THE FERMI@Elettra DISTRIBUTED REAL-TIME FRAMEWORK *

L. Pivetta, G. Gaio, R. Passuello, G. Scalamera, Sincrotrone Trieste, Trieste, Italy

Abstract

FERMI@Elettra is a Free Electron Laser (FEL) based on a 1.5 GeV linac. The pulsed operation of the accelerator and the necessity to characterize and control each electron bunch requires synchronous acquisition of the beam diagnostics together with the ability to drive actuators in real-time at the linac repetition rate. The Adeos/Xenomai real-time extensions have been adopted in order to add real-time capabilities to the Linux based control system computers running the Tango software. A software communication protocol based on Gigabit Ethernet and known as Network Reflective Memory (NRM) has been developed to implement a shared memory across the whole control system, allowing computers to communicate in real-time. The NRM architecture, the real-time performance and the integration in the control system are described.

INTRODUCTION

FERMI@Elettra is a light source based on a linear accelerator designed to operate two FEL undulator chains covering the wavelength range from about 100 nm to 4 nm. Designed to work at 50 Hz repetition rate, FERMI@Elettra requires the control system to be able to track and tag each shot with a unique timestamp called bunch number. Any relevant data must be tagged with the same timestamp eventually allowing to correlate machine parameters with the electron and photon beams characteristics. Moreover, a number of control loops are required to stabilize the main parameters of the beams. The control system must guarantee the mechanism, the performance and the reliability necessary to acquire all the relevant sensors, perform the feedback calculations and drive the actuators in real-time.

LINUX AND REAL-TIME

Linux, as a general purpose operating system, historically belongs to the server application field and typically privileges throughput over responsiveness. Thus, the *vanilla* linux kernel is not suitable to predict precisely the execution of a task, neither the scheduling time nor the real duration. Three different approaches have been tested to improve the real-time behaviour.

* The work was supported in part by the Italian Ministry of University and Research under grants FIRB-RBAP045JF2 and FIRB-RBAP06AWK3

Performance Tuning

A number of methods are availabile in the plain Operating System (OS) to tune the behavior and obtain a better timing of the tasks:

- Increase the scheduling priority via the `nice` command:
  `nice -20 ./executable`.
- Change scheduling policy programmatically:
```
sched.sched_priority = \
        sched_get_priority_max(SCHED_RR)-1;
sched_set_scheduler(0, SCHED_FIFO, &sched);.
```
- In multicore processors, reserve a core to a task via the `taskset` command:
  `taskset -c1 ./executable`.
- Avoid memory pages of the process to be swapped via the `mlockall` system call.

A quick and dirty solution could be rewriting some Interrupt Service Routine (ISR) to include the desired code. The ISR cannot be scheduled and preempts any other task, except other ISRs, assuring the real-time execution. Good candidate ISRs to be modified could be those related to:

- A *data availabile* interrupt request (IRQ): when data are availabile an interrupt is raised and the critical task code could run in the end of the ISR code.
- A timer IRQ: the critical code runs on a timer period.

It is even possible to reserve a core to just one IRQ.

This approach has nevertheless an important drawback: code running in the ISR is required to be short and fast and not all applications could fit.

Preemptible Kernel

A better solution relies on the adoption of a fully preemptible kernel based on the PREEMPT_RT patch [1] and availabile on the mainline since release 2.6.23. This patch reduces the kernel code protected by *big locks*, implements the priority inheritance mechanism for in-kernel spinlocks and semaphores, adds the possibility to perform deferred operations and enables the support for High Resolution Timers (HRT). This approach leads to a much more responsive kernel/system altough it cannot guarantee an unfailing respect of the deadlines: the system is still not real-time.

Real-time Subsystem

A real-time subsystem environment cooperating with the GNU/Linux kernel provides a pervasive hard real-time support to both kernel-space and user-space applications and allows for the best performances.

For FERMI@Elettra the Adeos/Xenomai [2] real-time subsystem has been adopted. It supports several processor architectures on embedded systems, such as ARM, Blackfin, NiosII, PowerPC, x86 and, of course, runs also on standard workstations and servers. To port and optimize the Adeos/Xenomai subsystem to the embedded platform selected for the FERMI@Elettra control system [3] some work has been outsourced [4].

A complete set of Application Programming Interfaces (API) to develope real-time applications in kernel/user space, such as Inter-Process Communication (IPC), synchronization, mailboxes, etc. are availabile.

To exploit all the performance of the real-time subsystem some care must be taken: standard GNU/Linux system calls should be avoided in the real-time application and the device drivers must be eventually patched in order to use the API availabile in the real-time domain.

REAL-TIME PERFORMANCE

A laboratory testbench, shown in Fig. 1, has been set up to measure the performance of a VME based system consisting of an Emerson MVME7100 CPU and a couple of digital I/O boards.

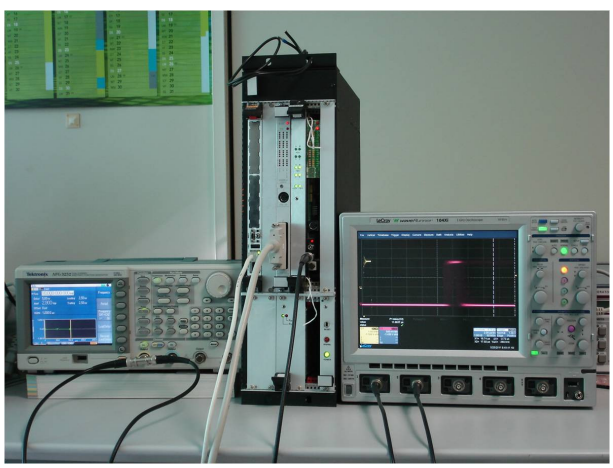

Figure 1: Real-time measurement testbench.

A pulse, referred below as *input pulse*, produced by a signal generator is acquired by a digital input board which rises an interrupt on the VME bus. The IRQ is then managed by the ISR and the digital line of an output board is driven producing an output pulse. The system has been loaded with tasks involving a large volume of interrupts: multiple *ping flood*, processes generating network activity and high-speed serial I/O traffic. Two scenarios have been analyzed.

Running the Code in the ISR

Some measurements have been carried out to characterize the performance of the described setup with the application code running inside the interrupt service routine of both GNU/Linux kernel domain and Adeos/Xenomai domain. A digital oscilloscope, triggered on the input pulse, measures the latency of the output pulse.

Table 1 and Table 2 show the results of the test for 50 Hz and 10 KHz input pulse repetiton rate respectively.

Table 1: Latency at 50 Hz Repetition Rate

	Min	Max	Mean	Std
Linux ISR	10μs	33μs	15μs	1.5μs
Xenomai ISR	11μs	23μs	16μs	1.4μs

Table 2: Latency at 10 KHz Repetition Rate

	Min	Max	Mean	Std
Linux ISR	10μs	29μs	11μs	0.8μs
Xenomai ISR	11μs	14μs	12μs	0.2μs

Measurements show that the worst case latency decreases when running at higher repetition rates, as the probability of the ISR code generating a *cache miss* decreases.

As already noticed, anyway, inserting the application code into the ISR is not good practice. The ISR must always be as fast and light as possbile.

Running the Code in an OS Task

Either the GNU/Linux kernel or the Adeos/Xenomai real-time scheduler have complete synchronization primitives that allow to execute a specific task triggered by an asynchronous event. In both cases the ISR code, servicing the IRQ, releases a semaphore that unlocks the execution of a suspended task.

A GNU/Linux task running in user space is blocked on a `ioctl` waiting on a semaphore. A GNU/Linux ISR, triggered by the IRQ generated by the digital input board, unlocks the semaphore. The GNU/Linux task generates a pulse driving a digital output line. An oscilloscope acquires both the input pulse and the output pulse. Table 3 shows the resulting measurements for a repetition rate of 10 Hz.

Table 3: GNU/Linux Task Latency Measurements

	Min	Max	Mean	Std
No load	15μs	48μs	23μs	2.5μs
Heavy load	196μs	37180μs	175μs	1374μs

As expected, the system shows outstanding performances without machine load. No guarantee,

even at 10 Hz repetiton rate, to meet the deadline when the system is heavily loaded.

In the Adeos/Xenomai case the IRQ is serviced by a Xenomai ISR. A Xenomai task is pending on a `ioctl` waiting on a Xenomai semaphore. The ISR releases the semaphore unlocking the task that drives the digital output line. The results are shown in Table 4.

Table 4: Adeos/Xenomai Task Latency Measurements

Rep. rate	Min	Max	Mean	Std
50Hz	17μs	69μs	42μs	5.7μs
10KHz	16μs	49μs	21μs	3.2μs

The real-time behaviour is guaranteed and is not affected by the laod on the system; the performance suffers a slight degradation with respect to the code running into the ISR but with a big advantage: the task runs in user space.

The Fig. 2 summarizes the measured overall latencies of the testbench. The GNU/Linux heavy load measurements are not shown.

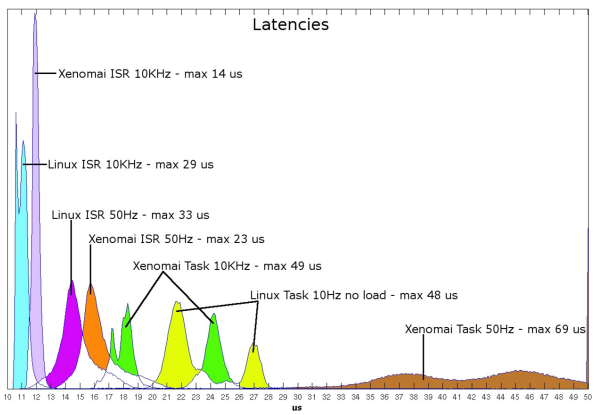

Figure 2: Overall system latency distributions.

HARDWARE

Two hardware platforms are currently supported and could be integrated into the real-time framework. VME crates, designed in accordance to the VME64x extensions, host the front-end computers; on this bus, double-edged source-synchronous transfer (2eSST) compliant boards can reach transfer rates up to 320 MB/s [5].

Emerson MVME7100 PowerPC single board computers (SBC) featuring 1.3 GHz CPU clock, 2 GB ECC RAM, 8 GB soldered flash disk, 4 Gb/s Ethernet interfaces, 5 RS232 serial lines, 2 user configurable hardware timers are used as main processors on the VME.

Intel-based processing servers have been adopted for the acquisition of digital cameras (CCD) based on Gigabit Ethernet, because of the closed-source binary-only proprietary support libraries. Two Xeon QuadCore 3.0 GHz processors, 4 GB of DDR3 RAM and up to six Gb/s Ethernet interfaces allow an effective acquisition and processing of up to three CCDs on each Gigabit link.

NETWORK REFLECTIVE MEMORY

A real-time framework tightly integrated in the control system has been implemented. Every control system cabinet is reached both by a standard control/supervisory network and by an Ethernet network dedicated to real-time. The standard Ethernet device driver has been customised in order to be able to transmit and receive raw Ethernet packets from the real-time domain. All the machines involved in real-time activities connect to this network using an additional Ethernet port.

To share the real-time data in a simple end effective way, a software infrastructure called Network Reflective Memory (NRM) has been developed. It consists of dedicated real-time device drivers and threads together with a dedicated API, availabile in both kernel space and user space, which implement the mechanisms of data-transparent memory shared among computers. Currently the device drivers of two families of Ethernet chipsets have been modified to support the NRM: the four channel Gianfar Ethernet Controller used on the MVME7100 and the Intel PRO/1000 used on the 1U rack-mount CCD acquisition servers [6].

The system topology is star shaped. At the centre a master system is in charge of collecting the packets coming from each slave and broadcasts them to all the stations. In order to guarantee the consistency of the shared memory the master broadcast is used as the trigger: upon reception, each slave syncronizes its local copy of the NRM with the data belonging to the master packet and at the same time starts sending its specific data to the master.

The NRM packets are handled in kernel space; the modified Ethernet device driver allows raw access to the interrupt handler and the transmission routines. Two First In Firts Out (FIFO) queues serve the write operations to the NRM coming from kernel space and user space respectively.

The raw packet is forged to comply to the link layer of the TCP/IP model; the header contains the MAC address of the sender and the destination, the timestamp, the sequence number and the CRC followed by the data. The data starts with a 4 bytes header containing the data type, the segment size, and the shared memory offset; the minimum payload size is therefore 4+4=8 bytes. To safeguard the availabile bandwidth shared among all stations, currently 1 Gb/s, the maximum packet size is fixed to 256 bytes for the slaves and could be increased up to the jumbo packet size for the master. Also, the address space of the NRM is limited to 1 MB.

Repetition rates up to 10 KHz are sustainable when the master and the slaves are all connected to the same network switch; otherwise, when the packets have to cross two switches the maximum repetition rate lowers to roughly 5 KHz.

An approximate calculation of the sustainable data traffic could be done considering the size of the master data packet and the usable repetiton rate; the maximum on a single Gbit Ethernet connection spanning at most two hops is roughly 9000 bytes × 5 KHz = 45 MB/s. Without using the jumbo packet payload and at the actual repetition rate, the NRM maximum data exchange rate through for the FERMI@Elettra control system is 1500 bytes × 3.3 KHz ≈ 5 MB/s.

The actual size of the NRM data currently in use is 12 KB at the Linac repetiton rate of 50 Hz, thus the data rate is 12 KB × 50 Hz = 600 KB/s. Moreover, with a NRM refresh rate of 3.3 KHz and a Linac repetition rate of 50 Hz the number of availabile NRM cycles between two shots, e.g. 20 ms, is 65. Considering the worst case of the data payload, 4 bytes data plus 4 bytes header, the number of NRM cycles used in the FERMI@Elettra control system is 12 KB / 1500 / 2 = 16 over 65, i.e. about 25% utilization.

USE CASES

A large number of devices are currently interfaced, i.e. acquired and/or driven, shot by shot:

- Electron/photon beam diagnostics: electron beam position monitors (BPM), photon beam position monitors (PHPBM), current monitors (CM), CCDs, bunch lenght monitors (BLM), beam arrival monitors (BAM), laser power meters, i0 monitors (photon counters), experimental station detectors.
- Power supplies: corrector power supplies, quadrupole power supplies.
- Radio frequency systems: linac low level RF.
- Machine protection systems: Cherenkov optic fibers, ionization chambers.

An example of a real-time FERMI@Elettra application is the Machine Protection System (MPS)[7][8]. An Equipment Controller [9] is in charge of monitoring the radiation levels along the undulator chain to avoid damaging the permanent magnets. For this purpose a number of Cherenkov optic fibers have been placed into appropriate grooves along the undulator magnets.

The waveform signals coming from the Cherenkov fibers front-ends are synchronously acquired shot by shot by a Caen V1720 VME digitizer at 250 MHz sampling rate.

A signal processing algorithm, running in real-time, analyzes the waveforms and, when over a predefined threshold, drives a digital output line requesting the switching off ot the electron beam to the MPS PLC.

Moreover, in order to monitor the operation of the software routines inside the VME system, a keep-alive signal with the repetition frequency of 50 Hz is transmitted from the VME to the PLC.

The same VME system is also in charge of the acquisition of the current monitors in real-time through a dedicated Ethernet network. Also in this case the algorithms to calculate the charge loss run in real-time on a shot by shot basis and the results are used to drive the inputs of the MPS PLC.

CONCLUSIONS

The Adeos/Xenomai realtime subsystem has been adopted to achieve hard real-time performances on GNU/Linux based systems running on PowerPC and x86 processor architectures. The measurements show that even loaded systems behave very well with the advantage of the programming API availabile in user space. The NRM has beed designed to share small amounts of data in real-time, i.e. with a known and reproducible latency, in a transparent and cheap way, leveraging the standard Gigabit Ethernet hardware. The hardware, the network topology, the software implementation of the data transmission make the NRM a good candidate for real-time distributed applications with repetition rates in the range of some hundreds of Hz and data size of tens of KB.

REFERENCES

[1] I. Molnar, `http://www.kernel.org/pub/linux/kernel/projects/rt`.

[2] `http://www.xenomai.org`.

[3] M.Lonza et al., "The control system of the FERMI@Elettra Free Electron Laser", ICALEPCS 2009, Kobe, Japan

[4] Denx Software Engineering, `http://www.denx.de`.

[5] `http://www.vita.com`.

[6] G.Gaio et al., "The FERMI@Elettra CCD image acquisition system", PCAPAC 2010, Saskatoon, Saskatchewan, Canada.

[7] F.Giacuzzo et al., "Equipment and Machine Protection Systems for the FERMI@Elettra FEL facility", these proceedings.

[8] L.Frölich et al., "Instrumentation for Machine Protection at FERMI@Elettra", DIPAC'11, Hamburg, Germany

[9] M.Lonza et al., "Status report of the FERMI@Elettra control system", these proceedings.