# Timerlat: Real-time Linux Scheduling Latency Measurements, Tracing, and Analysis

Daniel Bristot De Oliveira, Daniel Casini *Member, IEEE,* Juri Lelli, and Tommaso Cucinotta *Member, IEEE*

**Abstract**—A trend in many embedded devices is the move from hardware-based to software-defined, such as software-defined networks and software-defined PLCs. This trend is motivated by multiple aspects, including the availability of complex software stacks and the consolidation of multiple devices into a single larger system. Due to its real-time capabilities and flexibility, Linux is the operating system of choice for many applications, including time-sensitive ones. However, assessing and debugging timing violations, especially those caused by scheduling latency, is challenging with the current state-of-the-art tools. This paper presents `timerlat`, a tool that integrates scheduling latency measurements, tracing, and analysis in an easy-to-use interface. Its output includes an auto-analysis, providing insightful details on the composition of the scheduling latency. Experimental results are reported, evaluating the effectiveness of timerlat in assessing the latencies, considering different setups and workloads.

**Index Terms**—Real-time Linux, Scheduling latency, Tracing.

## 1 Introduction

In embedded systems, a series of motivations is leading the move from hardware-based to software-defined systems [1]. These support faster deployment and update, device consolidation, and easy access to modern software stacks, e.g., as needed for artificial intelligence (AI) use cases.

A common requirement for these systems is a low-latency and predictable response to external events [2]. Use cases increasingly relying on software-defined systems of this kind include industrial automation, automotive software, and low latency communications, e.g., a 5G radio access network (RAN) [3]. Therefore, the operating system needs to provide low latency, in the order of tens of microseconds, as in the case of software-defined networking for Virtualized Radio Access Network (vRAN) [4]–[6].

Linux is the preferred platform for many of these applications, because it supports a wide range of hardware, from embedded to server-based devices. It also offers multiple virtualization/containerization methods [7] and native applications/libraries for AI [8]. In particular, Linux's ability to provide low scheduling latency is essential for time-sensitive applications. On a properly configured Linux system, the highest priority task on a CPU can be activated with a scheduling latency in the order of a few microseconds. This is achievable with a PREEMPT_RT kernel[1] and a proper set of real-time priorities.

However, as shown in the experimental evaluation in Section 4, the scheduling latency also depends on the workload that runs on each CPU. For example, in a consolidation case where multiple virtual Programmable Logic Controllers (PLCs) with different real-time requirements share the same CPU [9], a lower-priority application can influence the scheduling latency of the highest-priority one.

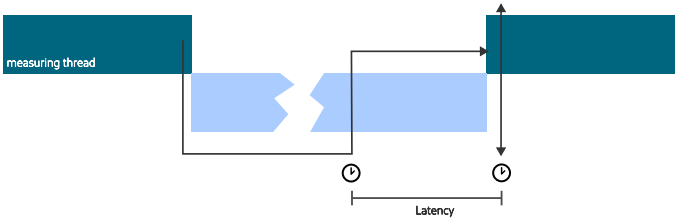



Fig. 1: Black-box latency measurement approach.

To date, the scheduling latency is measured using a black-box approach, for example, using *cyclictest*. This tool measures the scheduling latency as shown in Fig. 1: a "sampling" thread, typically assigned to the highest priority, sets a timer by leveraging an external clock reference and suspends waiting for the timer. The thread is then awakened when the timer expires, computing the scheduling latency as the difference between the measured current time and the expected timer wake-up time.

The black box approach is a simple and convenient method to measure the scheduling latency. Nevertheless, it has a principal limitation: it does not provide any insight about the *composition* of the scheduling latency, which could provide a notable help in understanding the causes of an unexpectedly high value. Current tools stop the tracing session when a value higher than a configurable threshold is met. Then, a tracing expert needs to set up a tracing session and discover its cause, which is time-consuming and requires expert knowledge. This is a significant barrier for many users since tracing is often inaccessible for non-experts. Timerlat makes this process much easier by integrating measurements and tracing.

**Contributions.** The contributions of this paper are threefold:

1) It presents timerlat, a new tool for measuring the Linux scheduling latency. In particular, it integrates workload

*Daniel Bristot De Oliveira and Juri Lelli are with the Real-time Scheduling Team, Red Hat, Inc., Italy. Daniel Casini and Tommaso Cucinotta are with the Scuola Superiore Sant'Anna, Pisa, Italy.*
*E-mail:{bristot, juri.lelli}@redhat.com,*
*{d.casini, t.cucinotta}@santannapisa.it*
*Manuscript received April 19, 2021; revised August 16, 2021.*

1. https://wiki.linuxfoundation.org/realtime/start

execution, tracing, and automatic root cause analysis into a single toolkit. The auto-analysis summarizes the possible root causes for latency spikes without needing the expert knowledge required to perform advanced tracing activities on Linux. Notably, the tool has already been integrated into mainline Linux (since kernel version 5.17) and is hence available in every computing platform with an updated Linux kernel;
2) It discusses the components of the latency as shown by the auto-analysis, along with insights about how to improve the system latency; and
3) It shows latency results on a system with different levels of optimization and background workload.

## 2 Related Work

The latency of Linux has been studied for several decades. For example, in 2002, Abeni et al. [10] studied the OS latency components of a set of Linux kernels and defined a latency metric similar to the one implemented by cyclictest. Their work is, however, limited to the kernels available in 2002. Reghenzani et al. [11] presented an empirical measurement study of the latencies of real-time Linux under stress considering a mixed-criticality scenario. Herzog et al. [12] presented a tool that measures the interrupt latency at runtime, targeting the standard Linux kernel. The evaluation of the empirical timing behavior of Linux has also been addressed by Regnier et al. [13]. An extensive empirical study using cyclictest has been presented by Cerqueira and Brandenburg [14], who evaluated the scheduling latency under three different Linux variants, i.e., vanilla Linux, PREEMPT_RT Linux, and LITMUS$^{RT}$, a real-time extension of the Linux kernel widely popular in the academia. The study in [15] compares the performance of PREEMPT_RT Linux with QNX and Windows Embedded Compact 7.

Most related to us is the work by De Oliveira et al. [16]. It provided a theoretically sound method to compute the Linux scheduling latency, leveraging an automata formalism. It easily integrates with perf to provide insights on the causes of a high latency value. Still, the work in [16] has different features with respect to Timerlat:

(i) It targets only the PREEMPT_RT kernel, while timerlat is compatible with any preemption model.
(ii) It focuses on the worst-case latency while this work provides statistics on the *latency distribution*, which allows reasoning on the jitter. This is particularly important for several application scenarios, including industrial automation.
(iii) It requires using several tracepoints that are typically not enabled since they cause overhead. While this overhead is negligible for the worst-case latency, it is not the case when looking at the overall latency distribution (in which a fine-grained accounting of small latency samples is also important). Timerlat uses more efficient tracepoints so that to enable an accurate accounting of latency samples of all kinds.

Other works targeted the investigation, via tracing, of unexpected performance metrics values.

The *ftrace preemptirqsoff* tracer [17] tries to estimate the longest time window in which OS-related blocking occurs by searching for the longest window with IRQs or preemptions disabled. The tool cannot distinguish between blocking arising from IRQs or preemptions and adds overhead to the measurement since it adds tracing capabilities to functions, potentially affecting the results. Other tracing solutions are perf, kutrace [18], and the aforementioned cyclictest. However, they both significantly differ from timerlat. Perf and kutrace are tracers used for general-purpose performance profiling. None of them measures the latency by means of the black-box approach discussed in Section 1, which is the standard approach used by practitioners. Furthermore, kutrace is not part of the mainline Linux kernel. Differently, cyclictest, performs an analogous latency measurement, but it does not include tracing features to point to the root cause of a high latency sample. Differently, timerlat allows for simultaneously measuring the latency and performing tracing, as extensively discussed in the following. This allows for avoiding setting up separate and complex tracing sessions, which can typically need to combine standard and custom-tailored tracepoints. Furthermore, timerlat provides three different latency measurements from the interrupt handler, thread, and user perspective. Instead, cyclictest considers just the user latency. As shown in the Section 7, the overhead introduced with respect to cyclictest is negligible. Finally, the *osnoise* tool [19] allows for an integrated solution like the one presented in this paper, allowing both measure and trace simultaneously. However, osnoise does not consider the scheduling latency but a different metric, the operating system noise (i.e., OS-level interference occurring during execution).

Overall, no other tool provides an integrated solution to measure the scheduling latency in Linux while seamlessly providing insights into the root cause of high latency values by leveraging integrated tracing facilities. Furthermore, timerlat has been integrated into mainline Linux *real-time Linux analysis tool*, called *rtla*, and is thus available for every user running an updated version of Linux.

## 3 Background

Linux has four main execution contexts that can contribute to the scheduling latency: **(I)** *NMIs* (non-maskable interrupts); **(II)** *IRQs* (maskable interrupts); **(III)** *Softirqs*, (deferred IRQs); and **(IV)** regular *threads*. It is worth noting that the softirqs run at thread context in the fully preemptive mode (PREEMPT_RT). We refer to a generic execution context as a task. Each execution context is characterized by a different preemption behavior, described by the following rules, classified for each task type.

- **R-NMI:** The (per-CPU) NMI cannot be preempted. It can preempt any other task.
- **R-IRQs:** IRQs can be preempted by the NMI. It can preempt softirqs and threads but not other IRQs.
- **R-Softirqs:** Softirqs are preempted by IRQs and the NMI. It can preempt threads, but not other softirqs.
- **R-Threads:** Threads can be preempted by the NMI, IRQs, softirqs, and by other threads, according to the *scheduling policy*. It cannot preempt softirqs, IRQs, and the NMI.

This preemption behavior is well-known from prior work (e.g., see [20]) and is of primary importance for understanding the tool's behavior. For the sake of completeness, we briefly summarize the Linux schedulers.

**Linux schedulers.** Linux has five schedulers, which are queried in order to determine the thread to be executed, thus forming a hierarchy of schedulers. At the top of the hierarchy, there is the *stop-machine* pseudo-scheduler, which runs kernel facilities. It is followed by the SCHED_DEADLINE [21] scheduler, a real-time scheduler implementing the Earliest Deadline First algorithm. After that, Linux provides a POSIX fixed-priority real-time scheduler, that implements the SCHED_FIFO and SCHED_RR scheduling classes. The last two schedulers are then, in order, the completely fair scheduler (CFS) (or, EEVDF, Earliest Eligible Virtual Deadline First, in Linux kernel versions from 6.6), used for general-purpose activities (implemented by the *SCHED_OTHER* scheduling class), and the *IDLE* scheduler, which always returns the idle thread when there is no other thread to execute.

**Linux trace.** Linux has an advanced set of tracing methods, which are mainly applied in the runtime analysis of kernel latencies and performance issues. The most popular tracing methods are the *function tracer* that enables the trace of kernel functions, and the *tracepoint* that enables the tracing of hundreds of events in the system, like the *wakeup* of a new thread or the occurrence of an interrupt. An essential characteristic of the Linux tracing feature is its efficiency. Nowadays, almost all Linux-based OSes have these tracing methods enabled and ready to be used in production kernels. Indeed, these methods have nearly zero overhead when disabled, thanks to the extensive usage of runtime code modification techniques that allow for a greater efficiency than using conditional jumps when tracing is disabled. The tracing interface is available via a pseudo-file system that can be used via shell or via special libraries used by programs like *perf* and *trace-cmd*.

## 4 Linux Scheduling Latency Analysis

This section introduces the timerlat tool. First, it focuses on the tool components. Then, it discusses how to use timerlat to measure the scheduling latency. Finally, it discusses the timerlat tracing features. In the examples and experiments provided next in the paper, we use *rteval* [22], a tool used by Linux practitioners for performance tuning, which starts and stops a typical workload for experimentation purposes. These workloads are a CPU-bound scheduler benchmark called *hackbench* and a Linux kernel compiling process, which is CPU, memory, and IO bound. More details on *rteval* [22] are discussed in Section 7.

### 4.1 Timerlat components

Timerlat consists of two components: the *timerlat tracer* and the *rtla interface*, shown in Fig. 2.

The timerlat tracer is an *in-kernel component*. In-kernel processing allows for reduced overhead in tracing, leveraging lockless synchronization (*ftrace*'s lockless ring buffer) and reducing the amount of tracing data. The tracer manages the timer handling and, optionally, configures the in-kernel workload used to measure the latency. The timerlat

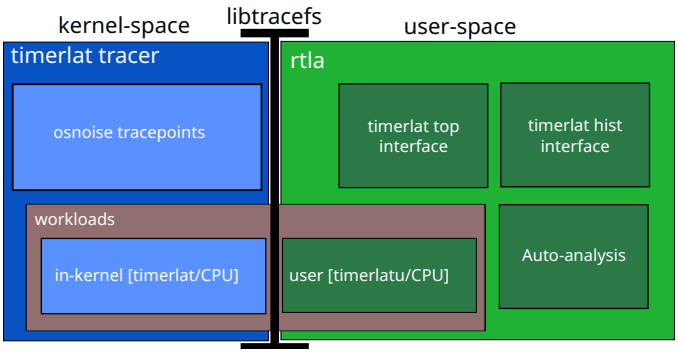



Fig. 2: Timerlat Architecture

tracer leverages the *osnoise tracepoints* [19] that reports the relevant latency values, also decomposed in several sub-components, as discussed in Section 5. For each task type (see Section 3), timerlat leverages a corresponding tracepoint to obtain execution time values free from any nested interference. For example, the *osnoise:thread_noise* tracepoint is free from any NMI, IRQ, and softirq interference (recall the **R-Threads** preemption rule from Section 3).

Therefore, the four tracepoints *osnoise:nmi_noise*, *osnoise:irq_noise*, *osnoise:softirq_noise*, and *osnoise:thread_noise* provides net values that are already discounted by any other nested interference that a task could face according to rules **R-NMI**, **R-IRQs**, **R-Softirqs**, and **R-Threads**.

The *rtla* (real-time Linux analysis) suite provides a benchmark-like interface for timerlat, accessing the tracing interface using *libtracefs* library, like *perf* and *trace-cmd*. The *rtla timerlat* tool sets up, collects, and parses trace data with either a top-like or a histogram interface. The tool also offers an interface to enable and collect advanced tracing features of Linux, such as *tracepoints* and histograms. Furthermore, it provides the *auto-analysis* functionality for debugging long latency values. Finally, it allows running timerlat on top of user-space workloads, which users often prefer.

### 4.2 Measuring the latency with timerlat

Fig. 3 shows the output of the *top interface* of timerlat with user-space workload. It is immediately possible to notice that three latencies are reported: the IRQ Timer (**IRQ for short**), the Thread Timer (**Thr**), and the Ret user Timer latency (**Usr**).

Similar to cyclictest, discussed in Section 1, timerlat provides an interface accessible by a measurement thread. It performs the following operations, which implement the measurement:

1) First, it sets up a timer at a known time instant $t_w$ and suspends the thread.
2) Then, the kernel manages the timer and services its expiry with an IRQ handler, which will run at a certain time $t_{\text{IRQ}}$, for which a timestamp value is obtained.
3) The IRQ handler awakens the measurement thread, which then reads the current time $t_{\text{Thr}}$ when it runs.
4) If a user-space workload is used, the tracer also reports the "return from user-space" latency, which is obtained by reading the timestamp at time $t_{\text{Usr}}$ when such workload is put back in execution by the in-kernel *timerlat* measurement thread.

```
$ sudo timerlat -u
                              Timer Latency
  0 00:01:01   |          IRQ Timer Latency (us)        |        Thread Timer Latency (us)      |    Ret user Timer Latency (us)
CPU COUNT      |      cur        min        avg        max |      cur        min        avg        max |      cur        min        avg        max
  0 #60005     |        1          0          1         16 |        6          2          5         20 |        9          2          8         29
  1 #60005     |        1          0          1         15 |        7          2          7         19 |       13          3         12         27
  2 #60003     |        1          0          1         13 |        8          2          7         27 |       12          3         11         32
  3 #60004     |        1          0          1         17 |        8          2          8         22 |       13          3         13         29
  4 #60004     |        0          0          1         14 |        7          3          6         22 |       11          4         11         26
  5 #59999     |        1          0          1         14 |        7          2          7         26 |       12          3         12         31
  6 #60003     |        0          0          1         19 |        6          2          7         24 |       11          3         12         28
  7 #60003     |        1          0          1         15 |        7          2          8         33 |       12          3         13         45
```

Fig. 3: The *rtla timerlat* top interface.

The previous operations define the three following latency metrics:

- **(IRQ)** The first latency value $L_{\text{IRQ}} = t_{\text{IRQ}} - t_w$ is the difference between the known timer expiration value and the timestamp of a special IRQ handling function added by the tracer timer at the beginning of the handler.
- **(Thr)** The second latency value $L_{\text{Thr}} = t_{\text{Thr}} - t_w$ is analogous, but measured concerning the thread and using the corresponding timestamp.
- **(Usr)** Finally, the third latency value is measured when execution comes back to the userspace component, occurring at time $t_{\text{Usr}}$. The corresponding latency is computed as $L_{\text{Usr}} = t_{\text{Usr}} - t_w$.

The three latency values are graphically represented in Fig. 4. As detailed in the following, one of the key differences between timerlat and cyclictest is that cyclictest just measures the Usr latency, and it provides no insights on the root cause of a latency value. Different, timerlat integrates the latency measurements with lightweight tracing, offering detailed traces that help practitioners understand the cause behind a latency value. This is particularly useful when the latency value is too high for the use case, and the system needs to be tuned or modified to avoid it. Without *timerlat*, practitioners would need to measure the Usr latency separately with *cyclictest* and set up a separate tracing session - typically combining standard and custom-tailored tracepoints. This approach complicates the tracing process and does not provide insights into IRQ and Thr latency. In fact, the ability to easily distinguish between IRQ and Thr latency is invaluable for root cause analysis of latency violations, i.e., when the latency reported by timerlat exceeds a threshold that can be configured in the tool, as extensively discussed later in Section 5. It allows practitioners to focus selectively on events that occurred either before (or during) the servicing of the IRQ or after the IRQ, which might have delayed the selection and execution of the Thr component. Moreover, not only *timerlat* provides integrated latency measurements and superior tracing information, but it does that with negligible overhead, as shown in Section 7.

The current, minimum, and maximum registered values are shown by the tool for each type of latency.

Timerlat is compatible with all the schedulers available to the user, i.e., it can configure the measurement thread with any of them. However, the default option and standard practice is to run it as a SCHED_FIFO thread at priority *95* (i.e., a "reasonably high" priority

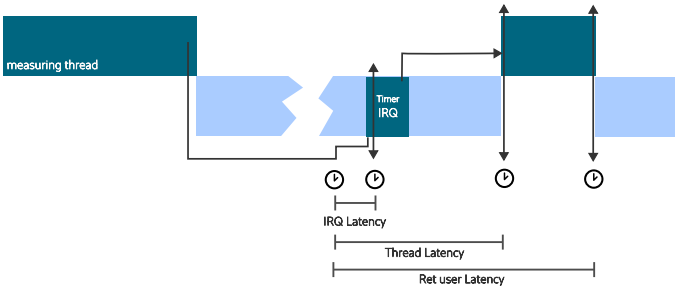



Fig. 4: *IRQ*, *Thread* (Thr) and *Ret user* (Usr) latency measured by *timerlat*.

```
# RTLA timerlat histogram
# Time unit is microseconds (us)
# Duration:   0 00:01:01
Index   IRQ-000   Thr-000   Usr-000   IRQ-001   Thr-001   Usr-001
0         56523         0         0     35696         0         0
1          3476     53462         0     24302      1607         0
2             1      6507     57606         2     57868     51600
3             0        30      2297         0       485      7414
4             0         1        87         0        22       888
5             0         0         9         0         9        64
6             0         0         1         0         9        15
7             0         0         0         0         0         3
8             0         0         0         0         0         6
9             0         0         0         0         0         9
10            0         0         0         0         0         1
over:         0         0         0         0         0         0
count:    60000     60000     60000     60000     60000     60000
min:          0         1         2         0         1         2
avg:          0         1         2         0         1         2
max:          2         4         6         2         6        10
```

Fig. 5: The *rtla timerlat* hist interface.

typically used by Linux practitioners). This can be configured with the `-P` option of timerlat, which needs to be detailed with the parameter, for example, `r:prio`, `f:prio` to priority *prio* with the SCHED_OTHER, SCHED_RR, or SCHED_FIFO schedulers, respectively, and with the parameter `d:runtime[us|ms]:period[us|ms]` when using SCHED_DEADLINE. Timerlat is also compatible with all the preemption models of Linux, including PREEMPT_RT.

In addition to the top interface, timerlat provides the *hist interface* that reports, for each interfering task, the number of occurrences of each latency value, with some statistics such as minimum, average, and maximum. An example is shown in Fig. 5, which reports the latencies of the first two CPUs (option `-c 0-1`) for 60 seconds (`-d 60`).

### 4.3 Timerlat tracing

The *tracepoints* are one of the basic bricks of the Linux kernel tracing. The tracepoints are points in the kernel code where

a probe can be attached to run a function. They are most commonly used to collect trace information. For example, *ftrace* registers a callback function to the tracepoints. These callback functions collect the data, saving it to a trace buffer. The data in the trace buffer can then be accessed by a tracing interface.

The tracepoints have been leveraged for many other use cases. For instance, to do runtime verification of the kernel [23].

The `timerlat` tracer leverages the current tracing infrastructure by re-using `osnoise` [19] tracepoints to collect information within kernel pre-processed information.

Linux already has tracepoints that intercept the entry and exit of IRQs, softirqs, and threads (i.e., `irq_handler_entry`, `irq_handler_exit`, `softirq_entry`, `softirq_exit`, `sched_switch`), and the `osnoise` tracepoint attaches a probe to all entry and exit events and uses it to:

1) account for the number of times each class of tasks added blocking or interference for the workload;
2) to compute the execution time of the current interfering task;
3) to subtract the noise occurrence duration of a preempted noise occurrence by leveraging the rules discussed in Section 3.

*timerlat* uses those tracerpoints, with some additional changes to optimize the output for its use-case.

To this end, timerlat performs the following operations:

1) print thread blocking and interference and softirq interference only when the thread is already awakened;
2) compute the execution time of the blocking task from the activation time of the workload (during the timerlat IRQ);
3) capture a *stracktrace* at the timerlat IRQ time, to print information about the state of the blocking thread when the interruption was served.

Fig. 6 shows a code snippet related to 2) and 3). First, it shows how the IRQ latency is obtained and stored in the `diff` variable by subtracting the expected timer wakeup time from the current timestamp, as discussed in Section 4.2. Second, it illustrates how the stacktrace is stored at `IRQ_CONTEXT` level, so that it is available for printing in case of a latency violation later on.

Fig. 7 shows an example of the *timerlat* tracer output.

In the first line, it is possible to see the *timerlat* tracer output with information about the IRQ latency. The second line shows the *osnoise:irq_noise* reporting the execution time of the IRQ handler. The third line shows the *blocking thread* noise execution time from the IRQ occurrence, providing an output that can be leveraged directly by the user (in kernel computation). Then the *timerlat* thread latency is printed, followed by a stack trace of the blocking thread at the moment of the timerlat IRQ: this includes the data that was collected in Fig. 6.

### 4.4 RTLA Timerlat tracing

*rtla timerlat* is a front end for general tracing. The `-t` option enables a tracing session, including timerlat and osnoise events enabled. When the `-t` option is set, if a threshold on the IRQ or thread latency is also set using the `-i` or `-T` options (by specifying the threshold in microseconds), timerlat copies the content of the trace buffer to the timerlat_trace.txt file when either the IRQ or thread latency thresholds are surpassed, respectively. timerlat can enable any Linux trace events by enabling the `-e` ⟨*event*⟩. Events can also be filtered and triggered using the `-filter` ⟨*filter*⟩ and `-trigger` ⟨*trigger*⟩ options. The `-trigger` option can be particularly useful to enable the collection and saving of files of histograms, e.g., to understand the contribution of each type of task to the latency, being a step towards discovering the hypothetical worst-case latency.

The *timerlat tracer* and *rtla timerlat* are integral parts of the Linux kernel, and a complete list of options is provided with Linux kernel documentation.

```
/*
 * timerlat_irq - hrtimer handler for timerlat.
 */
static enum hrtimer_restart timerlat_irq(struct hrtimer *timer)
{
        struct osnoise_variables *osn_var = this_cpu_osn_var();
        struct timerlat_variables *tlat;
        struct timerlat_sample s;
        u64 now;
        u64 diff;
        ...
        osn_var->thread.arrival_time = time_get();
        ...
        /*
         * Compute the current time with the expected time.
         */
        diff = now - tlat->abs_period;

        tlat->count++;
        s.seqnum = tlat->count;
        s.timer_latency = diff;
        s.context = IRQ_CONTEXT;

        trace_timerlat_sample(&s);
        ...
        if (osnoise_data.print_stack)
                timerlat_save_stack(0);
        ...
}
```

Fig. 6: A code snippet of the *timerlat* IRQ.

## 5 TIMERLAT AUTO-ANALYSIS

A key feature of *timerlat* is the auto-analysis. This feature is enabled by specifying the `-a` *threshold*, which stops the tracing if the latency crosses the threshold value through the tracing backend. When a latency sample crosses the threshold, the *rtla timerlat* parses the trace, seeking the root cause.

The auto-analysis splits the thread latency into several variables. Before proceeding to discuss them, we briefly recall some definitions we borrow from real-time scheduling theory [24]: *interference*, *blocking*, *release jitter*, and *execution time*:

- The *interference* is the delay caused by a higher-priority task delaying a lower-priority one.
- Conversely, the *blocking* time is the delay caused by a lower-priority task delaying a higher-priority one, which can exist, for example, due to synchronization delays or non-preemptive execution.
- The *release jitter* is a delay in the release of a task due to an external event, e.g., due to the hardware.
- Finally, the *execution time* is the time required to accomplish the goal.

```
            sh-18550   [010] d.h.. 20700.546662: #112225 context    irq timer_latency       852 ns
            sh-18550   [010] dNh1. 20700.546667: irq_noise: local_timer:236 start 20700.546661734 duration 5400 ns
            sh-18550   [010] d..3. 20700.546670: thread_noise:       sh:18550 start 20700.546662104 duration 2595 ns
    timerlat/10-5409    [010] ....1 20700.546671: #112225 context thread timer_latency      9268 ns
    timerlat/10-5409    [010] ...11 20700.546671: <stack trace>
=> timerlat_irq
=> __hrtimer_run_queues
=> hrtimer_interrupt
=> __sysvec_apic_timer_interrupt
=> sysvec_apic_timer_interrupt
=> asm_sysvec_apic_timer_interrupt
=> __memcg_slab_free_hook
=> kmem_cache_free
=> exit_mmap
=> __mmput
=> begin_new_exec
=> load_elf_binary
=> bprm_execve
=> do_execveat_common.isra.0
=> __x64_sys_execve
=> do_syscall_64
=> entry_SYSCALL_64_after_hwframe
```

Fig. 7: The *timerlat* tracer and *osnoise* tracepoints.

```
$ sudo timerlat -a 30
Timer Latency
0 00:00:01   |           IRQ Timer Latency (us)        |         Thread Timer Latency (us)
CPU COUNT    |      cur       min       avg       max |      cur       min       avg       max
0 #763       |        1         0         1         9 |        8         4         8        18
1 #763       |        1         0         1         8 |       12         4        12        21
2 #763       |        1         0         1         5 |       13         4        15        23
3 #763       |        1         0         1         8 |       16         4        14        21
4 #763       |       12         0         1        16 |       28         5        12        28
5 #763       |        1         0         1         8 |       12         4        11        22
6 #763       |       32         0         1        32 |       52         5        13        52
7 #763       |        0         0         1        11 |        7         5        12        20
rtla timerlat hit stop tracing
## CPU 6 hit stop tracing, analyzing it ##
IRQ handler delay:                              31.00 us (59.56 %)
IRQ latency:                                    32.17 us
Timerlat IRQ duration:                          9.57 us (18.38 %)
Blocking thread:                                8.77 us (16.84 %)
objtool:1164402                                 8.77 us
Blocking thread stack trace
-> timerlat_irq
-> __hrtimer_run_queues
-> hrtimer_interrupt
-> __sysvec_apic_timer_interrupt
-> sysvec_apic_timer_interrupt
-> asm_sysvec_apic_timer_interrupt
-> _raw_spin_unlock_irqrestore
-> cgroup_rstat_flush_locked
-> cgroup_rstat_flush_irqsafe
-> mem_cgroup_flush_stats
-> mem_cgroup_wb_stats
-> balance_dirty_pages
-> balance_dirty_pages_ratelimited_flags
-> btrfs_buffered_write
-> btrfs_do_write_iter
-> vfs_write
-> __x64_sys_pwrite64
-> do_syscall_64
-> entry_SYSCALL_64_after_hwframe
------------------------------------------------------------------------
Thread latency:                                 52.05 us (100%)

Saving trace to timerlat_trace.txt
```

Fig. 8: The timerlat auto analysis.

Timerlat maps these well-known concepts to the sets of variables, as mentioned earlier. Four of them are related to the interference: `NMI interference`, `IRQ interference`, `Softirq interference`, and `Thread interference`. Interference occurs according to the preemption rules reported in Section 3.

Another variable is instead related to the blocking time: `Thread blocking`. For example, a thread can suffer blocking from another (lower-priority) thread because it disabled preemption. All these variables measure interference and blocking suffered by the measuring thread and are relative to the completion time of the timer IRQ handler.

The time difference elapsed between the expected absolute time in which the timer should fire and the actual start of the timer IRQ is captured by the `IRQ handler delay` variable, reported by timerlat (see Fig. 8).

The `IRQ handler delay` variable can include interference and blocking time experienced by the timer IRQ handler (e.g., due to another higher-priority IRQ or a lower-priority IRQ that already started executing non-preemptively) and release jitter (e.g., an IRQ handle can be delayed if it begins in an idle CPU that needs to leave a deep idle state).

The `IRQ latency`, which is instead based on the timestamp taken by the the IRQ and hence also includes the time spent inside the IRQ, is used by timerlat to obtain the `IRQ`

handler delay.

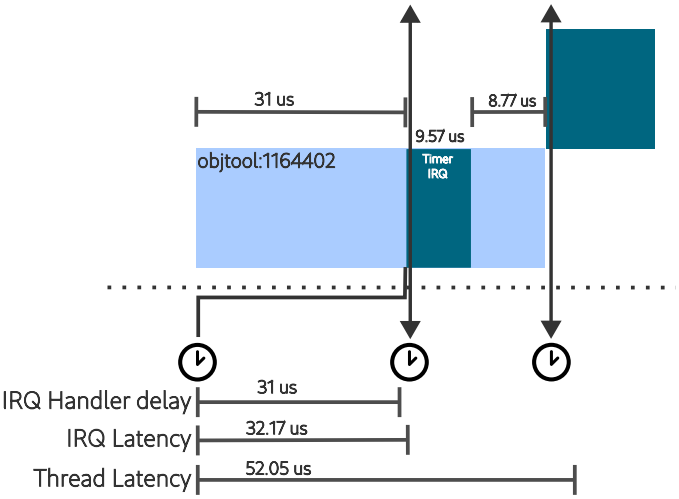



Fig. 9: Latency timeline for the example of Fig. 8

.

The execution time of the timer IRQ handler only partially influences the `IRQ latency`. This is because the latency is measured and printed right at the beginning of the handler. It has, however, a more substantial impact on the `Thread latency` variable reported by the auto analysis. The execution time of the handler is also reported by the auto analysis in the `Timerlat IRQ duration` variable. More formally, the IRQ latency can be expressed with the following summation:

$$L_{\text{IRQ}} = I^{\text{HP-IRQ}}_{[t_w,t_{\text{IRQ}})} + B^{\text{LP-IRQ}}_{[t_w,t_{\text{IRQ}})} + I^{\text{NMI}}_{[t_w,t_{\text{IRQ}})} + B^{\text{THR}}_{[t_w,t_{\text{IRQ}})} + J + e'_{\text{IRQ}},$$

in which:

- $I^{\text{HP-IRQ}}_{[t_w,t_{\text{IRQ}})}$ represents the interference due to high-priority IRQs to the timerlat IRQ, in the time interval $[t_w, t_{\text{IRQ}})$, defined in Section 4.2;
- $B^{\text{LP-IRQ}}_{[t_w,t_{\text{IRQ}})}$ represents the blocking due to low-priority IRQs (e.g., disabling interrupts) on the timerlat IRQ;
- $I^{\text{NMI}}_{[t_w,t_{\text{IRQ}})}$ represents the interference due to the NMI in $[t_w, t_{\text{IRQ}})$, which always has higher priority than IRQs (see Section 3);
- $B^{\text{THR}}_{[t_w,t_{\text{IRQ}})}$ is the blocking time due to threads;
- $J$ is the release jitter;
- $e'_{\text{IRQ}}$ is the partial execution time of the timerlat IRQ handler, as already discussed.

SoftIRQs are executed either in interrupt or thread context depending on the preemption model.

Fig. 10 shows an example in which the analysis of a high IRQ latency spike using the timerlat stacktrace reveals that its root cause is the release jitter. Indeed, it is possible to notice that the blocking thread, i.e., the one that caused the over-threshold latency sample, is the CPU 9 idle thread `swapper/9:0`. Therefore, it is possible to assume that the causes of the delays are external factors (e.g., exiting a CPU idle state) when the IRQ is delayed and the CPU is idle. This integrated root cause analysis would not have been possible with previous tools such as cyclictest, and it would have required complicated tracing sessions based on both

```
## CPU 9 hit stop tracing, analyzing it ##
IRQ handler delay: (exit from idle)     39.01 us (76.59 %)
IRQ latency:                            40.49 us
Timerlat IRQ duration:                  5.85 us (11.49 %)
Blocking thread:                        3.99 us (7.83 %)
                swapper/9:0  3.99 us
Blocking thread stack trace
-> timerlat_irq
-> __hrtimer_run_queues
-> hrtimer_interrupt
-> __sysvec_apic_timer_interrupt
-> sysvec_apic_timer_interrupt
-> asm_sysvec_apic_timer_interrupt
-> pv_native_safe_halt
-> default_idle
-> default_idle_call
-> do_idle
-> cpu_startup_entry
-> start_secondary
-> __pfx_verify_cpu
------------------------------------------------
Thread latency:  50.93 us (100%)

Max timerlat IRQ latency from idle: 40.49 us in cpu 9
```

Fig. 10: Auto analysis pointing to hardware-related jitter.

standard and custom tracepoints. Similar to the previous case, the Thr latency can be formalized as:

$$L_{\text{Thr}} = L_{\text{IRQ}} + I^{\text{IRQ}}_{[t_{\text{IRQ}},t_{\text{Thr}})} + I^{\text{NMI}}_{[t_{\text{IRQ}},t_{\text{Thr}})} + I^{\text{HP-THR}}_{[t_{\text{IRQ}},t_{\text{Thr}})} + B^{\text{LP-THR}}_{[t_{\text{IRQ}},t_{\text{Thr}})} + e.$$

Terms $I^{\text{IRQ}}_{[t_{\text{IRQ}},t_{\text{Thr}})}$, $I^{\text{NMI}}_{[t_{\text{IRQ}},t_{\text{Thr}})}$, and $I^{\text{HP-THR}}_{[t_{\text{IRQ}},t_{\text{Thr}})}$ refers to the interference due to all IRQs, NMI, and higher-priority threads on the timerlat measurement thread; $B^{\text{LP-THR}}_{[t_{\text{IRQ}},t_{\text{Thr}})}$ refers to the low-priority blocking; finally, $e = e''_{\text{IRQ}} + e'_{\text{Thr}}$ refers to the remaining execution time of the timerlat IRQ handler not accounted in $L_{\text{IRQ}}$ ($e''_{\text{IRQ}}$) and to the partial execution time of the timerlat measurement thread before taking the $t_{\text{Thr}}$ timestamp (term $e'_{\text{Thr}}$). The Usr latency can be written as $L_{\text{Usr}} = L_{\text{Thr}} + e''_{Thr}$, with $e''_{Thr}$ denoting the remaining part of execution time of the timerlat thread.

Similar performance optimization activities as those shown in Fig. 10 can also be performed by considering the Thr and Usr latency.

### 5.1 Understanding the causes of the IRQ latency

In the example of Fig. 8, the trace stopped because a `Thread latency` of 52.05 us was registered, surpassing the threshold of 30 us. The corresponding timeline representation of such a latency sample is shown in Fig. 9. Timerlat eases the root cause investigation by separating `Thread latency` and `IRQ latency`. From the stack trace, it can be noticed that the `objtool:1164402` thread called a cgroup operation (a feature of the Linux kernel that allows to control the amount of resources assigned to a group of tasks), that disabled IRQs on a `raw_spinlock_operation` (from line `_raw_spin_unlock_irqrestore`), causing the `IRQ handler delay`. One may further investigate the cause of such IRQ disabling operation, finding out that it is just a normal use case for the raw spinlock (e.g., by looking at the corresponding patch[2]). The same operation is also used to disable preemption: indeed, `objtool:1164402` also caused a thread blocking time of 8.77 us, reported by timerlat. Note that, in general, it is hard to distinguish

2. Linux kernel commit `b1e2c8df0f00 cgroup: use irqsave in cgroup_rstat_flush_locked()`

```
## CPU 18 hit stop tracing, analyzing it ##
IRQ handler delay:                        0.00 us (0.00 %)
IRQ latency:                              1.64 us
Timerlat IRQ duration:                    9.52 us (1.80 %)
Blocking thread:                          501.68 us (94.96 %)
        kworker/u40:0:306130              501.68 us
Blocking thread stack trace
-> timerlat_irq
-> __hrtimer_run_queues
-> hrtimer_interrupt
-> __sysvec_apic_timer_interrupt
-> sysvec_apic_timer_interrupt
-> asm_sysvec_apic_timer_interrupt
-> ZSTD_compressBlock_fast
-> ZSTD_buildSeqStore
-> ZSTD_compressBlock_internal
-> ZSTD_compressContinue_internal
-> ZSTD_compressEnd
-> ZSTD_compressStream2
-> ZSTD_endStream
-> zstd_compress_pages
-> btrfs_compress_pages
-> compress_file_range
-> async_cow_start
-> btrfs_work_helper
-> process_one_work
-> worker_thread
-> kthread
-> ret_from_fork
IRQ interference                          3.68 us (0.70 %)
        local_timer:236                   3.68 us
Softirq interference                      4.21 us (0.80 %)
        TIMER:1                           3.71 us
        RCU:9                             0.49 us
Thread interference                       6.21 us (1.17 %)
        migration/18:125                  6.21 us
------------------------------------------------
Thread latency:                           528.31 us (100%)

Max timerlat IRQ latency from idle: 10.34 us in cpu 12
Saving trace to timerlat_trace.txt
```

Fig. 11: A case in which the kernel causes a large latency.

between the interference from high-priority IRQs and blocking from other IRQs or threads disabling IRQs (as in this case). This is because the interrupt scheduling policy in the interrupt controller may be unknown in general. For this reason, timerlat does not provide a decomposition of the `IRQ latency`, which can be, however, investigated case-by-case by leveraging the insightful information contained in the stack trace.

### 5.2 Understanding the causes of the thread latency

Timerlat is mainly helpful in understanding the causes of thread latency. An example is in Fig. 11, in which a high thread latency (9.52 us, fourth line) follows a low IRQ latency value (1.64 us, third line). This example was run on a non-PREEMPT_RT Linux. The thread latency is mainly caused by a `btrfs` file system operation in the kernel, managed by the `kworker/u40:0:306130` (kworker threads are general worker threads used by the kernel to perform background tasks) for 501.68 us (see line "Blocking thread") since the kernel is non-preemptive, it causes blocking until a scheduling point is reached. The blocking also extended the window for capturing high-priority interference: indeed, the timerlat thread suffered interference from the local timer IRQ handler, 3.68 us, the timer and RCU softirqs (the latter are mechanisms used in the kernel to manage deferred processing of Read-Copy-Update - RCU - operations), 3.71 us and 0.49 us, respectively, and the migration thread that runs at the highest thread priority in the system (6.21 us). Again, thanks to the integrated measurement and tracing features of timerlat, Linux practitioners can more easily

```
now = get_time()
absolute_next_period = now + relative_period
wait_next_period(absolute_next_period) <-- hrtimer sleep
while (should_measure()) {
     now = get_time()
     diff = now - absolute_next_period
     if (diff > max_latency_threshold)
         exit and report
     collect measured latency (diff)
     absolute_next_period = absolute_next_period + relative_period
     wait_next_period(absolute_next_period)
}
```

Fig. 12: *timerlat* in a nutshell.

understand the root cause of a latency spike, which allows for a faster resolution of latency issues and time-to-market with respect to what was allowed before by cyclictest and separate tracing sessions.

## 6 TIMERLAT IMPLEMENTATION DETAILS

A key advantage of `timerlat` is its capability to differentiate between the different components that account for the total latency reported by the measuring workload. This feature is essential to eliminate wrong conjectures concerning the factors that caused such latency, leading the troubleshooting and debugging activities in the wrong directions and causing delays in resolving issues in expensive and critical systems.

As described in Section 4.2, the *timerlat* measuring thread uses a high-resolution timer to cause periodic wakeups and observe the latency of such events, essentially the pseudo-code depicted in Fig. 12. This is of course very similar to how existing tools (e.g., *cyclictest*) approach measuring wakeup latency. The key differentiating feature of *timerlat* is that it is capable of decomposing latency in three components: IRQ (`IRQ_CONTEXT`), Thread (`THREAD_CONTEXT`) and Ret user (`THREAD_URET`), which corresponds to the IRQ, Thr, and Usr latencies discussed in Section 4.2. Starting from the top (Ret user), each user-space thread issues a blocking read call onto a per-CPU file descriptor created by the kernel while initializing *timerlat* (`timerlat_fd`), and it then records the measured latency for `THREAD_URET` when it returns from such call. The in-kernel implementation of the read function will perform the bulk of the behavior detailed in Fig. 12, by getting the reference time for the periodic activation, sleeping until next period and reporting the measured latency for `THREAD_CONTEXT`. To complete the picture, instrumentation in the timer call back records the measured latency for `IRQ_CONTEXT`. In this way, `timerlat` is able to provide a complete and detailed picture of latency, enabling precise reporting within $\mu$s of granularity required by several use cases, such as, for example, network-function virtualization.

Furthermore, *timerlat* is designed to introduce minimal overhead to the system under analysis and being lightweight. This objective is pursued by leveraging the highly optimized tracing facilities inherited by the Linux kernel. These facilities are optimized through by the following mechanisms: **(i)** they operate primarily in kernel space to avoid costly context switches, **(ii)** they utilize per-CPU buffers to reduce locking overhead in multicore environments, and **(iii)** they employ ring buffer designs for efficient

data handling with minimal contention. Additionally, their integration into the kernel allows for direct access to low-level events with minimal latency, while features such as dynamic tracing enable targeted instrumentation, ensuring that only relevant events are captured. Being a tracer in itself, *timerlat* can perform pre-processing of collected data before storing it in the ftrace buffer, further reducing runtime computational demands and minimizing the potential for trace-related interference with system performance. These design choices make *timerlat* well-suited for performance-critical environments where accurate and unobtrusive latency measurements are essential.

## 7 Evaluation

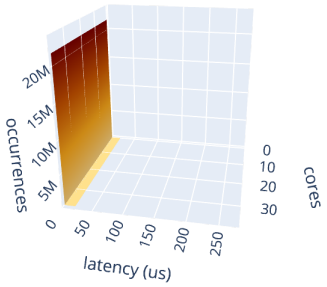
This section reports on an evaluation that has been performed to assess the latency values obtained by *timerlat* under different configurations. The evaluation has been carried out on a Dell PowerEdge 650 server with 40 Intel(R) Xeon(R) Platinum 8380 cores running at 2.30GHz, with Red Hat Enterprise Linux (RHEL) version 9.2. In the experiments, *timerlat* has been executed in user space. Several configurations have been tested. A first classification considers the usage of the PREEMPT_RT patch **(i)** RHEL with PREEMPT_RT and **(ii)** RHEL without PREEMPT_RT. For each of these two cases, the system has been configured to work **(a)** in isolation (defined as the *isol* scenario hereafter), meaning that interfering workload has been restricted to two housekeeping cores with *timerlat* measuring threads running on separate isolated cores and, **(b)** with interfering workloads (defined as the *workload* scenario hereafter), meaning that interfering workload and timerlat measuring threads has been co-scheduled on all the available cores. Interfering workloads consists of running a scheduler benchmark, *hackbench*, (CPU bound) while compiling the Linux kernel (CPU, memory and IO bound). The *rteval* [22] tool manages (starts and stops) such workloads and controls the execution of measuring threads (*timerlat* or *cyclictest*, which is used for comparison in one of the experiments). More information about the workloads and how to replicate the setup is available in Section 7.2. For each of the four configurations, i.e., (i-a), (i-b), (ii-a), (ii-b), an experiment of 6 hours has been performed, collecting the IRQ, Thread, and Ret user latencies.

### 7.1 Evaluation Results

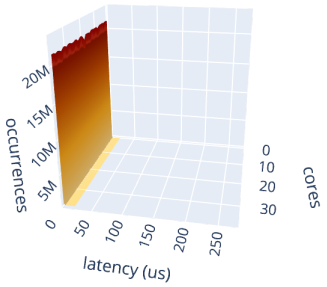
**Latency occurrences.** Fig. 13 and Fig. 14 shows the number of occurrences of IRQ, Thread, and Ret user (IRQ, Thr, and Usr, for short) latency samples for each of the $40$ cores in the range $[0, 250]$ $\mu$s in the *isol* configurations under RHEL (first row, without PREEMPT_RT) and RHEL-RT (second row, with PREEMPT_RT). The figures are 3D histograms with fixed parameters that report the occurrences of latency values (on the horizontal axis) for each core in the platform (40 cores for the *workload* configuration and 38 for the *isol* configuration since the latter two cores are used for housekeeping). Different levels of yellow represent an increasing number of latency occurrences: darker yellow corresponds to a higher number of occurrences; light yellow corresponds to a low level of occurrences; finally, white corresponds to no occurrences of a certain latency level. Fig. 13 targets the *isol* configuration, and Fig. 14 the *workload* configuration. Fig. 13 shows that both RHEL and RHEL-RT achieve small latency values in the *isol* configuration, with values generally below $20$ $\mu$s. Instead, Fig. 14 illustrates how in the presence of interfering workloads (*workload* configuration), latency values are generally much different. With RHEL (without PREEMPT_RT), the latency distribution covers the whole interval $[0, 250]$ $\mu$s, with considerably higher observed latencies. Under RHEL-RT with PREEMPT_RT, the latency values are much smaller, with the latency distribution covering the interval $[0, 60]$ $\mu$s. This is because one of the main objectives of the PREEMPT_RT patchset is to provide bounded wake-up latencies, as those measured by timerlat. Finally, by comparing IRQ, Thr, and Usr latencies, it can be observed that in all the configurations, IRQ latencies are generally much smaller (more occurrences are registered for low latency values), while Thr and Usr latencies are slightly higher. Furthermore, Usr latencies are higher than Thr latencies. This whole trend is expected because the measurement interval of the Usr latency is longer than in Thr's case, which is, in turn, longer than in IRQ's case. In all the configurations, the behavior with respect to the cores is consistent, with similar latencies registered across them. However, it is worth noting that all the plots refer to the interval $[0, 250]$ $\mu$s, which is the default interval used by timerlat to collect latency samples. However, some configurations led to higher latencies. This information can still be interpreted with timerlat thanks to the maximum aggregate latency provided by timerlat.

**Maximum latency.** Fig. 15 and Fig. 16 show the maximum IRQ, Thr, and Usr latencies obtained in the *workload* and *isol* configuration, respectively, with RHEL (violet bars) and with RHEL-RT (green bars), considering the same 6 hours runs as in the previous plots. In Fig. 15, the y-axis is in the log scale. From Fig. 15, it can be observed how RHEL-RT is effective in limiting the maximum latency, showing values up to $58$ $\mu$s. Instead, the RHEL configuration leads to much higher maximum latencies, up to $4454$, $26223$, and $26229$ $\mu$s for IRQ, Thr, and Usr latency, respectively. As expected, Fig. 16 shows lower values of latency in all the configurations. RHEL-RT achieves maximum latency values of $10$ $\mu$s, for both IRQ, Thr, and Usr latency. RHEL also achieves generally good performance, but it has a spike of 38 $\mu$s of latency. By comparing Fig. 15 and Fig. 16, we highlight a key message to achieve very low latencies: both using PREEMPT_RT and a proper system configuration in which interfering workloads are restricted to separated cores are needed. Indeed, by using PREEMPT_RT only (RHEL_RT workload configuration), the maximum latency still reaches $58$ $\mu$s. Instead, by leveraging housekeeping cores for interfering workloads, the maximum latency can be reduced to $10$ $\mu$s only, thus making it practical to use Linux for low-latency use cases, such as vRANs. Timerlat enables reaching such a low latency figures thanks to the root-cause analysis features it provides, discussed in Section 5.

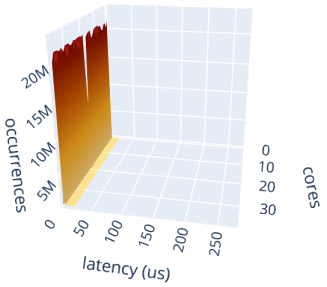
**Overhead and comparison with cyclictest.** To empirically verify that *timerlat* adds a negligible overhead for measuring latencies, a comparison (using the PREEMPT_RT kernel) has been made against a different legacy tool implementing the



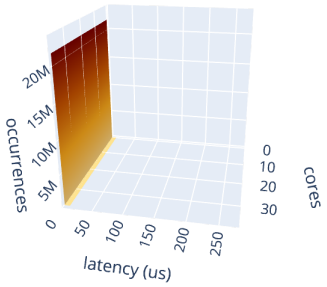

(a) RHEL - isol - IRQ latency



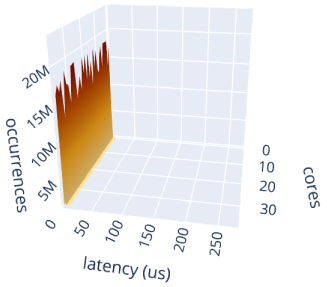

(b) RHEL - isol - Thr latency



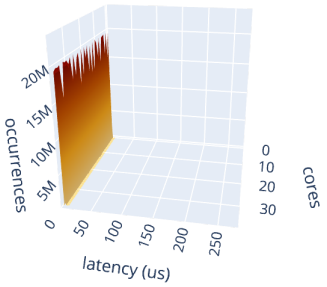

(c) RHEL - isol - Usr latency




(d) RHEL-RT - isol - IRQ latency

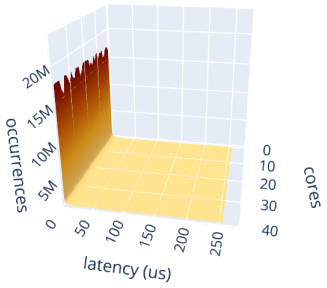



(e) RHEL-RT - isol - Thr latency

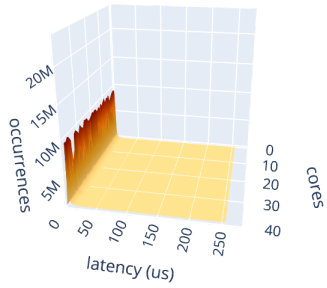



(f) RHEL-RT - isol - Usr latency

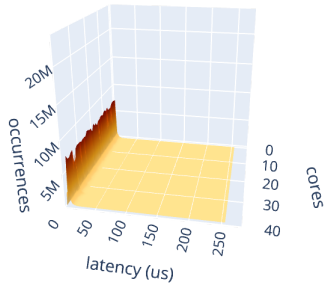
Fig. 13: Occurrences of latency values (IRQ, Thr, and Usr, in $\mu$s) obtained by running Timerlat for 6 hours in the *isol* configuration on 40 cores running RedHat Enterprise Linux with (second row) and without (first row) PREEMPT_RT.



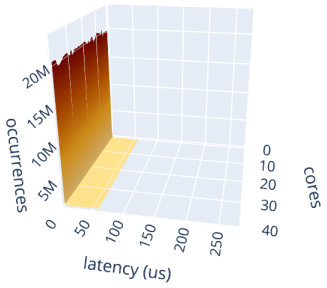

(a) RHEL - workload - IRQ latency



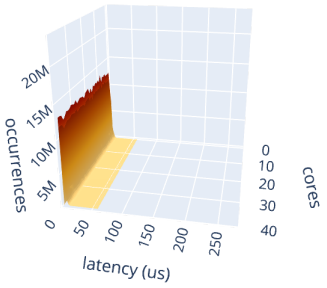

(b) RHEL - workload - Thr latency



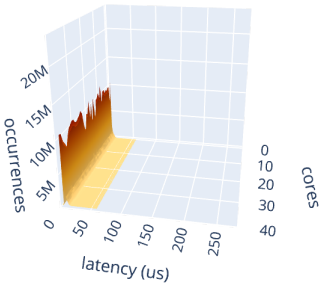

(c) RHEL - workload - Usr latency




(d) RHEL-RT - workload - IRQ latency




(e) RHEL-RT - workload - Thr latency




(f) RHEL-RT - workload - Usr latency

Fig. 14: Occurrences of latency values (IRQ, Thr, and Usr, in $\mu$s) obtained by running Timerlat for 6 hours in the *workload* configuration on 40 cores running RedHat Enterprise Linux with (second row) and without (first row) PREEMPT_RT.

| | *timerlat* | *cyclictest* |
|---|---|---|
| workload | max: 102us / avg: 3.9us | max: 109us / avg: 3.2us |
| isol | max: 8us / avg: 2.0us | max: 7us / avg: 2.1 |

TABLE 1: Comparison with cyclictest

same functionality, which was, de-facto, the state-of-the-art tool used by Linux practitioners before: *cyclictest*, discussed in Section 1. Timerlat measures the same quantity (Usr latency) of cyclictest: in addition, it also reports on IRQ and Thr latencies and provides integrated tracing features, as extensively discussed in the paper. Therefore, the overhead introduced by timerlat with respect to the previous de-facto standard practice is measured by comparing the two. Two setup scenarios (workload and isol) for each tool have been run for 3 hours on 40 cores. Average and maximum are computed over all the values collected on all 40 cores, amounting to over one billion samples per configuration. We run a three-hour experiment considering both the workload and isol configurations. The results are reported in Table 1 and show that both the reported average and maximum latencies have negligible differences: the difference in terms of average latency is below 1 $\mu$s in all configurations; the difference in terms of maximum latency is 1 $\mu$s for the isol configuration and 7 $\mu$s in the workload configuration (in which the maximum latency is more variable due to the interfering workload). The experiment shows that timerlat introduces a negligible overhead while providing more advanced features, such as extensive support to find the root causes of the latency, as discussed in Section 5.1 and Section 5.2.

### 7.2 Replicating the Experiments

All the software used for the experimental evaluation in this section is open-source and publicly available through repositories or as packages distributed depending on the Linux distribution. Replicating the experiments is therefore straightforward from a setup perspective. In this evaluation, the low-latency profile BIOS setting was used, which, among other optimizations, configures the Performance Energy Efficient Policy, disables C states (idle states), and sets the memory to maximum performance. However, system tuning at the BIOS level is beyond the scope of this paper.

The key steps for running similar experiments involve:

1) Obtaining or compiling a stock (i.e., non PREEMPT_RT) and a PREEMPT_RT kernel.
2) Tune the system for low latency using the `tuned` tool, using the `realtime` profile;
3) Run the experiments using timerlat and cyclictest through rteval [22], a benchmarking tool designed to evaluate a Linux system's real-time performance and, in particular, to measure the system's ability to handle real-time tasks with minimal latency.
4) Collect the results.

Detailed steps for conducting the experiments are available at this link [25].

## 8 Conclusions

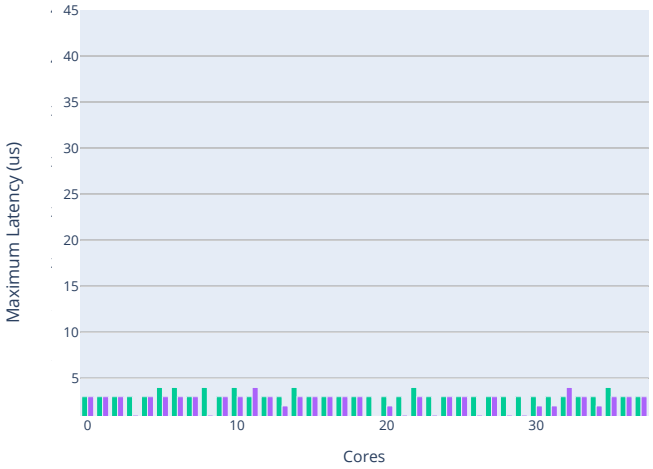
This paper presented timerlat, a new tool for measuring the Linux scheduling latency while allowing us to understand


(a) IRQ latency

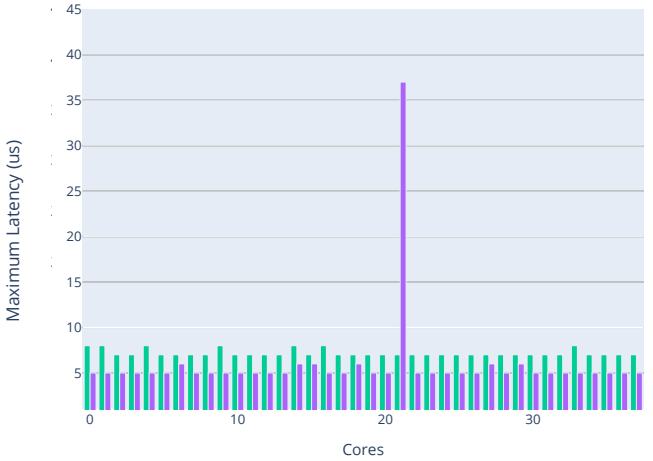
(b) Thr latency

(c) Usr latency

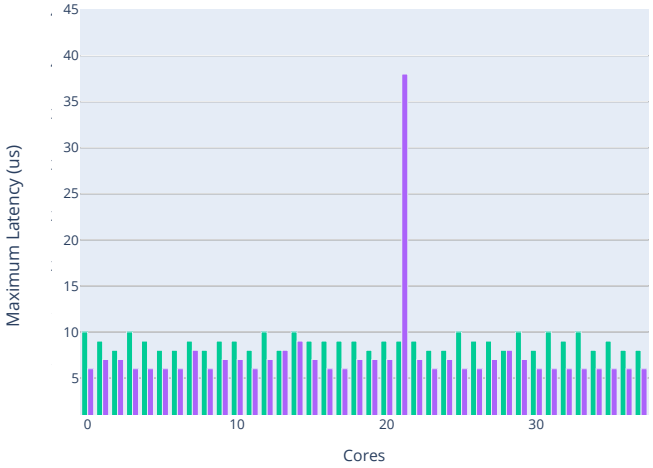
Fig. 15: Bar chart of the maximum latency values (logscale) obtained by running Timerlat for 6 hours in the *workload* configuration on 40 cores running RHEL (in violet) and RHEL-RT (in green).




(a) IRQ latency




(b) Thr latency




(c) Usr latency


Fig. 16: Bar chart of the maximum latency values obtained by running Timerlat for 6 hours in the *isol* configuration on 40 cores running RHEL (in violet) and RHEL-RT (in green).

the root causes of a latency spike through its integration with the Linux tracing infrastructure. Thanks to timerlat, troubleshooting high latency values is also possible for non-experts by leveraging timerlat's user-friendly interface. The tool is a part of the mainline Linux kernel, thus being easily accessible to all Linux users. Future work includes the extension of timerlat to support hardware-related features, e.g., by integrating information from the performance counters.

## Acknowledgment

*In memory of Daniel Bristot De Oliveira: a great professional, researcher, and friend.*

This work has been partially supported by the European Union's Horizon Europe Framework Programme project NANCY under the grant agreement No. 101096456, and the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

## References

[1] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

[2] P. Rad, R. V. Boppana, P. Lama, G. Berman, and M. Jamshidi, "Low-latency software defined network for high performance clouds," in *2015 10th System of Systems Engineering Conference (SoSE)*. IEEE, 2015, pp. 486–491.

[3] N. Bhushan, T. Ji, O. Koymen, J. Smee, J. Soriaga, S. Subramanian, and Y. Wei, "5g air interface system design principles," *IEEE Wireless Communications*, vol. 24, no. 5, pp. 6–8, 2017.

[4] R. Hat, "What is nfv?" https://www.redhat.com/en/topics/virtualization/what-is-nfv, 2021.

[5] Intel, "Flexran," https://github.com/intel/FlexRAN, 2021.

[6] L. Mandyam and S. Hoenisch, "Ran workload performance is equivalent on bare metal and vsphere," https://blogs.vmware.com/telco/ran-workload-performance-tests-on-vmware-vsphere/, 2021.

[7] T. Cucinotta, L. Abeni, M. Marinoni, R. Mancini, and C. Vitucci, "Strong temporal isolation among containers in openstack for nfv services," *IEEE Transactions on Cloud Computing*, vol. 11, no. 1, pp. 763–778, 2021.

[8] A. Parvat, J. Chavan, S. Kadam, S. Dev, and V. Pathak, "A survey of deep-learning frameworks," in *2017 International Conference on Inventive Systems and Control (ICISC)*. IEEE, 2017, pp. 1–7.

[9] P. Denzler, J. Ruh, M. Kadar, C. Avasalcai, and W. Kastner, "Towards consolidating industrial use cases on a common fog computing platform," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2020.

[10] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of linux," in *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, Sep. 2002.

[11] F. Reghenzani, G. Massari, and W. Fornaciari, "Mixed time-criticality process interferences characterization on a multicore linux system," in *2017 Euromicro Conference on Digital System Design (DSD)*, Aug 2017.

[12] B. Herzog, L. Gerhorst, B. Heinloth, S. Reif, T. Hönig, and W. Schröder-Preikschat, "Intspect: Interrupt latencies in the linux kernel," in *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, Nov 2018, pp. 83–90.

[13] P. Regnier, G. Lima, and L. Barreto, "Evaluation of interrupt handling timeliness in real-time linux operating systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, p. 52–63, 2008.

[14] F. Cerqueira and B. Brandenburg, "A comparison of scheduling latency in linux, preempt-rt, and litmus rt," in *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. SYSGO AG, 2013, pp. 19–29.

[15] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, "Linux preempt-rt vs commercial rtoss: How big is the performance gap?" *GSTF Journal on Computing*, vol. 3, no. 1, 2013.
[16] D. B. de Oliveira, D. Casini, R. S. de Oliveira, and T. Cucinotta, "Demystifying the real-time linux scheduling latency," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
[17] S. Rostedt, "Finding origins of latencies using ftrace."
[18] "Kutrace: Lightweight tracing for performance analysis," https://github.com/dicksites/KUtrace, 2024.
[19] D. B. de Oliveira, D. Casini, and T. Cucinotta, "Operating system noise in the linux kernel," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 196–207, 2023.
[20] D. B. de Oliveira, R. S. de Oliveira, and T. Cucinotta, "A thread synchronization model for the preempt_rt linux kernel," *Journal of Systems Architecture*, p. 101729, 2020.
[21] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the Linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016.
[22] Kernel.org Community, "rteval - a tool for real-time performance evaluation," https://git.kernel.org/pub/scm/utils/rteval/rteval.git, accessed: 2024-12-03.
[23] D. B. de Oliveira, T. Cucinotta, and R. S. de Oliveira, "Efficient formal verification for the linux kernel," in *International Conference on Software Engineering and Formal Methods*. Springer, 2019, pp. 315–332.
[24] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*. Springer, 2011.
[25] "Timerlat experiments," https://gist.github.com/jlelli/be016d87b7868f8b0f5734a9ed9a81cd, accessed: 2024-12-02.

**Daniel Bristot De Oliveira** Daniel Bristot de Oliveira (Red Hat) has a joint Ph.D. degree in Automation Engineering from UFSC (BR) and Embedded Real-Time systems from Scuola Superiore Sant'Anna (IT). Currently, he is Senior Principal Software Engineer at Red Hat, working on developing the real-time features of the Linux kernel. Daniel helps in the maintenance of real-time related tracers and toolings for the Linux kernel and the SCHED_DEADLINE. He is an affiliate researcher at the Retis Lab, and researches real-time and formal methods. He is an active member of the real-time academic community, participating in the technical program committee of academic conferences, such as RTSS, RTAS, and ECRTS.

**Daniel Casini** (IEEE Member) Daniel Casini is Assistant Professor at the Real-Time Systems (ReTiS) Laboratory of the Scuola Superiore Sant'Anna of Pisa. He graduated (cum laude) in Embedded Computing Systems Engineering, a Master degree jointly offered by the Scuola Superiore Sant'Anna of Pisa and University of Pisa, and received a Ph.D. in computer engineering at the Scuola Superiore Sant'Anna of Pisa (with honors), working under the supervision of Prof. Alessandro Biondi and Prof. Giorgio Buttazzo. In 2019, he has been visiting scholar at the Max Planck Institute for Software Systems (Germany). His research interests include software predictability in multi-processor systems, schedulability analysis, synchronization protocols, and the design and implementation of real-time operating systems and hypervisors.

**Juri Lelli** Juri Lelli (Red Hat) is a Senior Principal Software Engineer at Red Hat working on the RHEL for Real-Time kernel and related technologies. He is among the original authors of the SCHED_DEADLINE scheduling policy in Linux and a Linux kernel scheduler maintainer. He has a PhD degree from the Scuola Superiore Sant'Anna of Pisa, Italy (ReTiS Lab). His research area covered Real-Time systems, Real-Time Operating systems and Scheduling algorithms.

**Tommaso Cucinotta** has a MSc in Computer Engineering from University of Pisa (Italy), and a PhD in Computer Engineering from Scuola Superiore Sant'Anna (SSSA) in Pisa, where he has been investigating on real-time scheduling for soft real-time and multimedia applications, and predictability in infrastructures for cloud computing and NFV. He has been MTS in Bell Labs in Dublin (Ireland), investigating on security and real-time performance of cloud services. He has been a software engineer in Amazon Web Services in Dublin (Ireland), where he worked on improving the performance and scalability of DynamoDB. Since 2016, he is Associate Professor at SSSA and head of the Real-Time Systems Lab (RETIS) since 2019.