Schedulable Device Drivers: Implementation and Experimental Results [1]

Nicola Manica, Luca Abeni, Luigi Palopoli,
University of Trento - Trento (Italy),
Dario Faggioli
ReTiS Lab, Scuola Superiore Sant'Anna - Pisa (Italy),
Claudio Scordino,
Evidence Srl - Pisa (Italy)

6, July 2010

[1] This work has been partially supported by European Commission under the ACTORS project (FP7-ICT-216586) and by the "Provincia Autonoma di Trento" under the PAT/CRS Project RoSE

Motivation for This Work

Real-time applications needs guaranteed CPU time.

If they access hardware devices, properly serving hardware interrupt requests (*IRQs*) is equally important:

- device drivers must have some CPU time to execute;
- device drivers must not steal CPU time to real-time tasks;
- device drivers must run with correct timing to serve hardware requests.

Contribution of This Work

We use Linux PREEMPT_RT where device drivers (ISRs and bottom halves) are treated as tasks (*IRQ threads*) so that:

- their execution is not accounted to any other task,
- they can preempt or be preempted by other tasks.

We show how to use the SCHED_DEADLINE policy to:

- schedule real-time tasks,
- schedule IRQ threads,
- schedule task sets composed by real-time tasks **and** IRQ threads.

We show ho to gather statistical information about interrupt requests and how to assign proper scheduling parameters to the IRQ threads.

Contribution of This Work

We use Linux PREEMPT_RT where device drivers (ISRs and bottom halves) are treated as tasks (*IRQ threads*) so that:

- their execution is not accounted to any other task,
- they can preempt or be preempted by other tasks.

We show how to use the SCHED_DEADLINE policy to:

- schedule real-time tasks,
- schedule IRQ threads,
- schedule task sets composed by real-time tasks **and** IRQ threads.

We show ho to gather statistical information about interrupt requests and how to assign proper scheduling parameters to the IRQ threads.

Contribution of This Work

We use Linux `PREEMPT_RT` where device drivers (ISRs and bottom halves) are treated as tasks (*IRQ threads*) so that:

- their execution is not accounted to any other task,
- they can preempt or be preempted by other tasks.

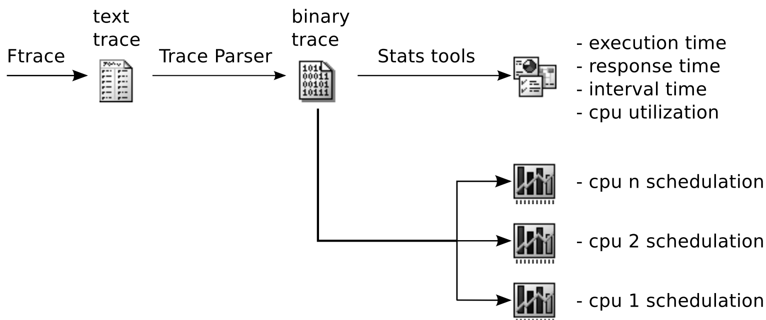
We show how to use the `SCHED_DEADLINE` policy to:

- schedule real-time tasks,
- schedule IRQ threads,
- schedule task sets composed by real-time tasks **and** IRQ threads.

We show ho to gather statistical information about interrupt requests and how to assign proper scheduling parameters to the IRQ threads.

The Tracing Pipeline

Statistical characterization of *inter-arrival* and *processing* time of IRQ handling can be extracted from the output of the Linux scheduling tracer (`ftrace`) by our pipeline of tools:



Resource Reservation scheduling with SCHED_DEADLINE

Scheduling class based on EDF+CBS [2]:

- tasks are reserved Q (*budget*) every T time units (*period*), and have $\frac{Q}{P}$ bandwidth (or utilization);
- no restrictive assumption on the characteristics of the tasks (periodic, sporadic or aperiodic).

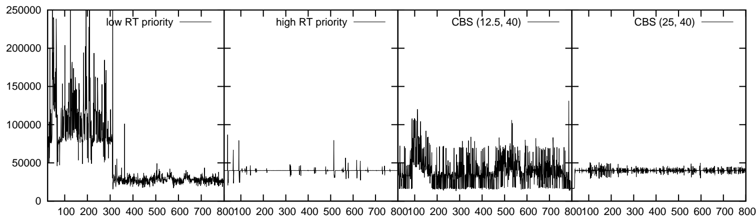
[2] http://gitorious.org/sched_deadline

Resource Reservation scheduling with SCHED_DEADLINE

Scheduling class based on EDF+CBS [2]:

- tasks are reserved Q (*budget*) every T time units (*period*), and have $\frac{Q}{P}$ bandwidth (or utilization);
- no restrictive assumption on the characteristics of the tasks (periodic, sporadic or aperiodic).

Example: inter-frame times for two video players executing under SCHED_FIFO or SCHED_DEADLINE:



[2] http://gitorious.org/sched_deadline

Resource Reservation and IRQ Scheduling

If an IRQ thread is limited to execute for Q CPU time units every T, and if P is the minimum inter-interrupt time for some device, at most $\frac{T-Q}{P}$ interrupts will be delayed due to IRQ thread not being allowed to run.

This means that, when selecting the reservation period for an IRQ thread, we must respect

$$\frac{T-Q}{P} < N_c$$

where N_c is the number of IRQs that can be buffered inside the hardware device.

Resource Reservation and IRQ Scheduling

If an IRQ thread is limited to execute for Q CPU time units every T, and if P is the minimum inter-interrupt time for some device, at most $\frac{T-Q}{P}$ interrupts will be delayed due to IRQ thread not being allowed to run.

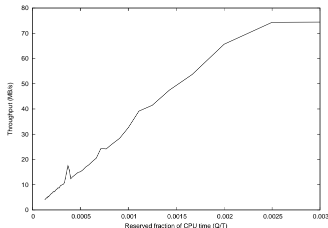
This means that, when selecting the reservation period for an IRQ thread, we must respect

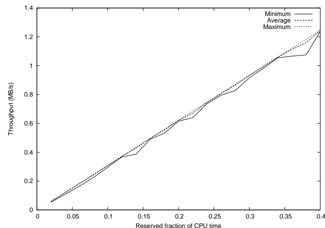
$$\frac{T-Q}{P} < N_c$$

where N_c is the number of IRQs that can be buffered inside the hardware device.

Controlling I/O Throughput

Disk throughput figures when using different reservation for the disk IRQ thread. OS caching mechanisms are disabled, DMA is enabled on the left, disabled on the right.



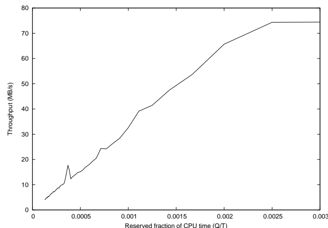


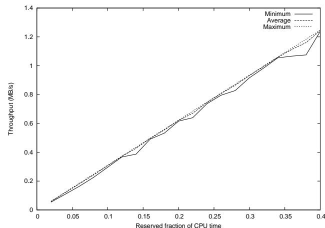
What can we say?

- DMA makes it possible to achieve much higher throughput but shows some more unpredictability.
- disk throughput is proportional to IRQ thread's bandwidth $\frac{Q}{T}$, both with and without DMA!

Controlling I/O Throughput

Disk throughput figures when using different reservation for the disk IRQ thread. OS caching mechanisms are disabled, DMA is enabled on the left, disabled on the right.





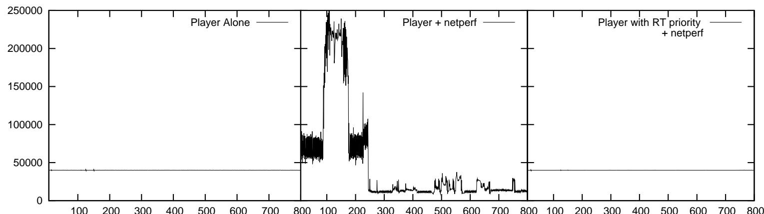
What can we say?

- DMA makes it possible to achieve much higher throughput but shows some more unpredictability.
- disk throughput is proportional to IRQ thread's bandwidth $\frac{Q}{T}$, both with and without DMA!

Balancing Throughput and Latency (I)

A **video player** is used as a real-time application, while IRQ load from the network card is generated with `netperf`.

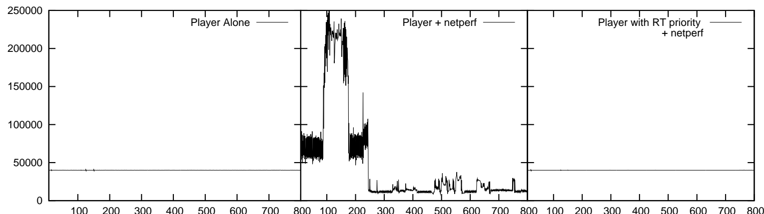
Raising the priority of the video player *above* the one of the network IRQ thread stabilizes the player performances...



... but the **network throughput drops from** $88 Mbps$ **to** $58 Mbps$.

A **video player** is used as a real-time application, while IRQ load from the network card is generated with `netperf`.

Raising the priority of the video player *above* the one of the network IRQ thread stabilizes the player performances…

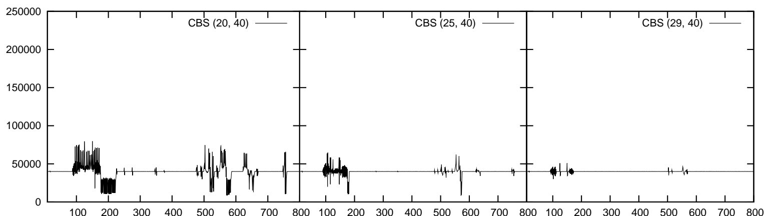


…but the **network throughput drops from** $88Mbps$ **to** $58Mbps$.

Balancing Throughput and Latency (II)

Using SCHED_DEADLINE for both (video player and netperf):

Test	Player CBS	net IRQ CBS	Throughput
Test1	$(29ms, 40ms)$	$(9ms, 100ms)$	$59.75Mbps$
Test2	$(28ms, 40ms)$	$(12ms, 100ms)$	$65.43Mbps$
Test3	$(26ms, 40ms)$	$(13ms, 100ms)$	$70.83Mbps$
Test4	$(25ms, 40ms)$	$(14ms, 100ms)$	$76.14Mbps$
Test5	$(20ms, 40ms)$	$(18ms, 100ms)$	$88.55Mbps$



Both throughput and latency can be controlled by means of resource reservations!

Conclusions and Future Work

We showed that some recent developments in the Linux kernel can be exploited to schedule the IRQ threads together with time sensitive tasks, using the resource reservation framework, such that both their interference on real-time tasks and the achieved device throughput are kept under control.

We also showed how to derive consistent scheduling parameters for the IRQ threads by statistically analyzing Linux scheduling traces.

Future Works:

- investigating thoroughly effectiveness and usability of the stochastic analysis framework;
- considering even more different workloads and resources;
- try to simplify the Markov model used inside such analysis.

Conclusions and Future Work

We showed that some recent developments in the Linux kernel can be exploited to schedule the IRQ threads together with time sensitive tasks, using the resource reservation framework, such that both their interference on real-time tasks and the achieved device throughput are kept under control.

We also showed how to derive consistent scheduling parameters for the IRQ threads by statistically analyzing Linux scheduling traces.

Future Works:

- investigating thoroughly effectiveness and usability of the stochastic analysis framework;
- considering even more different workloads and resources;
- try to simplify the Markov model used inside such analysis.

Conclusions and Future Work

We showed that some recent developments in the Linux kernel can be exploited to schedule the IRQ threads together with time sensitive tasks, using the resource reservation framework, such that both their interference on real-time tasks and the achieved device throughput are kept under control.

We also showed how to derive consistent scheduling parameters for the IRQ threads by statistically analyzing Linux scheduling traces.

Future Works:

- investigating thoroughly effectiveness and usability of the stochastic analysis framework;
- considering even more different workloads and resources;
- try to simplify the Markov model used inside such analysis.

Thank You for Your Time...

Any questions?

Effectivenes of the Statistical Tools

Test	Average	Std Dev	Max	Min
T1	1190	29	1569	1040
T5	5198	22	5278	5058
T10	10195	22	10277	10062
T50	50207	27	50298	50081
T100	100207	25	100290	100093

Test	Average	Std Dev	Max	Min
T1	1207	1011	14336	0
T5	5212	1019	6144	4096
T10	10210	271	12288	8192
T50	50229	1023	51200	49152
T100	100204	530	100352	98304

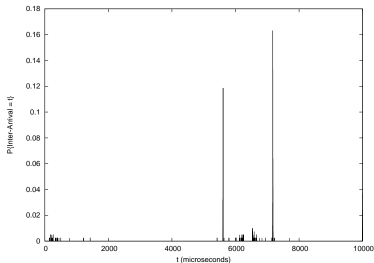
Inter-Packet times as measured in the sender and in the receiver (μs).

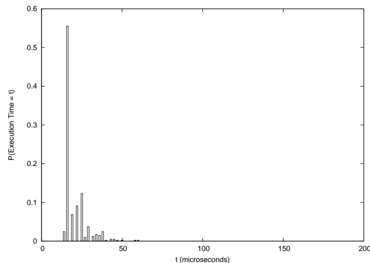
Test	Average	Std Dev	Max	Min
T1	1210	32	1424	59
T5	5222	117	5385	63
T10	10264	60	10353	10093
T50	50832	627	50353	50082
T100	100424	9342	100313	76

Test	Average	Std Dev	Max	Min
T1	15	5	63	9
T5	19	1	68	18
T10	14	1	29	13
T50	16	2	28	15
T100	21	3	23	12

Statistics about the inter-arrival and execution times of the IRQ thread (μs).

Full Disk Experiments (I)





PMF of the inter-arrival and execution times for the disk IRQ thread.

Full Disk Experiments (II)

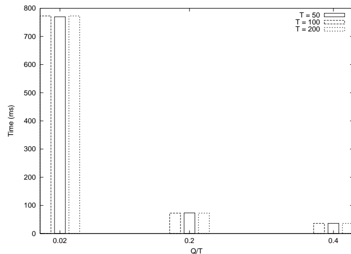
Test	Average
$(2ms, 100ms)$	1.9858%
$(6ms, 100ms)$	6.38374%
$(20ms, 1000ms)$	2.16843%
$(60ms, 1000ms)$	6.49811%
$(10ms, 100ms)$	10.6726%
$(20ms, 100ms)$	21.5825%
$(60ms, 100ms)$	62.4476%
$(80ms, 100ms)$	82.5952%
$(90ms, 100ms)$	92.9371%
$(95ms, 100ms)$	100%
$(100ms, 1000ms)$	11.778%
$(200ms, 1000ms)$	23.261%
$(600ms, 1000ms)$	65%
$(800ms, 1000ms)$	84.5455%
$(900ms, 1000ms)$	93.007%
$(950ms, 1000ms)$	100%

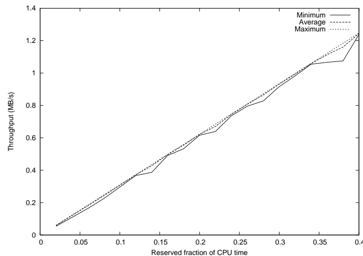
Disk IO-throughput when IRQ thread is scheduled with deadline scheduler.

Test	Average	Std Dev	Max	Min
No Reservations	$16.89s$	$0.12s$	$17.05s$	$16.67s$
$(30ms, 100ms)$	$52.85s$	$0.87s$	$55.22s$	$52.36s$
$(40ms, 100ms)$	$39.52s$	$0.61s$	$41.25s$	$39.27s$
$(50ms, 100ms)$	$31.49s$	$0.12s$	$31.74s$	$31.40s$
$(60ms, 100ms)$	$26.23s$	$0.03s$	$26.30s$	$26.19s$
$(70ms, 100ms)$	$22.58s$	$0.20s$	$23.14s$	$22.47s$
$(80ms, 100ms)$	$19.77s$	$0.19s$	$20.31s$	$19.69s$
$(90ms, 100ms)$	$17.59s$	$0.04s$	$17.66s$	$17.55s$

Time needed to perform a file copy, when the disk IRQ thread is scheduled with different parameters.

Full Disk Experiments (III)





Total time needed to read a large file and achieved throughput, as a function of the reserved CPU time.