Low Latency Real-Time Computing on Multiprocessor Systems Running Standard Linux

Through proper configuration and adherence to guidelines, a multicore/multiprocessor standard Linux® kernel can run applications requiring low latency and fast interrupt response.

A system configured with SGI® React provides standard Linux processes with real-time response using a standard distribution kernel. No underlying real-time kernel required.

Latency Effects

Interrupt latency—time to process an interrupt and wake a sleeping thread.
– Time until handler invoked
– Time until application thread woken
– Time until application thread reaches userspace

Jitter—periods of cpu unavailability during thread execution.

Image courtesy of the Air Force Research Laboratory

Configuring the System

Once you have a distribution free from unexpected latencies installed, configure the system by isolating a subset of cpus from:

- Processes and threads not associated with the application
- Kernel threads when possible
- Load Balancing Effects
- Interrupts not associated with the application
- Unnecessary timers not associated with the application
- Effects of hyperthreading (turn hyperthreading off)

SGI's React configuration does this for you.
It also sets up cpusets to aid in easily attaching your threads to specific cpus.

Steps to achieving a low latency Linux OS installation

Install a distribution already suited to real-time work (such as an SGI React supported distro)
– The easiest way
– Vendor supported

OR

Work with the community and a vendor to test for latency issues and get fixes to those issues into their distribution
– Must check new release for new problems and address those
– Vendor support

OR

Test for latency issues and modify a kernel with your own patches for fixing latencies.
– Fixes likely need to be reapplied with each new release
– Must check new release for new problems and address those
– No vendor support

Configure a set of cpus for realtime use

Design and run your application following recommended guidelines

Running Application Threads

Pin to an isolated cpu (using 'taskset' or attach to appropriate cpuset).

Lock memory (mlock/mlockall).

Avoid spending time in the kernel.

Use shared memory for IPC.

Avoid locking when possible.

Follow rules for avoiding priority inversion if locking required.

Redirect application interrupt to same cpu running interrupt handling thread.

Best with single thread per isolated cpu, but multi-threading is fine if little or no time is spent in the kernel and priority inversion is strictly avoided.

Create user level drivers where possible. Facilities such as userspace PCI bus access, SGI ULI (User Level Interrupts), and SGI KBAR (Kernel Barriers) can aid in this.

SGI FRS (Frame Rate Scheduler) allows you to schedule threads periodically using an RTC clock or external interrupts as a sync trigger, again eliminating the need for kernel-level driver programming in certain instances.

With the above strategy, non-preemptive thread latencies can be kept within the 10's of usec range. Jitter will still be in the multi-usec range.

To further reduce jitter during periods of time critical processing, cpu timer interrupts can be switched off for short durations, allowing near jitter-free operation. The SGI provided SGI-shield API allows you to do this from program control.

Highest interrupt response times

4 CPU

system	without React	with React
noload	349.30	11.95
load	556.95	11.15

64 CPU

system	without React	with React
noload	484.95	10.50
load	545.35	18.75

Systems running sles10sp1 2.6.16.46-0.12 kernel with ProPack 5SP2 Itanium 2 processors 5,000,000 interrupts received per cpu
4 CPU system tests were done on 1000 MHz Madison B1 cpus with 1 cpu receiving interrupts. Load was 5-7 aim7 tasks.
64 CPU system tests were done on 1500 MHz Madison B1 cpus with 28 cpus receiving interrupts. Load was 20-23 aim7 tasks.

SGI poster
Sliced into 12 panels—each at 15" x 11"