Operating System Noise in the Linux Kernel

Daniel Bristot de Oliveira, Daniel Casini, *Member, IEEE*, and Tommaso Cucinotta, *Member, IEEE*

Abstract—As modern network infrastructure moves from hardware-based to software-based using Network Function Virtualization, a new set of requirements is raised for operating system developers. By using the real-time kernel options and advanced CPU isolation features common to the HPC use-cases, Linux is becoming a central building block for this new architecture that aims to enable a new set of low latency networked services. Tuning Linux for these applications is not an easy task, as it requires a deep understanding of the Linux execution model and the mix of user-space tooling and tracing features. This paper discusses the internal aspects of Linux that influence the Operating System Noise from a timing perspective. It also presents Linux's `osnoise` tracer, an in-kernel tracer that enables the measurement of the Operating System Noise as observed by a workload, and the tracing of the sources of the noise, in an integrated manner, facilitating the analysis and debugging of the system. Finally, this paper presents a series of experiments demonstrating both Linux's ability to deliver low OS noise (in the single-digit μs order), and the ability of the proposed tool to provide precise information about root-cause of timing-related OS noise problems.

Index Terms—Linux kernel, operating system noise, high-performance computing, soft real-time systems

1 Introduction

The Linux Operating System (OS) has proved to be a viable option for a wide range of very niched applications, despite its general-purpose nature. For example, Linux can be found in the High-Performance Computing (HPC) domain, running on all the top 500 supercomputers.[1] It can also be found in the embedded real-time systems domain, not only in the area of industrial automation and robot control but even reaching out the space [1]. These achievements are possible thanks to the great flexibility in the configuration options of Linux, and specifically its kernel.

Another remarkable domain where Linux plays a central role is the one of developing core services supporting modern networking infrastructures and the Internet. With Network Function Virtualization (NFV) [2] and Software-Defined Networking (SDN) [3], this domain is shifting from the traditional paradigm of hardware appliances sized for the peak-hour, to the new one of flexible software-based and programmable networking services with horizontal elasticity abilities to adapt dynamically to the workload conditions. These new architectures often rely on general-purpose hardware [4] and software stacks based on Linux [5].

1. By Nov. 2021, according to https://www.top500.org/.

- *Daniel Bristot de Oliveira is with Real-time Scheduling Team, Red Hat, Inc., 56025 Pontedera, PI, Italy, and also with TeCIP Institute, Scuola Superiore Sant'Anna, 56127 Pisa, Italy. E-mail: danielbristot@gmail.com.*
- *Daniel Casini is with the TeCIP Institute, Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, 56127 Pisa, Italy. E-mail: d.casini@sssup.it.*
- *Tommaso Cucinotta is with TeCIP Institute, Scuola Superiore Sant'Anna, 56127 Pisa, Italy. E-mail: tommaso.cucinotta@santannapisa.it.*

Manuscript received 29 January 2022; revised 24 May 2022; accepted 26 June 2022. Date of publication 30 June 2022; date of current version 13 December 2022.
(Corresponding author: Daniel Bristot de Oliveira.)
Recommended for acceptance by Daniel Mosse, Tam Chantem, Enrico Bini.
Digital Object Identifier no. 10.1109/TC.2022.3187351

The 5 G network stack is built upon this paradigm, and it is enabling a new set of services characterized by strict timing requirements [6]. These were generally satisfied using physical appliances in traditional networks. However, in the new 5 G stack, these requirements need to be achieved by software-based appliances, requiring the support of a real-time operating system. For example, in Virtualized Radio Access Network (vRAN), latencies are in the order of tens of microseconds [4], [7]. Such a need made time and processing latency one of the main metrics for vendors in this market [8], [9], [10].

To meet these tight timing requirements, both hardware and Linux are configured according to standard practices from both the HPC and the real-time domains. To this end, the hardware is configured to achieve the best trade-off between performance and determinism. This setup includes adjusting the processor speed and power savings setup while disabling features that could cause hardware-induced latencies, such as system management interrupts (SMIs).

Regarding the Linux configuration, the system is usually partitioned into a set of *isolated* and *housekeeping* CPUs, which is a typical setup for HPC systems. The *housekeeping* CPUs are those where the tasks necessary for the regular system usage will run. This includes kernel threads responsible for in-kernel mechanisms, such as RCU (read-copy-update) callback threads [11], kernel threads that perform deferred work such as `kworkers` and threads dispatched by daemons and users. General system's IRQs (Interrupt ReQuests) are also routed to *housekeeping* CPUs. This way, the *isolated* CPUs are then dedicated to the NFV work. However, despite the high-grade CPU isolation level currently available on Linux, some housekeeping work is still necessary on all CPUs. For example, the timer IRQ still needs to happen under certain conditions, and some kernel activities need to dispatch a kworker for all online CPUs. Drawing from real-time setups, NFV threads are often configured with real-time priorities, and the kernel is generally configured with the fully preemptive

mode (using the PREEMPT_RT patch-set [12]) to provide bounded wakeup latencies.

In order to debug and evaluate the system setup, Linux practitioners use syntectic workloads that mimic the behavior of these complex scenarios. NFV applications run both triggered by an interrupt or by polling the network device while waiting for packets, running non-stop. While the Linux wakeup latency has been extensively studied from the real-time perspective [13], [14], this is not the case for the interference suffered by threads. This subject however was extensively covered by another community: the HPC one, in a metric named OS noise [15], [16]. In this paper, we focus on the practicalities of OS noise measurement and analysis on Linux, from a real-time viewpoint.

Why Yet Another Tool? Several tools have been proposed over the years to measure OS noise, and they can be classified into two categories: *workload* and *trace-based* methods. Both of these have advantages and disadvantages, extensively discussed later in the paper. In summary, workload methods simulate a workload, being capable of accounting for the OS noise measurement as a metric reported by the workload. For instance, by detecting a large amount of time elapsed between two consecutive reads of the time or by the number of finished operations. The limitation of workload-based tools is that they do not provide any insight into the root cause of the noise. Conversely, trace-based methods show potential causes of latency spikes, but they cannot account for how the workload perceives the noise.

Differently from previous work, we cover both worlds by designing and implementing a comprehensive kernel tracer to deal with the OS noise on Linux, called `osnoise`. It uses a hybrid approach, leveraging both the workload and a tracing mechanism synchronized together to account for the operating system noise while still providing detailed information on the root causes of OS noise spikes, and also reducing the tracing overhead using in-kernel tracing features. While the tool was developed with extreme isolation cases in mind, targeting the detection of single-digit μs noise occurrences, it is not limited to this use case. Indeed, it can be applied to any HPC system setup.

The `osnoise` tracer is officially part of the Linux kernel since version *5.14*, passing by the thorough kernel revision process, including experts in real-time, scheduling, and tracing, evidencing the agreement on the abstractions and technologies used by `osnoise`. Since Linux kernel version *5.17*, the tracer can be used as an user-space tool available via the `rtla` (Real-Time Linux Analysis) toolset, becoming easily accessible both by practitioners to test their systems and developers to extend it.

Paper Contributions. The contributions of this paper are three-fold: (I) propose a precise definition of the causes of OS noise in Linux, from the real-time perspective; (II) present a kernel tracer that is able to measure the OS noise using the workload approach, while also providing tracing information essential to pinpoint the tasks suffering of OS noise, not only caused by the OS, but also from the hardware or the virtualization layer; (III) report on empirical measurements of the OS noise from different configurations of Linux, commonly found in NFV setups, showing how the tool can be used to find the root causes of high latency spikes, thus enabling finer-grained tuning of the system.

2 Background

We start presenting the needed background. First, Section 2.1 presents the Linux execution contexts and their relation. Then, Section 2.2 summarizes the Linux schedulers hierarchy and the most commonly used tracers.

2.1 Linux Execution Contexts and Their Relation

In Linux, there are four main execution contexts: non-maskable interrupts (NMIs), maskable interrupts (IRQs), softirqs (deferred IRQ activities), (note that in the PREEMPT_RT, the softirq context is moved from its own execution context to run as a regular thread), and threads [12]. When there is no explicit reason to distinguish among them, we hereafter refer to all of them as tasks. Interrupts are managed by the interrupt controller, which queues and dispatches multiple IRQs and one NMI for each CPU. The NMI handler is the highest-priority activity on each CPU, it is non-maskable, and hence it is capable of preempting IRQs and threads. IRQs, in turn, are able to preempt threads and softirqs, unless they have been temporarily disabled within critical sections of the kernel. Ssoftirq is a software abstraction, and in the standard kernel configuration, runs after IRQ execution, preempting threads. Finally, threads are the task abstraction managed by the Linux schedulers.

Linux's execution contexts are characterized by the following rules:

- R1 The per-CPU NMI preempts IRQs, softirqs and threads;
- R2 The per-CPU NMI, once started, runs to completion.
- R3 IRQs can preempt softirqs and threads.
- R4 Once an IRQ is started, it is not preempted by another IRQ.
- R5 Softirqs can preempt threads.
- R6 Once a softirq is started, it is not preempted by any other softirq.
- R7 Threads cannot preempt the NMI, IRQs and sofirqs.

The rule-set is derived from the automata-based Thread Synchronization Model of Linux [17], which showed to model the Linux synchronization behavior faithfully, and by industrial expertise.

2.2 Linux Schedulers and Tracing

Next, we proceed introducing the Linux's scheduler and tracing mechanisms.

Schedulers. Linux has a hierarchy of five schedulers, which handle all the threads irrespectively of their memory contexts (e.g., kernel threads, process context in user-space). The five schedulers are queried in a fixed order to determine the next thread to run. The first one is the*stop-machine*, a pseudo scheduler used to execute kernel facilities. The second one is`SCHED_DEADLINE` [18], a deadline-based real-time scheduler based on Earliest Deadline First (EDF). The third one is a POSIX-compliant fixed-priority real-time scheduler. A thread using this scheduler can be either a`SCHED_RR` or a`SCHED_FIFO` thread. The difference between the two is only for threads at the same priority: in this case,`SCHED_RR` threads are scheduled in a round-robin fashion with a given time slice, while`SCHED_FIFO` threads release the CPU only on suspension, termination or preemption. The fifth scheduler, is

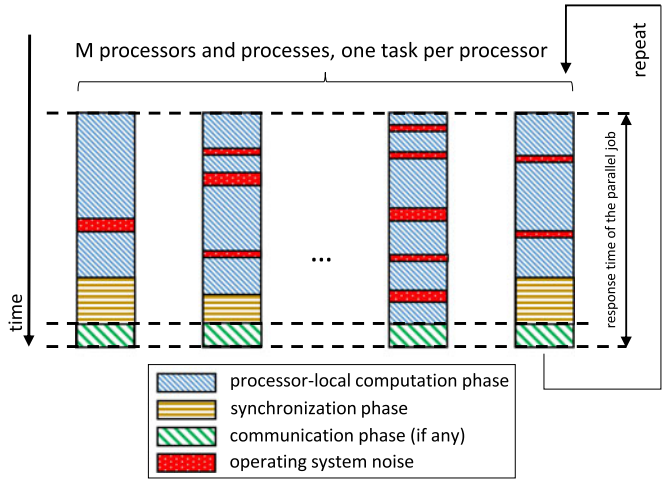


Fig. 1. The single-program multiple-data (SPMD) model used for HPC workloads, and the effects of the OS noise (adapted from [19]).

the general-purpose scheduler, the completely fair scheduler (`CFS`), also called `SCHED_OTHER`. Finally, when no ready threads are available from these schedulers, the *IDLE scheduler* returns the idle thread.

Tracers. Linux has a rich set of tracing features. For instance, it is possible to trace specific events such as scheduling decisions and many functions called in kernel context. These features are ready to use in most Linux distributions, mainly because they do not add overhead to the system if they are not in use. Ftrace is an in-kernel set of tracing features designed to aid practitioners in observing in-kernel operations.

3 Motivations and Proposed Approach

Next, we start introducing the problem and the motivations for this work.

The OS noise, sometimes named OS jitter, is a well-known problem for the HPC field [15], [16].

Generally, HPC workloads follow the single-program multiple-data (SPMD) model, shown in Fig. 1. In this model, a system is composed of M processors, and a parallel job consists of one process per processor [19]. All processes are dispatched simultaneously at the beginning of the execution. At the end of the execution, the process synchronizes to compose the final work, and repeat cyclically.

Ideally, the parallel job process should be the only workload assigned to the processor. However, some operating system-specific jobs need to run on all processors for the correct operation of the system, like the periodic scheduler tick, critical kernel threads or others. In this scenario, the scheduler decisions of each local processor significantly impact in response time of a parallel job. These delays caused to a parallel workload by OS activities running on the same processor(s) is named Operating System Noise.

One of the main reasons that led Linux to dominate the top 500 super-computers list is the flexibility of the system configuration. These systems' setup involve selecting a small set of CPUs to be in charge of all tasks necessary for the system execution and operation, such as running system daemons, periodic maintenance tasks, managing user access to the system for monitoring activities, etc., leaving a large set of CPUs *isolated* from most of the operating noise that users or the OS could cause.

In NFV, to achieve high throughput, the generic network stack of the operating system is often bypassed, with all the network packet processing done in user-space by a specific process that handles the network flows. Similar to the HPC case, these processes receive dedicated resources, including dedicated isolated CPUs.

To reduce even more the latency for handling new packets, some of these network applications poll the network in a busy-wait fashion, most notably using the DPDK's Poll Mode Driver (PMD).[2] In this use case, the Linux setup follows the same script as HPC. *The difference between this use-case and the HPC one is the real-time constraints, for example in the order of tens of microseconds for vRAN.*

Although some turn-key options to provide CPU isolation are available, such as moving all threads and IRQs to a reduced set of *house-keeping* CPUs, the fine-tuning of the configuration for these time-sensitive use-cases is not a simple task. The reason is that the OS still requires some per-CPU actions, such as scheduler tick, virtual memory stat operations, legacy network packet processing, etc. While some of these noise sources can be mitigated via fine-tuning of the configuration, like enabling `NOHZ_FULL` that reduces the scheduler tick frequency, others might even require reworking the currently existing kernel algorithms to either remove the cause of noise or add a method to mitigate the issue.

Despite of the appealing use-case, Linux is a general-purpose operating system, with the main focus on developing general-purpose features. Specialized communities, such as the real-time and HPC ones, need to constantly monitor the evolution of the OS to adapt possible non-HPC and non-RT aware functionality for these specific use-cases. It is impractical to force Linux developers to test their new algorithms to all the specific use-cases' metrics when there is no simple way to observe and debug them.

To measure the OS noise, a practitioner generally starts by spawning a synthetic workload. An example of workload is `sysjitter`, and its clone `oslat`. These tools loop reading the time using architecture-specific instructions. They define a *jitter* when two consecutive readings of the time have a gap larger than a given *threshold*. These tools do not attempt to correlate a *jitter* to a root cause.

To find a root cause, practitioners need to observe the system. The most efficient way to observe the system is using tracing. The challenge of using tracing is in the tradeoff between information and overhead. With a workload and tracing features in place, the user must identify a correlation between the noise and tracing information. This correlation is not always possible, mainly because the workload and the tracing features are unaware of each other.

It is essential to notice that at the dozens of microseconds figure, hardware-induced noise is also noticeable by the workload. Hardware noise can be a side effect of hardware stalls caused by shared resources, as happening in hyper-threading enabled processors or execution contexts with a higher priority than the OS, like SMIs. Because these actions

2. https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html

are not a side effect of the operating system, it is impossible to observe the events via trace. This noise observed by the workload but not by the trace creates a grey area that often misleads the analysis.

3.1 Proposed Approach

In this paper, we propose an integrated tracing and synthetic workload solution that aims to join workload- and tracing-based approaches benefits while minimizing the drawbacks of each solution.

The steps taken for such an approach include:

- Define the composition of the OS Noise on Linux from the real-time HPC point of view;
- Define the minimum set of tracing events to provide evidence of the root cause of each noise, at a limited overhead;
- Create a synthetic workload aware of tracing, enabling an unambiguous correlation of the trace and the noise;
- Make the approach production-ready, with a standard and easy-to-use interface.

3.2 OS Noise Composition for Real-Time HPC Workload

This paper adopts the following generalized definition of OS noise:

Definition 1 (Generalized (OS)-Noise). *The OS noise is defined as all the time spent by a CPU executing instructions not belonging to a given application task assigned to that CPU while the task is ready to run.*

The definition generalizes the usual interpretation of OS noise, which typically only includes OS-related activities and overheads, by accounting also for the time used by *any* interfering computational activity, not limited to the OS but also from regular user-space threads. This makes a difference when multiple user threads can run in the same CPU, as any computational activity that can interfere with the measurement thread would also interfere with any user thread running with the same scheduler and scheduler settings (e.g., priority), irrespectively to whether it belongs to the OS or not. Therefore, it would constitute an actual source of noise which the fine-grained tuning of the system needs to account for.

This extended definition gives room for an interesting link between the OS noise, a metric from the HPC domain, and the high-priority interference commonly considered in real-time systems theory.

This generalizes the approach beyond the HPC and NFV use cases, allowing to practically profile all the sources of interference that can affect a task running with a given configuration of the scheduler: for example, a thread running at a given priority under the fixed-priority scheduler of Linux.

Indeed, thanks to this generalized definition, the `osnoise` tracer can be used not only to monitor the noise strictly-related to the operating system but all high-priority interference in a broader sense.

Generalized (OS)-Noise Under Fixed-Priority Scheduling. As a highly-relevant example, we consider the case in which a designer needs to determine whether a thread of interest τ_i (for example, a constrained-deadline sporadic task [20]) will complete within the deadline, under a partitioned scheduling setting where workloads are considered for each processor separately. To this end, the classical worst-case response time equation can be leveraged

$$R_i = e_i + \sum_{\tau_h \in hp_i} \eta_h(R_i) \cdot e_h, \tag{1}$$

where e_h is the worst-case execution time (WCET) of τ_h, $\eta_h(\Delta)$ is its arrival curve [21] bounding the maximum number of release events of τ_h in a time window[3] of length Δ, and the set hp_i contains the higher-priority activities that can interfere with the thread τ_i under analysis.

While using Equation (1) at design time is in principle possible, it is quite often hard. Indeed, in modern heterogeneous computing platforms, many design principles used to increase average-case performance (e.g., complex cache hierarchies [22], un-revealed memory controller policies [23], out-of-order execution, etc.) are making it hard to obtain reliable WCET estimates for user threads. This is even harder for OS threads and interrupt service routines, for which also the arrival pattern is unknown, and therefore it is difficult to obtain an arrival curve.

Adopting such computing platforms in a small subset of strictly hard real-time systems, e.g., avionics, calls for comprehensive solutions allowing to know all the parameters involved in Equation (1), e.g., by leveraging static analysis tools for WCET estimation [24].

However, most real-time systems are robust enough to tolerate small uncertainties in the estimation of the parameters, and they can tolerate a small amount of deadline misses [25] (e.g., in multimedia [26]).

In these cases, `osnoise` can be used to empirically measure the high-priority interference in Equation (1). For example, to estimate the high-priority interference faced by a NFV workload running at a given priority under `SCHED_FIFO` (a common use-case), the system engineer can setup `osnoise` to run under `SCHED_FIFO` at the same priority, thus exposing the measurement thread to the same sources of noise.

4 Related Work

The adverse effects to workload performance due to the operating system noise have been known for a long time [27]. One of the first works addressing the problem of detecting the OS noise is due to Petrini *et al.* [15], which identified and eliminated some source of noise for an HPC application running on the ASCI Q supercomputer. The study has been extended in a later paper [28]. Ferreira *et al.* [29] provided a characterization of application sensitivity to the noise by injecting interference at the OS level.

As discussed at the beginning of the paper, Linux tools for detecting OS noise are divided into two categories: workload and trace-based methods.

Some workload-based methods run micro-benchmarks with a known duration, and they measure the difference between the expected duration of the microbenchmark and

3. e.g., if τ_h is a sporadic thread with minimum inter-arrival time T_i, it holds $\eta_h(\Delta) = \lceil \Delta / T_h \rceil$).

```
[root@f35 tracing]# cat trace
# tracer: osnoise
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              || /                                  MAX
#                              ||||                RUNTIME   NOISE  SINGLE  Interference counters:
#           TASK-PID     CPU# ||||   TIMESTAMP    IN US     IN US  NOISE   +-----------------------------+
#              | |         |  ||||      |           |         |      |      HW    NMI    IRQ    SIRQ THREAD
          <...>-859     [000] ....    81.637220: 1000000       190  9         18     0      1007   18     1
          <...>-860     [001] ....    81.638154: 1000000       656  74        23     0      1006   16     3
          <...>-861     [002] ....    81.638193: 1000000      5675  202       6      0      1013   25     21
          <...>-862     [003] ....    81.638242: 1000000       125  45        1      0      1011   23     0
          <...>-863     [004] ....    81.638260: 1000000      1721  168       7      0      1002   49     41
```

Fig. 2. `osnoise` summary output from *ftrace* interface.[4]

the actual time needed to process it. For this category, one relevant example is due to Sottile and Minnich [30], who proposed the Finite-Time Quantum (FTQ) benchmark. FTQ measured the number of basic operations done within a fixed time quantum. Another work from Tsafrir *et al.* [31] used microbenchmarks in conjunction with a mechanism based on "smart timers" to measure OS noise.

A widely used by practitioners workload-based tool is `sysjitter` [32]. It measures OS noise by running a thread on each CPU and keeping track of the duration of each time interval in which the thread is not running, e.g., due to OS activities. Other similar tools are `oslat` [33], jHiccup [34], and MicroJitterSampler [35]. However, all these tools are not capable of pointing to the root causes of the OS noise.

This problem can be solved with other tools, which use a trace-based approach. For example, De *et al.* [36] presented a tool to identify the sources of the OS jitter using kernel instrumentation by leveraging OProfile. The authors obtained statistics on preemptions and interruptions. Later, Morari *et al.* [16] proposed a different tool to measure the OS noise but using a similar (but more extended) technique based on kernel instrumentation and building on the LTTng [37] tracer. With respect to [36], the work by Morari *et al.* allows for capturing additional causes of the OS noise, e.g., softirqs. Nataraj *et al.* [38] proposed another approach to instrument the Linux kernel and measure the OS noise during application execution by using KTAU [39]. More recent work is due to Gonzalez *et al.* [40], which presents Jitter-Trace, a tool that uses the information provided by the `perf` tracer (see Section 5.3). However, all these trace-based methods do not account for how workloads perceive the noise.

Different from previous work, the `osnoise` tool proposed in this paper takes the best of both workload-based and trace-based methods, allowing to point to the root causes of the operating system noise while also accounting for how the workload perceives the noise.

And finally, but most relevantly, `osnoise` is the only tool for directly monitoring the OS noise that has been just recently made available in the mainline Linux kernel [41], and hence it is ready-to-use on billions of devices.

5 The *osnoise* Tracer

This section presents the `osnoise` tracer, which leverages the rules presented in Section 2.1 to correctly profile the execution time of each task by correctly subtracting the time required by each interfering activity from its measured runtime. The tool is not limited to a specific preemption model of Linux, and it can work with any of its preemption models, from the non-preemptive kernel to PREEMPT_RT.

Before discussing the internals, we present the tool at a high level. As mentioned, `osnoise` has two components: the workload and the tracing components.

5.1 The *osnoise* Workload Threads

The `osnoise` workload threads used for measurements work on a per-CPU basis.

By default, `osnoise` creates a *periodic* kernel thread on each CPU. The kernel thread can be assigned to any Linux scheduler, such as `SCHED_DEADLINE`, `SCHED_FIFO`, `SCHED_RR`, or `CFS`.

Each thread runs for a pre-determined amount of *runtime*. The primary purpose of the workload thread is to detect the *time* stolen from its execution, which is considered OS noise. Each `osnoise` thread works by reading the time in a loop. When it detects a gap between two consecutive readings higher than a given *tolerance threshold*, a new *noise sample* is collected. The time is read using the `trace_local_clock()` function. This architecture-specific non-blocking function provides a lightweight CPU-level coherent timestamp, at the nanoseconds granularity, at the same accuracy used by other ftrace tracing mechanisms.

The thread runs with preemption and IRQs enabled. This way, it can be preempted at any time by any task abstraction present in Linux.

After *runtime* microseconds are elapsed since the first time read of the current *period*, the workload reports a summary of the OS noise faced by the current activation. This summary is reported using tracing features of Linux, as in Fig. 2.

The `osnoise` summary reports:

- `RUNTIME IN US`, i.e., the amount of time in μs in which `osnoise` looped reading the timestamp.
- `NOISE IN US`, i.e., the overall amount of noise in μs observed in the associated runtime.
- `PERCENTAGE OF CPU AVAILABLE`, i.e., the percentage of CPU available to the `osnoise` thread in the measuring period.
- `MAX SINGLE NOISE IN US`, i.e., the longest observed occurrence of noise in μs during the runtime.
- The *interference counters:* for each type of interference among the classes NMI, IRQs, softirqs, and threads,

```
osnoise/3-4417  [003] d.... 203398.433218: sched_switch: prev_comm=osnoise/3 prev_pid=4417 prev_prio=120 prev_state=R+
                                           ==> next_comm=sleep next_pid=5842 next_prio=120
    sleep-5842  [003] d.... 203398.433414: sched_switch: prev_comm=sleep prev_pid=5842 prev_prio=120 prev_state=Z
                                           ==> next_comm=bash next_pid=5802 next_prio=120
     bash-5802  [003] d.... 203398.433830: sched_switch: prev_comm=bash prev_pid=5802 prev_prio=120 prev_state=S
                                           ==> next_comm=bash next_pid=5843 next_prio=120
    sleep-5843  [003] d.h.. 203398.434017: local_timer_entry: vector=236
    sleep-5843  [003] d.h.. 203398.434022: local_timer_exit: vector=236
    sleep-5843  [003] d.... 203398.434629: sched_switch: prev_comm=sleep prev_pid=5843 prev_prio=120 prev_state=S
                                           ==> next_comm=osnoise/3 next_pid=4417 next_prio=120
```

Fig. 3. Example of tracepoints: IRQ and thread context switch events read from *ftrace* interface[4].

`osnoise` maintains an interference counter that is increased in correspondence of an entry event of activity of that type.

It is worth noting that Fig. 2 shows a high number of hardware noise samples: this is because `osnoise` was running on a virtual machine, and the interference due to virtualization is detected as hardware noise.

5.2 The *osnoise* Parameters

The `osnoise` tracer has a set of parameters. These options are accessible via *ftrace*'s interface, and they are:

- `osnoise/cpus`: CPUs on which a `osnoise` thread s will execute.
- `osnoise/period_us`: the period (in μs) of the `osnoise` thread s.
- `osnoise/runtime_us`: how long (in μs) an `osnoise` thread s will look for noise occurrences.
- `osnoise/stop_tracing_us`: stop the system tracing if a single noise occurrence higher than the configured value in μs happens. Writing 0 disables this option.
- `osnoise/stop_tracing_total_us`: stop the system tracing if total noise occurrence higher than the configured value in μs happens. Writing 0 disables this option.
- `tracing_threshold`: the minimum delta between two time reads to be considered as a noise occurrence, in μs. When set to 0, the default value will will be used, which is currently five μs.

5.3 The *osnoise* Tracing Features

The *tracepoints* are one of the key pillars of the Linux kernel tracing. The tracepoints are points in the kernel code where it is possible to attach a probe to run a function. They are most commonly used to collect trace information. For example, `ftrace` register 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. Fig. 3 shows an example of tracepoint output via *ftrace* interface.

The usage of tracepoints is not limited to saving data to the buffer. They have been leveraged for many other use-cases. For instance, patch the kernel at runtime or transform network packets [42]. Tracepoints can also be used to optimize tracing itself. While saving data to the trace buffers has been optimized to the minimum overhead, it is also possible to pre-process data in the tracepoints in such a way as to minimize the amount of data written to the trace buffer. This method has shown good results, reducing the tracing overhead when the trace processing presents lower overhead than writing trace to the buffer [42].

The `osnoise` tracer leverages the current tracing infrastructure in two ways. It adds probes to existing tracepoints to collect information and adds a new set of tracepoints with pre-processed information.

Linux already has tracepoints that intercept the entry and exit of IRQs, softirqs, and threads. `osnoise` attaches a probe to all entry and exit events and uses it to: 1) account for the number of times each of these classes of tasks added noise to the workload; 2) to compute the value of the *interference counter* used by the workload to identify how many interferences occurred between two consecutive reads of the time;[5] 3) to compute the execution time of the current interfering task; 4) to subtract the noise occurrence duration of a preempted noise occurrence by leveraging the rules discussed in Section 2.1.

At the exit probe of each of these interference sources, a single tracepoint from `osnoise` is generated, reporting the noise-free execution time of the task's *noise observed via trace*.

In addition to the tracepoints and the summary at the end of the period, the `osnoise` workload emits a tracepoint anytime a noise is identified. This tracepoint informs about the *noise observed via workload*, and the amount of interference that happened between the two consecutive time reads. The interference counters are fundamental to unambiguously defining the root cause for a given noise.

For example, in Fig. 4, the first four lines represent the noise as identified by the trace, while the last line is the tracepoint generated by the workload, mentioning the previous four interferences.

Both Figs. 3 and 4 were extracted from the same trace file. The difference is that the former contains the previous existing tracepoints, while the latter includes the new tracepoints added to the kernel with `osnoise`. With these two examples, it is possible to notice that the amount of information reported by the `osnoise` tracepoints is reduced and more intuitive.

Regarding the noise reported in Fig. 4, it is important to notice that the duration reported by the `irq_noise` and `thread_noise` are free of interference. For example, the `local_timer:236` has a *start time* later than the `sleep-5843`. This means that `local_timer:236` preempted `sleep-5843`, in a case of nested noise. The `local_timer:236`, however,

4 All *ftrace* excerpts share the same column description, as in the header in Fig. 2.

5. The single per-CPU NMI is a particular case without tracepoints; in this case, a special function was added to collect the same information.

```
    sleep-5842  [003] d.... 203398.433413: thread_noise:     sleep:5842 start 203398.433217481 duration 195472 ns
    bash-5802   [003] d.... 203398.433829: thread_noise:     bash:5802 start 203398.433413330 duration 415172 ns
    sleep-5843  [003] d.h.. 203398.434022: irq_noise:        local_timer:236 start 203398.434016335 duration 5627 ns
    sleep-5843  [003] d.... 203398.434629: thread_noise:     sleep:5843 start 203398.433829263 duration 793261 ns
osnoise/3-4417  [003] ..... 203398.434631: sample_threshold: start 203398.433215747 duration 1414624 ns interference 4
```

Fig. 4. Example of tracepoints: `osnoise` events read from *ftrace* interface with equivalent data highlighted[4].

discounted its own *duration* from the duration of `sleep-5843` This facilitates the debugging of the system by removing the fastidious work of computing these values manually or via a script in user-space. This also reduces the amount of data saved in the trace buffer, reducing resource usage and overhead.

Another important thing to notice is that the total *noise observed via trace* accounts for 1409532 ns,[6] but the *noise obseved via workload* reports 5092 ns more (1414624 ns), as illustrated in Fig. 5. The reasons behind are multiple. For example, the overhead added by the tracepoints enabled in Figs. 3 and 4; the delays added by the hardware to manage context switch and the dispatch of IRQs handlers; delays caused by cache inlocality after an interrupt [43]; low level the code that enables the tracing at IRQ context, like making the RCU aware of the current context;[7] and the scheduler call caused by the thread noise.

This justifies the dual approach and motivates the novelty to prior work that used only one of the main distinguishing factors with respect to prior work (as extensively discussed in Section 4): using both the measuring thread and the tracing. Indeed, the trace cannot be used as the only source of information because it cannot account for the overheads occurring outside the scope of the tracing. Similarly, the measurement thread alone cannot capture the reasons for the OS noise, and hence it does not provide essential information to understand and reduce the OS-related interference.

Hardware-Induced Noise. To identify hardware-induced noise, introduced in Section 2, Linux includes a tracer named `hwlat`. It works running a workload in the kernel, with preemption and IRQs disabled, avoiding all the sources of interference except the hardware and NMIs noise, which cannot be masked. While running a busy-loop reading the time, when `hwlat` detects a gap in two subsequent reads, it reports a hardware-induced noise.

The resemblance of `hwlat` and `osnoise` is not a coincidence because the latter was indeed inspired to the former tool. `osnoise` is also able to detect hardware noise. Because it tracks all the tasks execution, when a sample noise is detected without a respective increase in any of the interference counters, it is safe to assume that a layer below the operating system generated the noise.

5.4 The *osnoise* Interface

The Ftrace Interface. The `osnoise` was integrated into the Linux kernel as a tracer part of `ftrace` in version *5.14*. This is the basic interface that allows users to start and stop the workload, and set the parameters and tracing options.

The rtla Interface. Since version *5.17*, Linux includes the *real-time Linux analysis tool*, named `rtla`. This is a meta-tool that aims at analyzing the real-time properties of Linux by exploiting its tracing features to provide information on the causes of measurements. `rtla` includes an user-space *tool* named `rtla osnoise` [44], transforming the tracer into a benchmark-like tool. This tool works by configuring, dispatching and collecting data from the *osnoise* tracer via *ftrace* interface. The `rtla osnoise` can either collect the periodic *workload* summary, creating a long run summary, or create a histogram with data from the `sample_threshold` tracepoint.

5.5 The *osnoise* Internals

The `osnoise` tracer aims to measure possible sources of noise at the single-digit μs scale, and this represents a challenge when dealing with parallel and re-entrant code as in the Linux kernel. This section presents some of these challenges and how they have been tackled.

5.5.1 Task and Memory Model

`osnoise` aims to simulate applications built using the SPMD model, presented in Section 2. When dispatched, the tracer creates a per-CPU kernel thread to run the `osnoise` workload component. Each per-CPU thread has its affinity configured to run on the target CPU only. All threads share the same configuration that the user can update as the workload runs. This data is only accessed outside the main workload loop, thus not representing a problem for the measurement phase. A `mutex` protects the configuration. The runtime data utilized during the measurement phase is organized on per-CPU structures, and the thread only accesses the one related to the CPU where it is executing.

`osnoise` aims to simulate a user-space workload that follows the rules reported in Section 2.1. Specifically, a user-space workload on Linux can be preempted by all types of OS tasks mentioned in Section 2. While there are methods to temporarily disable thread preemption, softirqs, and interrupts, they have some undesired drawbacks. For example, disabling interrupts is a costly operation, and it should be avoided in cases where overhead needs to be minimized, like on the Linux tracing subsystem. Moreover, it is not

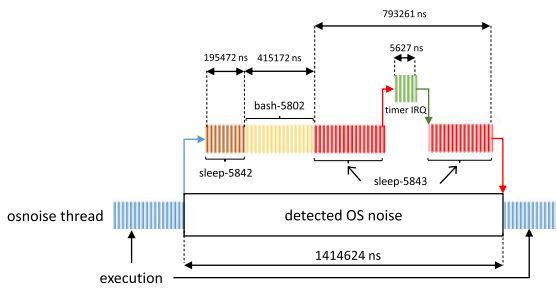


Fig. 5. Graphical representation of Fig. 4.

6. The quantity 1409532 ns is the sum of the four individual noise contribution detected by the tracer and reported in Fig. 4, i.e., $195472 + 415172 + 5627 + 793261$.

7. See `irq_enter()` and `irq_exit()` functions of Linux.

```
time = Current time
-------> IRQ entry
           increment local interference counter
<------- IRQ exit
int_counter = Read the interrupt counter
/* Interference counter not coherent
 * with the time read */
```

Fig. 6. Code reentrancy problem when incrementing the interference counter.

possible to mask non-maskable interrupts (NMIs). Finally, with such an operation, `osnoise` would influence the behavior of the noise tasks. Therefore, `osnoise` has been developed with the constraint of not using methods that prevent preemption of any class of tasks while measuring the OS noise. If the kernel is configured as non-preemptive, the `osnoise` measurement thread would be non-preemptive too. In this particular case, a fully-preemptive behavior is simulated by adding a function to check for the need for re-scheduling between two reads of the timer. This operation does not represent a significant source of overhead because Linux uses a single variable that informs whether there is a higher-priority thread waiting to run.

5.5.2 Dealing With Code Reentrance

One of the main benefits of `osnoise` is the ability to associate the number of interferences that lead to an observed noise by the workload. This feature is essential to avoid speculation concerning the events that caused a noise, leading the debugging to wrong directions and causing delays in troubleshooting critical and costly systems. However, reading the current time coherently with the *interference counter* is not a straightforward operation when considering the possibility of preemption. For example, take into consideration the pseudo-code in Fig. 6. The *interference counter* would account for one interrupt more than the number at the time in which the timer was read. Even simple operations such as incrementing a counter in a variable are not guaranteed to be atomic on all architectures supported by Linux.

In `osnoise`, these problems have been solved using mainly two solutions: *local atomic variables* and *compiler barriers*. An excerpt from the current code is shown in Fig. 7.

The *interference counter* is defined as a *local atomic integer*, a atomic type of integer coherent in the local CPU. To ensure that the *current time* is consistent with the *interference counter*, the *current time* is read inside two reads of *interference counter*. Additionally, compiler barriers are utilized to avoid compiler optimizations that could rearrange the *interference counter* read, moving the *current time* outside of protection. A similar approach is used for other non-atomic operations, such as computing the *interference free* noise for the `osnoise` tracepoints shown in Fig. 4. In this way, `osnoise` is able to provide coherent information while avoiding more costly methods, enabling measurements within one μs of granularity required by NFV use-cases.

6 Experimental Results

This section reports on the `osnoise` usage for the measurement and trace of a system. The system is a Dell workstation with an AMD Ryzen 9 5900 processor, with 12 cores and 24 threads. The system is configured with Fedora Linux 35 server and runs the kernel 5.15 patched with the PREEMPT_RT patchset. `osnoise` has been executed via `rtla osnoise` tool, both to collect a summary of the OS noise and a histogram of each noise occurrence.

The first considered configuration of the system has no tuning applied and is so-called *As-is*. The system is said to be *Tuned* when the best practices for CPU isolation are applied. In this case, CPUs $\{0, 1\}$ are reserved for housekeeping work of users and the operating system tasks. CPUs $\{2, \ldots, 23\}$ are reserved for the workload execution, and `osnoise` is set to run on these CPUs. The system tuning includes 1) set the *performance* CPUFreq governor;[8] 2) using cpu isolation features; 3) enabling RCU callbacks offload and *nohz_full* configuration; and 4) moving all possible kernel threads and IRQs to the CPUs $\{0, 1\}$. By default, `osnoise` workload threads runs with the default's task priority (`SCHED_OTHER` with 0 nice). However, for the NFV use case, it is common for the users to set a real-time priority for the workload. To also evaluate this specific scenario, additional experiments have been performed. In the experiments marked as `FIFO:1`, the `osnoise` workload has been set to run with priority 1 under `SCHED_FIFO`.

For the regular case (i.e., `SCHED_OTHER` with 0 nice) the workload has been configured with default parameters, so it runs with *one second* runtime and period, attempting to monopolize the CPU.

The `FIFO:1` case required some additional system and workload configuration. The system configuration includes disabling the *runtime throttling mechanism* of Linux, allowing the real-time threads to use more than 95% of CPU time, and setting the `ksoftirqd` (thread responsible for softirq processing when using PREEMPT_RT) and RCU per-CPU threads with *FIFO 2* priority to avoid *RCU* [11] stalls. The workload has been configured with a 10 seconds period, and the runtime has been set to allow a 100 μs between each period, allowing any starving thread to have a chance to run. Finally, the *tolerance threshold* was configured to 1 μs.

6.1 Percentage of OS Noise

A six-hours experiment has been conducted for all *tune/FIFO priority* cases collecting the *OS noise* summary. A summary of the *percentage OS noise* is shown in Fig. 8.

The maximum observed total noise was 0.5484% with the system *As-is* , while the minimum was 0.00001% for both *Tuned* cases. It is also possible to notice the main factor for OS noise reduction is provided by the CPU isolation features. The reason is that most of the work left for isolated CPUs is essential for the OS, mostly in the IRQ context.

It worth to highlight that these experiments have been conducted with the PREEMPT_RT kernel, which incurs additional overheads on the average case to bound the worst-case scheduling latencies. In this setup, Linux is able to provide both low latency for interrupt-based workloads and low OS noise for polling-based workloads in a single solution. This flexibility is fundamental for NFV

8. Linux CPUFreq: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

```
static u64 set_int_safe_time(struct osnoise_variables *osn_var, u64 *time) {
   u64 int_counter;
   do {
       int_counter = local_read(&osn_var->int_counter);
       barrier();          /* synchronize with interrupts */
       *time = time_get();
       barrier();          /* synchronize with interrupts */
   } while (int_counter != local_read(&osn_var->int_counter));
   return int_counter;
}
```

Fig. 7. Code excerpt of `set_int_safe_time()`: how `osnoise` deals with reentracy problems.

deployments with dynamic and diverse workloads in the same host.

6.2 OS Noise Occurrence Analysis

A six-hours experiment has been conducted for all *tune/FIFO priority* cases collecting a histogram of each detected noise occurrence. This experiment is important for the NFV use-case because a single long noise occurrence might cause the overflow of queues in the network packets processing. The results are presented in Fig. 9.

With this experiment, it is possible to see the main problem of using the system *As-is* in Fig. 9a. The `osnoise` workload detected 230 out-of-scale noise samples, with the maximum value as long as 13045 μs. Fig. 9b also shows that using `FIFO:1` in the system *As-is* represents an easy-to-use option to reduce the maximum single noise occurrence value. The reason being is that because the workload causes starvation of non-real-time threads, these threads are migrated to the CPUs with time available for them to run.

As-is using FIFO:1 however has two major drawbacks when compared against the *Tuned* options with or without *using FIFO:1* in Figs. 9c and 9d. The first is the high count of noise occurrences. The *Tuned* experiment includes the `nohz_full` option that reduces the occurrence of the scheduler tick, reducing the execution of the *ksoftirqd* kernel thread that checks for expired timers and activities that follow. Another difference is the tail latency, which is lower on the *Tuned* cases. This difference is explored in Section 6.3.

The results with system *Tuned* in Figs. 9c and 9d show that the tune dramatically changes the entries and duration of each noise occurrence when compared with the system *As-is*. Figs. 9e and 9f have been added to better visualize the *Tuned* cases.

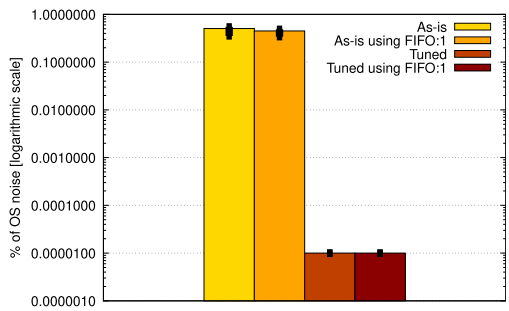


Fig. 8. Average percentage of OS noise observed by the workload on different scenarios. Error bars represent the range between minimum and maximum percentage.

The *Tuned* kernel was able to deliver consistent results, with the kernel *Tuned using FIFO:1* was able to provide below 5 μs maximum single noise occurrence. That is because background OS activities that run as threads are deferred by the real-time scheduler, without creating a fault in the system. For example, jobs dispatched on all CPUs via `kworkers` threads that execute deferrable work [45]. However, these still need to have the possibility to run to avoid major problems. Thus, a wise choice for the development of high-performance applications with low latency requirements is to be aware of this property and yield some CPU time when it would not cause performance issues (for instance, when network buffers are empty), even for a small amount of time, like 100 μs every 10 seconds.

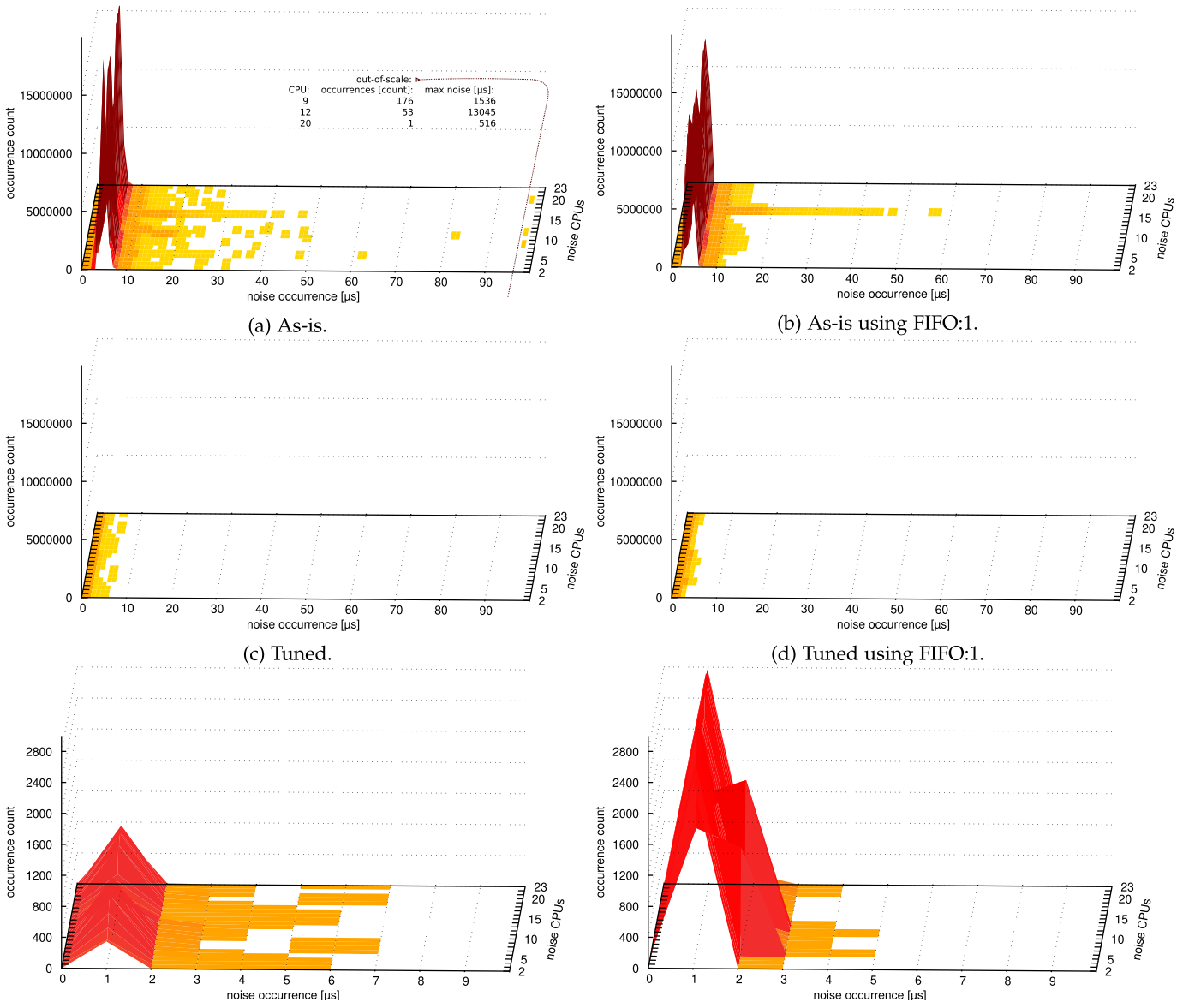
These experiments also serve to show the low impact that the `osnoise` internals imposes in the evaluation, allowing the user to receive information in the μs granularity used by practitioners on other tools like cyclictest.

It is important to highlight that the results presented in this section are only valid for this specific scenario. Different hardware, CPU count, auxiliary operating system services, and conditions will likely provide different results. Thus the importance of such a tool, providing an integrated OS noise benchmark and guidance for the fine tune of the system.

6.3 Using *osnoise* to Trace Sources of Latency

The experiment of the system `As-is with FIFO:1` presented an interesting result with regard to the tail latency, as only few samples passing the 30 μs mark. To understand the reasons behind these cases that go over 30 μs, the `osnoise` tracer was set to trace the `osnoise` events, stopping the tracer when a noise occurrence over 30 μs was detected. The trace with sole `osnoise` events is shown in Fig. 10. It shows an interrupt noise, caused by the interrupt 62, responsible for the `eno1` ethernet driver. Right after the `ksoftirqd` is scheduled, causing a long -duration noise occurrence. The `ksoftirqd` thread is responsible for running the `softirq` jobs context in the PREEMPT_RT kernel (recall from Section 2.1 that the softirq context does not exist under PREEMPT_RT, and `softirq` jobs run in the thread context).

Following the evidence that the problem is caused by `softirqs`, the sole events that provide information about softirqs were enabled, as selecting a small amount of tracing events helps to avoid influencing too much in the timing behavior of the system due to overhead. The trace again reported a similar output in Fig. 11. This trace shows that the *network receive* (`NET_RX`) `softirq` was the reason for



(a) As-is.

(b) As-is using FIFO:1.

(c) Tuned.

(d) Tuned using FIFO:1.

(e) Tuned - Zoom of Figure 9c.

(f) Tuned using FIFO:1 - Zoom of Figure 9d.

Fig. 9. `osnoise` *noise occurrence* per-cpu histogram under different system setup, mixing CPU isolation tune and real-time priority for the workload (less noise occurrence and less occurrence count is better).

```
     osnoise/16-2373    [016] d.h2   127.490797: irq_noise: eno1:62 start 127.490793954 duration 2204 ns
  ksoftirqd/16-129      [016] d..3   127.490844: thread_noise: ksoftirqd/16:129 start 127.490798012 duration 45816 ns
     osnoise/16-2373    [016] ....   127.490844: sample_threshold: start 127.490793483 duration 50946 ns interference 2
     osnoise/16-2373    [016] ....   127.490847: osnoise_main: stop tracing hit on cpu 16
```

Fig. 10. `osnoise` tracer finding source of latencies[4].

```
     osnoise/16-2501    [016] d.h2   533.347969: irq_noise: eno1:62 start 533.347965225 duration 3165 ns
  ksoftirqd/16-129      [016] ..s.   533.347970: softirq_entry: vec=3 [action=NET_RX]
  ksoftirqd/16-129      [016] ..s.   533.347994: softirq_exit: vec=3 [action=NET_RX]
  ksoftirqd/16-129      [016] d..3   533.347995: thread_noise: ksoftirqd/16:129 start 533.347969964 duration 25438 ns
     osnoise/16-2501    [016] ....   533.347996: sample_threshold: start 533.347964865 duration 30938 ns interference 2
     osnoise/16-2501    [016] ....   533.347996: osnoise_main: stop tracing hit on cpu 16
```

Fig. 11. `osnoise` tracer finding source of latencies augmented with other events[4].

the `ksoftirqd` activation. The `NET_RX softirq` is activated by the network driver that is running in the same CPU. This it is an side effect of the `eno1` ethernet driver causing the interrupt. Following this evidence, the IRQ 62 was configured to fire in the CPUs $[0:1]$. With this configuration applied, the experiment in Fig. 9b was re-executed for six hours, the results is shown in Fig. 12. This configuration alone was reposible to reduce the the tail latency to

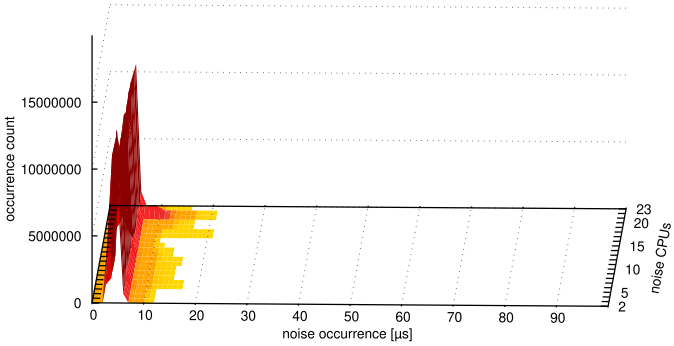


Fig. 12. As-is using FIFO:1 after moving the network IRQ as suggest from trace in Fig. 11.

figures like the system *Tuned*, with the debug facilitated by `osnoise`.

7 Conclusion and Future Work

Network function virtualization and modern low latency communications are creating the need for Linux systems with low latency for both scheduling latency and OS noise. These real-time HPC workloads require noise to be in the order of a few tens of microseconds.

However, debugging these cases is not an easy task. Workload-based tools are precise for measurements but do not point to a root cause. Trace-based measurements provide information about the cause but without an accurate picture of the actual noise observed by the thread.

Practitioners use both methods together, but this requires advanced knowledge of the tracing features, and it can often mislead the investigation because the trace is not synchronized with the workload or adds too much overhead.

The `osnoise` tool puts together the tracing and the workload, providing precise information at low overhead by processing and exporting only the necessary information for pointing to the root causes of the latency, serving as a good starting point for the investigation.

The experimental results show that the tool is able to serve both as a tracer and benchmark tool, facilitated by the usage of `rtla osnoise` interface to collect data. The experiment shows that Linux can deliver extremely low OS noise, achieving maximum sample noises as low as less than 5 μs. But more importantly, the tool is able to follow the kernel, delivering results in the desired scale.

Both the `osnoise` tool and `rtla osnoise` interfaces are an integral part of the Linux kernel, thus accessible for the entire Linux user base.

Because the `osnoise` tracer uses the most basic building blocks of the Linux tracing sub-system, it can be combined with many other existing tracing tools, such as performance counters provided via `perf` tool, or to be used with graphical interfaces provided by `LTTng` and `KernelShark` . This creates an end-less set of possibilities for future work, extending the `osnoise` measurements to include data from the memory/cache, to include workload-dependent methods, other clock sources, and energy-aware methods, for example. Extending the analysis with a more formal approach is another possibility, as well as conducting experimental evaluations based on other real-time schedulers of Linux, e.g.,`SCHED_DEADLINE`.

References

[1] L. Tung, "SpaceX: We've launched 32,000 linux computers into space for starlink internet," Jun. 2011. [Online]. Available: https://www.zdnet.com/article/spacex-weve-launched-32000-linux-computers-into-space-for-starlink-internet/

[2] Network Functions Virtualisation – Introductory White Paper – An Introduction, Benefits, Enablers, Challenges & Call for Action," SDN and OpenFlow World Congress. Oct. 2012. [Online]. Available: https://portal.etsi.org/NFV/NFV_White_Paper.pdf

[3] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A. comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[4] Z. Li, L. Ruan, C. Yao, Y. Dong, and N. Cui, "A best practice of 5G layer 2 network on Intel architecture," in *Proc. IEEE Globecom Workshops*, 2019, pp. 1–5.

[5] T. Cucinotta, L. Abeni, M. Marinoni, R. Mancini, and C. Vitucci, "Strong temporal isolation among containers in OpenStack for NFV services," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2021.3116183.

[6] GSM Association, "Cloud infrastructure reference model version 1.0," [Online]. Available: https://www.gsma.com/newsroom/wp-content/uploads//NG.126-v1.0-2.pdf

[7] N. Bhushan *et al.*, "Industry perspective," *IEEE Wireless Commun.*, vol. 24, no. 5, pp. 6–8, Oct. 2017.

[8] L. Mandyam and S. Hoenisch, "RAN workload performance is equivalent on bare metal and vSphere," [Online]. Available: https://blogs.vmware.com/telco/ran-workload-performance-tests-on-vmware-vsphere/

[9] *Intel*, "FlexRAN," [Online]. Available: https://github.com/intel/FlexRAN

[10] *Red Hat*, "What is NFV?" [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-nfv

[11] P. E. McKenney, J. Fernandes, S. Boyd-Wickizer, and J. Walpole, "RCU usage in the linux kernel: Eighteen years later," *SIGOPS Oper. Syst. Rev.*, vol. 54, no. 1, pp. 47–63, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3421473.3421481

[12] D. B. de Oliveira and R. S. de Oliveira, "Timing analysis of the PREEMPT_RT linux kernel," *Softw., Pract. Exper.*, vol. 46, no. 6, pp. 789–819, 2016.

[13] D. B. de Oliveira, D. Casini, R. S. de Oliveira, and T. Cucinotta, "Demystifying the real-time linux scheduling latency," in *Proc. 32nd Euromicro Conf. Real-Time Syst.*, 2020, pp. 9:1–9:23.

[14] F. Cerqueira and B. Brandenburg, "A comparison of scheduling latency in linux, PREEMPT-RT, and LITMUS RT," in *Proc. 9th Annu. Workshop Operating Syst. Platforms Embedded Real-Time Appl.*, 2013, pp. 19–29.

[15] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proc. ACM/IEEE Conf. Supercomputing*, 2003, pp. 55–55.

[16] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of OS noise," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 852–863.

[17] D. B. de Oliveira, R. S. de Oliveira, and T. Cucinotta, "A thread synchronization model for the PREEMPT_RT linux kernel," *J. Syst. Archit.*, vol. 107, 2020, Art. no. 101729.

[18] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the linux kernel," *Softw.: Pract. Exp.*, vol. 46, no. 6, pp. 821–839, 2016.

[19] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective distributed scheduling of parallel workloads," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 1996, pp. 25–36.

[20] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proc. Real-Time Syst. Symp.*, 1989, pp. 166–171.

[21] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the SymTA/S approach," in *IEEE Proc. - Comput. Digit. Techn.*, 2005.

[22] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, Nov. 2015, Art. no. 32.

[23] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2020, pp. 239–252.

[24] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *Proc. 17th Int. Workshop Worst-Case Execution Time Anal.*, 2017, pp. 8:1–8:12.
[25] B. Brandenburg, "The case for an opinionated, theory-oriented real-time operating system," in *Proc. 1st Int. Workshop Next- Gener. Oper. Syst. Cyber- Phys. Syst.*, 2019.
[26] L. Abeni and G. Buttazzo, "Adaptive bandwidth reservation for multimedia computing," in *Proc. 6th Int. Conf. Real-Time Comput. Syst. Appl.*, 1999, pp. 70–77.
[27] R. Mraz, "Reducing the variance of point to point transfers in the IBM 9076 parallel computer," in *Proc. Conf. Supercomputing*, 1994, pp. 620–629.
[28] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of system overhead on parallel computers," in *Proc. 4th IEEE Int. Symp. Signal Process. Inf. Technol.*, 2004, pp. 387–390.
[29] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proc. ACM/IEEE Conf. Supercomputing*, 2008, pp. 1–12.
[30] M. Sottile and R. Minnich, "Analysis of microbenchmarks for performance tuning of clusters," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2004, pp. 371–377.
[31] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications," in *Proc. 19th Annu. Int. Conf. Supercomputing*, 2005, pp. 303–312. [Online]. Available: https://doi.org/10.1145/1088149.1088190
[32] D. Riddoch, "sysjitter v1.4," [Online]. Available: https://github.com/alexeiz/sysjitter
[33] RT-Tests. [Online]. Available: https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git
[34] G. Tene, "jHicckup," [Online]. Available: http://www.azulsystems.com/jHiccup
[35] P. Lawrey, "MicroJitterSampler," [Online]. Available: http://blog.vanillajava.blog/2013/07/micro-jitter-busy-waiting-and-binding.html
[36] P. De, R. Kothari, and V. Mann, "Identifying sources of operating system jitter through fine-grained kernel instrumentation," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2007, pp. 331–340.
[37] The LTTng Project, "LTTng," [Online]. Available: https://lttng.org/
[38] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman, "The ghost in the machine: Observing the effects of kernel operation on parallel application performance," in *Proc. ACM/IEEE Conf. Supercomputing*, 2007, pp. 1–12.
[39] A. Nataraj, A. D. Malony, S. Shende, and A. Morris, "Kernel-level measurement for integrated parallel performance views: The KTAU project," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2006, pp. 1–12.
[40] N. M. Gonzalez, A. Morari, and F. Checconi, "Jitter-trace: A low-overhead OS noise tracing tool based on linux perf," in *Proc. 7th Int. Workshop Runtime Oper. Syst. Supercomputers*, 2017, Art. no. 2.
[41] D. B. de Oliveira, "Osnoise tracer," [Online]. Available: https://www.kernel.org/doc/html/latest/trace/osnoise-tracer.html, 2021.
[42] D. B. de Oliveira, T. Cucinotta, and R. S. de Oliveira, "Efficient formal verification for the linux kernel," in *Proc. Int. Conf. Softw. Eng. Formal Methods*, 2019, pp. 315–332.
[43] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation*, 2010, pp. 33–46.
[44] D. B. de Oliveira, "rlta-osnoise: Measure the operating system noise," 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/tools/rtla/rtla-osnoise.html
[45] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Driver*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2005.

Daniel Bristot de Oliveira received the joint PhD 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 with Red Hat, working on developing the real-time features of the Linux kernel. He helps in the maintenance of real-time related tracers and toolings for the Linux kernel and the SCHED_DEADLINE. He is an affiliate researcher with 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 (Member, IEEE) received the graduate (cum laude) degree in embedded computing systems engineering, the master's degree jointly offered by the Scuola Superiore Sant'Anna of Pisa and University of Pisa, and the PhD degree in computer engineering from the Scuola Superiore Sant'Anna of Pisa (with honors), working under the supervision of Prof. Alessandro Biondi and Prof. Giorgio Buttazzo. He is an assistant professor with the Real-Time Systems (ReTiS) Laboratory of the Scuola Superiore Sant'Anna of Pisa. In 2019, he has been visiting scholar with 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.

Tommaso Cucinotta (Member, IEEE) received the MSc degree in computer engineering from the University of Pisa, Italy, and the PhD degree 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 with SSSA and head of the Real-Time Systems Lab (RETIS) since 2019.

Open Access funding provided by 'Scuola Superiore 'S.Anna' di Studi Universitari e di Perfezionamento' within the CRUI CARE Agreement