f) **Linux$^{\mathbf{Prt}}$ - with loads, option** `IRQF_NODELAY`
M: 15.6, SD: 6.6, Mn: 5.0, Mx: 107.2



(g) **Linux$^{\mathbf{Xen}}$ - without load**
M: 6.5, SD: 0.5, Mn: 5.0, Mx: 13.2



(h) **Linux$^{\mathbf{Xen}}$ - with loads**
M: 11.1, SD: 3.0, Mn: 5.0, Mx: 25.7

9: Activation latencies with PP-IRQ triggered on E_M by E_T at a $20Hz$ frequency.

task. However, the experiments were conducted without loading the processor and the methodology used was not precisely described. [19] realizes a comparative study of Linux$^{\mathbf{Prt}}$, RT-Linux [21] and Linux$^{\mathbf{RTAI}}$ [14] in which they use both LMbench benchmark [12] and measures of deviations in the scheduling of a periodic task. In these experiments, the authors applied an "average" load"of the processor, without considering interrupt load. In another work, only published on the Internet [2], the developers of the project Adeos present comparative results for Linux with the patches Preempt-RT and Adeos. This assessment, rather comprehensive, uses LMbench benchmark LMbench [12] to characterize the performance of the two platforms and presents results of measures taken to interrupt latencies with the parallel port.

This paper presents results of latency of interruption that confirms the results obtained in [2] for the Linux platform Linux$^{\mathbf{Xen}}$. Nevertheless, the results found here for Linux$^{\mathbf{Prt}}$, without the option `IRQF_NODELAY`, differed from those submitted by [2], as a degradation of time guarantees by the platform was observed, as seen in Section 6.3.1. As for activation latencies, we are not aware of any other comparative work. Experiments similar to those reported here were conducted for the Linux$^{\mathbf{RTAI}}$ platform [16] and the results are similar to those presented for Linux$^{\mathbf{Xen}}$, since both platforms use the same Adeos *nanokernel*.

8. CONCLUSION

In this work, the evaluation of two RTOS solutions based on Linux was held. The methodology has allowed experimental measurements of interrupt and activation latencies, in situations of variable load, both of the processor and of external events processed by interruption. Two experimental configurations were used, the first based on local measurements was compared with a more regular configuration based on external measurements through the parallel port. As both configuration gives similar results, it appears that the first methodology is founded and can be efficiently used for the purpose of real-time platforms comparisons.

While the 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 ordrer to achieve this behaviour with Linux$^{\mathbf{Prt}}$, it was necessary to disable the interruption threading for the parallel port IRQ line, making the system less flexible. With such thread, the behavior of Linux$^{\mathbf{Prt}}$ suffers considerable deterioration of its temporal predictability. The Linux platform Linux$^{\mathbf{Xen}}$ was found more appropriate since offers a user-mode programming environment as well as temporal predictability characteristic of real-time system.

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 *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 dez. 07, 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, G. Lima, and A. Andrade. TLA+ formal specification and verification of a real-time ethernet protocol. Submitted to SBMF, 2008.

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

[18] 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.

[19] 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.

[20] 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.

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

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