



## WHITE PAPER

### **A Comparison of Linux Alternatives for Hard Real-Time – Revised<sup>1</sup>**

Status: Released

December 17, 2004

Prepared by

Peter Laurich  
Akamina Technologies  
[www.akamina.com](http://www.akamina.com)

---

<sup>1</sup>Revised to update the LXRT data



## Table of Contents

|                                  |    |
|----------------------------------|----|
| 1 Foreword.....                  | 3  |
| 2 Introduction.....              | 3  |
| 3 OS Options Evaluated.....      | 4  |
| 3.1 Linux 2.4.....               | 4  |
| 3.2 Linux 2.6.....               | 4  |
| 3.3 Linux 2.4 with RTAI.....     | 4  |
| 3.4 Linux 2.4 with LXRT.....     | 5  |
| 4 Hardware Platform.....         | 5  |
| 5 Measurements.....              | 6  |
| 6 Test Set-up.....               | 7  |
| 6.1 Process Description.....     | 9  |
| 7 Detailed Process Models.....   | 9  |
| 7.1 Linux 2.4 and Linux 2.6..... | 9  |
| 7.2 Linux 2.4 with RTAI.....     | 10 |
| 7.3 Linux 2.4 with LXRT.....     | 12 |
| 8 Measurement Results.....       | 13 |
| 8.1 Interrupt Latency.....       | 14 |
| 8.2 Preemption Latency.....      | 17 |
| 8.3 Overall.....                 | 20 |
| 9 Conclusions.....               | 20 |
| 10 Resources.....                | 20 |



## 1 Foreword

This paper is a revised version of the original whitepaper: *A Comparison of Linux Alternatives for Hard Real-Time* that was released on November 19, 2004. This version includes updated data for the Linux 2.4 with LXRT configuration. The hard real-time LXRT task that was originally used to collect the preemption latency data included a `printf()` call after the call to make the task hard real-time. Since Release 24.1.13 and 3.0x this has the effect of forcing the task back to soft real-time where in previous releases it would not have. The original tests, that showed the LXRT results to be very similar to the Linux 2.4 alone, are what would be expected for LXRT tasks in soft real-time.

Thanks to Paulo Mantegazza for identifying the circumstances that led to the performance problem. We chose to re-run the LXRT tests without the print statement rather than the alternative of setting the LXRTmode to allow it. Consideration is being given to reverting back to a non-zero default for LXRTmode in Release 3.2.

## 2 Introduction

This paper provides an evaluation of a number of options for creating a hard real-time system based on Linux. A hard real-time system is a system that requires a guaranteed response to specific events within a defined time period. The failure of a hard real-time system to meet these requirements typically results in a severe failure of the system.

This work was carried out as part of an analysis to determine the suitability of using Linux in a hard real-time digital control system. The control loops in this system are executed at a rate of 100 Hz. A fundamental requirement of the system is that all of the control calculations must be completed within the 10 ms window available – regardless of any other loading on the system. In order to achieve this, it was felt that a maximum delay from the time the beginning of a control interval is signalled (the event) to the time that the control task is started should be less than 0.5 ms.

There are a number of Linux configurations that are available for real-time. Some of these are based on Linux alone, some use Linux with a sub-kernel and some use kernel patches to improve the real-time behaviour of Linux. This paper focuses on four of the possible configurations. The options selected are all freely available for download from official websites.

In this evaluation, Linux and any sub-kernels used were treated as black-boxes. The approach followed was to configure Linux and program the system for the various options and then measure results. No attempts were made to investigate the kernel specifics to determine why the configurations responded the way they did nor were any attempts made to improve the performance using custom patches or custom configurations.

Tests were carried out on a lightly loaded system and on a system under relatively heavy communications loading. Communications loading was used since it was easy to set up, it provided a way to exercise interface hardware and the device driver for the hardware, and it provided a realistic situation where hard real-time and communications messaging must be possible simultaneously.

This evaluation was done on an Intel x86 processor. Some of the options that were tested may not be supported on all processor architectures.



## 3 OS Options Evaluated

The four Linux options evaluated include:

1. the most up-to-date 2.4 Linux distribution downloaded from kernels.org (2.4.27)
2. a recent 2.6 Linux distribution downloaded from kernels.org (2.6.8.1)
3. a 2.4 Linux distribution with RTAI downloaded from rtai.org (3.1-test5)
4. a 2.4 Linux distribution with LXRT distributed with RTAI (3.1-test5)

Each of these options are defined further below.

### 3.1 Linux 2.4

Linux 2.4.27 was the most recent 2.4 release available at the time of the investigation. The 2.4 kernel is a stable kernel that has well documented deficiencies when it comes to hard real-time. Various approaches have been used to provide 2.4 with real-time capabilities. Linux 2.4 was included in the study as a baseline against which to compare the other options.

### 3.2 Linux 2.6

Linux 2.6 includes a number of significant improvements over Linux 2.4. Many of these improvements were made to enhance the real-time capabilities of Linux 2.6. Some of these include:

- New scheduler algorithm – O(1) algorithm should have superior performance especially under higher loads and on multiprocessor systems
- Kernel preemption patch – before 2.6, a user application could not preempt a task operating in kernel mode. With 2.6, preemption is now possible resulting in lower latencies for user interactive applications.
- Improved threading model and in-kernel support for Native Posix Threading Library (NPTL) increases the performance of threading operations and provides support for more threads.
- Merging of much of the uClinux project (Linux for microcontrollers). This provides support for processors that do not feature an MMU (Memory Management Unit)
- Improvements to the module subsystem

Please consult the resources listed in the resource section for additional information on Linux 2.6.

### 3.3 Linux 2.4 with RTAI

RTAI – Real-Time Application Interface – is a hard real-time extension to the Linux kernel. The RTAI project is a Free Software project that was founded by the Department of Aerospace Engineering of Politecnico di Milano (DIAPM). It has evolved into a community project coordinated by Professor Paulo Mantegazza of DIAPM.

RTAI is a sub-kernel that runs under Linux. It provides hard real-time response by running Linux as the idle task. Interrupts are intercepted by RTAI where they may be processed by an RTAI interrupt handler or passed up to Linux. RTAI runs only in kernel-space and, therefore, all RTAI tasks also need to run in kernel-space. Versions of RTAI are available that can be used with the Linux 2.4 kernel and the Linux 2.6 kernel.



Release 3.1 of RTAI is based on the Adeos nano-kernel – a migration away from the Real-Time Hardware Abstraction Layer (RTHAL) that is the centre of a patent infringement claim by FSMLabs. Future releases of RTAI will include fusion technology. RTAI/Fusion is the point of convergence of Adeos, LXRT, the preemptable Linux kernel and low latency enhancements. Version 0.6.2 of Fusion has been released but was not tested as part of the work on this paper as it was listed as being experimental.

For this option, all hard real-time tasks were implemented in kernel-space as RTAI tasks.

### **3.4 Linux 2.4 with LXRT**

LXRT is an extension of RTAI. It allows hard and soft real-time programs to run in user-space using the RTAI API. The advantages of working in user-space with LXRT include:

- tasks execute under Linux memory protection
- support for IPC calls with standard Linux processes
- user-space tasks can be debugged with standard debug tools

LXRT and RTAI work together to provide real-time performance to user-space tasks. RTAI kernel-space modules are still required for LXRT and any implementation will generally include LXRT tasks that run in user-space and RTAI tasks that run in kernel-space.

For this option, all hard real-time tasks were implemented in user-space using LXRT support.

## **4 Hardware Platform**

The GE Fanuc, VMIVME-7700 CPU board was selected for use in the control system. The board is a VMEbus single board computer based on the Intel Celeron CPU running at 650 MHz. The board is populated with 512 MB SDRAM and 128 MB CompactFlash. Other features of the board that were used in the evaluation of the real-time options include:

- 10/100 BaseT interface,
- video graphic controller,
- keyboard interface,
- remote Ethernet booting and
- programmable timer.

The Ethernet interface allowed the target to be connected to the host PC using a LAN. The host PC was an x86-based PC running the Fedora Core 2 Linux distribution. The target had its root filesystem NFS mounted to the host PC.

The video graphic controller and keyboard interface provided a convenient interface to monitor and control the target.

The remote Ethernet booting capabilities were used to allow the kernel image to be downloaded to the target from a tftp server running on the host PC.

The programmable timer provided a mechanism to obtain an indication of the latencies inherent in the various OS options considered. The driver for the timer was provided by GE Fanuc.

## 5 Measurements

An important measure of an operating system's ability to meet the real-time requirements for a control system is the length of time from the instant that the event marking the beginning of a control interval is generated to the time that the control task begins execution. We define this time as the preemption latency. As the preemption latency increases, the time available for the control calculations decreases. The preemption latency is shown in Figure 1.

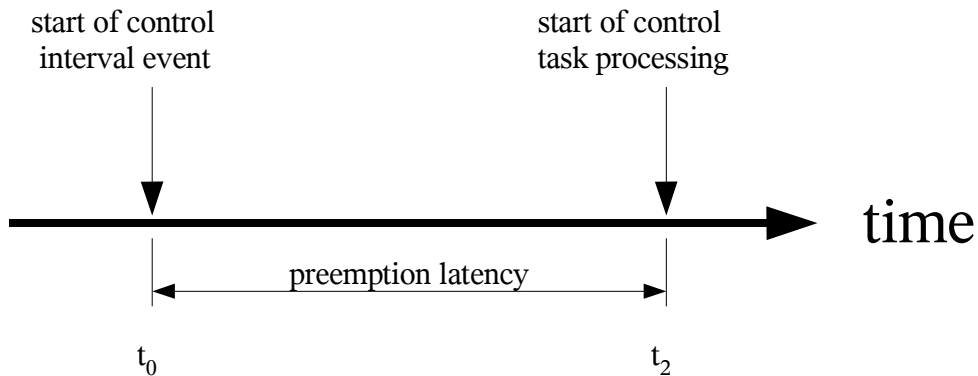


Figure 1 - Preemption Latency

The preemption latency is the sum of a number of shorter delays including: hardware response time, interrupt service time and context switch time. We chose to measure the preemption latency and the interrupt latency defined according to the more detailed Figure 2 shown below.

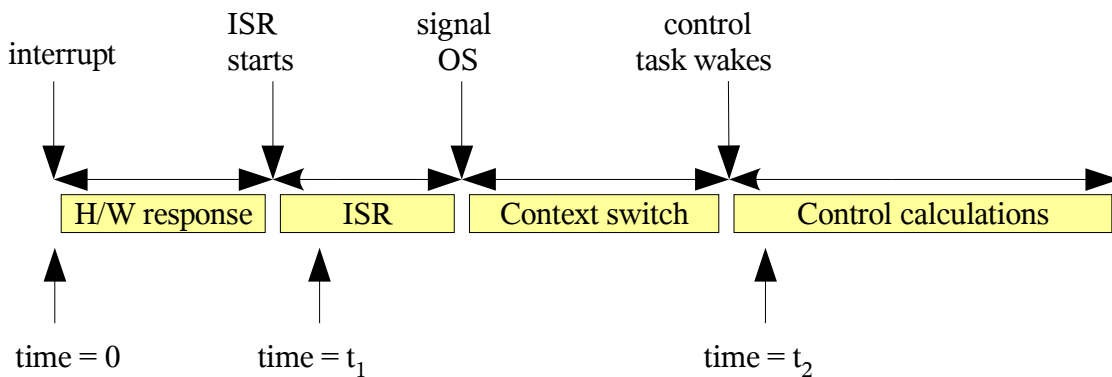


Figure 2 - Components of Latency

where:

- time = 0 is the instant that the start of control interval interrupt is generated
- time =  $t_1$  is the point at which we can first record the time during the interrupt service routine and
- time =  $t_2$  is the point at which we can first record the time during the control task

and we define:

- $t_1$  as the interrupt latency
- $t_2$  as the preemption latency

The interrupt signalling the start of the control interval was generated by one of the timers (timer 3) in the VMIC FPGA timer device. The timer is loaded with a count value that is decremented on each clock tick. When the timer count reaches zero, an interrupt is generated and the count value is reloaded. This allows the timer to generate periodic interrupts.

Time values were measured by reading the value of the timer counter. The difference between the initial counter value and the counter value at  $t_1$  or  $t_2$  indicate the number of clock ticks that have elapsed. Knowledge of the clock rate and the count provides all of the information required to measure time.

The timer counter was set up as follows:

- clock rate: 2 MHz
- count: 20,000

The set up above will cause the timer to generate an interrupt every 10 ms with a resolution of  $0.5 \mu\text{s}$  per count.

The counts recorded in the ISR and in the control task were read as early in each procedure as was possible. Although the measured counts include some time spent processing within the procedures, this time was short and the measurements were considered to be representative measurements of the actual interrupt latency and the preemption latency.

Time series of preemption latency and interrupt latency were recorded for the OS options indicated above (2.4, 2.6, 2.4 with RTAI and 2.4 with LXRT). Measurements were made for both the loaded condition and the unloaded condition. A total of 1,500,000 samples, acquired at a rate of 100 samples per second, were acquired during each of the 8 tests (4 OS options; loaded and unloaded). The samples were collected and a basic data reduction analysis was completed as part of each test.

The data reduction analysis consisted of counting the number of times the latency measurements fell into the range of a series of bins. For the preemption latency, a series of 2,500 bins were used with each bin spanning a latency of  $2 \mu\text{s}$ . This allowed the recording of latency measurements of up to  $5,000 \mu\text{s}$ . For the interrupt latency, a series of 2,000 bins were used with each bin spanning a latency of  $0.5 \mu\text{s}$  for a maximum latency measurement of  $1,000 \mu\text{s}$ . The final latency bin counts were stored in files for further analysis.

## 6 Test Set-up

The components of the test set-up included the development PC, the target hardware and the network connection between the PC and the target.

The development PC provided the following:

1. development platform with our Eclipse-based IDE
2. bootp, NFS and tftp servers
3. client application to provide network comms loading for a server running on the target

The target included the following:

1. Linux run-time environment including BusyBox
2. the necessary drivers for the hardware (those that are not part of the kernel)
3. RTAI and LXRT modules when required
4. software processes to acquire latency information and store the data in files
5. a server application to exchange large data packets with the development PC client

The network connection was a 100 BaseT internal LAN using a switch to connect the development PC and the target. The root filesystem for the target was NFS mounted to the development PC. This set-up is shown in Figure 3 below.

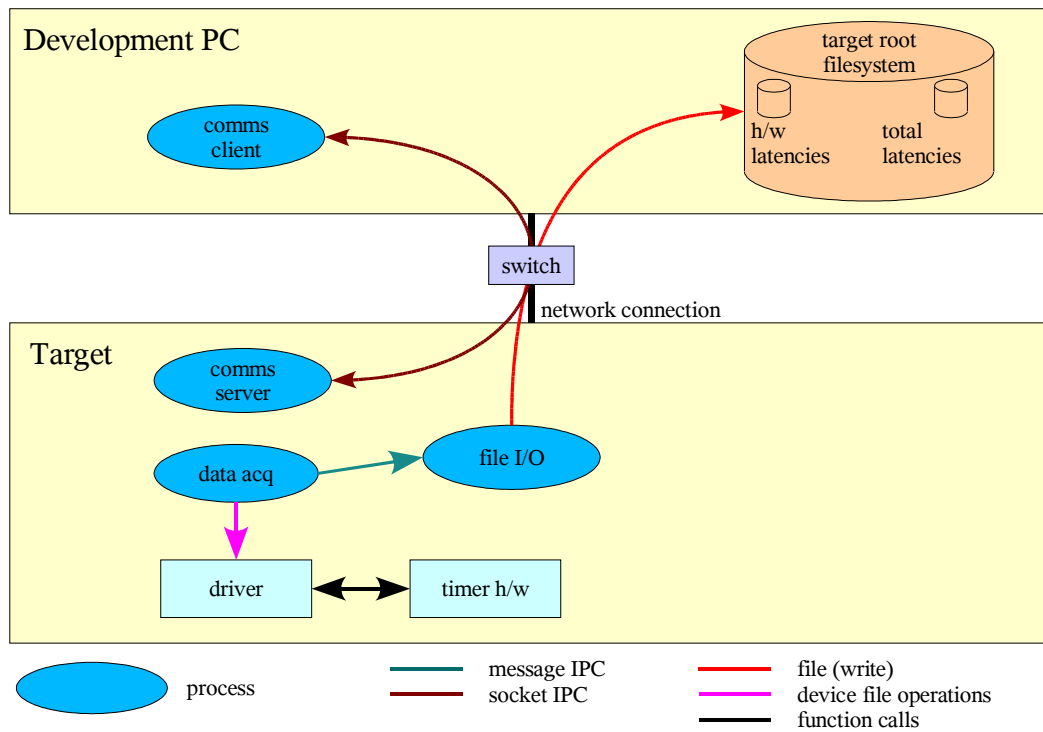


Figure 3 - Measurement Set-up

The set-up of the tests involved the following steps:

1. boot the target and load the desired kernel image
2. load the necessary kernel modules (drivers, RTAI, LXRT)
3. start the comms server on the target
4. start the comms client on the development PC
5. start the data acquisition process on the target
6. start the file I/O process on the target

Note that steps 3 and 4 are only required if the latency measurements are to be made with additional comms and processing loading on the target.

## 6.1 Process Description

**comms server and comms client:** The comms server and comms client provide communications and processing loading on the target. The comms server process waits for a message to arrive from the comms client, copies the message into an output buffer and sends the output buffer to the client. The messages are blocks of 60,000 bytes of data.

**data acq:** the data acquisition process collects latency information from the timer. The process waits for a signal to indicate the start of a control interval, reads the current counter value (to determine the preemption latency) and then requests the counter value stored by the driver interrupt handler. The two latency measurements are sent to the file I/O process.

**file I/O:** the file I/O process waits for messages to arrive at a mailbox. The latency values that are contained within the message are used to update arrays that keep track of the number of times latencies have been measured in the range of each measurement bin. Once the specified number of samples have been collected, the final latency bin counts are written to output files and the process terminates.

## 7 Detailed Process Models

The high-level process model shown in Figure 3 does not provide sufficient information to adequately describe the processes that run on the target. In some cases, these processes run in user-space and in other cases processes are run in kernel-space. A more detailed description of each of the cases is provided below.

### 7.1 Linux 2.4 and Linux 2.6

The process models for Linux 2.4 and Linux 2.6 are identical. This model is shown in Figure 4 and the components in the model are described in Table 1.

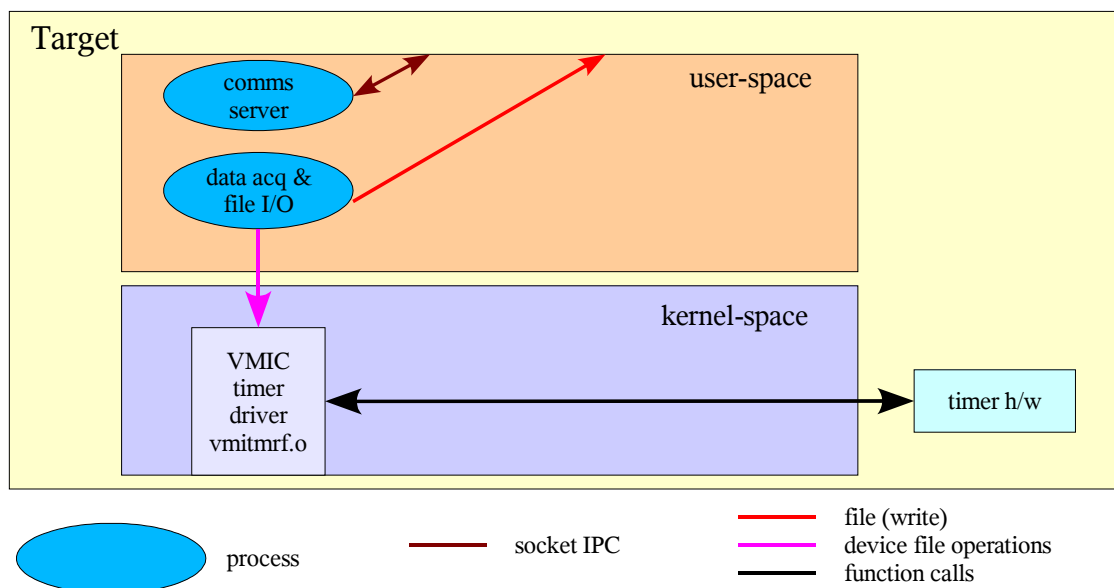


Figure 4 - Linux 2.4 and Linux 2.6 Process Model



| <i>Space</i> | <i>Component</i>    | <i>Type</i> | <i>Description</i>  |
|--------------|---------------------|-------------|---|
| User-Space   |                     |             |   |
|              | comms server        | process     | this is the same comms server described earlier. It accepts a large message from the client, copies the message into an output buffer and sends the message back to the comms client.                                 |
|              | data acq & file I/O | process     | this process sets up the timer and then collects the data from the driver. When all of the samples have been acquired, the process writes the summary data into output files and terminates.                          |
| Kernel-Space |                     |             |   |
|              | vmitmrf.o           | driver      | Timer driver module made available by GE Fanuc. The timer driver provides the file interface that can be accessed by user-space processes. The module is inserted into kernel-space using the BusyBox insmod utility. |

*Table 1 - Components of the Linux 2.4 and Linux 2.6 Model*

## **7.2 Linux 2.4 with RTAI**

For the Linux 2.4 with RTAI configuration, the data acquisition and file I/O process was split into a hard real-time process (data acquisition) and soft real-time process (file I/O). The data acquisition process was implemented as a kernel-space process using RTAI and the file I/O process was implemented as a user-space process. The file I/O process was coded as a soft real-time process that included calls to the RTAI API. This functionality was made possible by LXRT.

RTAI requires special driver modules if the driver is to be used by an RTAI task. In the case of this investigation, the VMIC timer driver was converted into an RTAI compatible driver. This conversion involved the following:

- creation of an RTAI interrupt handler and registering the handler with RTAI
- developing an internal mechanism to suspend tasks waiting for an interrupt to occur (we used semaphores for this)
- creating an API that other kernel modules can call to wait for a specific timer

The process model for Linux 2.4 with RTAI is shown in Figure 5 below. The components in the model are described in Table 2.

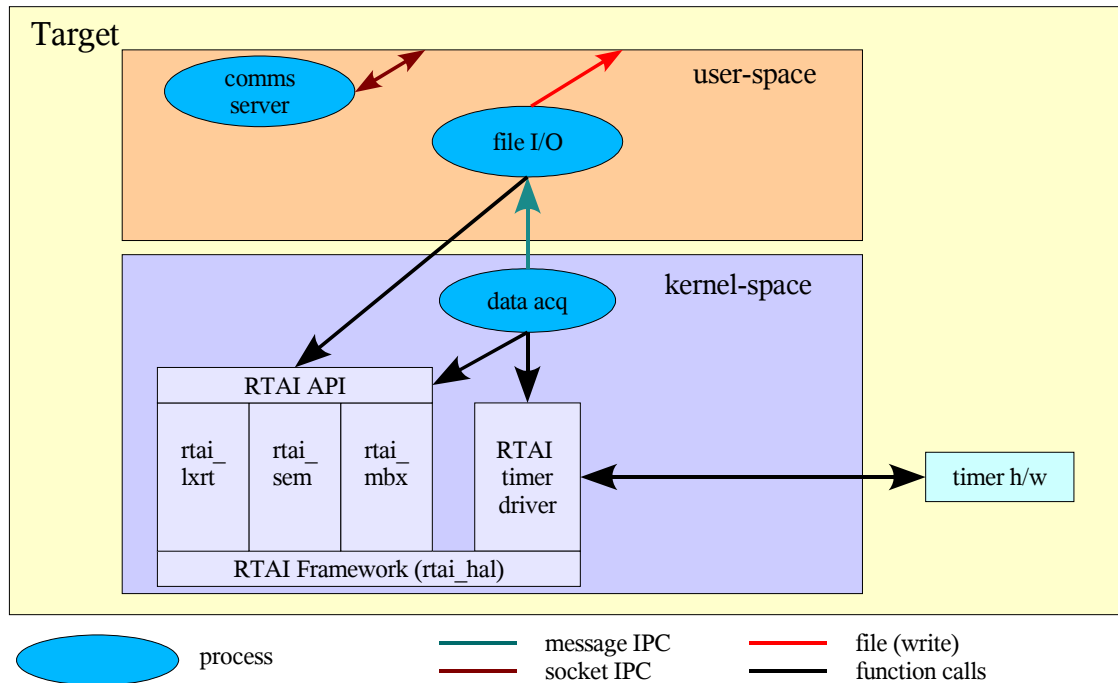


Figure 5 - Linux 2.4 with RTAI Process Model

| <i>Space</i> | <i>Component</i> | <i>Type</i> | <i>Description</i>  |
|--------------|------------------|-------------|---|
| User-Space   |                  |             |   |
|              | comms server     | process     | this is the same comms server described earlier.  |
|              | file I/O         | process     | this process waits on a mailbox for counter data to be sent to it from the data acquisition process. Non-blocking messages are sent at the control update rate of 100 Hz. When all of the samples have been acquired, the process writes the summary data into output files and terminates. The process is an LXRT soft real-time task. |
| Kernel-Space |                  |             |   |
|              | data acq         | task        | The hard real-time task that sets up the timer and then waits on signals from the timer driver. When a signal is received, the task reads the current timer count and the ISR timer count and sends them in a message to the mailbox used by the file I/O process.  |
|              | rtai_hal.o       | module      | Basic RTAI framework  |
|              | rtai_lxrt.o      | module      | Support for LXRT  |
|              | rtai_sem.o       | module      | Support for semaphores  |
|              | rtai_mbx.o       | module      | Support for mailboxes   |
|              | RTAI API         | API         | API supported by the 4 RTAI modules   |

| <i>Space</i> | <i>Component</i>  | <i>Type</i> | <i>Description</i>   |
|--------------|-------------------|-------------|--|
|              | RTAI timer driver | driver      | Modified driver to work with RTAI. Uses semaphores to create an API that other kernel modules can call to wait on a timer. |

Table 2 - Components of the Linux 2.4 with RTAI Model

### 7.3 Linux 2.4 with LXRT

The process model for Linux 2.4 with LXRT is very similar to that of Linux 2.4 with RTAI since both use LXRT for the file I/O user-space process and both rely on the same RTAI timer driver. The essential differences between the models are: the data acquisition process has been pulled into user-space and a new proxy module has been added to kernel space. The proxy module extends the timer driver API provided by the RTAI module to user-space. Without this proxy module, the data acquisition process would not be able to call the RTAI timer driver API.

The process model for Linux 2.4 with LXRT is shown in Figure 6 below. The figure shows that communication between the user-space data acquisition process and the kernel-space timer driver was through the Proxy API kernel module. We also tried using device file operations between the data acquisition process and the RTAI timer driver to set up the timer/counter and then using a semaphore to allow the data acquisition process to wait for a task switch. The latencies measured using the two different approaches were virtually identical. A description of the components in the model shown in Figure 6 are provided in Table 3.

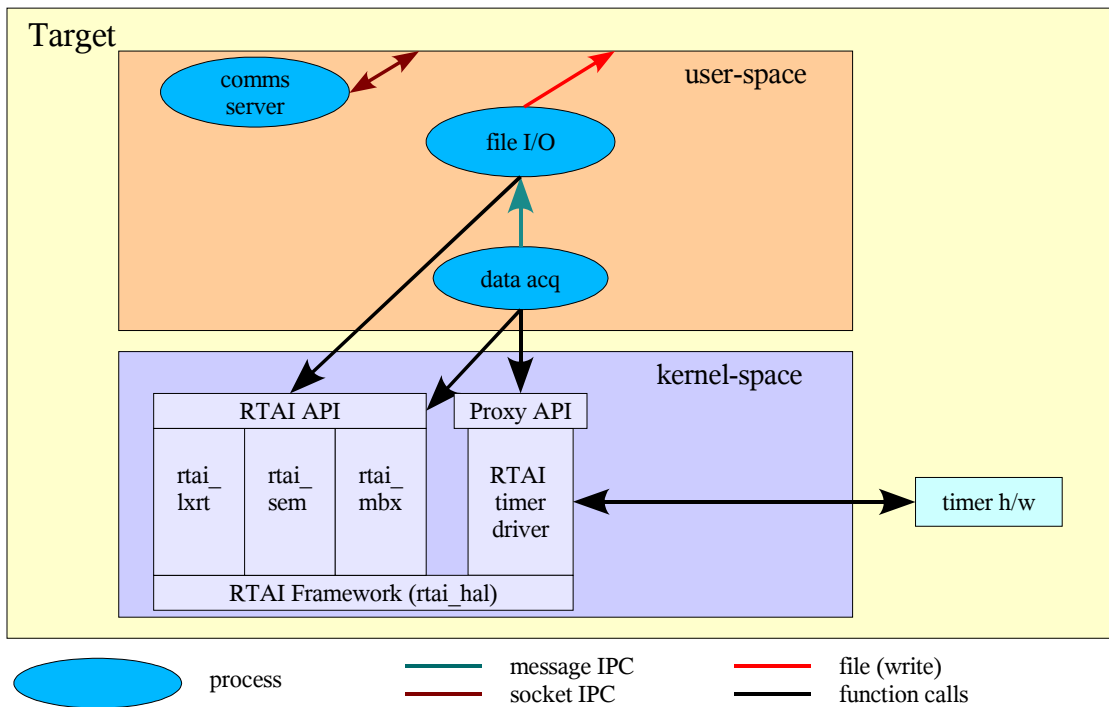


Figure 6 - Linux 2.4 with LXRT Process Model

| <i>Space</i> | <i>Component</i>  | <i>Type</i> | <i>Description</i>   |
|--------------|-------------------|-------------|--|
| User-Space   |                   |             |  |
|              | comms server      | process     | this is the same comms server described earlier.   |
|              | file I/O          | process     | this is the same file I/O process that was used for the Linux 2.4 with RTAI option.  |
|              | data acq          | task        | The hard real-time task that sets up the timer and then waits on signals from the timer driver. When a signal is received, the task reads the current timer count and the ISR timer count and sends them in a message to the mailbox used by the file I/O process. |
| Kernel-Space |                   |             |  |
|              | Proxy API         | module      | Proxy module that allows LXRT tasks in user-space to call the RTAI timer driver API  |
|              | rtai_hal.o        | module      | Basic RTAI framework   |
|              | rtai_lxrt.o       | module      | Support for LXRT   |
|              | rtai_sem.o        | module      | Support for semaphores   |
|              | rtai_mbx.o        | module      | Support for mailboxes  |
|              | RTAI API          | API         | API supported by the 4 RTAI modules  |
|              | RTAI timer driver | driver      | Modified driver to work with RTAI. Uses semaphores to create an API that other kernel modules can call to wait on a timer.   |

*Table 3 - Components of the Linux 2.4 with LXRT Model*

## 8 Measurement Results

The data showed the expected results; the vast majority of latencies measured were confined to a small range. The range for the preemption latency was wider than the range for the interrupt latency – also, as expected. For a hard real-time system it is important to understand the distribution of the maximum latencies.

To understand the distribution of the maximum latencies, we chose to look at curves of the cumulative percentage measurements vs latency. At any point on the cumulative percentage curve, the cumulative percentage value (y-value) is the percentage of measurements that had a latency less than or equal to the latency value (x-value). The latency at which the cumulative percentage curve reaches 100% represents the worst-case latency measured. The ideal cumulative percentage curve is one that is steep with a minimal decrease in slope as the curve approaches 100%.

The real-time operating system for a control system must be able to guarantee that the control calculations will be executed every control interval and that there must be sufficient time for the control calculations to complete before the control interval ends. For the system for which this analysis was completed, this means that all control calculations must start and end within a 10 ms window. Of the 10 ms available, the preemption latency between the start of the control interval (interrupt generated) to the start of the control calculations must be less than 0.5 ms (500  $\mu$ s) or 5% of the control interval.

### 8.1 Interrupt Latency

The interrupt latency curves are shown in Figures 7, 8 and 9. Figure 7 shows the loaded and unloaded interrupt latencies for all of the OS options evaluated. In general, the interrupt latencies for the tests without communications loading are to the left of the graph and are quite steep. The notable exception is Linux 2.6 where the curve flattened considerably, ultimately having higher latencies than some of the tests with communications loading.

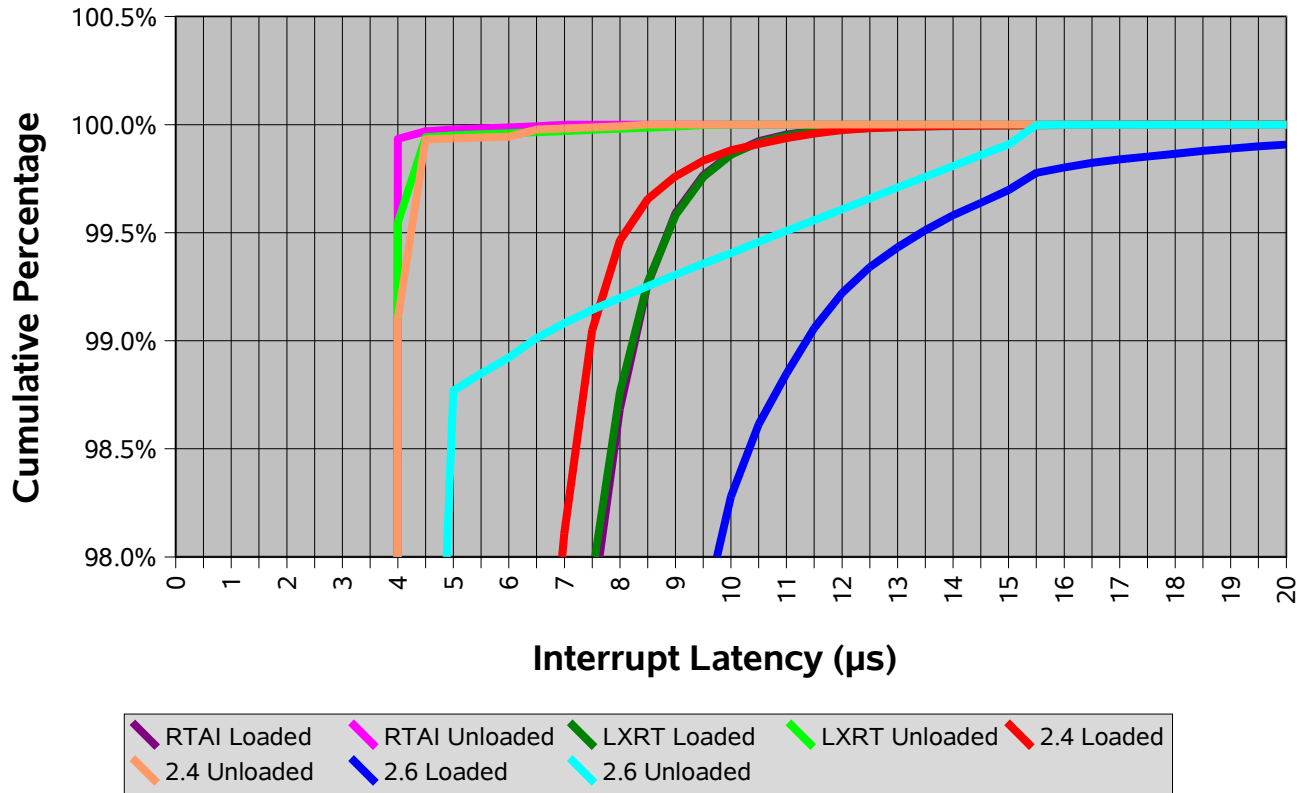


Figure 7 - Interrupt Latency Curves

Figure 8 shows the top 2% of the cumulative percentage curves for the unloaded tests and Figure 9 shows the top 2% for the loaded tests. Note that in Figures 8 and 9, the curves for Linux 2.4 with RTAI and for Linux 2.4 with LXRT are nearly identical. This is expected since the interrupt latencies were measured in the same RTAI timer driver module ISR.

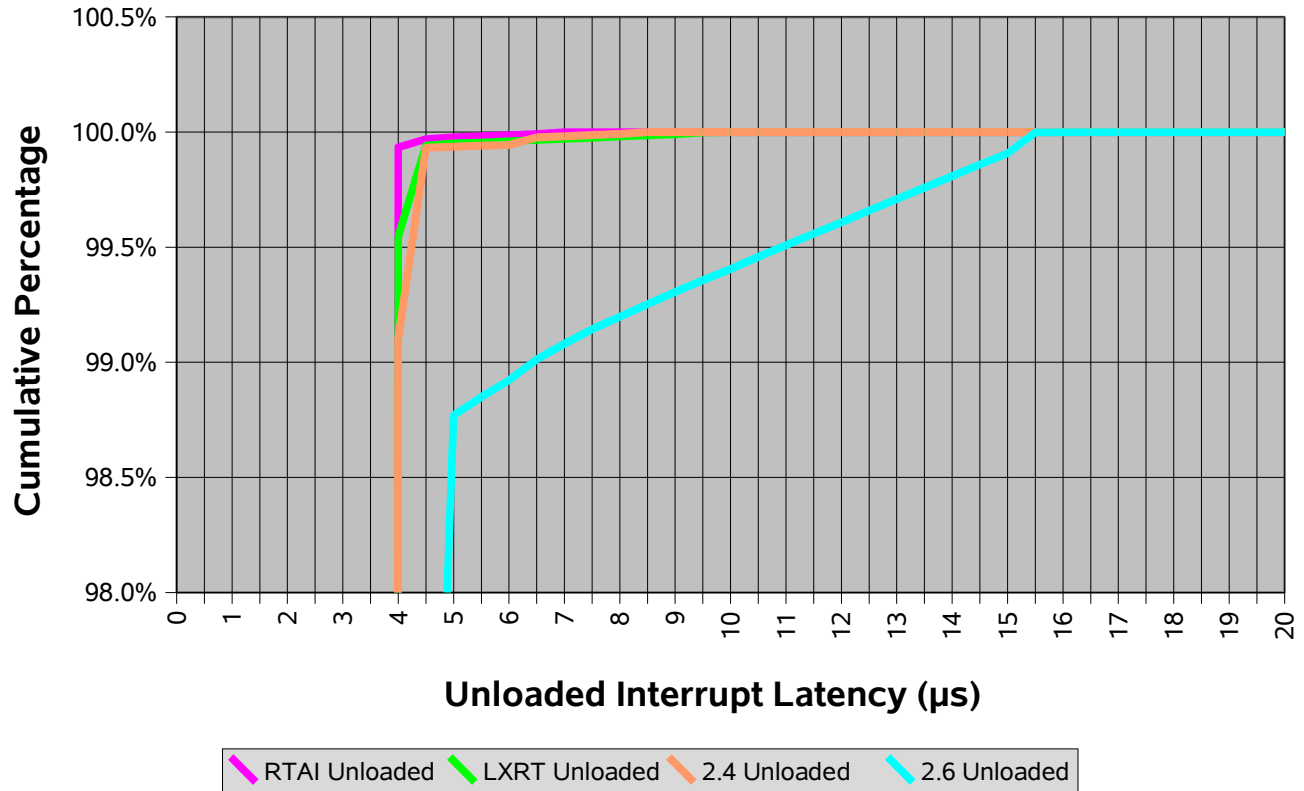


Figure 8 - Interrupt Latency Without Loading

Under loading (Figure 9), Linux 2.6 continued to exhibit higher latencies than the other options. The other 3 curves (Linux 2.4, 2.4 with RTAI and 2.4 with LXRT) are all similar. Even under loading, the interrupt latency curves for Linux 2.4 with RTAI and Linux 2.4 with LXRT were nearly identical. This too is to be expected since the interrupts are processed in kernel-space and the same RTAI timer driver module was used for both OS options. The difference between RTAI and LXRT was only evident in the preemption latency curves. It is interesting to note that the interrupt latencies with loading for Linux 2.4 were generally lower than the other options.

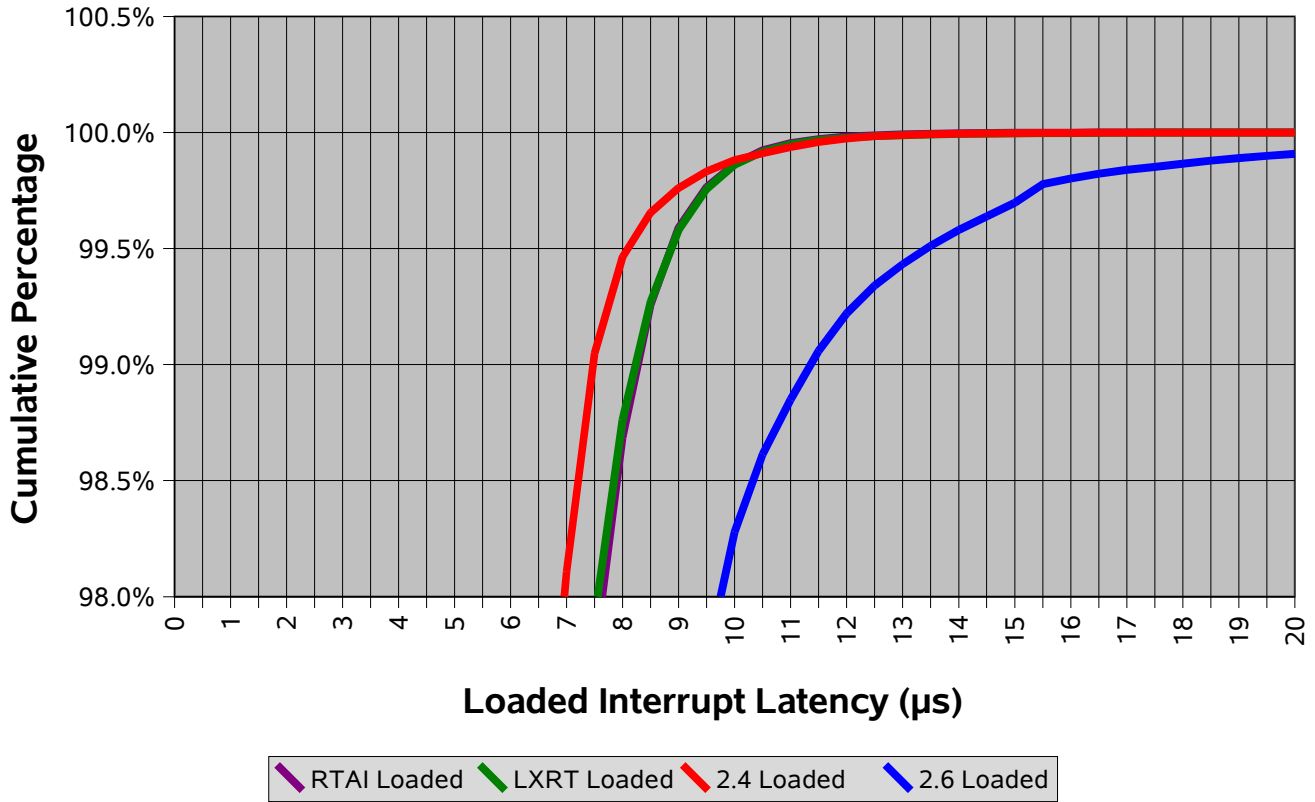


Figure 9 - Interrupt Latency With Loading

The interrupt latency information is summarised in Table 4. The maximum latency as well as the 99.999% latency threshold are shown in the table for all test configurations. The 99.999% latency threshold is the latency at which 99.999% of the measurements had latencies less than or equal to the threshold. At a sample rate of 100 samples per second, we would expect one latency measurement to exceed this threshold every 17 minutes.

| <b>OS Option</b>    | <b>Maximum Unloaded (µs)</b> | <b>Maximum Loaded (µs)</b> | <b>99.999% Threshold Unloaded (µs)</b> | <b>99.999% Threshold Loaded (µs)</b> |
|---------------------|------------------------------|----------------------------|--|--------------------------------------|
| 2.4 Linux           | 8.5                          | 113.5                      | 8.5                                    | 16.5                                 |
| 2.6 Linux           | 31.0                         | 49.5                       | 16.0                                   | 39.5                                 |
| 2.4 Linux with RTAI | 15.5                         | 16.5                       | 7.0                                    | 15.5                                 |
| 2.4 Linux with LXRT | 10.0                         | 20.0                       | 7.0                                    | 16.5                                 |

Table 4 - Interrupt Latencies Summary

## 8.2 Preemption Latency

The preemption latency curves are shown in Figures 10, 11 and 12. Figure 10 shows the loaded and unloaded preemption latencies for all OS options on the same graph. In general, the preemption latencies for the tests without communications loading are to the left of the chart and are quite steep.

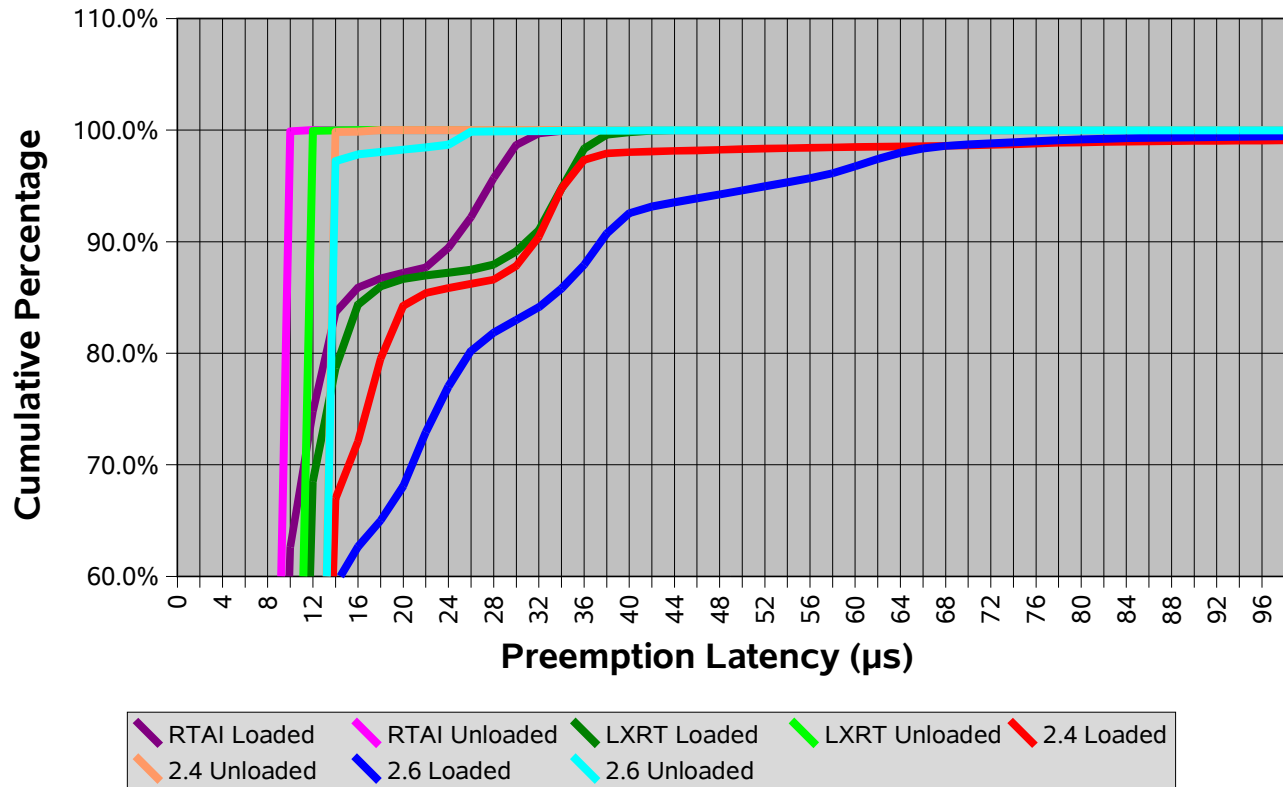


Figure 10 - Preemption Latency Curves

Figure 11 shows the top 3% of the cumulative percentage curves for the unloaded tests. The unloaded test curves for Linux 2.4, Linux 2.4 with RTAI and Linux 2.4 with LXRT are all very similar in shape. In each of these 3 cases, over 99% of the measurements were confined to a 2 μs range for the configuration. For Linux 2.4 with RTAI the range was 8 to 10 μs; for Linux 2.4 with LXRT the range was 10 to 12 μs; and for Linux 2.4 the range was 12 to 14 μs. Although some of the preemption latencies measured for the unloaded condition with Linux 2.6 were higher, over 97% of the measurements were in the same 12 to 14 μs range observed for Linux 2.4. For the 3% of the measurements where Linux 2.6 showed higher preemption latencies, these latencies were often 2 to 3 times that of both Linux 2.4.

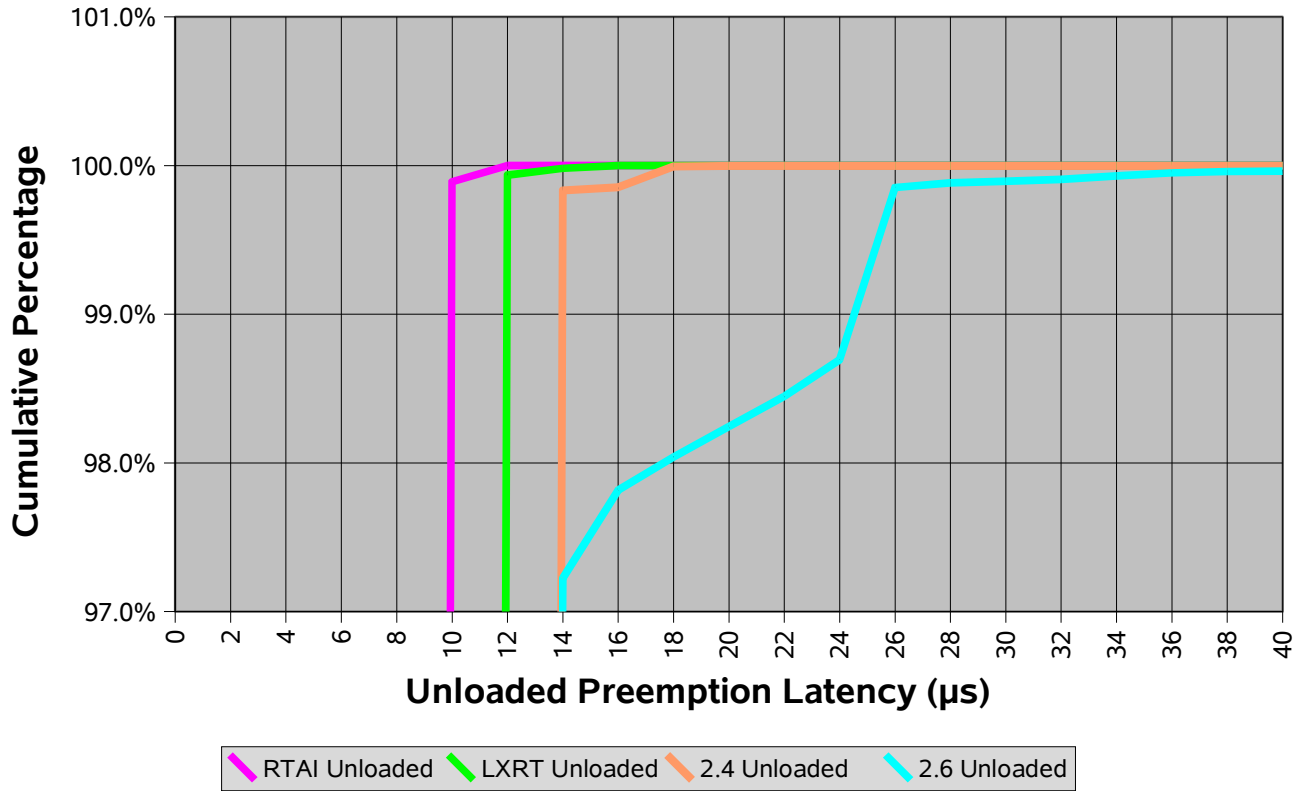


Figure 11 - Preemption Latency Without Loading

The difference between the interrupt latency and the preemption latency is largely the context switch time. The table below summarises the range of context switch times for the unloaded conditions for each of the 4 OS options. The table is based on the majority of the measurements (97%) where the latency times were within the range of single measurement bin (0.5 µs for the interrupt latency and 2 µs for the preemption latency). The data show that for more than 97% of the measurements, the context switch time for Linux 2.4 with RTAI is significantly faster than the other configurations.

| <i>OS Option (unloaded)</i> | <i>Interrupt Latency (µs)</i> | <i>Preemption Latency (µs)</i> | <i>Context Switch Times (µs)</i> |
|-----------------------------|-------------------------------|--------------------------------|----------------------------------|
| 2.4 Linux                   | 3.5 – 4.0                     | 12 – 14                        | 8.0 – 10.5                       |
| 2.6 Linux                   | 4.0 – 4.5                     | 12 – 14                        | 7.5 – 10                         |
| 2.4 Linux with RTAI         | 3.5 – 4.0                     | 8 – 10                         | 4.0 – 6.5                        |
| 2.4 Linux with LXRT         | 3.5 – 4.0                     | 10 – 12                        | 6.0 – 8.5                        |

Table 5 - Context Switch Times Without Loading (based on 97% of the data)

Figure 12 shows the top 20% of the latency measurements for the loaded tests. Under loading, the results showed that Linux 2.4 with RTAI was the best of the OS options. This curve had the desired characteristics of being very steep without flattening too much as the curve approached 100%. The curve for Linux 2.4

with LRTX was very similar to that of Linux 2.4 with RTAI but the latencies were higher for the top 12% of the measurements. The curve for Linux 2.4 showed that the majority of the latencies (97%) were quite comparable to those of RTAI and LXRT but the curve then flattened out considerably and approached 100% much more slowly. The results for Linux 2.6 are interesting in that its performance was better than Linux 2.4 for latencies above 70  $\mu$ s (top 1.3% of the measurements).

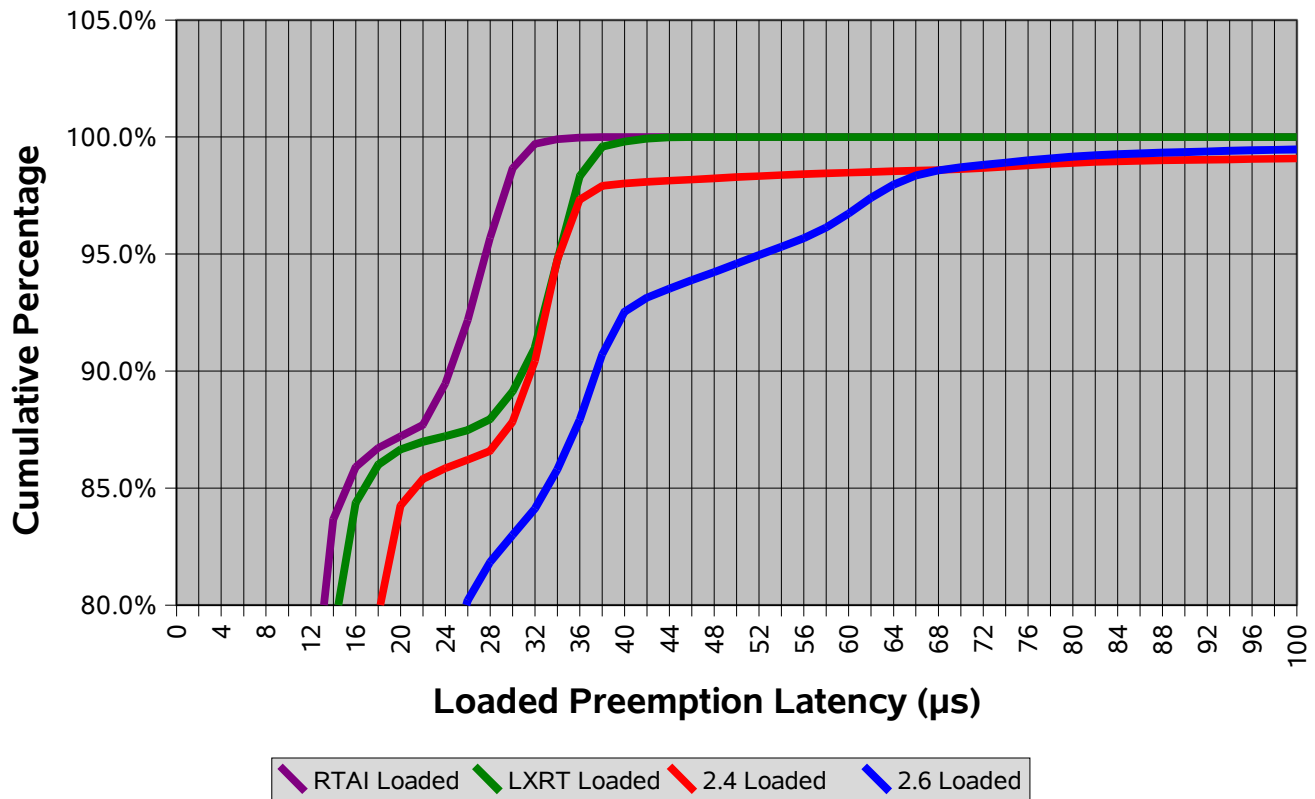


Figure 12 - Preemption Latency With Loading

Table 6 below summarises the preemption latency data collected. The maximum latency as well as the 99.999% latency threshold are shown in the table for all test configurations. The 99.999% latency threshold is the latency at which 99.999% of the measurements had latencies less than or equal to the threshold. At a sample rate of 100 samples per second, we would expect one latency measurement to exceed this threshold every 17 minutes.

| <i>OS Option</i>    | <i>Maximum Unloaded (<math>\mu</math>s)</i> | <i>Maximum Loaded (<math>\mu</math>s)</i> | <i>99.999% Threshold Unloaded (<math>\mu</math>s)</i> | <i>99.999% Threshold Loaded (<math>\mu</math>s)</i> |
|---------------------|---|---|---|---|
| 2.4 Linux           | 104   | 4446                                      | 48  | 760   |
| 2.6 Linux           | 412   | 578                                       | 396   | 440   |
| 2.4 Linux with RTAI | 32  | 42  | 12  | 40  |
| 2.4 Linux with LXRT | 26  | 50  | 16  | 48  |

*Table 6 - Preemption Latencies Summary*

### 8.3 Overall

The results show that among the options tested, the best performance was measured with RTAI followed closely by LXRT. For these two options, the measured interrupt and preemption latencies for both loaded and unloaded conditions were more consistent and shorter than the other options. Both Linux 2.4 with RTAI and Linux 2.4 with LXRT meet the maximum preemption latency requirement of less than 500  $\mu$ s. Linux 2.6 is the next most suitable option followed by Linux 2.4.

The data collected provided a single data series collected over 4 hours for each of the 8 cases. If these tests were repeated or if data were collected for a longer period of time, the maximum latencies would likely be higher. It is the small percentage of measurements where the latencies are much higher that drive the choice of an OS for use in systems requiring hard real-time performance.

## 9 Conclusions

Based on the latency measurements made using the hardware described in this paper:

1. Of the options considered, both Linux 2.4 with RTAI and Linux 2.4 with LXRT meet the stated latency requirements for a real-time 100-Hz control system
2. Only RTAI and LXRT provide what could be considered deterministic interrupt response times and task switch times 100% of the time
3. Linux 2.4 can not be used for hard real-time systems
4. If the maximum preemption latency were to be increased to 600  $\mu$ s, all of Linux 2.4 with RTAI, Linux 2.4 with LXRT and Linux 2.6 would meet the requirements

## 10 Resources

The following resources were used in the work described in this paper:

1. The Wonderful World of Linux 2.6 by Joseph Pranevich, 2003. <http://www.kniggit.net/wwol26.html>
2. Towards Linux 2.6 – A look into the workings of the next new kernel by Anand K Santhanam, , 23 Sep 2003. <http://www-106.ibm.com/developerworks/linux/library/l-inside.html>
3. HOWTO Port your C++ GNU/Linux application to RTAI/LXRT. <http://people.mech.kuleuven.ac.be/~psoetens/portingtolxrt.html>
4. Various RTAI pages on Capitan’s Universe (see <http://www.captain.at/programming/>)