


RTLA: finding sources of OS noise on Linux

Daniel Bristot de Oliveira
Senior Principal Software Engineer

Real-time Linux

- ▶ Linux has been used as an RTOS – it is a fact!
- ▶ There are multiple reasons for people to use it
 - Software stack and availability
 - Man-power
- ▶ But also because Linux achieves the desired timing behavior
- ▶ Some key features to help on that are:
 - The fully preemptive mode
 - Real-time scheduling
 - SCHED_DEADLINE

Who am I?

- I am Daniel 'bristot' de Oliveira
- Senior Principal Software Engineer at Red Hat 
 - Kernel developer in the real-time and scheduling team
 - Help to maintain rtla, tracers for rt/low-latency, and SCHED_DEADLINE
- Affiliate researcher at the Retis Lab/Scuola Superiore Sant'Anna
 - I have a Ph.D. in Automation Engineering & Real-time Embedded systems
 - Research about real-time and runtime verification
 - I'd say that this is my hobby

About me:

<https://bristot.me>



Agenda

- Motivation: RT and HPC - a real world request
- Characterization of the metrics
- Current and new approach for measurements and debugging
- The kernel tracers
- RTLA
- DEMO
- Some OS noise measurements

All these tools are integral part of the Linux kernel, and their documentation can be found in the Linux kernel documentation.

Linux as Real-time and HPC OS

- Linux has been used as an RTOS - it is a fact!
- Linux dominates the TOP 500 HPC list
- There are multiple reasons for people to use it
 - Software stack and availability
 - Human resources
 - Flexibility
- Both are high-performance setups but with different targets
 - RT focuses determinism, generally event-driven
 - HPC focuses on high-throughput, generally single-program multiple-data (SPMD)

The PREEMPT_RT is a patch set that improves the full preemptive mode of Linux. The patch set is on the final steps of the merge. So, Linux itself can be considered a real-time OS on its own.

Real-time and HPC: a real-world request

- Network Function Virtualization relies on HPC like setup
 - DPDK Poll Mode Driver
 - CPU isolation is set up to provide maximum throughput
 - Attempt to avoid any source of noise
- 5G is enabling low-latency event-driven communication
 - 10s of microseconds for vRAN
- NFV for 5G requires the best from RT and HPC setup

It is not easy to debug
`10s of microseconds
noise cases because
debugging can easily cost
more than the goal itself.

Real-time and HPC: a real-world request

- Linux has good CPU isolation features
- But isolation is not perfect as there is still residual work
- These scenarios only get worse when trying to use more complex and dynamic platforms like Kubernetes
 - HPC-RT workload mixed with regular workload
 - The regular workload can indirectly cause noise
- Moreover, the hardware itself can cause such noise
 - Without any traceable evidence

The lack of tools that can unambiguously point to a root cause creates a nightmare for those supporting such environment.

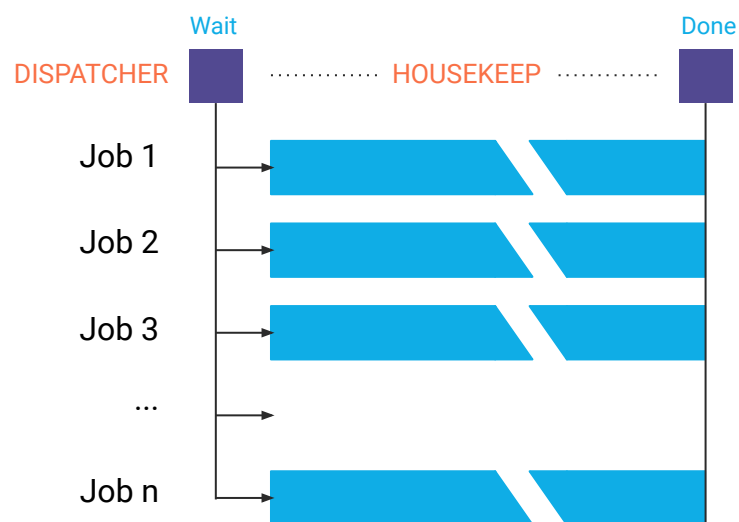
Speculations is your worst enemy.



RT and HPC metrics characterization

OS Noise

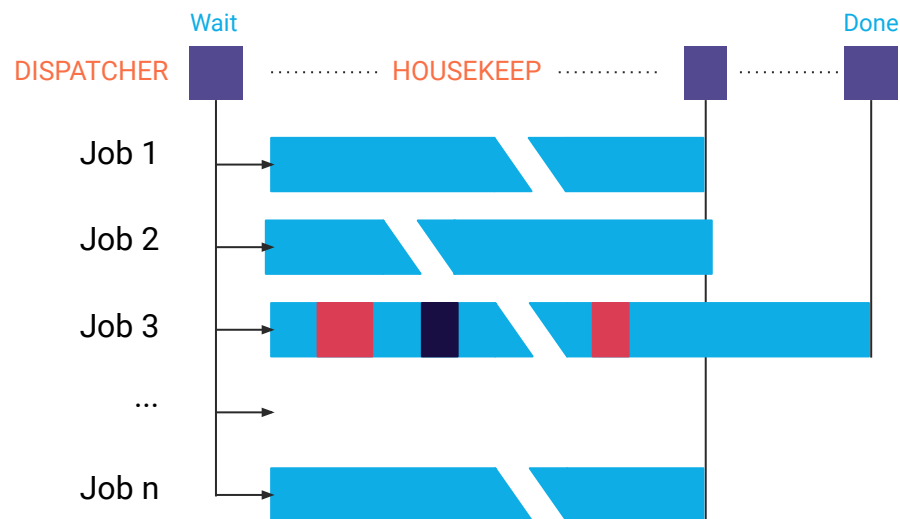
- Generally, HPC workloads are composed of parallel jobs
- The system is configured with CPUs dedicated to the jobs
- A dispatcher lunches jobs to these CPUs and waits for completion.



In real-time terms, the OS Noise can be seen as the interference of a high priority OS task to the user application.

OS Noise

- In practice, other OS and user workloads can disturb the HPC job, creating the so called operating system (OS) noise.



The same problem replicates for serial pipelines, with OS Noise influencing in the response time - real-time metric.

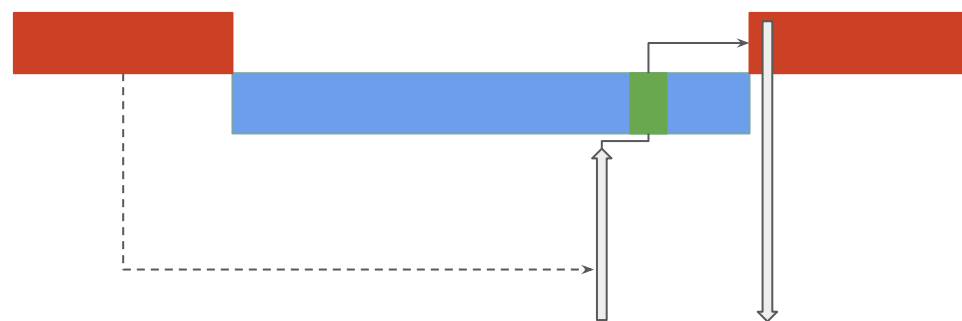
Sources of OS Noise

- Any "task" abstraction on the OS that can preempt the HPC job
- On Linux:
 - NMIs
 - IRQs
 - Softirqs
 - Threads
- Hardware & Virtualization sources of noise
 - SMIs
 - VM preemption by the host
 - ...

It doesn't matter if it is an OS/Kernel thread or an User/User space thread. From the scheduler perspective they are equivalent.

Activation latency

- Activation latency is the delay on replying to an external event
- Can be simulated with an external timer



Set a timer at t and
goes to sleep t $\Delta = t' - t$

It is the delta between the time that the timer was set and the time in which the thread after activated could read the current time.

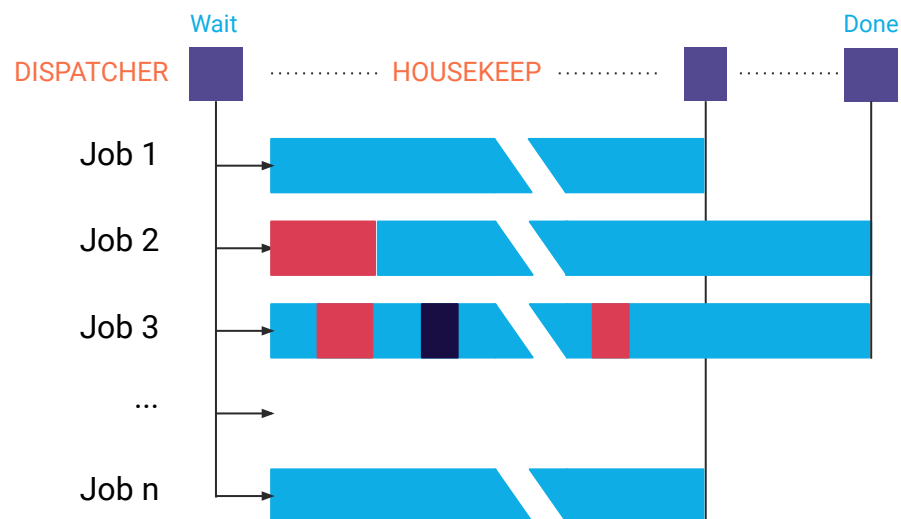
Sources of timer latency

- Any "task" abstraction on the OS that can preempt the RT job
- All HPC metrics (OS tasks, hardware latencies)
- Low priority thread running with preemptions or IRQ off
 - Well, it is a **little bit more complex than that...** See ->

For a formal definition,
please see:



OS Noise and Activation latency



For low latency HPC,
activation latency is also a
problem.



Current approach

Current approach

- Nowadays, practitioners use a set of black-box tools that mimic typical workload:
 - Event-driven application: **cyclicttest**
 - Polling like application: **sysjitter/oslat**
- They report a "latency," and this is important for many use-cases. For example:
 - The kernel-rt has to deliver < 150 us cyclicttest latency under stress
 - cyclicttest latency of 10~20 us on isolated & tuned systems.

```

$ cat -c cyclicttest.txt
$]# cyclicttest --smp -p 95 -m
ency set to 0us
avg: 14.90 6.21 3.98 2/387 2735923

$ I:1000 C: 66520 Min: 4 Act: 5 A
$ I:1500 C: 44341 Min: 4 Act: 6 A
$ I:2000 C: 33254 Min: 4 Act: 6 A
$ I:2500 C: 26598 Min: 4 Act: 5 A
$ I:3000 C: 22162 Min: 4 Act: 5 A
$ I:3500 C: 18993 Min: 4 Act: 6 A
$ I:4000 C: 16647 Min: 4 Act: 5 A
$ I:4500 C: 14762 Min: 4 Act: 6 A
$ I:5000 C: 13290 Min: 4 Act: 5 A
$ I:5500 C: 12080 Min: 4 Act: 6 A
$ I:6000 C: 11072 Min: 8 Act: 12 A
$ I:6500 C: 10219 Min: 4 Act: 5 A
$ I:7000 C: 9488 Min: 5 Act: 6 A
$ I:7500 C: 8854 Min: 5 Act: 5 A
$ I:8000 C: 8200 Min: 4 Act: 6 A
$ I:8500 C: 7801 Min: 4 Act: 9 A
$ I:9000 C: 7370 Min: 5 Act: 6 A
$ I:9500 C: 6987 Min: 4 Act: 6 A
$ I:10000 C: 6638 Min: 5 Act: 9 A

```


Linux Tracing and black-box testing

- If a bad value happens, you need to start getting the hands dirt
- The developer/practitioner needs try to understand the root cause
 - It is hard to do when you have someone else operating the machine
- Manual interpretation of a lot of data
 - Speculation goes on (many times misleading)

There are problems that a really difficult to reproduce, so it is easy to get mislead by different problems. Speculation is a real problem without precise information.

Linux Tracing

- There are multiple ways to observe Linux runtime behavior
- One can use **top** command to monitor CPU usage
- But for a low overhead of fine-grained observation, there is nothing better than tracing
- Linux has a powerful set of tracing tools
 - It is possible to trace all functions
 - It is possible to trace dynamic events
 - It is possible to extend it

```

$ cat -c cyclictst.txt
$]# cyclictst --smp -p 95 -m
ency set to 0us
avg: 14.90 6.21 3.98 2/387 2735923

```

```

$ I:1000 C: 66520 Min: 4 Act: 5 A
$ I:1500 C: 44341 Min: 4 Act: 6 A
$ I:2000 C: 33254 Min: 4 Act: 6 A
$ I:2500 C: 26598 Min: 4 Act: 5 A
$ I:3000 C: 22162 Min: 4 Act: 5 A
$ I:3500 C: 18993 Min: 4 Act: 6 A
$ I:4000 C: 16647 Min: 4 Act: 5 A
$ I:4500 C: 14762 Min: 4 Act: 6 A
$ I:5000 C: 13290 Min: 4 Act: 5 A
$ I:5500 C: 12080 Min: 4 Act: 6 A
$ I:6000 C: 11072 Min: 8 Act: 12 A
$ I:6500 C: 10219 Min: 4 Act: 5 A
$ I:7000 C: 9488 Min: 5 Act: 6 A
$ I:7500 C: 8854 Min: 5 Act: 5 A
$ I:8000 C: 8200 Min: 4 Act: 6 A
$ I:8500 C: 7801 Min: 5 Act: 9 A
$ I:9000 C: 7370 Min: 4 Act: 6 A
$ I:9500 C: 6987 Min: 5 Act: 6 A
$ I:10000 C: 6638 Min: 5 Act: 9 A

```

Linux Tracing

- Linux tracing is available on production systems
- They are optimized to the state of the art
- They cause no overhead when disabled
- But the overhead can be noticeable when too much tracing is done

```

$ cat -c cyclictst.txt
$]# cyclictst --smp -p 95 -m
ency set to 0us
avg: 14.90 6.21 3.98 2/387 2735923

```

```

$ I:1000 C: 66520 Min: 4 Act: 5 A
$ I:1500 C: 44341 Min: 4 Act: 6 A
$ I:2000 C: 33254 Min: 4 Act: 6 A
$ I:2500 C: 26598 Min: 4 Act: 5 A
$ I:3000 C: 22162 Min: 4 Act: 5 A
$ I:3500 C: 18993 Min: 4 Act: 6 A
$ I:4000 C: 16647 Min: 4 Act: 5 A
$ I:4500 C: 14762 Min: 4 Act: 6 A
$ I:5000 C: 13290 Min: 4 Act: 5 A
$ I:5500 C: 12080 Min: 4 Act: 6 A
$ I:6000 C: 11072 Min: 8 Act: 12 A
$ I:6500 C: 10219 Min: 4 Act: 5 A
$ I:7000 C: 9488 Min: 5 Act: 6 A
$ I:7500 C: 8854 Min: 5 Act: 5 A
$ I:8000 C: 8200 Min: 4 Act: 6 A
$ I:8500 C: 7801 Min: 5 Act: 9 A
$ I:9000 C: 7370 Min: 4 Act: 6 A
$ I:9500 C: 6987 Min: 5 Act: 6 A
$ I:10000 C: 6638 Min: 5 Act: 9 A

```

Linux Tracing and black-box testing

- It is possible to observe black-box testing tools using trace
- The problem is defining what is important to trace
- Obtaining the maximum of information with the minimum possible overhead

One could enable all traceable events, but the analysis is impractical due to the amount of information, and additional overhead.



A new approach

A new approach

- Measuring without tracing is not productive
- Without always tracing, you never know if the problem you faced in the first place is the same one you are seeing while tracing.
 - That is especially hard when the target values are tight, and a lot of information is traced.
- After 10+ years of doing this, the trace became a mechanical thing:
 - irq events, sched: events, compute deltas.
- Can't we join these two things?

One could enable all traceable events, but the analysis is impractical due to the amount of information, and additional overhead - noticeable at 10s of us scale.

Tracing + Workload

- **osnoise and timerlat are kernel tracers that also dispatches the workload**
- The workload runs in the kernel:
 - **osnoise: A busy-loop kernel thread that reads time() in a loop**
 - Reports problem when $\text{time()} - \text{time()} > \text{threshold}$ - aka noise.
 - **timerlat: A periodic task that is awakened by an hrtimer**
 - Reports IRQ latency and Thread latency

There is a DEMO video
later on...

osnoise: tracepoints

- The tracers provided a **new set of tracepoints** that automatize the trace:
 - osnoise:nmi_noise/irq_noise/softirq_noise/thread_noise:
 - Report the interference of tasks to the tracer workload
 - Account for the interference and report net values of it.
 - The osnoise: tracepoints work by hooking to existing events
 - Instead of tracing irq_entry & irq_exit, osnoise:irq_noise reports the delta
- The tracers can also collect other information such as stack traces

Tracepoints are one of the most versatile tracing methods. There are multiple ways to consume a tracepoint output, and it is also possible to run code on tracepoints!

Benefits of the selected approach

- The workload and the trace are synchronized
- The workload can have atomic access to information collected by the trace
 - E.g., osnoise workload also reports the \$ of interference that happens between two time() reads
 - Precise information (no false positives/speculation)
- Minimum overhead:
 - The osnoise: tracepoints reduce the amount of events by a half
 - Only the necessary information is exported via tracepoints

osnoise and **timerlat**

tracers are already enabled on multiple Linux Distro, for example on Fedora/CentOS/Red Hat and on OpenSUSE/SLE.



osnoise tracer

OS Noise tracer

- **osnoise** is a kernel tracer that also dispatches the workload
 - The workload runs in the kernel
- Mimics HPC workload
 - One thread per CPU
- It detects high priority tasks that interfere with the osnoise workload
 - osnoise can also detect hw/vm induced latency

The tracer is inspired on
hwlat tracer.

OS Noise tracer: summary report

```
[root@f32 ~]# cd /sys/kernel/tracing/
[root@f32 tracing]# echo osnoise > current_tracer
[root@f32 tracing]# cat trace
# tracer: osnoise
#
#
#          _-----> irqs-off
#          / _-----> need-resched
#          | / _----> hardirq/softirq
#          || / _--> preempt-depth
#          || /
#                                     MAX
#                                     SINGLE
#                                     Interference counters:
#                                     +-----+
# TASK-PID      CPU#  | TIMESTAMP   |   |   |   |   |   |   |   |   |   |   |
#      | |      |    |         |   |   |   |   |   |   |   |   |   |
# <...>-859    [000]  ....   81.637220: 1000000      190  99.98100      9   18    0   1007    18    1
# <...>-860    [001]  ....   81.638154: 1000000      656  99.93440     74   23    0   1006    16    3
# <...>-861    [002]  ....   81.638193: 1000000     5675  99.43250    202    6    0   1013    25   21
# <...>-862    [003]  ....   81.638242: 1000000     125  99.98750     45    1    0   1011    23    0
# <...>-863    [004]  ....   81.638260: 1000000    1721  99.82790    168    7    0   1002    49   41
# <...>-864    [005]  ....   81.638286: 1000000     263  99.97370     57    6    0   1006    26    2
# <...>-865    [006]  ....   81.638302: 1000000     109  99.98910     21    3    0   1006    18    1
# <...>-866    [007]  ....   81.638326: 1000000    7816  99.21840    107    8    0   1016    39   19
```

OS Noise tracer: options

- Configuration files inside **/sys/kernel/trace/osnoise**
 - **cpus:** CPUs at which a osnoise thread will execute.
 - **period_us:** the period of the osnoise thread.
 - **runtime_us:** how long an osnoise thread will look for noise in the period
 - **stop_tracing_us:** stop the system tracing if a single noise is \geq than set here
 - **stop_tracing_total_us:** stop the system tracing if total noise is \geq than set here
- **/sys/kernel/trace/tracing_threshold**
 - The minimum delta between two time() reads to be considered as noise, in us.
 - When set to 0, the default value will be used, which is currently 5 us.

OS Noise tracer: fine-grained tracing

```
[root@f32 ~]# cd /sys/kernel/tracing/
[root@f32 tracing]# echo osnoise > current_tracer
[root@f32 tracing]# echo osnoise > set_event
[root@f32 tracing]# echo 8 > osnoise/stop_tracing_us
[root@f32 tracing]# cat trace
[...]
```

osnoise/8-960	[007]	d.h.	5789.857530:	irq_noise: local_timer:236	start 5789.857527123	duration 1867 ns
osnoise/8-961	[008]	d.h.	5789.857532:	irq_noise: local_timer:236	start 5789.857529929	duration 1845 ns
osnoise/8-961	[008]	dNh.	5789.858408:	irq_noise: local_timer:236	start 5789.858404871	duration 2848 ns
migration/8-54	[008]	d...	5789.858413:	thread_noise: migration/8:54	start 5789.858409300	duration 3068 ns
osnoise/8-961	[008]	5789.858413:	sample_threshold:	start 5789.858404555	duration 8812 ns interferences 2

hw noise

- As the osnoise tracer tracks all sources of noise:
 - NMI
 - IRQs
 - softirqs
 - threads
- Any noise sample that is not classified as OS Noise, is then a hardware (or VM) noise.

hwlat tracer is a more specialized tool. It works with interruptions disabled, so it can only be interfered by NMIs and hardware itself.

osnoise is sufficient, but not necessary, to detect hw noise. Being sufficient is enough to reduce time debugging time.

OS Noise tracer: hw noise

```
[root@f32 ~]# cd /sys/kernel/tracing/
[root@f32 tracing]# echo osnoise > current_tracer
[root@f32 tracing]# cat trace
# tracer: osnoise
```

```
#
#           _-----=> irqsoft
#         / _-----=> need-resched
#       | / _-----=> hardirq/softirq
#     || / _--=> preempt-depth
#    || /
#   |||
#                                     RUNTIME
# TASK-PID          CPU#  ||||      TIMESTAMP      IN US
#   | |             |    ||||      |              |
#
```

#	TASK-PID	CPU#		TIMESTAMP	RUNTIME IN US	NOISE IN US	% OF CPU AVAILABLE	NOISE IN US
#								
	<...>-859	[000]	81.637220:	1000000	190	99.98100	9
	<...>-860	[001]	81.638154:	1000000	656	99.93440	74
	<...>-861	[002]	81.638193:	1000000	5675	99.43250	202
	<...>-862	[003]	81.638242:	1000000	125	99.98750	45
	<...>-863	[004]	81.638260:	1000000	1721	99.82790	168
	<...>-864	[005]	81.638286:	1000000	263	99.97370	57
	<...>-865	[006]	81.638302:	1000000	109	99.98910	21
	<...>-866	[007]	81.638326:	1000000	7816	99.21840	107

Interference counters:

HW	NMI	IRQ	SIRQ	THREAD
18	0	1007	18	1
23	0	1006	16	3
6	0	1013	25	21
1	0	1011	23	0
7	0	1002	49	41
6	0	1006	26	2
3	0	1006	18	1
8	0	1016	39	19

OS Noise tracer: hw noise

```
[root@x1 osnoise]# cd /sys/kernel/debug/tracing/
[root@x1 tracing]# echo osnoise > current_tracer
[root@x1 tracing]# echo osnoise > set_event
[root@x1 tracing]# cat per_cpu/cpu1/trace | grep -B 2 "interference 0"
[...]
```

osnoise/1-32160	[001]	d.h1.	31240.380886:	irq_noise: thermal_apic:250	start 31240.380884026	duration 1715 ns
osnoise/1-32160	[001]	31240.380886:	sample_threshold:	start 31240.380883588	duration 2763 ns interference 1
osnoise/1-32160	[001]	31240.381105:	sample_threshold:	start 31240.381090803	duration 14384 ns interference 0



timerlat tracer

Timer latency

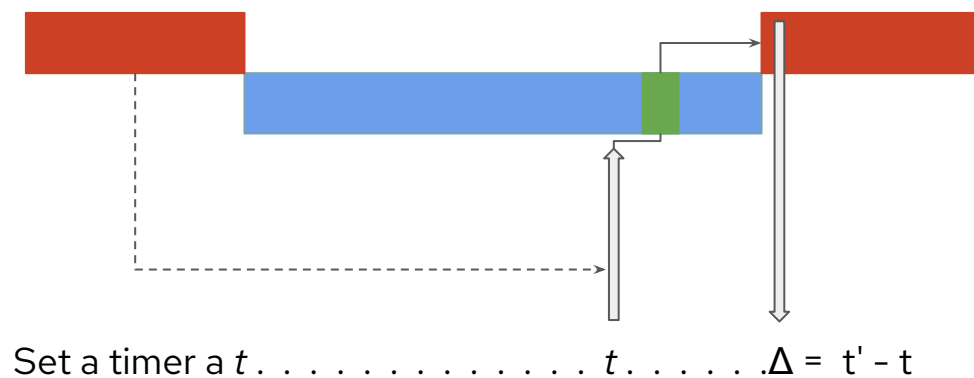
- The timer latency has been used as a metric by the real-time Linux kernel developers
- cyclicttest is indeed a timer testing tool
- It empirically measures the observed scheduling latency of the highest priority thread - or a thread at any priority
- timerlat tracer measure the same metric, but it is integrated with tracing.

The usage of the timer latency to measure the wakeup latency is a controversial topic.

See this:

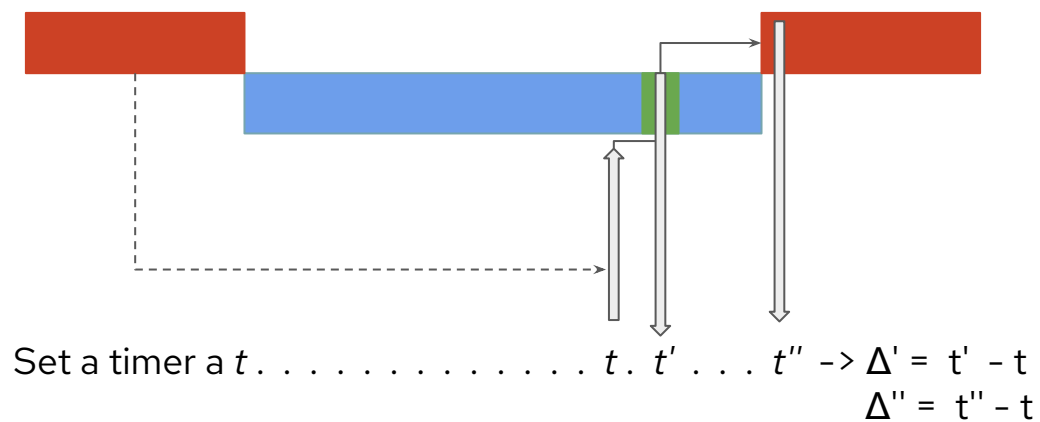


Timer latency: Thread latency



It is the delta between the time that the timer was set and the time in which the thread after activated could read the current time.

Timer latency: IRQ and Thread



Because timerlat is in the kernel, it has a special IRQ handler that also notifies the IRQ activation latency

Timerlat tracer: summary output

```
[root@f32 ~]# cd /sys/kernel/tracing/
[root@f32 tracing]# echo timerlat > current_tracer
[root@f32 tracing]# cat trace
# tracer: timerlat
#
#          _-----> irqsoff
#         /_-----> need-resched
#        |/_----=> hardirq/softirq
#       ||/_--=> preempt-depth
#      ||/_
#     |||/
#    |||| ACTIVATION
#   TASK-PID   CPU#  ||||  TIMESTAMP   ID             CONTEXT           LATENCY
#      ||      |    ||||  |              |             |               |
<idle>-0      [000] d.h1   54.029328: #1      context  irq timer_latency    932 ns
<...>-867     [000] ....   54.029339: #1      context thread timer_latency 11700 ns
<idle>-0      [001] dNh1   54.029346: #1      context  irq timer_latency    2833 ns
<...>-868     [001] ....   54.029353: #1      context thread timer_latency  9820 ns
<idle>-0      [000] d.h1   54.030328: #2      context  irq timer_latency     769 ns
<...>-867     [000] ....   54.030330: #2      context thread timer_latency  3070 ns
<idle>-0      [001] d.h1   54.030344: #2      context  irq timer_latency     935 ns
<...>-868     [001] ....   54.030347: #2      context thread timer_latency  4351 ns
```

timerlat tracer options

- Configuration files inside `/sys/kernel/trace/osnoise`
 - **cpus:** CPUs at which a timerlat thread will execute.
 - **period_us:** the timer period
 - **stop_tracing_us:** stop the system tracing if IRQ latency \geq than set here
 - **stop_tracing_total_us:** stop the system tracing if thread latency is \geq than set here
 - **print_stack:** save the IRQ stack trace to print in case of latency \geq than set

What can cause timer latency?

- Linux's task abstractions:
 - NMI
 - IRQs
 - softirqs
 - Higher priority thread
- Previously running thread with preemption || irq disabled

Kernel has multiple preemption models. But also the fully preemptive mode can suffer from preemption delays because preemption can be temporarily disabled.

timerlat tracer: stack trace of preempt disable

```
[root@f32 ~]# cd /sys/kernel/tracing/
[root@f32 tracing]# echo timerlat > current_tracer
[root@f32 tracing]# echo 1 > events/osnoise/enable
[root@f32 tracing]# echo 500 > osnoise/stop_tracing_total_us
[root@f32 tracing]# echo 500 > osnoise/print_stack
[root@f32 tracing]# tail -21 per_cpu/cpu7/trace
    insmod-1026    [007] dN.h1..  200.201948: irq_noise: local_timer:236 start 200.201939376 duration 7872 ns
    insmod-1026    [007] d..h1..  200.202587: #29800 context    irq timer_latency    1616 ns
    insmod-1026    [007] dN.h2..  200.202598: irq_noise: local_timer:236 start 200.202586162 duration 11855 ns
    insmod-1026    [007] dN.h3..  200.202947: irq_noise: local_timer:236 start 200.202939174 duration 7318 ns
    insmod-1026    [007] d...3..  200.203444: thread_noise:  insmod:1026 start 200.202586933 duration 838681 ns
    timerlat/7-1001 [007] .....  200.203445: #29800 context thread timer_latency    859978 ns
    timerlat/7-1001 [007] ....1..  200.203446: <stack trace>
=> timerlat_irq
=> __hrtimer_run_queues
=> hrtimer_interrupt
=> __sysvec_apic_timer_interrupt
=> asm_call_irq_on_stack
=> sysvec_apic_timer_interrupt
=> asm_sysvec_apic_timer_interrupt
=> delay_tsc
=> dummy_load_1ms_pd_init
=> do_one_initcall
=> do_init_module
=> __do_sys_finit_module
=> do_syscall_64
=> entry_SYSCALL_64_after_hwframe
```



RTLA

Real-time Linux Analysis!

- RTLA is a meta-tool that includes a **set of commands that aims to analyze the real-time properties of Linux.**
- **rtla is a user-space tool** that serves as the front-end for setup, tracing, and interpretation of data.
- **It is in C, hosted inside the kernel source.**

It is a meta tool because it does not aim to do only one analysis, but to become a tool set, home for multiple types of analysis.

rtla osnoise

- **rtla osnoise** is an interface for osnoise tracer
- Two different modes:
 - osnoise top: shows an interactive view of the osnoise summary output
 - osnoise hist: shows a histogram of the osnoise sample tracepoint

The tool can be called either using "rtla osnoise" or "osnoise" only.

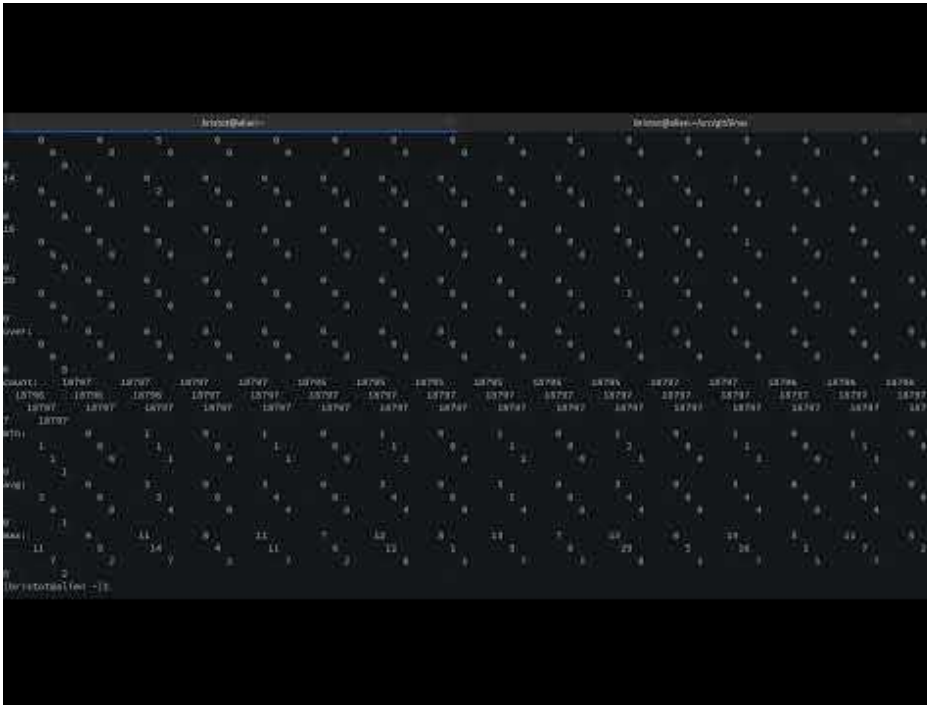
rtla timerlat

- **rtla timerlat** is an interface for timerlat tracer
- Two different modes:
 - timerlat top: shows an interactive view of the timer latencies
 - timerlat hist: shows a histogram of the timer latencies

The tool can be invoked either using "rtla timerlat" or "timerlat" only.

demo

<https://www.youtube.com/watch?v=3sGM076mLRQ>



There are more rta tools to be developed, but the current implementation is already useful for many users!

Some osnoise measurements

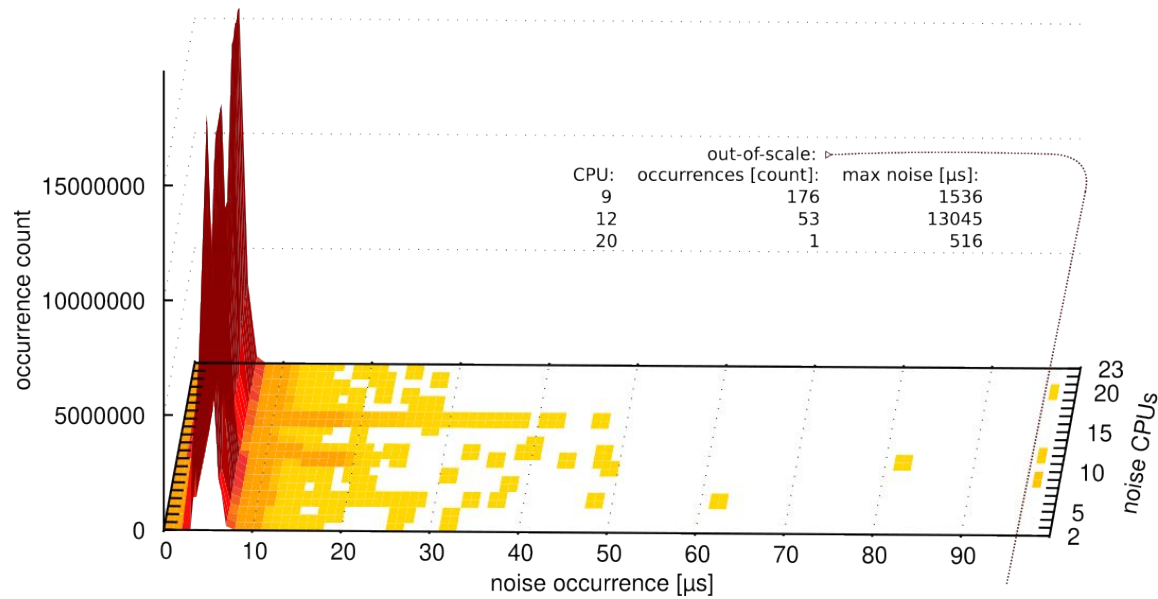
- Measurements made in a system with 24 CPUs
 - 12 cores/24 threads
- Two configurations, four experiments:
 - the system **as is** has no isolation setup (fresh install), while the **isolated** is tuned for this purpose.
 - the osnoise threads run with **regular priority (0 nice)** or with **real-time priority (FIFO:1)**
- The system runs for six hours on each setup

The tool can be invoked either using "rtla timerlat" or "timerlat" only.

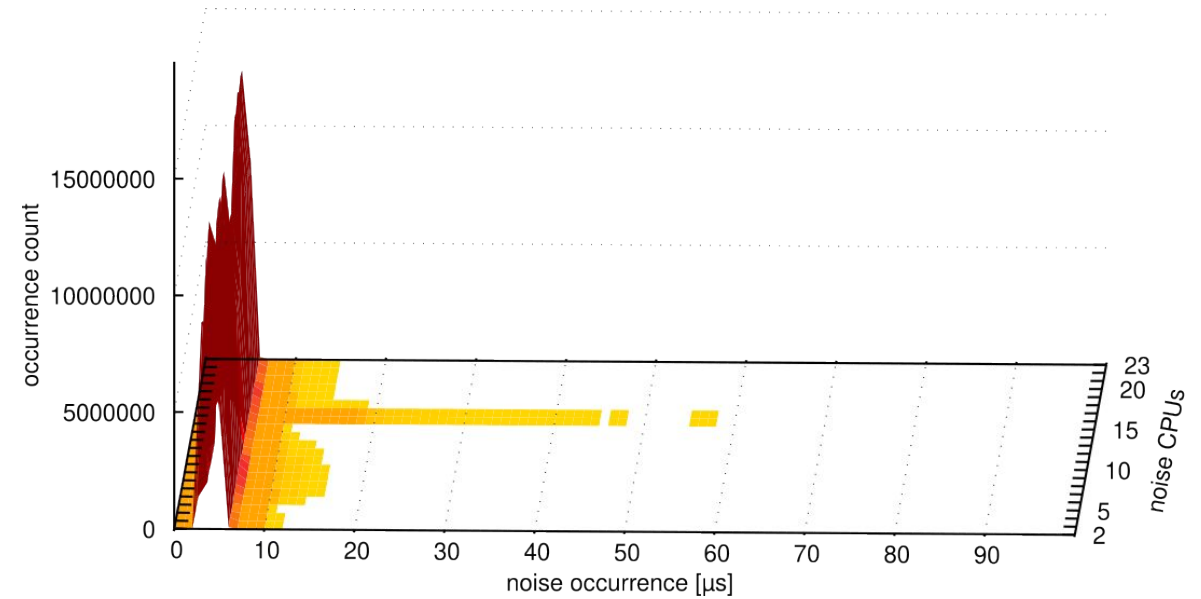


Some measurements

osnoise histograms: system as is

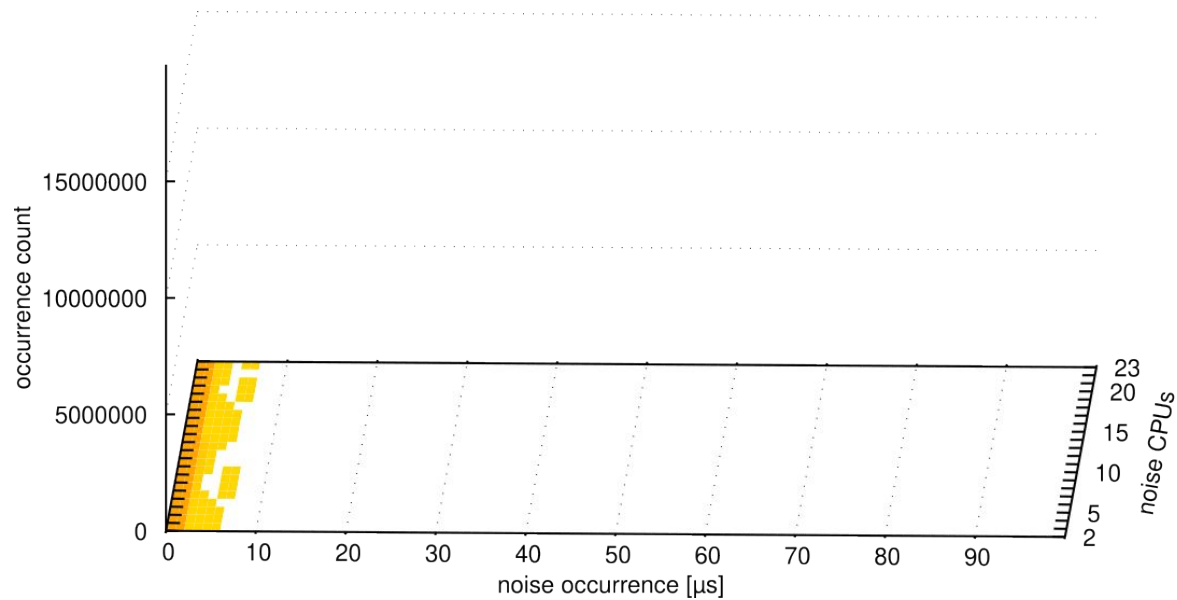


system as is

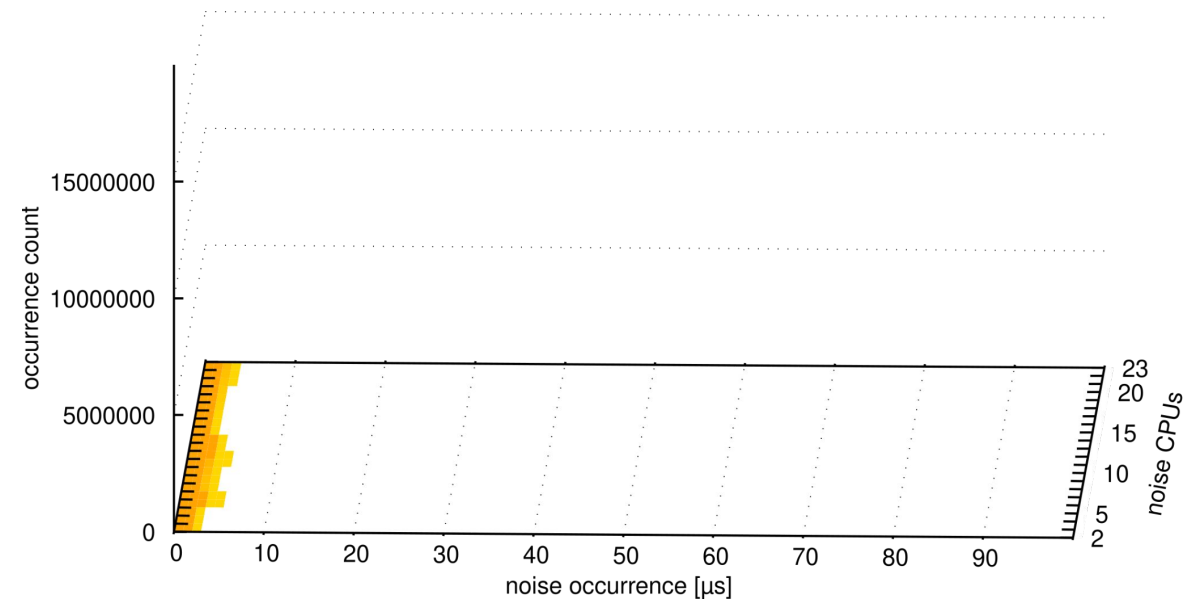


system as is using FIFO

osnoise histograms: system tuned



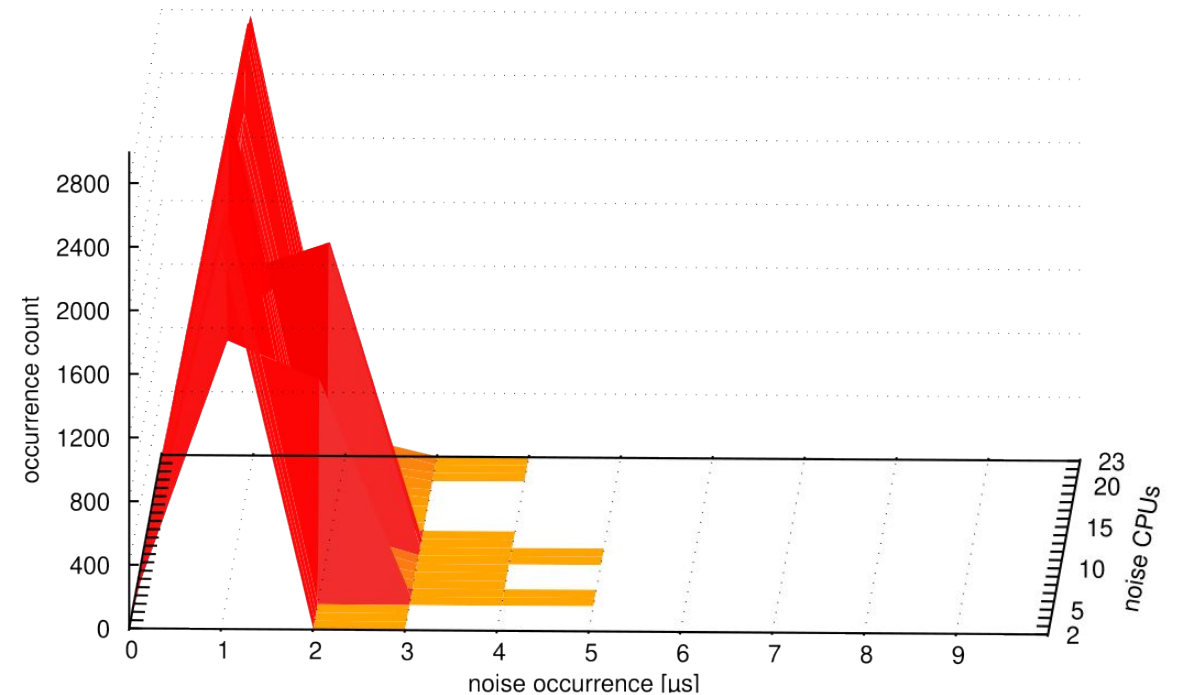
system with isolated CPUs



system with isolated CPUs using FIFO

osnoise histograms: system tuned

- With proper tune, Linux delivers:
 - Linux can deliver single digit osnoise occurrences - FIFO:1 delivered < 5 us in this setup
 - Linux consistently delivered 99.99999% of CPU time
- But the results depends on the machine and can change at every kernel release due to non-HPC/RT aware algorithms.



system with isolated CPUs using FIFO

Remarks

- The flexibility of Linux enables a set of new applications
 - Mixing RT and HPC configs
- The timerlat and osnoise tracers allow the measurement and tracing of the desired metrics in an integrated way
- The RTLA transforms these tracers into a benchmark tool
- All tools discussed here are an integral part of the Linux kernel:
 - timerlat and osnoise in the 5.14
 - RTLA in 5.17

RTLA will continue evolving to be the home for multiple types of analysis.

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

 linkedin.com/company/red-hat

 youtube.com/user/RedHatVideos

 facebook.com/redhatinc

 twitter.com/RedHat