Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 11, November-2023

Linux Kernel Noise from The Operating System

Aheesh Bharadwaj K K
Department of Computer Science and engineering,
P.E.S College Of Engineering, Mandya

Likhith H L
Department of Computer Science and engineering,
P.E.S College Of Engineering, Mandya

Abstract— A new set of specifications are put forth for operating system developers as modern network infrastructure transitions from hardware-based to software-based employing Network Function Virtualization. For this new architecture that promises to provide a new set of low latency networked services, Linux is becoming a key building piece by utilising the real-time kernel choices and enhanced CPU isolation capabilities common to the HPC use-cases. The Linux execution model, as well as the combination of user-space tooling and tracing features, must be thoroughly understood in order to tune Linux for these applications. This paper explains how Linux's intrinsic components affect operating system noise from a time standpoint. The relevant application areas, difficulties, and most crucially, the gaps for future research, are described in depth. Additionally, it introduces Linux's in-kernel osnoise tracer, which enables integrated measurement of operating system noise asexperienced by a workload and tracing of the sources of the noise to aid in system analysis and debugging. This study concludes with a series of experiments showing Linux's capability to give minimal OS noise (in the single-digit ms order) as well as the capability of the suggested tool to provide accurate information about the root-cause of timing-related OS noise concerns.

Keywords— High-performance computing, the Linux kernel,operating system noise, and soft real-time systems.

1. INTRODUCTION

Despite being general-purpose, the Linux Operating System (OS) has shown to be a viable solution for a variety of extremely specialised applications. For instance, all 500 of the top supercomputers in the High-Performance Computing (HPC) sector run Linux.1 It can also be found in the field of embedded real-time systems, extending beyond the realms of space and industrial automation and robot control [1]. These accomplishments are made possible by Linux's extensive configuration choices, particularly its kernel.The creation of fundamental services that enable contemporary networking infrastructures and the Internet is another noteworthy area where Linuxplays a crucial role.

This domain is transitioning from the old paradigm of hardware appliances sized for 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 with the help of Network Function Virtualization (NFV) [2] and Software-Defined Networking (SDN) [3]. These new architectures frequently rely on all-purpose hardware [4] and Linux-based software stacks [5]. This paradigm serves as the foundation for the 5G network stack, which is enabling a new set of services with stringent timing constraints [6]. In previous networks, these were typically satisfied by using physical equipment. However, the new5 G stack mandates that these standards be met by software- based appliances, which calls for the assistance of a real-time operating system. Latencies, for instance, are on the order of tens ofmicroseconds in the Virtualized Radio Access Network (vRAN) [4]. Time and processing delay became one of the key criteria for suppliers in this market as a result of this necessity [3], hardware.and Linux are set in accordance with best practises from both the real- time and HPC domains to meet these stringent timing constraints.

To this purpose, the hardware is set up to provide the best performance/determinism trade-off. This configuration involves changing the processor's speed and power-saving settings while eliminating elements like system management interrupts (SMIs) that can result in hardware- induced latencies.

In terms of the Linux configuration, an HPC system will often be divided into a number of separated and housekeeping CPUs. The CPUs used for housekeeping are those that perform the tasks required for routine system operation. This includes threadsdispatched by daemons and users as well as kernel threads in charge of internal operations like RCU (read-copy-update) call- back threads [1] and kworkers, which are kernel threads that carry out postponed tasks. Housekeeping CPUs are also forwarded the IRs (Interrupt Requests) from general systems. In this manner, the isolated CPUs are subsequently assigned to the NFV tasks. Although Linux now offers a high level of CPU isolation, all CPUs still require some housekeeping work. For instance, in some circumstances, the timer IRQ must still occur, and certain kernel operations require the dispatch of a kworker for all active CPUs. In order to offer bounded wakeup latencies, the kernel is typically configured with the fully preemptive mode (using the PREEMPT_RT patch-set [2]) and NFV threads are frequently configured with real-time priority.

Linux experts utilise synthetic workloads that imitate the behaviour of these intricate circumstances in order to diagnose and assess the system configuration. While waiting for packets, NFV applications can be

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

continuously running or interrupted by polling the network device. However, this is not the case for the interference that threads experience. The Linux wakeup latency has been thoroughly researched from the real-time perspective [3]. However, the HPC community has already extensively examined this topic in a metric known as OS noise [5], [6]. This study focuses on the Linux OS noise measurement and analysis in practise from a real-time perspective.

Why Another Tool Again? Over the years, a number of techniques have been presented to measure OS noise, and they can be divided into two groups: workload-based methods and trace-based methods.Both of them have benefits and drawbacks, which are covered in great detail later in the paper. In conclusion, workload simulation techniques can consider the OS noise measurement as a statistic supplied by the workload. For instance, by determining how much time passed between two consecutive time reads or by counting the number of processes that were successfully completed. The drawback of workload- based solutions is that they don't offer any insight into what is causing the noise in the first place. Contrarily, trace-based approachesreveal plausible reasons for latency spikes but are unable to take into consideration how the workload interprets thenoise

Since version 5.14, the os noise tracer has been a recognised component of the Linux kernel. This shows that the abstractions and technologies utilised by os noise have been approved by real-time, scheduling, and tracing experts during the extensive kernel revision process. Since the 5.17 release of the Linux kernel, the tracer is a user-space tool that can be used through the rtla (Real-Time Linux Analysis) toolset. This makes it simple for developers to add new features and practitioners to test their systems.

Contributions in paper. This article offers three contributions:

present a kernel tracer that can measure OS noise using theworkload approach and provide tracing information necessary to identify the tasks affected by OS noise, which can be caused not onlyby the OS but also by the hardware or virtualization layer; (II)propose a precise definition of the causes of OS noise in Linux, from the real-time perspective; (III) describe empirical measurements of the OS noise from several Linux configurations that are frequently used in NFV systems, demonstrating how the tool may be used to identify the underlying causes of excessive latency spikes and allow for more precise system tuning.

2 BACKGROUND

We begin by providing the necessary context. The Linux execution contexts are first presented in Section 2.1 along with theirrelationship. The hierarchy of Linux schedulers is then outlined in Section 2.2, along with a list of the most popular tracers.Keep your text and graphic files separate until after the text has been formattedand styled. Do not use hard tabs, and limit use of hard returns to only one return at the end of a paragraph. Do not add any kind of pagination anywhere in the paper. Do not number text heads-the template will do that for you.

2.1 Linux Execution Contexts and Their Relation

Non-maskable interrupts (NMIs), maskable interrupts (IRQs), softirqs (delayed IRQ activities) (notice that in the PREEMPT_RT, the softirq context is relocated from its own execution context to operate as a regular thread), and threads are the four primary execution contexts in Linux [12]. In the following, we simply refer to all of them as tasks when there is no clear reason to differentiate between them. The interrupt controller, which queues and dispatches numerous IRQs and one NMI for each CPU, is in chargeof managing interrupts. Since the NMI handler has the greatest priority on each CPU and cannot be hidden, it has the ability to preempt IRQs and threads. Softirqs and threads can both be preempted by IRQs, unless they have been momentarily disabled within crucial kernel regions. Software abstraction called Ssoftirq executes after IRQ execution in the default kernel setup, preemptingthreads. The task abstraction that is controlled by Linux schedulers is threads.The following guidelines define the execution contexts in

LINUX:

The per-CPU NMI, once begun, runs to completion. R1 The per- CPU NMI preempts IRQs, softirqs, and threads.Softirqs andthreads may be preempted by R3 IRQs. R4 An IRQ that has already begun is not interrupted by another IRQ.Threads may preempt using R5 Softirqs. R6 A softirq cannot be preempted byanother softirq once it has begun. R7 The NMI, IRQs, andSOFIRQs are not preemptibleby threads.

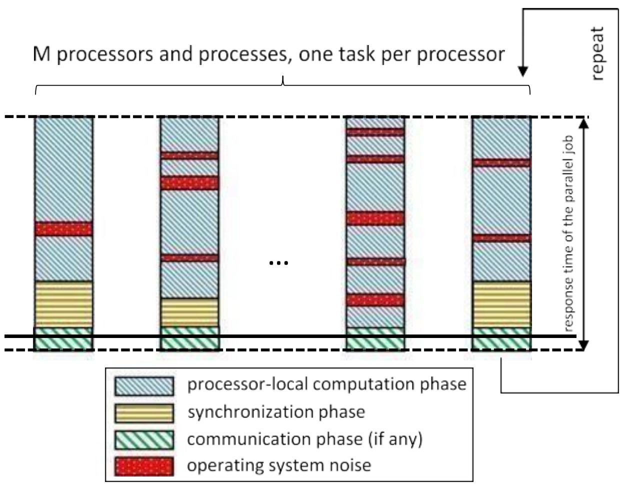

M processors and processes, one task per processor
repeat
response time of the parallel job
...
processor-local computation phase
synchronization phase
communication phase (if any)
operating system noise

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

2.2 Schedulers and tracing for Linux

The Linux scheduler and tracing mechanisms are then introduced.

Schedulers. Linux contains a hierarchy of five schedulers thatmanage all threads regardless of their memory contexts (such as kernel threads or user-space process context). The following thread to run is chosen by querying the five schedulers in a predetermined order. The first one is thestop-machine, which functions as a phoney scheduler for kernel facilities. The second is SCHED_DEADLINE [1], a real-time scheduler based on earliest deadline first (EDF) that uses deadlines. A POSIX-compliant fixed-priority real-timescheduler is the third option. This scheduler supports both aSCHED_RR and aSCHED_- FIFO thread types. In this scenario, SCHED_RR threads are scheduled in a round-robin fashion with a specific time slice, while SCHED_FIFO threads only release the CPU on suspension, termination, or preemption. The difference between the two is only for threads with the same priority. The perfectly fair scheduler (CFS), also known as SCHED_OTHER, is the fifth scheduler and is a general-purpose scheduler. Finally, theIDLE scheduler returns the idle thread when no ready threads are available from these schedulers.

Tracers. There are many tracing features available in Linux. For instance, various functions called in kernel context can be tracked, ascan individual events like scheduling choices. Due in large part to thefact that they do not increase system overhead when not in use, these features are available for usage in the majority of Linux distributions.Ftrace is a set of tracing features built into the kernel that are intended to make it easier for users to monitor in-kernel operations.

MOTIVATING FACTORS AND PROPOSED STRATEGY

The issue and the driving forces behind this study are then introduced. A common issue in the HPC community is OS noise, often known as OS jitter [5], [6]. Most HPC workloads adhere to the single-program multiple-data (SPMD) model, which is seen in Figure 1. A parallel job in this model consists of one process for each of the M processors that make up the system [1]. The execution starts witha simultaneous dispatch of all processes. After the execution is complete, the process synchronises to create the finished product and then repeats itself repeatedly. The scheduler decisions of each local processor significantly affect the response time of a parallel workloadin this scenario, as some operating system-specific jobs need to run on all processors for the correct operation of the system, such as the periodic scheduler tick, critical kernel threads, or others. The adaptability of the system setup is one of the primary factors that contributed to Linux dominating the list of the top 500 supercomputers. In the configuration of these systems, a small number of CPUs are chosen to handle all operations required for system execution and operation, such as managing user access to thesystem for monitoring activities, running system daemons, and performing routine maintenance tasks, while a large number of CPUs are shielded from the majority of operating noise that users or the OSmay produce. In NFV, the generic network stack of the operating system is frequently bypassed in order to achieve high throughput, with all network packet processing taking place in user-space by a dedicated process that manages the network flows. Similar to the HPC scenario, these processes are given exclusive access to resources, such as exclusive isolated CPUs. Some of these network applications poll the network in a busy-wait mode, most notably using the DPDK's Poll Mode Driver (PMD), to further reduce the latency for handling new packets.2 The Linux setup follows the sameprocedure as HPC in this use scenario. Real-time limitations, like those in the order of tens of microseconds for vRAN, are what set this usecase apart from the HPC one.

Although there are some ready-made alternatives for providingCPU isolation, such as transferring all threads and IRQs to a smaller number of house-keeping CPUs, it is not easy to fine-tune the settings for certain time-sensitive use-cases. The OS still needs to dosome per-CPU operations like scheduler ticks, virtual memory stat operations, processing of legacy network packets, etc. While some ofthese noise sources can be reduced by fine-tuning the configuration, such as turning on NOHZ_FULL to lower the scheduler tick frequency, others may even require reworking the current kernel algorithms to either eliminate the noise source or add a technique to reduce the problem.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

Linux is a general-purpose operating system with a primary focus on providing general-purpose functionality, notwithstanding the compelling use-case. Real-time and HPC groups in particular need to continuously monitor OS development in order to adapt potential non-HPC and non-RT aware capabilities for these particular use- cases. When there is no straightforward way to examine and debugthem, forcing Linux developers to test their new algorithms againstall the metrics for each individual use-case is impractical.

A practitioner typically starts by producing a synthetic workload tomeasure the OS noise. Workload examples include sysjitter and itsclone, oslat. These utilities use architecture-specific instructions toloop over reading the time.
When there is a gap between two successive time

readings that is more than a specific threshold, they classify that as a jitter. These tools don't try to link jitter to an underlying issue.

Practitioners must watch the system to determine the root cause. Tracing is the most effective method of system observation. The trade-off between information and overhead is where employingtracing runs into trouble. The user must establish a relationshipbetween the noise and the tracing data using a workload and tracing features. The fundamental reason why this link isn't always achievable is that the workload and the tracing features aren't alwaysaware of one another. It is important to note that workload can also detect hardware-induced noise at dozens of microseconds. Hardwarestalls brought on by shared resources, such as those in hyper-threaded processors or execution contexts with a higher priority than the OS, such as SMIs, can have a side effect called hardware noise. It is difficult to see the occurrences via trace because these acts are not a consequence of the operating system. This noise, which the workloadsees but the trace doesn't, generates a grey region that frequently leads the analysis astray.

Proposed Approach

In this research, we offer an integrated synthetic workload and tracing solution that tries to combine the benefits of workload- and tracing-based approaches while reducing the shortcomings of each solution.

- Define the composition of the OS Noise on Linux from a real-time HPC point of view, among other actions performed for such an approach.
- Identify the minimal number of tracing events necessary to demonstrate the underlying cause of each noise with a finiteoverhead.
- Make a synthetic workload that is aware of tracing to enablea clear correlation between the trace and the noise.
- Make the method production-ready with a uniform and user-friendly interface.

1.2 Composition of OS Noise for Real-Time HPC Workload:The generalised definition of OS noise adopted in this study is as follows:((OS)-Generalized Noise) Definition 1. When a job is allocated to a CPU and is ready to begin, the OS noise is all the time the CPU spends executing instructions that are not related to that task.

The definition broadens the traditional interpretation of OS noise, which normally only accounts for OS-related activities and overheads, by taking into account the time consumed by any intervening computational activity, including both OS-related activities and threads running in regular user-space. This is important because it means that any computational activity that could interfere with the measurement thread would also interfere with any other user thread running with the same scheduler and scheduler settings (for example, priority), regardless of whether it is an OS thread or not. As a result, it would be a real source of noise that the system's fine-grained tuning would need to take into account.
This expanded definition allows for an intriguing connection to be made between the high-priority interference frequently taken into account in real-time systems theory and OS noise, a metric from the HPC realm.
This broadens the scope of the approach beyond the HPC and NFV use cases, making it possible to practically profile all the sources of interference that may impact a task running under a specific scheduler configuration, such as a thread running at a specific priority under Linux's fixed-priority scheduler. In fact, because of this more inclusive definition, the osnoise tracer can now be used to monitor all high-priority interference in a broader sense, in addition to to noise that is specifically connected to the operating system.
Under Fixed-Priority Scheduling, Generalised (OS)-Noise. We consider the situation where a designer must decide whether a thread of interest (such as a constrained-deadline sporadic task [2]) will finish on time in a partitioned scheduling setting where workloads are considered for each processor separately. In order to achieve this, the conventional worst-case response time equation might be used.

Although it is theoretically possible to use Equation at design time, it is frequently challenging. In fact, it is challenging to obtain accurate WCET estimates for user threads on modern heterogeneous computing platforms due to a number of design principles used to improve average-case performance (e.g., complex cache hierarchies [2], un-revealed memory controller policies [3], out-of-order execution, etc.). It is considerably more challenging to obtain an arrival curve for OS threads and interrupt service routines because

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

they also lack knowledge about the arrival pattern. In these circumstances, the high-priority interference in Equation (1) can be empirically measured using osnoise. For instance, the system engineer can configure osnoise to run under SCHED_FIFO at the same priority, exposing the measurement thread to the same sources of noise, to estimate the high-priority interference faced by an NFV workload running at a given priority under SCHED_FIFO (a common use-case).

4. CONNECTED WORK

Operating system noise has long been acknowledged to have detrimental effects on workload performance [7]. Petrini et al.'s [5] study is one of the earliest efforts addressing the issue of identifying OS noise, and it discovered and reduced several sources of noise for an HPC application operating on the ASCI Q supercomputer. In a subsequent work, the investigation was expanded [2]. By injecting interference at the OS level, Ferreira et al. [2] presented a characterization of application sensitivity to the noise.

Workload and trace-based approaches are the two categories into which Linux tools for detecting OS noise are split in the paper's introduction.

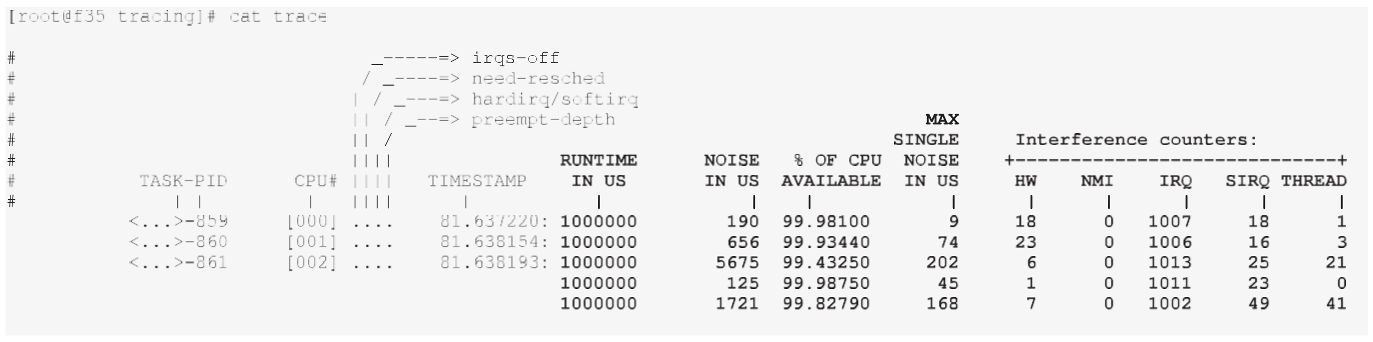
For example, if Γh is a sporadic thread with minimum inter-arrival time Ti, it holds ηhΔ=Th. Some workload-based methods run micro- benchmarks with a known duration and measure the difference between the expected duration of the micro benchmark and the actual time needed to process it. For this category, Sottile and Minnich's [3] Finite-Time Quantum (FTQ) benchmark is one pertinent example. The number of fundamental operations carried out in a given amount of time was assessed by FTQ. Another study by Tsafrir et al. [1] measured OS noise using micro benchmarks in combination with a "smart timers"-based technique. of these technologies, however, are unable to identify the underlying sources of the OS noise.

The usage of additional tools that take a trace-based approach can solve this issue. De et al. [3], for instance, proposed a technique to locate the OS jitter sources using kernel instrumentation andOProfile. The authors gathered data on interruptions and preemptions. Later, Morari et al. [6] suggested an alternative instrument to measure OS noise, but they did so by utilising a similar(but extended) method based on kernel instrumentation and extending the LTTng [37] tracer. Regarding [3], the work by Morari et al. enables the capture of other OS noise causes, such as softirqs. Another method to instrument the Linux kernel and quantify OS noise during application execution by using KTAU [3] was suggested by Nataraj et al. [3]. Gonzalez et al. [4], in more recent work, introduce Jitter-Trace, a programme that makes use of data from the perf tracer. However, none of these trace-based approaches take into account how workloads interpret noise.

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

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 11, November-2023



```
[root@f35 tracing]# cat trace

#                                  _-----=> irqs-off
#                                 / _----=> need-resched
#                                | / _---=> hardirq/softirq
#                                || / _--=> preempt-depth                              MAX
#                                || /                                               SINGLE     Interference counters:
#                                ||||               RUNTIME      NOISE  % OF CPU  NOISE    +-----------------------------+
#           TASK-PID      CPU#   ||||   TIMESTAMP    IN US       IN US  AVAILABLE  IN US     HW    NMI    IRQ   SIRQ THREAD
#              | |          |    ||||      |           |           |       |         |        |      |      |      |      |
            <...>-859    [000] ....    81.637220: 1000000       190   99.98100       9       18      0   1007     18      1
            <...>-860    [001] ....    81.638154: 1000000       656   99.93440      74       23      0   1006     16      3
            <...>-861    [002] ....    81.638193: 1000000      5675   99.43250     202        6      0   1013     25     21
                                                  1000000       125   99.98750      45        1      0   1011     23      0
                                                  1000000      1721   99.82790     168        7      0   1002     49     41
```

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

Sysjitter [2] is a workload-based solution that is frequently usedby professionals. By running a thread on each CPU and keeping noteof how long the thread is not operating, for example because of OS operations, it analyses OS noise. Oslat [3], jHicckup [4], and MicroJitterSampler [5] are further comparable tools. All .

In contrast to earlier work, the osnoise tool suggested in this research combines the best aspects of workload- and trace-based approaches, making it possible to identify the underlying causes of operating system noise while also taking into account how the workload interprets the noise.

The only tool for directly monitoring OS noise that has lately been included in the core Linux kernel [1] is osnoise, which is available for use on a vast array of devices.

5. The Osnoise Tracer

This part introduces the osnoise tracer, which makes use of the guidelines in part 2.1 to accurately profile each task's execution time by deducting the time spent on each interfering activity from the task's measured runtime. The programme can be used with any of Linux's preemption models, from the non-preemptive kernel through PREEMPT_RT, and is not restricted to any one particular preemption model.We first give a high-level overview of the tool before getting into the internals. Osnoise includes two parts, the workload and the tracing components, as was already described.

5.1 The osnoise Workload Threads

Measurement threads for the osnoise workload operate per- CPU.On each CPU, osnoise automatically spawns a periodic kernel thread. Any Linux scheduler, including SCHED_DEADLINE, SCHED_FIFO, SCHED_RR, or CFS, may be given the task of scheduling the kernel thread.Every thread has a runtime that is predetermined. The workload thread's main goal is to identify time that was taken from its execution and is therefore regarded OS noise. Every osnoise thread functions by reading the time repeatedly. A fresh noise sample is gathered whenever it notices a gap between two successive measurements that is greater than a predetermined tolerance level.

The function trace_- local_clock() is used to read the time. This architecture-specific nonblocking function offers a compact CPU- level coherent timestamp with the same precision as other ftrace tracing techniques, at the nanosecond granularity.

Preemption and IRQs are turned on for the thread. This makes it possible for any Linux task abstraction to preempt it at any time.

The workload presents a summary of the OS noise encountered by the current activation after runtime microseconds have passed since the first time read of the current period. This summary is shown utilising Linux's tracing features, as shown in Fig. 2.

According to the osnoise summary, RUNTIME IN US, or the number of milliseconds that OSNOISE looped over the timestamp. NOISE IN US, or the total quantity of noise measured inmilliseconds during the related runtime. PERCENTAGE OF CPU AVAILABLE, or the portion of the CPU that the osnoise thread had access to throughout the measurement period. MAX SINGLE NOISE IN US, or the longest single instance of noise that wasrecorded during the runtime in milliseconds.

The interference counters: osnoise keeps an interference counter for each type of interference among the classes NMI, IRQs, softirqs, and threads. The interference counter is raised in response to an entrance event of an activity of that type.

Because osnoise was running on a virtual system, the interference caused by virtualization is identified as hardware noise, which is why Fig. 2 displays a high number of hardware noisesamples.

5.2 The Osnoise Dimensions

There are a few parameters for the osnoise tracer. These choices are available through the ftrace interface, and they are:

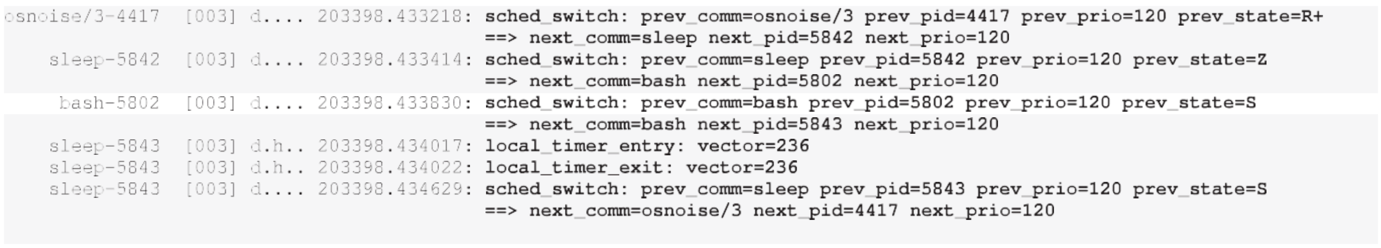
CPUs on which an osnoise thread will run are listed as osnoise/cpus. osnoise/period_us: the osnoise thread s's

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 11, November-2023

period (in milliseconds).osnoise/runtime_us: the duration (in milliseconds) for which an osnoise thread will search for noise occurrences.When a single noise event occurs that exceeds the configured value in milliseconds, the system tracing is stopped by the osnoise/stop_tracing_us command. This option is disabled by writing0.When total noise occurrence exceeds the configured value in milliseconds, the system tracing is stopped using the osnoise/stop_tracing_total_us command. This option is disabled by writing 0.tracing_threshold: the minimum difference in time betweentwo time reads to be regarded as a noise occurrence, expressed in milliseconds. The default value will be utilised if the value is set to 0,which currently 5µs.

osnoise tracepoints relay is less and more comprehensible.It is significant to note that the duration supplied by the irq_noise and thread_noise are free of interference with regard to the noise reported in Fig. 4.For instance, the start time of the local_timer:236 is later than that of the sleep-5843.This indicates that, in a case of nested noise, local_timer:236 preempted sleep-5843.However, the local_timer:236 subtracted its own length from that of sleep-5843.By eliminating the meticulous work of manually computing these variables or computing them using a script in user-space, this makes it easier to debug the system.



```
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
```

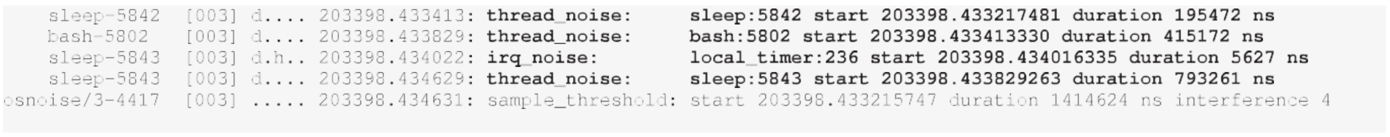
Fig. 3. Example of tracepoints: IRQ and thread context switch events read from ftrace interface4

entry and exit events and uses it to: 1) account for how often each of these task classes added noise to the workload; 2) compute the value of the interference counter used by the workload to determine how many interferences occurred between two consecutive reads of the time; and, 3) compute the value of the interference counter used by the workload to determine how long the present interrupting task willtake to complete; 4) using the procedures discussed in Section 2.1 to deduct the noise occurrence time of a preempted noise occurrence.A single tracepoint from osnoise is generated at the exit probe of each of these interference sources, indicating the noise-free execution duration of the task's noise detected via trace.The osnoise workload emits a tracepoint whenever a noise is discovered in addition to the tracepoints and the summary at the end of the period. This tracepoint provides information on the interference that occurred between the two successive time reads as well as the noise that was detected by workload. To clearly identify the source of a given noise, interference counters are essential.For instance, in Fig. 4, the last line represents the tracepoint produced by the workload and refers to the previous four interferences, while the first four lines reflect the noise as recognised by the trace. The data for Figs. 3 and 4 came from the same trace file.The former contains the tracepoints that were previously present, whereas the latter contains the new tracepointsthat osnoise added to the kernel. With these two examples, it is clear to see that the information the

The osnoise Tracing Features

One of the fundamental building blocks of Linux kernel tracing is the tracepoint. The tracepoints are locations in the kernel code where a probe can be connected to execute a function. They are mostfrequently used to gather trace data. Take the callback function that ftrace registers to the tracepoints as an example.These callback processes gather the information and store it in a trace buffer. A tracing interface can then access the data in the trace buffer. An illustration of tracepoint output using the ftrace interface is shown inFig. 3.Tracepoints can be used for more than just buffering data. Many more use cases have made use of them. For instance, alter network packets or modify the kernel at runtime [2].

(This work is licensed under a Creative Commons Attribution 4.0 International License.)



```
    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]

.

has been minimised, it is still feasible to pre-process data in the tracepoints in a way that reduces the amount of data written to the trace buffer. When trace processing has a lower overhead than writing trace to the buffer, this strategy has demonstrated good success in lowering tracing overhead [4].

The present tracing infrastructure is used by the osnoise tracer intwo different ways. It creates a new set of tracepoints with pre- processed information and adds probes to existing tracepoints to collect information.

Linux already provides tracepoints that stop threads, softirqs, andIRQs at their entry and exit points. Osnoise attaches a probe to all

Additionally, tracepoints can be used to enhance the tracing process.While the overhead associated with writing data to the trace buffers
Additionally, by lowering the amount of data stored in the trace buffer, resource consumption and overhead are decreased.Another crucial point to note is that, as shown in Fig. 5, the overall noise noticed via trace accounts for 1409532 ns6, whereas the noise observed by workload reports 5092 ns more (1414624 ns).

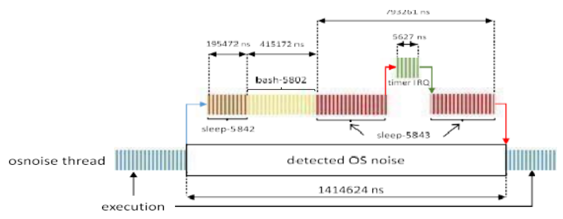


fig. 5. Graphical representation of Fig. 4.

5.3 The osnoise Internals

The Linux kernel's parallel and re-entrant code makes it difficultfor the osnoise tracer to measure potential noise sources at the single-digit millisecond scale. This section lists some of these difficulties along with solutions.

5.3.1 Task and Memory Model

The SPMD model, which is described in Section 2, is used in osnoise to simulate applications created with it. The osnoise workload component is run via a per-CPU kernel thread that iscreated when the tracer is dispatched. Each per-CPU thread's affinityis set up to only run on the target CPU. The user can modify the configuration of each thread while the workload is executed. Since this data is only retrieved sometimes outside of the primary workload loop, it is not a concern for the measurement stage. The configurationis secured with a mutex.

The thread only accesses the per-CPU structure that corresponds to the CPU on which it is now running, which is where the runtime data used during the measurement phase is organised. The goal of osnoise is to mimic a user-space workload that adheres to the specifications listed in Section 2.1. All of the OS job types discussedin Section 2 can specifically preempt a userspace workload on Linux. There are ways to temporarily turn off thread preemption, softirqs, and interrupts, but they come with certain unfavourable consequences. For instance, turning off interrupts is an expensive action that should be avoided when overhead is to be kept to a minimum, such as on the Linux tracing subsystem.

5.3.2 Addressing Code Reentry

Osnoise's capacity to correlate the workload with the quantity of interferences that result in an observed noise is one of its primary advantages. This capability is crucial to prevent debugging from going in the wrong directions and delaying the troubleshooting of crucial and expensive systems by preventing conjecture about the events that generated a noise. However, while taking into account the probability of preemption, reading the current time consistently with the interference counter is not a simple task. Think about the pseudo-code in Fig. 6, for instance. One interrupt more than the number at the moment the timer was read would be taken into consideration by the interference counter.

These issues have primarily been resolved in osnoise via two techniques: compiler barriers and local atomic variables. Fig. 7 displays an excerpt from the present code.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 11, November-2023

A local atomic integer, also known as an atomic type of integer coherent in the local CPU, is the definition of the interference counter. The current time is read between two interference counter reads to make sure it agrees with the interference counter. Compiler barriers are also used to prevent compiler improvements that can reorder the interference counter read and cause the current time to no longer be protected.Other non-atomic processes, include calculating the interference-free noise for the osnoise tracepoints in Fig. 4 using the interference-free noise technique.

6. EXPERIMENTAL RESULTS

This section provides information on how osnoise is used to measure and trace a system. The computer is a Dell workstation with a 12-core, 24-thread AMD Ryzen 9 5900 processor. The machine is set up with Fedora Linux 35 server and utilises a PREEMPT_RT patched kernel 5.15. To get a summary of the OS noise and a histogram of each noise event, the osnoise tool from rtla has been used.The system's initial configuration under consideration is referred to as "as-is" and has no adjustment applied. When best practises for CPU isolation are used, the system is considered to be tuned. In this scenario, the CPUs f0; 1g are set aside for user housekeeping and operating system duties.The workload execution isreserved for CPUs f2;... ; 23g, and osnoise is configured to execute on these CPUs. One step in system tuning is to set the CPU frequency governor for performance. Utilising CPU Isolation Features, Enabling RCU Callbacks Offload, Enabling Nohz_Full Configuration, and Transferring All Potential Kernel Threads and IRQs to the CPUs F0; 1G are the other four steps.

The task priority for osnoise workload threads is set to default (SCHED_OTHER with 0 nice). Users frequently establish a real- time priority for the task in the NFV use case, though. Additional experiments have been carried out to assess this particular circumstance as well. The osnoise workload has been configured to run with priority 1 under SCHED_FIFO in the experimentsdesignated as FIFO:1.

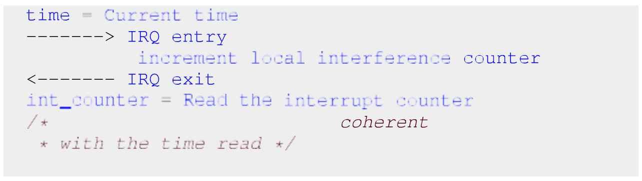
The workload has been configured with default parameters for the typical situation (i.e., SCHED_OTHER with 0 nice), thus it runs with a one-second runtime and period while attempting to monopolise the CPU.

It was necessary to configure the system and workload in more detail for the FIFO:1 situation. In order to prevent RCU [11] stalls, the system configuration includes turning off Linux's runtime throttling mechanism, allowing real-time threads to use more than 95% of the CPU, and setting the ksoftirqd (the thread in charge of processing softirqs when using PREEMPT_RT) and RCU per-CPU threads with FIFO 2 priority. The workload is set to run for 10 seconds at a time, and the runtime is adjusted to allow 100µs between each period so that any thread that is starving for time can execute. The tolerance threshold was set to 1 µs in the end.

6.1 Percentage of OS Noise

For all tune/FIFO priority scenarios collecting the OS noise summary, a six-hour experiment was run. Fig. 8 displays a summary of the percentage OS noise.The greatest observed total noise for the *As-is* system was 0:5484%, whereas the minimum for both Tuned cases was 0:00001%. It is also feasible to observe that the CPU isolation features are the primary source of OS noise reduction. The majority of the work still left for isolated CPUs is necessary for the OS, mostly in the IRQ context, which is the cause.In order to limit the worst-case scheduling latencies, these experiments were carried out using the PREEMPT_RT kernel, which has additional overheads in the average scenario.In this configuration, Linux is able to offer low OS noise for polling-based applications and low latency for interrupt-based workloads in a single solution. This adaptability is crucial for NFV deployments that include dynamic and varied workloads on the same host.

Fig. 6. Code reentrancy problem when incrementing the



```
time = Current time
-------> IRQ entry
          increment local interference counter
<------- IRQ exit
int_counter = Read the interrupt counter
/*                              coherent
 * with the time read */
```

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 11, November-2023

6.2 Analysis of OS Noise Occurrence

For every tune/FIFO priority situation, an experiment lasting six hours was run, capturing a histogram for every instance of noise that was found. This experiment is crucial for the NFV use case since a single extended noise event could result in a network packet processing queue overflow.

Fig. 9 presents the findings.

With the help of this experiment, it is feasible to demonstrate in Fig. 9a the primary issue with using the system *As-is*. The maximum value of the Theosnoise workload, which recognised 230 out-of-scale noise samples, was 13045 milliseconds. Additionally, Fig. 9b demonstrates that employing *FIFO:1* in the system as-is is a simple way to lower the maximum single noise occurrence value.

As-is using FIFO:1: however, when compared to the Tuned alternatives in Figs. 9c and 9d with or without employing FIFO:1, it has two significant drawbacks. The high frequency of noise occurrences is the first. The nohz_full option in the Tuned experiment limits the frequency of scheduler ticks, which in turn reduces the execution of the ksoftirqd kernel thread that checks for expired timers and subsequent actions. The tail latency, which is less on the Tuned instances, is another distinction. Section 6 explores this distinction.3. When compared to the system as-is, the findings with the system tuned in Figs. 9c and 9d reveal that the tune significantly alters the entries and length of each noise event. To make the Tuned examples easier to see, Figs. 9e and 9f have been included.

6.3. Using osnoise to Find Latency Sources

With regard to tail latency, the experiment using the system as-is and FIFO:1 produced an intriguing result, with just a small number of samples exceeding the 30 ms threshold. The osnoise tracer was configured to trace the osnoise events, stopping the tracer when a noise occurrence above 30 ms was detected, in order to better understand the causes of these occurrences that last longer than 30 ms. Figure 10 depicts the trace with only osnoise events. It displays an interrupt noise that is brought on by the eno1 ethernet driver's interrupt number 62. Immediately following the ksoftirqd's scheduling, a long-duration noise incident occurs. In the PREEMPT_RT kernel, the softirq jobs context is managed by the ksoftirqd thread (recall from Section 2.1 that under PREEMPT_RT, the softirq context does not exist, and softirq jobs run in the thread context).The only events that provide information on softirqs were enabled after it became clear that softirqs are what's causing the issue since choosing a minimal number of tracing events prevents overhead from having a significant impact on the system's timing behaviour. In Fig. 11, the trace once more displayed a comparable output. This trace demonstrates that the ksoftirqd's activation was caused by the network receive (NET_RX) softirq. The network driver, which is running in the same CPU, activates the NET_RX softirq. This is a consequence of the interrupt being caused by the eno1 ethernet driver. In light of this proof, the IRQ 62 was set up to fire in the CPUs 1:20:1. The experiment in Fig. 9b was repeated with this configuration applied for six hours, and the results are displayed in Fig. 12.

```
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

The tuned kernel was able to produce consistent results, and the tuned kernel employing FIFO:1 was able to deliver noise occurrences with a maximum time below 5 ms. This is due to the real-time scheduler deferring background OS processes that are carried out by threads without causing a systemic problem. As an illustration, jobs are distributed across all CPUs via k workers threads that carry out deferrable work [5] to prevent serious issues, these still need to be able to operate. As a result, it is wise to be aware of this property and give up some CPU time when it would not negatively impact performance (for example, when network buffers are empty), even if it is only for a brief period of time like 100 µs once every 10 seconds.

These tests further demonstrate the minimal influence the osnoise internals have on the evaluation, allowing the user to access data at the µs granularity commonly utilized by professionals on other tools like cyclic test. It is important to stress that the conclusions presented in this section only pertain to the specific case at hand. The hardware, CPU count, auxiliary operating system features, and ambient conditions may all affect the results. Thus, the value of a programme that provides an integrated OS noise benchmark and suggestions for system fine-tuning.

The tail latency could be reduced to values like the system with just this configuration. tuned, with osnoise's debugging assistance.

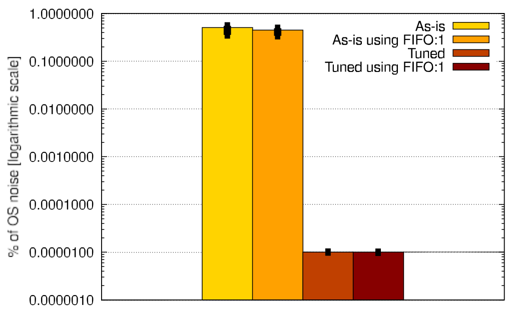
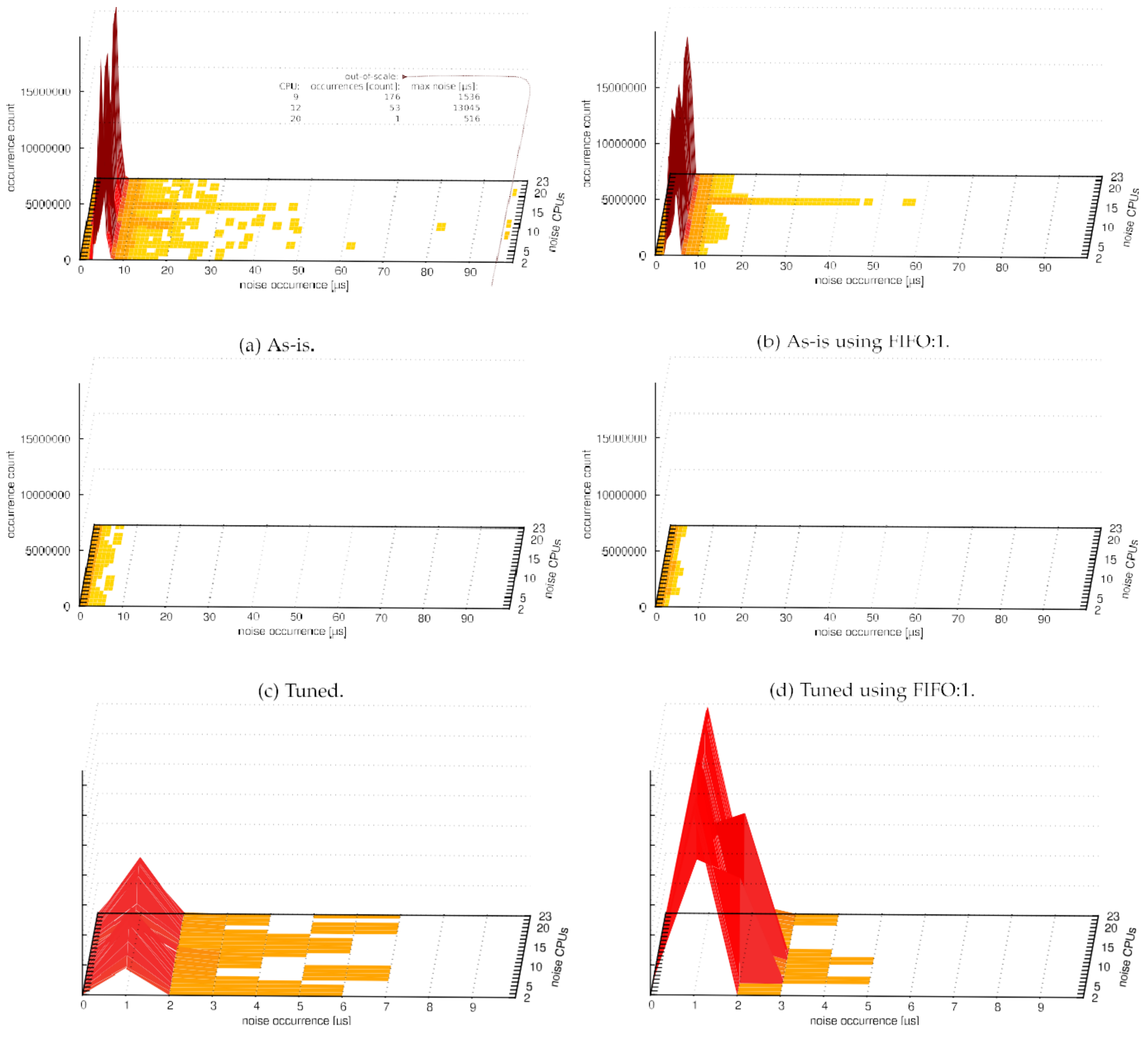


Fig. 8. Average percentage of OS noise observed by the workload on different scenarios. Error bars represent the range between minimum andmaximum percentage.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)



out-of-scale:

CPU:	occurrences [count]:	max noise [µs]:
9	176	1536
12	53	13045
20	1	516

(a) As-is.

(b) As-is using FIFO:1.

(c) Tuned.

(d) Tuned using FIFO:1.

(e) Tuned - Zoom of Figure 9c.

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: enol: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:                     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: enol: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:                     duration 30938 ns interference 2
     osnoise/16-2501    [016] ....   533.347996: osnoise_main: stop tracing hit on cpu 16
```

Fig. 11. `osnoise tracer supplemented with other events determining the source of latencies`[4].

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 11, November-2023

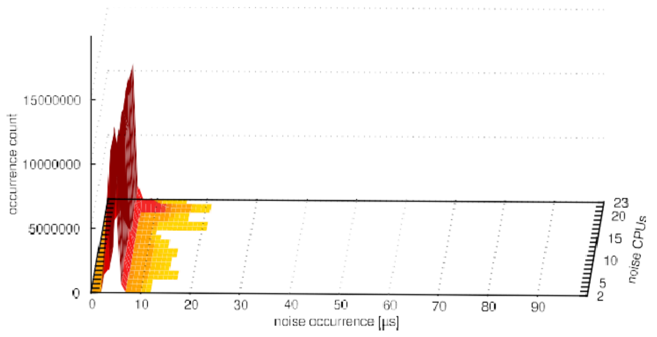


Fig. 12. After moving the network IRQ as suggested by the trace in Fig. 11, as-is utilising FIFO:1.

7. CONCLUSION AND FUTURE WORK

Modern low latency communications and network function virtualization have increased the demand for Linux systems with minimal OS noise and scheduling lag.These real-time HPC tasks demand noise of a few tens of microseconds or less.debugging these situations is a difficult effort, though.

Workload-based methods provide accurate measures but do not identify the underlying problem. Trace-based measures reveal the cause but do not accurately depict the noise that the thread has actually seen.Practitioners combine the two approaches, but doing sonecessitates a thorough understanding of the tracing features and frequently causes the inquiry to be misled because the trace is out of sync with the workload or adds too much overhead.

The osnoise tool combines the workload and tracing, delivering accurate information with little overhead by processing and exporting only the data required for identifying the primary reasons of the latency, providing a solid place to start the investigation.The use of the rtla osnoise interface to gather data has made it possible for the tool to function as a tracer and benchmark tool, according to the experimental results. The test demonstrates Linux's ability to produce exceptionally low OS noise, with maximum sample noises as low as less than 5 µs. However, the tool's ability to follow the kernel and produce results at the necessary scale is more significant.The osnoise

tool and rtla osnoise interfaces are both built into the Linux kerneland are thus available to all Linux users.The osnoise tracer may be used with many other existing tracing tools because it makes use of the most fundamental components of the Linux tracing sub-system, such as performance counters offered by the perf tool or graphical user interfaces offered by LTTng and KernelShark. This opens up an infinite number of opportunities for future research, such as expanding the osnoise measurements to incorporate data from the memory/cache, workload-dependent techniques, other clock sources, and energy-aware techniques. Another option is to extend the analysis using a more formal methodology. Another option is to carry out experimental evaluationsusing different Linux real-time schedulers, such asSCHED_DEADLIN

.

REFERENCES

[1] "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 2018 *(references)*

[2] J. Clerk Maxwell, Third Edition, Volume 2 of A Treatise on Electricity and Magnetism. Clarendon, 2019, Oxford, pp. 68– 73.

[3] C. P. Bean and I. S. Jacobs, "Fine Particles, Thin Films, and Exchange Anisotropy," in Magnetism, vol. III, G. T. Rado,and H. Suhl, Eds., New York: Academic, pp. 271-350.K. Elissa, ―Title of paper if known,‖ unpublished.

[4] R. Nicole, "Title of paper with only first word capitalised," Journal of Name Abbreviation, under consideration.

[5] "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol.2, pp. 740–741, August 2018 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 2019]; Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa.

[6] The Technical Writer's Handbook, M. Young [6]. University Science, Mill Valley, California, 2020.

[7] The Technical Writer's Handbook by M.Young. University Science,MillValley, California,2019

(This work is licensed under a Creative Commons Attribution 4.0 International License.)