Hard Real-Time Linux

(or: How to Get RT Performances Using Linux)

Andrea Bastoni

University of Rome “Tor Vergata”
System Programming Research Group
bastoni@sprg.uniroma2.it

Linux Kernel Hacking Free Course IV Edition

What is a “Real-Time” System

A general (informal and incomplete) definition:

- A real-time system should complete its work accordingly to precise temporal constraints

Is it enough?

What about the consequences of a malfunctioning?

What is a “Real-Time” System (Cont.)

- **Hard Real-Time**
- **Soft Real-Time**

Some possible criteria to draw their definitions:

1. Criticality of consequences of a failure
   (on both system and environment)
   - What is a generally acceptable definition of critical failure?
2. Usefulness of late work completion (job tardiness)
   - How to evaluate the usefulness of a late completion?
3. Probabilistic considerations
   - No accounting of possible consequences

What is a “Real-Time” System (Cont.)

An operational definition:

- Hard Real-Time: a *certification (formal proof, etc.)* is needed to show that deadlines will *always* be met.
- Soft Real-Time: such a *certification* is not needed: good *statistical averages* or *testing evidences* will generally suffice.

These definitions do not fully cover the complexity of the field though.

What is a “Real-Time” System (Cont.)

Some questions:

- What are *deadlines* and *who* provides them?
  And on which bases?
  - Generally settled starting from job *criticality*, output *usefulness* etc.
  - Risk assessment and deadline-based “countermeasures” defined in *Specification Documents*
- Is it so easy to prove (100%) a system to behave correctly?
  - Various methodologies: WCET Analysis, formal proofs, exhaustive testing (not always applicable)

Real-Time Applications Features

Distinctive features:

Real-Time side

- Predictability
- Reliability
- Performances (not always)

Embedded side

- Power awareness
- Compactness
- Scalability

Real-Time Operating Systems Features

A Real-Time Operating System (RTOS) should be able to offer the *right execution environment* to well suited RT Applications.

This means:

- *Predictable* and efficient *scheduling*
  - Fixed Priority Scheduling (e.g. Rate Monotonic)
  - Dynamic Priority Scheduling (e.g. Earliest Deadline First)
- *Predictable* interrupt handling and low-latency IRQ dispatching
- Task communication and synchronization support
- Resource allocation policies (Prio Inheritance, Prio Ceiling . . .)

Linux as RTOS

Is Linux a Real-Time Operating System?

Sometimes . . .

Linux as (S)RTOS

Linux is a *Soft Real-Time* Operating System:

It is optimized to provide:

- *Good average response time*
- *High throughput*

Suitable for:

- Multimedia Applications
- VoIP
- Video / Audio Streaming

Linux as RTOS

Main native features to support Real-Time:

- Two real-time scheduling policies
  - FIFO (`SCHED_FIFO`)
  - Round-Robin (`SCHED_RR`)
- Fast IRQ management (“two stage” handling)
- High responsiveness (*high resolution timers*, *1000Hz tick* . . .)
- Preemptive kernel

. . . and what about predictability?

Linux as (H)RTOS

Linux has “some” problems with predictability:

1. Several paths in the kernel *cannot be preempted*
2. *Interrupts are disabled* in critical sections of many handlers
3. *Non-predictable Interrupt Service Routines (ISR) management*: fast *ack* to devices, but most “bottom-half” handler’s durations are not predictable (e.g. Disk I/O)
4. IRQ management doesn’t consider priorities

Furthermore:

5. Quite high (w.r.t. HRT performances) scheduling latency for user mode processes

Linux as (H)RTOS

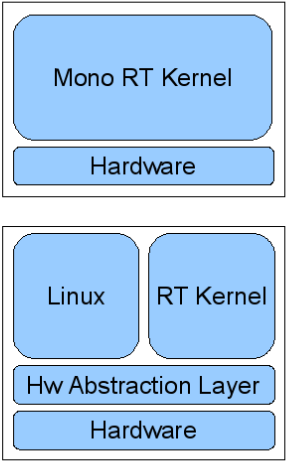
How to overcome these problems:

- *Mono Kernel Approach*
  - Changes are done directly into the kernel source
  - Porting should be done through kernel versions
  - Mostly commercial: *TimeSys Linux*, *MontaVista Linux* ...
- *Dual Kernel Approach*
  - Changes are done locally: simplified porting
  - New (and complex) intermediate layer between Hardware and OS
  - Mostly Open Source: *RTAI*, *RTLinuxFree - PaRTiKle*, *Xenomai*
  - But some commercial as well: *Wind River Real-Time Core Linux*

Dual Kernel Approach

Ideas:

- Insert an *Hardware Abstraction Layer* between HW and OSes
- Run Linux as a "normal" *low priority* process on top of a real-time scheduler



Dual Kernel Pros

Dual Kernel approach allows:

- Very Low latency IRQ response
- Predictable IRQ handling (response time can be bound from above)
- Reliable scheduling policy (FIFO, RR, RM, EDF)
- Fast task switching
- Resource allocation policies explicitly take into account tasks priority
- Ad-hoc synchronization mechanisms

Dual Kernel Cons

But:

- When calling standard library functions we may loose "real-time characteristics"
- So we must stay in kernel mode: calls to non real-time library functions are not allowed
- Drivers have to be suitable for "hard real-time"
- To exploit OS real-time performances we may have to use "non compliant" API (sometimes proprietary)
- Generally limited interprocess communication with Linux standard applications

What to do then?

- Positive aspects generally counterbalance negative ones
  - We can afford the extra programming effort and limitations
- In some cases though, dual kernel disadvantages are unacceptable
  - It would be great to (1) have a way to “do real-time” using the standard kernel and (2) to be able to obtain good performances staying in User Mode

Kernel's native support to Hard Real-Time

Some questions:

- What kernel's features can be exploited to get Hard Real-Time performances?
- How to cope with *load balancing* on SMP architectures
- Is there a way to increase kernel preemptability level?

...and some answers:

- Real-Time Scheduling Policies and *Scheduling Classes*
- CPU *affinity* (IRQs affinity, tasks affinity)
- Cpusets and `sched_domain`

Scheduler and Scheduling Classes

New ($\geq$ 2.6.23) scheduling approach:

- Mainly due to Ingo Molnar, working on Con Kolivas' "fair-scheduling" approach
- Introduction of *Completely Fair Scheduling* (for conventional processes), which models "an ideal, precise multi-tasking CPU"
- Introduction of *Scheduling Classes*:
  - *Hierarchy* of scheduler modules that *incapsulate* the details of their scheduling policy
  - Clean interface between the scheduler core and scheduler modules
  - Clear scheduler modules separation: one file per class (`sched_rt.c`, `sched_fair.c`, `sched_idletask.c`)

Scheduler and Scheduling Classes (cont.)

- The scheduler core can handle different classes (and different policies) without assuming too much about them
- It simply begins to walk the hierarchy from top and *delegates* tasks selection and management to classes
- It is easier to modify policies or introduce new ones
- `sched_rt.c`: `SCHED_FIFO` and `SCHED_RR` policies:
  - Highest prio module in the hierarchy
  - RT tasks management completely distinct from conventional processes one
  - Single runqueue with 100 priority levels
  - O(1) task selection bitmap-based

Scheduler and SMP management

Renewed SMP load-balancing

- Scheduler core relies on classes policies to choose which processes to move
- Selection of processes to move is done through iterators (provided by each class)
- Scheduler core is *unaware* of strategies chosen by classes to balance tasks
- Different classes may implement different strategies

CPU Affinity

The idea:

- Use affinity mechanisms provided in the kernel to *bind* a real-time task and *its relative interrupts* on a CPU
- Prevent other tasks and IRQs to be executed on that CPU

This path has been already followed in *ASymmetric MultiProcessor-Linux* which allows to obtain:

1. Deterministic execution time
2. Low system overhead
3. High performances and responsiveness

CPU Affinity and Cpusets

How to assign IRQs and tasks to CPUs:

- Binding interrupts on a CPU(s) can be easily done by using the *procfs*
- `sched_setaffinity` syscall can be used to bind a task to a CPU

Cpusets offer more flexibility:

- Cpuset provides a mechanism to *associate* a set of CPUs (and of Memory Nodes) with a set of tasks
- All task's children are automatically executed in the same *set* of their parent

Cpusets and Real-Time

- A cpuset defines a *scheduling domain* which covers all the CPUs included in the cpuset
- Load Balancing is done *only* inside a `sched_domain`
- Cpusets allow to easily control tasks distribution (and isolation) on system CPUs
- Cpusets can be effectively used to define a partitioning of system CPUs. This is often the first step of several multiprocessor real-time scheduling policies
- If used together with IRQ affinity we can enforce real-time tasks isolation w.r.t. other non-real-time tasks in the system

Still unanswered questions

Up to this point we have seen how to:

- Obtain RT scheduling policies through Linux Scheduler
- Use IRQ affinity and Cpusets to get CPU assignment determinism on multiprocessor architecture

But we still don't have a way to:

- *Preempt* the kernel in most critical paths
- *Assign priority* to IRQ handlers

The Real-Time Patch

The Real-Time Preemption Patch allows to cope with these problems:

- The patch is the continuation of the *Montavista*[1] real-time preemptive patch, mainly due to Ingo Molnar
- From Kernel 2.6.23 is completely integrated into mainline kernel projects

The patch enables:

- “Full” kernel preemption: non-preemptive kernel paths are reduced to less than 5%
  - Substitution of almost all spinlocks with semaphore locking mechanisms (preemptable mutexes)

[1] http://source.mvista.com/linux_2_6_RT.html

The Real-Time Patch (Cont.)

Further modification of RT scheduler load-balancing:

- Load-accounting and load-balancing are optimized for real-time tasks
- Balancing decisions are taken *only* before or after a context switch (“inside” `schedule()`)
- Try to keep runqueues from being overloaded:
  - Attempt to place all topmost priority real-time tasks on different CPUs
- When a high priority task wakes up (and it would preempt the currently executing one), check if it can run on a less loaded CPU

The Real-Time Patch (Cont.)

Threading of IRQ handlers:

- Both *soft* and *hard* interrupts handlers are threaded
  - All softirqs and all (or selected) hardirqs run in separate kernel threads
- We can control the priority assigned to an IRQ handler
- Higher priority handlers will be executed before lower priority ones

Improved synchronization mechanisms:

- Real-Time Mutexes with priority inheritance extends Priority Inheritance Mutexes (PI-futexes)
  - Used in `pthread_mutex` with prio inheritance implementation

Can we do better?

Recall high level of modularity of scheduling classes:

- Each class is placed in an hierarchy of classes; the real-time class is the topmost priority class and its tasks are evaluated first
- We can modify the RT scheduling policy:
  - Completely disable load-balancing heuristics
  - . . . speeding-up scheduler execution
- To introduce such a modification we “just” have to:
  - Implement the new scheduling class
  - Hook the scheduling class functions in the scheduler core through the `struct sched_class` structure

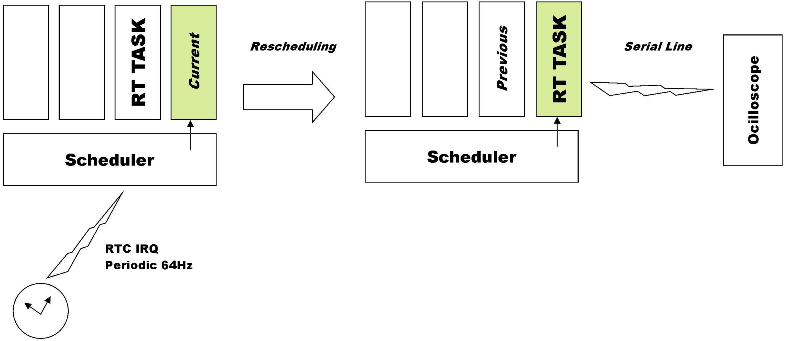
Some results (HRT)

I have talked a lot... but what about performance measures?

Hard Real-Time

- Platform: SBC Concurrent Technologies 417/03x; Intel Core 2 Duo T7400, 4GB RAM, Sata HDD
- OS: Linux 2.6.23.9 with RT patch and new real-time scheduling class which disables load balancing among processors
- We measure the period jitter (Standard Deviation) in the execution of a periodic real-time task
- Task periodicity is obtained by reprogramming the RTC (so that it ticks every $\approx 64Hz$)

Some results (HRT)



System was configured using cpusets and giving higher priority to RTC IRQ handler

Some results (HRT)

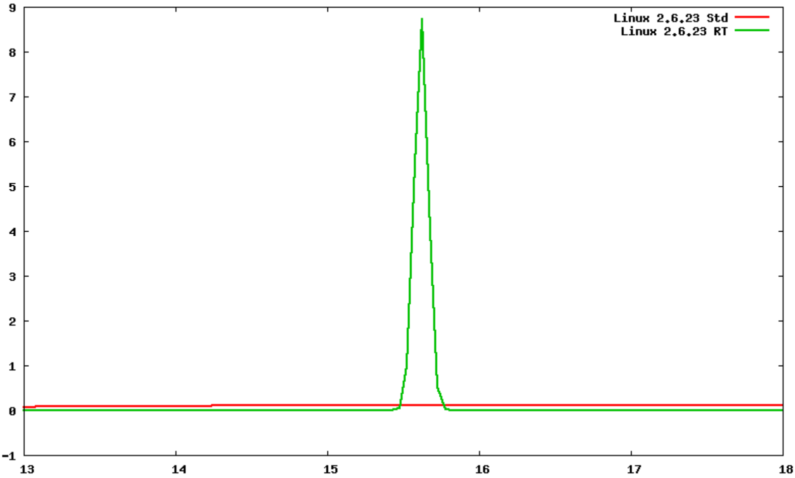
- A comparison between performances obtained using standard Linux kernel and kernel with RT patch, cpusets and IRQ prioritization.
- “Load” is composed by a mixture of different loads (CPU, memory and disk)

	NO LOAD			
	min(ms)	max(ms)	mean(ms)	StdDev(μs)
Linux 2.6.23 Std	15.59	15.71	15.63	46.30
Linux 2.6.23 RT	15.59	15.71	15.63	46.18

	LOAD			
	min(ms)	max(ms)	mean(ms)	StdDev(μs)
Linux 2.6.23 Std	1.04	33.16	16.11	**3310**
Linux 2.6.23 RT	15.59	15.71	15.62	45.21

Some results (HRT)

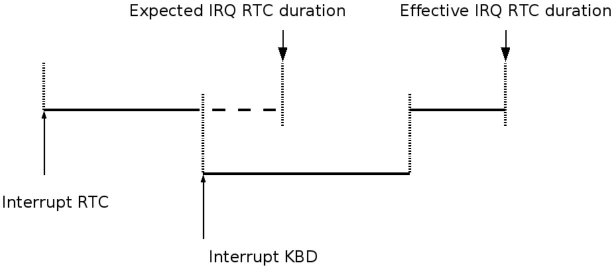
Linux 2.6.23 Standard vs Linux 2.6.23 RT (Load - NoLoad)



Some results (HRT)

IRQ threading and priority assignment to IRQ handlers:

- How to create a repeatable experiment to effectively verify IRQ prioritarization?
- Modify the IRQ handler of a popular device (e.g. Keyboard i8042) so that a single execution of the handler will last for a sensible amount of time



Some results (HRT)

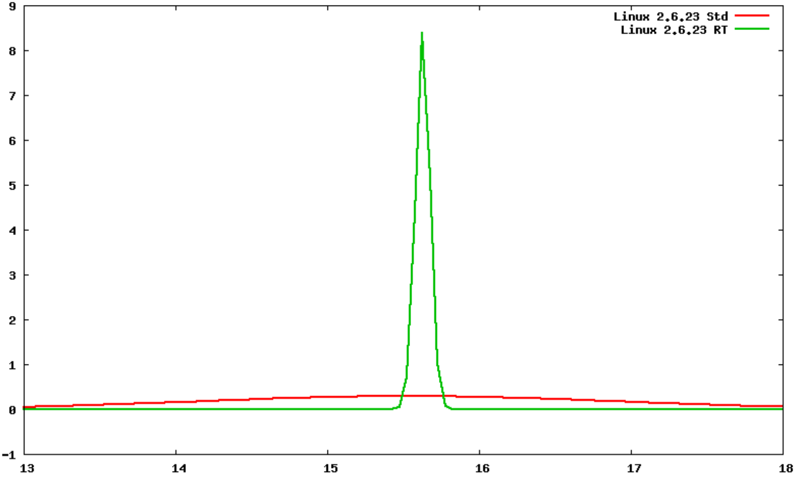
- Quantitative performances:

	NO Interrupts			
	min(ms)	max(ms)	mean(ms)	StdDev(μs)
Linux 2.6.23 Std	15.59	15.71	15.63	48.69
Linux 2.6.23 RT	15.59	15.71	15.62	45.44

	KBD interrupts			
	min(ms)	max(ms)	mean(ms)	StdDev(μs)
Linux 2.6.23 Std	7.280	19.86	15.43	**1368**
Linux 2.6.23 RT	15.59	15.71	15.63	47.44

Some results (HRT)

Linux 2.6.23 Standard vs Linux 2.6.23 RT (KBD test)



Some Results (SRT)

Soft Real-Time

- *Idea*: Test soft real-time kernel features in the typical application context of *Audio Streaming*
- We selected *VideoLan VLC* for both streaming server and clients
- Server offers 70 Audio Streams which are asked by clients using Real-Time Streaming Protocol (RTSP)
  - Stream transfer is done via RTP
- We measure the interarrival frame jitter (RFC 3550) relatively to frames in the same stream
- We remove head and tail jitter data and we focus on the central part of each audio stream transfer

Some Results (SRT)

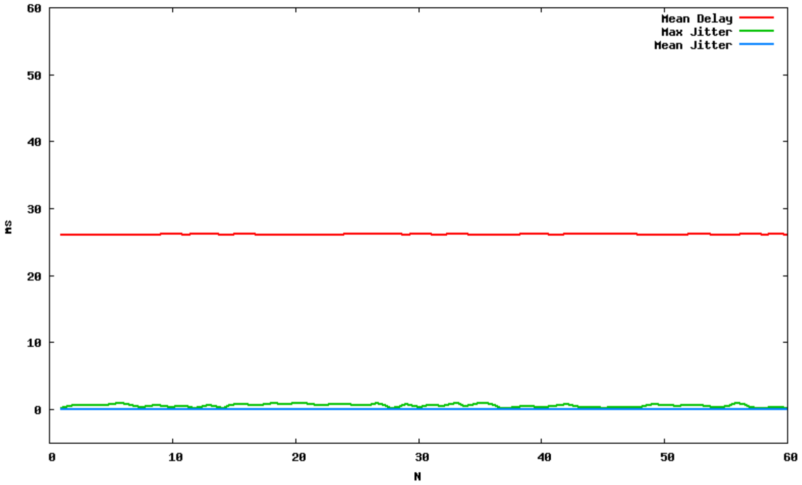
- Server: AMD Athlon 64 X2 Dual-Core 4000+ (2.1GHz), 1 GB RAM, Sata HDD, Slamd64, Linux 2.6.24.3
- Clients: Dual-Core AMD Opteron 8212 (4 Dual-Core processors, 2GHz each), 16 GB Ram, Sata HDD, Slamd64, Linux 2.6.24.3
- Gigabit Ethernet connection link
- One client CPU (two cores) is reserved to network traffic sniffing, while all the other CPUs are dedicated to VLC clients

Some Results (SRT)

- Two server configurations:
  - Normal: load-balancing is allowed
  - Cpusets: load-balancing is disabled by appropriate tasks partitioning
- Two load scenarios:
  - Light load (“only” the streaming server)
  - Heavy load (CPU an disk load plus streaming server load)

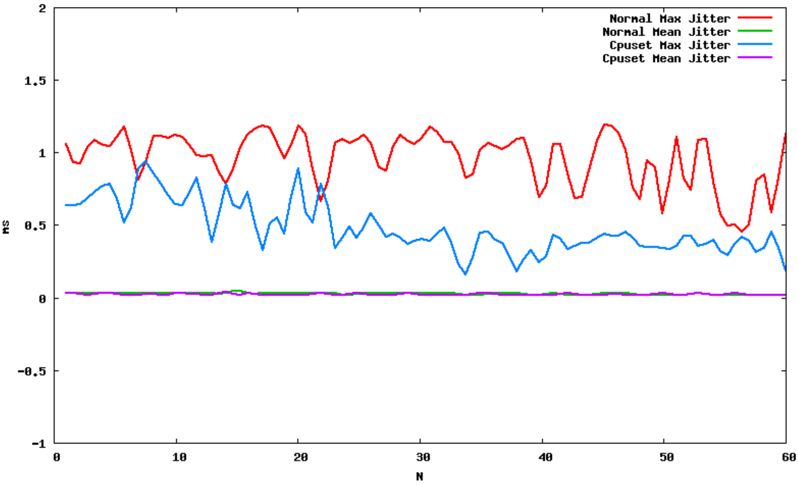
Some Results (SRT)

Light load, Normal configuration



Some Results (SRT)

Heavy load, Normal configuration vs. Cpuset configuration



Conclusions

- The times when speaking about Real-Time Linux was a scary topic are over. . .
- Linux is already a very good *soft real-time* system. . .
- Linux in its real-time variants (Mono / Dual Kernel) is able to provide predictable and reliable hard real-time performances
- In its rapid evolution, Linux is moving towards a good yet flexible *hard real-time* support
- Of course, the road to strong hard real-time performances or to certification is long and winding. . .