Detecting OS Noises: A Kernel Event Monitoring Approach for Reliable System Performance

Ben Grandy
Department of Computer Science
Brock University
St. Catharines, Canada
bg16pz@brocku.ca

Morteza Noferesti
Department of Computer Science
Brock University
St. Catharines, Canada
mnoferesti@brocku.ca

Naser Ezzati-Jivan
Department of Computer Science
Brock University
St. Catharines, Canada
nezzati@brocku.ca

***Abstract*—Understanding the impact of various OS noises on process performance and resource management presents significant challenges. The presence of diverse noise sources, combined with the multitude of events occurring within the OS, emphasizes the necessity for comprehensive analysis to effectively mitigate the adverse effects of OS noises. This paper introduces an approach based on kernel event monitoring to detect different types of OS noise and identify their root causes. The approach defines and monitors specific metrics on kernel events to detect CPU, Disk I/O, and Network I/O noises. sched_switches and timer events are employed for monitoring CPU noises, while the duration between block_rq_insert and block_rq_issue is measured for Disk I/O noise analysis. Similarly, the time delay between events related to network I/O is analyzed to identify potential network noise. The proposed approach's performance in detecting various OS noises is evaluated, confirming its effectiveness through test cases such as the *CPU Noise Test*, *Disk Noise Test*, and *Network Noise Test*. The experimental results demonstrate the efficiency and accuracy of the proposed approach in detecting OS noises through kernel-level event analysis.**

***Index Terms*—OS Noises, Process Performance, Kernel Event Monitoring, Root Cause Analysis, Reliability, Performance Interference**

I. Introduction

Performance analysis helps optimize the utilization of system resources such as CPU, memory, disk, and network. By identifying resource-intensive operations or inefficient algorithms, software performance analysis allows for improvements that maximize resource utilization, leading to better overall system efficiency. External noises can indeed significantly affect the performance of software [1], [2]. Noise (In this paper, we refer to noise instead of using the term "performance noise") refers to non-deterministic and undesired variations or fluctuations in system behavior that have an impact on the expected or desired performance levels. It represents interference or disturbances that introduce random or unpredictable elements, resulting in deviations from the anticipated performance patterns.

One example of noise impacting software performance is network congestion. In a distributed system or client-server architecture, high network traffic or congestion can introduce variations in response times and throughput. This noise can cause delays and affect the overall performance of the software, resulting in slower data transfers, increased latency, and reduced user experience. By analyzing and mitigating the effects of network noise, such as optimizing network protocols or implementing congestion control mechanisms, the performance of the software can be improved.

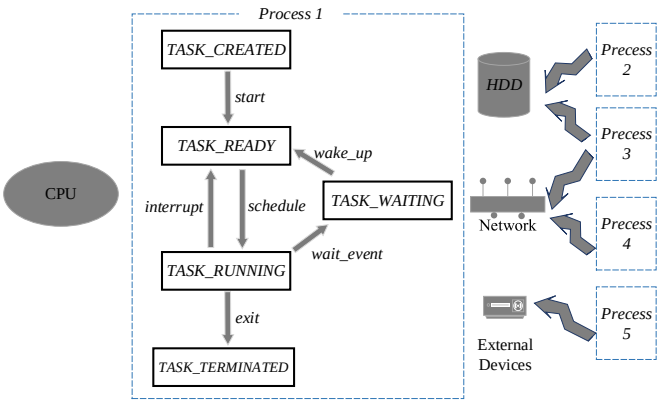


Fig. 1: Various noises throughout the lifecycle of a process can have a significant impact on its performance.

Noise can manifest in various forms, such as irregular delays, interruptions, or inconsistencies in system response times [1], throughput, resource utilization, or other relevant performance metrics [3]. It may result from various factors, such as hardware limitations, contention for shared resources [4], scheduling algorithms [1], [5], external interference [6], or workload variations [7].

Different types of noise root causes, particularly in high-performance computing, demand specific actions to effectively mitigate their impact [2], [8]. For example, addressing CPU noises may involve optimizing the balance between the number of threads and cores to enhance resource allocation. Similarly, mitigating network noises might require adjustments in traffic shaping or accounting techniques to efficiently manage network load. Conducting root cause analysis plays a crucial role in assisting system administrators in upholding system performance amidst noise. By identifying the underlying causes, administrators can implement targeted solutions to mitigate the effects and ensure optimal performance [9].

The performance of a process can be significantly affected

by various noises throughout its lifecycle. Fig. 1. illustrates a typical system where the lifecycle of *Process 1* is influenced by different processes [10]. When *Process 1* enters the *TASK_READY* state, its performance can be impacted by interruptions from the task scheduler. As the CPU is shared among all processes, the task scheduleŕs dispatch and allocation of the CPU can influence the performance of *Process 1*. Similarly, when the process transitions to the *TASK_WAITING* state, its performance can be influenced by other processes. If there is an extensive load on shared resources such as disk, network, or other external devices, it can adversely affect the performance of *Process 1*. Conducting a root cause analysis of the noises affecting *Process 1*ś performance is essential to identify and mitigate these issues, ultimately achieving optimal performance.

Kernel event tracing is a technique utilized to monitor and capture low-level events occurring within the kernel of an operating system [11]. It involves tracing and recording a diverse range of events, including interrupts, system calls, scheduler events, disk I/O operations, and network activities [12]. By exploring these events, kernel event tracing enables the detection of noise and facilitates root cause analysis. However, effectively analyzing a substantial volume of kernel-level events poses a significant challenge when applying noise detection and root cause analysis. Furthermore, accurately processing complex events generated at the kernel level are imperative for pinpointing the precise root causes of the observed noises.

In this paper, we address the significant challenges in understanding the impact of different OS noises on process performance and resource management. The presence of diverse noise sources and the multitude of events within the OS highlights the need for comprehensive analysis to effectively mitigate the adverse effects of OS noises. To address this, we propose an approach based on kernel event monitoring that enables the detection of various types of OS noise and the identification of their root causes. Our approach defines and monitors specific metrics on kernel events to detect CPU, Disk I/O, and Network I/O noises. For CPU noise analysis, we employ sched_switches and timer events, while Disk I/O noise is assessed by measuring the duration between block_rq_insert and block_rq_issue. Additionally, network noise is identified by analyzing the time delay between events related to the network I/O. We evaluate the performance of our approach in detecting different OS noises, confirming its effectiveness through test cases such as the *CPU Noise Test*, *Disk Noise Test*, and *Network Noise Test*. The experimental results demonstrate the efficiency and accuracy of our proposed approach in detecting OS noises through kernel-level event analysis.

The contributions of this paper are as follows:

1) Proposal of a framework for noise detection and root cause analysis through kernel-level event analysis.
2) Introduction of the "Disk Analyzing" technique, which leverages disk-related events, such as block_rq events, to detect noises associated with disk behavior.
3) Definition of the "CPU Analyzing" technique, which monitors various CPU events to detect noises related to CPU usage.
4) Definition of the "Network Analyzing" technique, which monitors different network-related events to detect noises associated with network behavior.
5) The *CPU Noise Test*, *Disk Noise Test*, and *Network Noise Test* were designed to showcase the capability of our approach in accurately identifying the underlying sources of noise in each respective domain.

The remaining sections of this paper are organized as follows: Section II provides a review of related works in the field of noise detection and root cause analysis. Section III presents the details of the proposed approach for noise detection and root cause analysis through kernel event monitoring. In Section IV, the effectiveness of the proposed approach is evaluated through *CPU Noise Test*, *Disk Noise Test*, and *Network Noise Test* use cases. Finally, Section V concludes the paper and discusses potential avenues for future research.

II. Related works

Numerous studies have been conducted to explore and tackle the complexities of performance noise detection. Considering the impact of noise on scalability and performance in a distributed environment, Edgar et al. [9] proposed an approach that utilizes Simultaneous Multi-Threading (SMT) to isolate applications from system interference, requiring no modifications to the operating system or the application itself. Instead of eliminating noise, their approach relocates it away from the critical path. By leveraging SMT, which enables multiple hardware threads per core, the method effectively mitigates noise and protects applications from system interference. Unlike core specialization, where specific cores are dedicated to system processing, this approach maximizes the utilization of all cores within a node for application execution. The results have demonstrated significant performance improvements, including a maximum enhancement of 2.4 times for a high-order finite elements shock hydrodynamics application.

In latency-sensitive applications, the interference from the operating system (OS) can significantly affect performance. Lameter in [5] defines OS noise as disturbances caused by the OS utilizing a processor, resulting from activities such as scheduling, interrupts, timers, and other events. These disturbances introduce delays and variations in execution time in user space processing. The Linux kernel, with its expanding range of features, contributes to this noise. To mitigate OS noise and enhance latency, additional measures beyond real-time scheduling policies (such as round robin and FIFO) are required.

Scientific applications often face interruptions from the operating system, despite the availability of multiple cores. Traditional task scheduling in Linux is not optimized for high-performance computing scenarios where a single job utilizes all cores. In [3], Hakan et al. investigate a soft-partitioning approach to isolate application processes from the operating system. They explore various methods, including configuration changes and invasive code modifications, to reduce

TABLE I: Noise detection approaches and different root causes.

Name	*IRQ noise*	*CPU noise*	*Disk noise*	*Network noise*	*Analyzing method*
Edgar et al. [9]	√	√	–	–	Monitoring CPU on distrbuted environment
Lameter in [5]	–	√	–	–	Monitoring OS scheduling policy
Hakan et al. [3]	√	√	–	–	Monitoring the adverse impact of OS clock
Hakan et al. [6]	√	√	–	–	Monitoring preemption caused by task scheduling
Nelson et al. in [13]	√	√	–	–	Monitoring Linux scheduler activity
Daniel Bristot et al. [1]	√	√	–	–	Kernel tracing to measure CPU-related noise
The proposed approach	√	√	√	√	multiple analysis on system-level execution traces

interruptions caused by the operating system, particularly by leveraging CPU affinity settings. The experimental results demonstrate that even at small scales, parallel applications benefit from these modifications, showing a 1.72% improvement. The research underscores the adverse impact of OS clock ticks on scalability and highlights the potential to achieve a noise-free system using a widely used commodity operating system.

In another study conducted by Hakan et al. [6] , the focus is on mitigating application interruptions by utilizing compile and run-time configurations within an unmodified Linux kernel. The authors also introduce an invasive approach that effectively eliminates involuntary preemption caused by task scheduling. Through experiments, they observe a 1.91% performance improvement in parallel applications, even at smaller scales. These findings highlight the potential benefits of optimizing system configurations and minimizing interruptions to enhance application performance.

Nelson et al. in [13] addresses the issue of Operating System (OS) noise, which can adversely affect the scalability of large-scale parallel applications. The authors present Jitter-Trace, a tool built on top of Linux Perf, that identifies and quantifies jitter sources in the OS. By analyzing the Linux scheduler activity, Jitter-Trace tracks the processes associated with the application and records jitter events when an application process is switched out for a system activity. The collected data provides task profiles and histograms of OS noise, enabling the identification of noise sources and the implementation of mitigation strategies. Jitter-Trace offers a low-overhead solution as it leverages the tracing and profiling capabilities of Linux Perf without requiring modifications to the kernel or the application.

Linux, with its real-time kernel options and advanced CPU isolation features, is increasingly important in Network Function Virtualization architecture, enabling low latency networked services. However, tuning Linux for these applications is challenging and requires a deep understanding of its execution model and tracing features. In [1], Daniel Bristot et al. explore the internal aspects of Linux that affect Operating System Noise (OS noise) from a timing perspective. They focus on CPU-related noise and define OS noise as the time spent by a CPU executing instructions not belonging to the assigned application task while the task is ready to run. The authors introduce ”osnoise,” an in-kernel tracer that measures and traces OS noise as observed by workloads, facilitating system analysis and debugging. The paper presents experiments demonstrating Linux's ability to deliver low OS noise and the effectiveness of osnoise in identifying the root causes of timing-related OS noise problems.

The paper focuses on the current state of noise analysis, which involves developing new heuristics to identify performance noise within a specific range of observable behavior. It introduces an approach for noise detection and root cause analysis based on system-level execution tracing. By analyzing low-level events, the approach enables efficient and transparent monitoring of system behavior. The proposed approach utilizes a three-module architecture with three analysis techniques dedicated to detecting CPU, disk, and memory noises. The effectiveness of the approach in noise detection is evaluated through multiple test cases.

Table I summarizes different noise detection approaches and the scopes of observable noise they analyze. Previous approaches have primarily focused on CPU-related noises that are specifically generated by the OS scheduler. However, the proposed approach aims to detect noise across all scopes by analyzing system-level execution traces. The table includes the names of the approaches, along with checkboxes indicating whether they analyze CPU noise, disk noise, and network noise. The proposed approach is highlighted as it analyzes all of these noise scopes using multiple analysis techniques on system-level execution traces.

III. The proposed approach

This section outlines the details of an approach for noise detection and root cause analysis using system-level execution. The proposed approach involves collecting events generated by the software and processing them through multiple analysis methods to detect noises across different scopes.

The proposed approach architecture is illustrated in Fig. 2. It consists of four main modules: *Event Gathering*, *Trace Handler*, *Noise and Root Cause Detector*, and *Visualization*. The *Event Gathering* module collects kernel-level events generated by the software to detect noise. These events are gathered with low overhead and without interfering with the software behavior. The *Trace Handler* module manages the events and creates traces that are associated with the same execution. Different analysis methods are employed in the *Noise and Root Cause Detector* module to detect noise and determine its root cause across various analyzing scopes, including *Disk Monitoring*, *CPU Monitoring*, and *Network Monitoring*. Finally, the *Visualization* module presents noise detection metrics in a

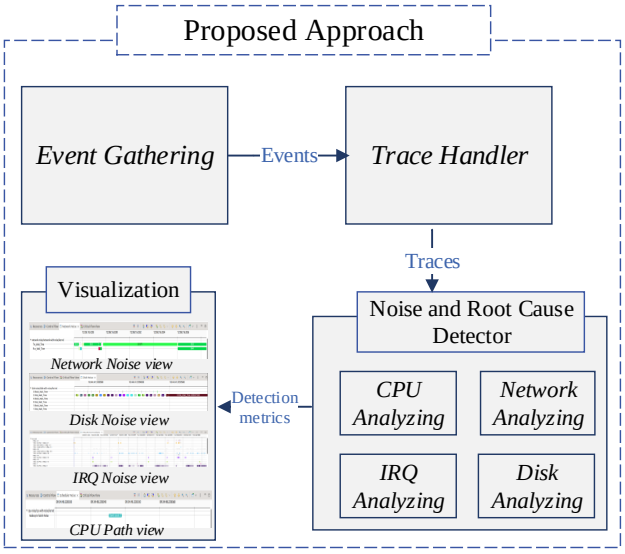


Fig. 2: The architecture of the proposed approach.

visual format, aiding system administrators in understanding the details of the noise and identifying mitigation strategies. The following sub-sections provide further details on each of the proposed modules.

A. Event Gathering module

During the execution of the software, there are numerous interactions between the operating system and the software itself. To detect noise without interference, the proposed approach captures events occurring between the operating system and the software. This is achieved through the *Event Gathering* module, which utilizes an open-source tracing tool called LTTng (Linux Tracing Toolkit: next generation) [14]. LTTng is known for its low overhead and scalability, making it suitable for tracing both the Linux kernel and userspace. It provides an easy-to-use interface, efficient memory usage, and precise timestamps across multiple processors, ensuring accurate noise detection without significant interference or performance impact.

The *Event Gathering* module is responsible for handling the high volume and diverse nature of events generated at the kernel level. Figure 3 illustrates the events that occur during software execution, each defined by attributes such as Timestamp, CPU, event type (e.g., system call, interrupt), event details (e.g., IP address), process ID, and thread ID. To collect the necessary attributes, the module utilizes tracepoints, which are hooks placed in the code that allow function probes to attach at runtime. When a tracepoint is reached, the probe is executed within the caller's context, and control returns to the caller after probe execution. By strategically placing tracepoints in Linux subsystems, valuable runtime information can be extracted. For instance, the `sched_switch` tracepoint indicates when one thread is replaced by another on a CPU,

Timestamp	CPU	Event type	Contents	TID	PID
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>
09:27:44.953 678 085	7	irq_softirq_entry	vec=1, context.packet_seq_num=0, cc	0	
09:27:44.953 678 981	1	irq_softirq_exit	vec=9, context.packet_seq_num=0, cc	0	
09:27:44.953 680 359	1	timer_hrtimer_start	hrtimer=0xffff8a45450a3220, functio	0	
09:27:44.953 712 454	1	sched_switch	prev_comm=swapper/1, prev_tid=0, ;	0	
09:27:44.953 719 625	1	sched_switch	prev_comm=kcompactd0, prev_tid=7	79	79
09:27:44.953 721 497	1	timer_hrtimer_cancel	hrtimer=0xffff8a45450a3220, context	0	
09:27:44.953 721 717	1	timer_hrtimer_start	hrtimer=0xffff8a45450a3220, functio	0	
09:27:44.953 726 519	7	irq_softirq_exit	vec=1, context.packet_seq_num=0, cc	0	
09:27:44.954 088 771	6	timer_hrtimer_cancel	hrtimer=0xffff8a4545323220, context	22774	22774
09:27:44.954 088 925	6	timer_hrtimer_expire_entry	hrtimer=0xffff8a4545323220, now=4C	22774	22774

Fig. 3: *Event Gathering* module - Collection of events and attributes

providing insights into the scheduling behavior of the Linux scheduler subsystem.

Multiple tracepoints and their corresponding recorded events are enabled in the *Event Gathering* module, including the following:

- `sched_switch`: Records events when a thread is switched from running on one CPU to another.
- `irq_handler_entry` and `irq_handler_exit`: Record events when an interrupt handler starts and finishes execution.
- `block_rq_insert`: Records events when a request is inserted into the block I/O request queue.
- `block_rq_complete`: Records events when a block I/O request is completed.
- `netif_receive_skb`: Records events when a network interface receives a new socket buffer.

These tracepoints allow the collection of specific events related to thread scheduling, interrupt handling, block I/O requests, and network interface activity. By enabling these tracepoints, the *Event Gathering* module can extract the necessary metrics for noise detection and root cause analysis.

B. Trace Handler module

The *Trace Handler* module is responsible for constructing the necessary relationships among events collected by the *Event Gathering* module. It analyzes the event stream, identifies causal patterns, and establishes temporal and logical dependencies between events. By understanding the order of events and extracting relevant metrics, such as the block queue and network queue waiting times, the *Trace Handler* module enables effective noise detection and root cause analysis techniques. It provides the necessary context and insights to analyze system behaviors and identify potential sources of noise.

To handle the stream of kernel events, the *Trace Handler* module is responsible for maintaining the sequence of required events for each process. Considering a multi-threaded application P, it is represented as a set $P = \{ES^k | 1 \leq k \leq m\}$, where each event sequence ES^k represents a sequence of events $< e_1^k, e_2^k, ..., e_l^k >$ generated during the runtime of P by the same thread. It should be noted that the timestamp of event e_i^k always precedes the timestamp of event e_j^k, for all $i \leq j$, ensuring the chronological order of events within

the event sequence ES^k. Additionally, all events in the event sequence ES^k belong to the same thread of the process P. By analyzing the event attributes such as timestamp, PID, and TID, the $TraceHandler$ module generates and organizes the sequence of events for each process, enabling further analysis and processing.

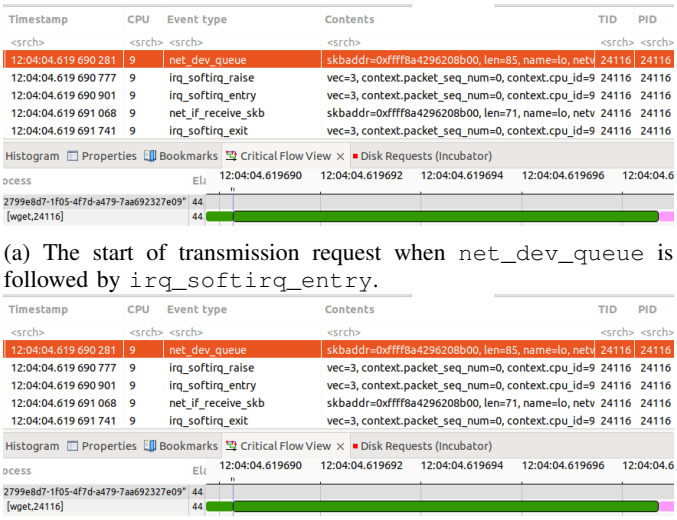
Let's consider a scenario where a process initiates a disk I/O operation, which involves a collection of events within the operating system, including system calls, interrupts, and function calls. The process goes through a series of events to complete the disk I/O operation. It starts with the `block_getrq` event, which represents the acquisition of a request for the disk I/O operation. Following that, the request is inserted into the waiting queue using the *block_rq_insert* event. Subsequently, the request is issued to the driver for processing through the *block_rq_issue* event. If the disk successfully processes the request, the *block_rq_complete* event is triggered, indicating the completion of the operation. This event releases the request data structure and awakens any blocked processes. The *Trace Handler* module organizes these events into a sequence ⟨`block_getrq`, `block_rq_insert`, `block_rq_issue`, `block_rq_complete`⟩ sorted based on the timestamp of the events. Furthermore, the *Trace Handler* module ensures that these events are associated with the same thread of the same program, maintaining their relationship. These sequences, along with other sequences generated during the runtime of the process, are then sent to the *Noise and Root Cause Detector* module for noise detection and root cause analysis.

C. Noise and Root Cause Detector

To effectively detect noise and determine the root cause of performance issues, the Noise and Root Cause Detector module incorporates three distinct analyses: *CPU Analyzing*, *Network Analyzing*, *Disk Analyzing*, and *IRQ Analyzing*.

1) CPU Analyzing: For CPU noise monitoring, *CPU Analyzing* component calculates the number of `sched_switch` events and the duration between them. The `sched_switch` event occurs when the CPU switches from executing one process (or thread) to another, indicating a change in the active execution context. A large number of `sched_switch` events indicates that the system has a high number of tasks and the processor switches frequently between them. This can be an indication of CPU noise, where the system is constantly switching between different processes, potentially impacting overall system performance.

On the other hand, analyzing the duration between `sched_switch` events and is `sched_wakeup` events is also important. The `sched_wakeup` event refers to the event where a task or thread in a system is awakened or signaled to start executing by the scheduler. The duration between a `sched_wakeup` event and a `sched_switch` event represents the time it takes for a task to be awakened and actually scheduled to run on a CPU. In a noisy system, this duration may increase due to the increased number of processes waiting for CPU time. To monitor CPU noise, the proposed approach



(a) The start of transmission request when `net_dev_queue` is followed by `irq_softirq_entry`.

(b) The completion of the `NETWORK` state when `net_if_receive_skb` is followed by `sched_waking`.

Fig. 4: The critical path of *wget* in a non-noisy network.

defines the *CPU_Wait_Time* metric, which is calculated using Equation 1 $sched\_switch_t^i$ and $sched\_wakeup_t^i$ indicates the time stamp of i-th occurrence of those events.

$$CPU\_Wait\_Time^i = sched\_switch_t^i - sched\_wakeup_t^i \quad (1)$$

2) Network Analyzing: The Network Analyzing component is responsible for monitoring the network activities of the software. It captures and analyzes network-related events, such as network packets, socket operations, and network protocol activities. This analysis helps identify network noises, such as network congestion, packet loss, or delays in network communication.

To monitor network noises, the proposed approach calculates metrics related to the send queue and to the receive queue. To explain the metrics, let's consider the critical path of a simple *wget* program trying to download a website, as depicted in Fig. 4a. In the figure, the green color represents the *wget* process in the `RUNNING` state, and the pink color indicates the `NETWORK` state. Before the *NETWORK* state begins, there is a `net_dev_queue` event. The tracepoint `net_dev_queue` records events when a network device queues a packet for transmission. In the absence of noise, we expect to see another event, `irq_softirq_entry`, in close proximity to this event, with the same process ID (PID) and thread ID (TID). The `irq_softirq_entry` tracepoint captures events when a software interrupt or softirq is triggered, providing information about the interrupt or softirq being processed. In simple terms, the occurrence of `irq_softirq_entry` indicates that the packet is being sent. However, if there is noise present, the duration between the `net_dev_queue` and `irq_softirq_entry` events becomes longer, indicating delays or disruptions in the packet transmission process.

The completion of the NETWORK state is depicted in Fig.4b. The `sched_waking` event occurs when a thread is awakened and scheduled to run on a CPU (indicated by the thread transitioning to the `PREEMPTED` state, shown in orange in the critical path view). In the context of network noise analysis, when the `net_if_receive_skb` event, which indicates the reception of a network packet by the network interface card, is immediately followed by the `sched_waking` event, it signifies the completion of the NETWORK state in a non-noisy network.

Considering $irq\_softirq\_entry_t^i$ and $net\_dev\_queue_t^i$ as the time stamp of i-th occurance of `irq_softirq_entry` and `net_dev_queue` respectively, the transmission wait time, denoted by Trs_Wait_Time is calculated by Equation 2. The receive wait time, denoted by $Rcv_Wait_Time$is calculated by Equation 3. By monitoring these durations, the proposed approach can detect network noises and evaluate their impact on the critical path of network-related tasks.

$$Trs\_Wait\_Time^i = irq\_softirq\_entry_t^i - net\_dev\_queue_t^i \tag{2}$$

$$Rcv\_Wait\_Time^i = sched\_waking_t^i - net\_if\_receive\_skb_t^i \tag{3}$$

3) Disk Analyzing: The *Disk Analyzing* technique focuses on monitoring the disk I/O operations performed by the software. It captures events related to disk read and write operations, disk queues, and disk latency. By analyzing these events, it can detect disk-related noises, such as high disk utilization, long disk queues, or slow disk access times.

In the context of disk I/O operations, various events are related to capturing different stages and aspects of the operation. As mentioned in the paper by Daoud et al. [15], the `block_getrq` event indicates that the software requires a specific disk block for its I/O operation. This event is followed by other events in the disk I/O sequence, including `block_rq_insert`, `block_rq_issue`, and `block_rq_complete`, which collectively represent the complete lifecycle of the disk I/O operation.

These generated events provide valuable insights into different noise problems that can occur during disk I/O operations. For example, if there is external noise affecting the disk, there may be a significant time interval between the `block_rq_issue` and `block_rq_complete` events, indicating delays in the completion of the I/O operation. On the other hand, if another process is generating an excessive number of hard requests, it can lead to a consistently busy `WAITING` queue in the driver, resulting in an increase in the time between the `block_rq_insert` and `block_rq_issue` events.

To effectively monitor and analyze noise problems caused by disk I/O operations, the *Disk Analysis* component defines two metrics that monitor requests in the $Block\_Queue$ and $Disk\_Queue$. The $Block\_Queue$ consists of requests that have encountered the `block_rq_insert` event but have not yet encountered the subsequent `block_rq_issue` event. Similarly, the $Disk\_Queue$ comprises requests that have encountered the `block_rq_issue` event but have not yet encountered the `block_rq_complete` event.

To calculate the waiting time for both queues, the Equation 4 and Equation 5 are defined:

$$Block\_Wait\_Time^i = block\_rq\_issue_t^i - block\_rq\_insert_t^i \tag{4}$$

$$Disk\_Wait\_Time^i = block\_rq\_complete_t^i - block\_rq\_issue_t^i \tag{5}$$

Where $block\_rq\_issue_t^i$, $block\_rq\_insert_t^i$, and $block\_rq\_complete_t^i$ represent the timestamps of related events for the same thread in the same process. The metrics $Block\_Wait\_Time$ and $Disk\_Wait\_Time$ are designed to measure and track various aspects of the I/O noise related to disk activity or process activity inside the system. By monitoring these metrics and analyzing the corresponding events, the *Noise and Root Cause Detector* module can detect and identify issues related to disk I/O performance and assist in root cause analysis.

4) IRQ Analyzing: The *IRQ Analyzing* technique focuses on monitoring interrupt noises during software execution. It captures events related to interrupt handling, specifically `irq_handler_entry` and `irq_handler_exit`. The `irq_handler_entry` event occurs when an interrupt request (IRQ) handler begins its execution. When a hardware device generates an interrupt, the CPU interrupts its current execution and jumps to the corresponding IRQ handler routine. This event marks the entry point of the handler, where the specific IRQ is being serviced. Monitoring and analyzing `irq_handler_entry` events are important for understanding the timing and behavior of interrupt handling in a system.

On the other hand, the `irq_handler_exit` event is the counterpart of the `irq_handler_entry` event in interrupt handling. It signifies the completion of the execution of an IRQ handler. After the handler has finished processing the interrupt, the CPU resumes its previous execution at the point where it was interrupted. The `irq_handler_exit` event is crucial for measuring the duration of IRQ handling and gaining insights into the overall performance of interrupt handling in a system.

The *IRQ Analyzing* technique calculates the *IRQ_Wait_Time* as the duration between the `irq_handler_entry` and `irq_handler_exit` events. The specific equation used for this calculation, referred to as Equation 6, Where $irq_handler_entry_t^i$ and $irq_handler_exit_t^i$represent the timestamps of related events for the same thread in the same process.

$$IRQ\_Wait\_Time^i = irq\_handler\_exit_t^i - irq\_handler\_entry_t^i \tag{6}$$

D. Visualization module

The *Visualizer module* utilizes the open-source trace analysis software Trace Compass [16] to provide graphical views that highlight noise in the process execution for system

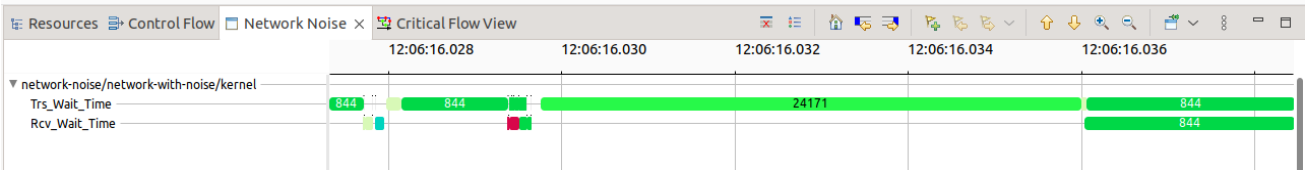


Fig. 5: *Visualizer* module - *Network Noise* view

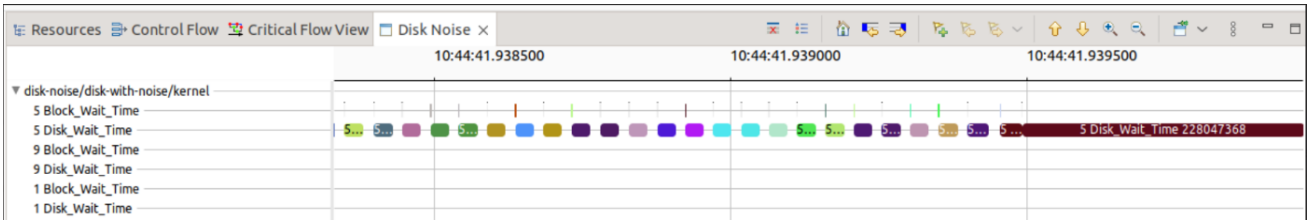
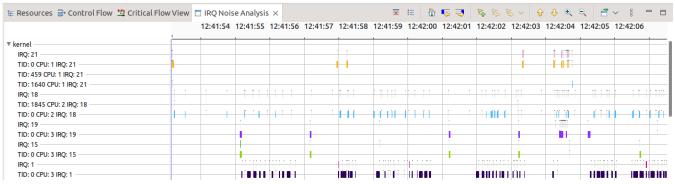
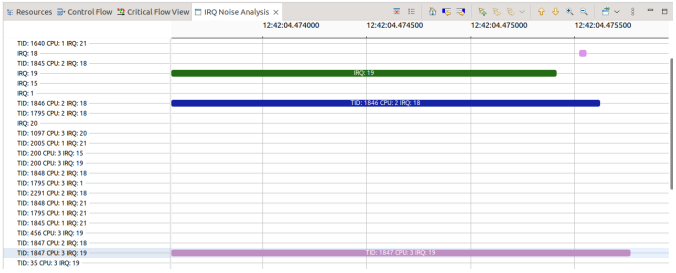


Fig. 6: *Visualizer* module - *Disk Noise* view



(a) The IRQ noise behavior overview.



(b) Closeup to the IRQ behavior.

Fig. 7: *Visualizer* module - *IRQ Noise* view

administrators. Trace Compass is a powerful performance analysis and visualization tool designed for importing and analyzing trace data, including system, kernel, and application traces. It offers a user-friendly graphical interface with a range of analysis capabilities, including event correlation, latency analysis, and resource utilization analysis, enabling users to identify performance bottlenecks and diagnose issues.

In the context of the *Visualizer module*, views in Trace Compass represent graphical representations of isolated events extracted from the state system, which is a core component of the Trace Compass framework. Leveraging the TMF (Trace Compass Trace and Log Analysis Framework), the *Visualizer module* introduces three specific noise views: *Network Noise*, *Disk Noise*, and *IRQ Noise*. These views enable system administrators to detect and analyze different types of noise, such as interrupt-related noise, disk I/O noise, and network noise. By examining these views and analyzing the corresponding events, administrators can gain deeper insights into the noise sources, identify root causes, and make informed decisions to optimize system performance.

1) Network Noise view: The *Network Noise* view provides a graphical representation of network activity and the associated thread IDs. In the view, different colors are used to differentiate between threads, and each color is accompanied by a thread ID number. The horizontal bars on the graph represent specific events related to network communication. The first horizontal bar indicates when the network wants to send a message. This bar shows the *Trs_Wait_Time* metric and starts at the beginning of the `net_dev_queue` event and ends at the occurrence of the `soft_irq_entry` event. The second horizontal bar, monitors the *Rcv_Wait_Time* metric and represent when the network device receives a packet and the corresponding thread gets woken up. The `net_if_receive_skb` event marks the start of this bar, and it continues until the `sched_waking` event occurs. By analyzing the Network Noise view, system administrators can identify patterns and anomalies in network activity. For example, in Fig. 5, thread ID 24171 is shown to be waiting for a long time to send a packet, indicating potential network noise or performance issues.

2) Disk Noise view: The *Disk Noise* view in the *Visualizer* module visualizes the noise detection metrics for disk I/O operations, focusing on the *Block_Wait_Time* and *Disk_Wait_Time*. These metrics monitor the time intervals between the `block_rq_insert`, `block_rq_issue`, and `block_rq_complete` events in the disk operation lifecycle. Fig. 6 illustrates the Disk Noise view, utilizing different colors to represent these metrics for different thread IDs and providing a visual representation of the noise in disk activity. The view displays the monitored wait times for each disk in the system, with disk number five being the primary focus in the given example. The figure highlights an increase in wait times for thread ID 228047368, indicating a disk-related performance noise.

3) IRQ Noise view: The IRQ Noise view illustrates the IRQ_Wait_Time metric, which measures the duration between `irq_handler_entry` and `irq_handler_exit`

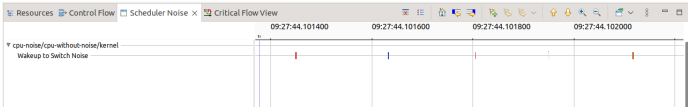
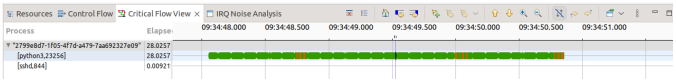
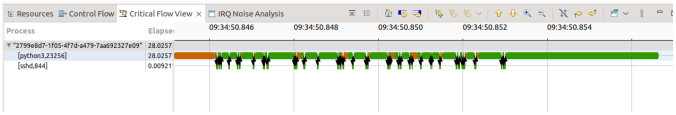


Fig. 8: *Visualizer* module - *CPU Noise* view



(a) The noise-free environment.



(b) The noisy system.

Fig. 9: The *Visualizer* module utilizes the Critical Path view to showcase CPU noise.

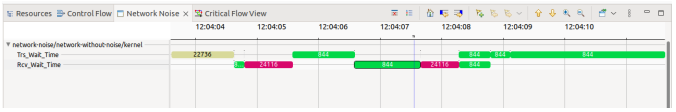
events. Fig. 7a provides an overview of the view, where horizontal bars represent different IRQs, and the colored sections within the bars depict the waiting time of thread IDs for the corresponding IRQs. In the overall view, the IRQ noise may not be immediately apparent, but system administrators can zoom in and explore individual lines to identify interesting IRQ wait times for different IRQs. For example, Fig. 7b highlights that TID 1846 and TID 1847 experienced long wait times for IRQ 18 and IRQ 19, respectively.

4) CPU Noise view: The CPU noise view, as proposed, is designed to monitor the duration between *sched_switch* and *sched_wakeup* events. This metric is depicted in Fig. 8. In this view, each vertical bar represents a thread, while the horizontal segments within the bar represent the duration between these events. Another valuable view in Trace Compass is the *Critical Path* view, which visualizes the occurrences of delays or blocks during execution. Fig. 9 demonstrates the difference between a process in a noise-free and a noisy environment. When CPU noise occurs, the Critical Path view reveals that numerous CPU switches take place within the critical path of the process. This suggests that CPU noise significantly impacts the performance of the system. These two views, the *Control Flow* view and the *Critical Path* view, enable users to analyze CPU performance noise and gain a deeper understanding of the effects on system execution.

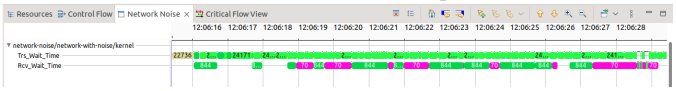
IV. Evaluation

In order to validate the proposed approach, we conducted several test cases to examine how it detects different types of performance noises. These test cases include:

- The *Network Noise Test-case*: This test case simulates a network noise environment where various network activities generate excessive traffic or delays. The goal is to assess how well the proposed approach can detect and analyze network-related performance noises.



(a) The noise-free setup.



(b) The noisy setup.

Fig. 10: The *Network Noise* view shows the noise in the *Network Noise* test case.

- The *Disk Noise Test-case*: This test case focuses on noise in disk operations, such as high disk queue wait times or prolonged block queue wait times.
- The *CPU Noise Test-case*: This test case aims to evaluate the approach's ability to detect and analyze CPU noise. It involves scenarios where the CPU is heavily loaded, or experiences frequent context switches.

Through these test cases, we evaluate the proposed approach's usefulness and efficiency in detecting and diagnosing different types of performance noises.

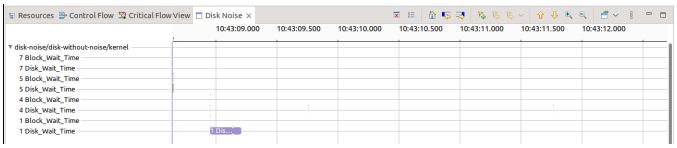
A. Experimental Setup

All the experiments were conducted on a host operating system, specifically *Linux Ubuntu 22.04*, running on a machine equipped with an *Intel® Core™ i7* CPU operating at a clock speed of 3.60GHz and 32.00 GB of RAM. The traces were recorded on a guest operating system, also *Ubuntu 22.04*, which was configured with 16 GB of RAM and 10 CPUs. The guest operating system was deployed as a virtual machine using *Oracle VM VirtualBox 6.1*.

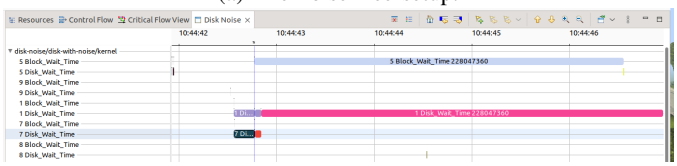
To capture and analyze the traces, we utilized *LTTng 2.13.9* as the tracing tool and *Trace Compass 8.3* as the analysis tool. These tools provided the necessary functionality to record and examine the performance-related events and metrics during the experiments.

B. Network Noise Test-case

To generate network noise, we implemented a test scenario using the *wget* command to retrieve data from the website "*www.google.com*", with a limited download rate of 4KB/s. Traces were collected during the execution of this application, capturing various network-related events and metrics. Subsequently, we introduced network noise deliberately by running another application, curl, which attempted to download a large file from the address "*https://speed.hetzner.de/100MB.bin*." This intentional overload on the network caused disturbances and introduced noise in the network trace. By comparing the timings between this noisy trace and a separate network trace, the proposed approach was able to identify the additional time incurred due to the presence of network noise. This analysis provided valuable insights into the impact of network noise on network performance and latency.

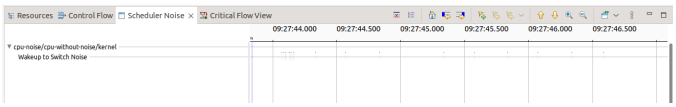


(a) The noise-free setup.

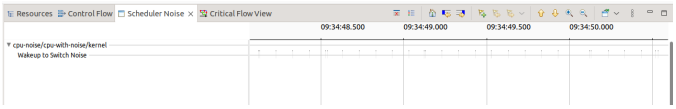


(b) The noisy setup.

Fig. 11: The *Disk Noise* view shows the noise in the *Disk Noise* test case.



(a) The noise-free environment.



(b) The noisy system.

Fig. 12: The CPU Noise view monitors the duration between `sched_switch` and `sched_wakeup` events.

Fig.11 represents the proposed *Network Noise* view, showcasing the effects of network noise on two monitored metrics in the *Network Noise* test case: *Trs_Wait_Time* and *Rcv_Wait_Time*. In the presence of network noise, as depicted in Fig.11b, both metrics exhibit high fluctuations. The *Trs_Wait_Time*, which represents the time between when the CPU intends to send a message over the network and when the message is actually sent, shows significant variations. Similarly, the *Rcv_Wait_Time*, which represents the delay between the network card receiving a packet and the associated thread receiving the waking event, also experiences fluctuations. Comparing Fig.11b with Fig.11a, which represents the metrics in a noise-free environment, clearly demonstrates the impact of network noise. In the noisy network scenario, where a file is being downloaded, the fluctuations in *Trs_Wait_Time* and *Rcv_Wait_Time* are more pronounced, indicating the disruptions caused by the network noise.

C. Disk Noise Test-case

To design the disk I/O noise test case, a Python script was developed to concurrently read 10 distinct files using 10 different threads. Each file contains random characters, resulting in an approximate size of 100 MB per file. Before tracing the behavior of this application, we took the precaution of clearing the cache by executing the command "sync; echo 3 /proc/sys/vm/drop_caches". This step was taken to eliminate any caching effects and ensure a more accurate representation of the impact of disk I/O noise on system performance. To create disk I/O noise, three recursive *grep*s were run concurrently in the background. The *grep* tool [17] is a command-line utility commonly used in Unix-based systems to search for specific patterns within text files. These three *grep*s running in the background generated a noisy environment in the test case, while the proposed approach aims to detect and visualize the root cause of the noise.

To do so, we timed how long it took each process in the waiting queue and elevator queue to see if the service time was delayed. We did this by iterating through events in TraceCompass and upon important block_rq events we took action. The events we were the most interested in were block_rq_insert and block_rq_issue events as these indicated when the CPU wants to insert a block operation request into the queue and when issuing a pending block IO request operation to the device driver respectively. These events can also be viewed as the process that wants to access the disk getting put into a queue and when it actually gets access and exits. in TraceCompass. upon encountering a block_rq_insert we pushed the event into a 2d array in order to match it up with its associated block_rq_issue event matching them up by the sector they're writing to and which CPU handled it. We can then determine how much longer the noisy trace takes based on additional time spent in the elevator or waiting queue.

D. CPU Noise Test-case

To create the *CPU Noise* test case, we developed a simple Python code that generates the first 100,000 prime numbers. The code was executed, and the corresponding traces were collected for noise free environment. To generate additional workload and introduce noise into the system, we employed Sysbench [18], a benchmarking tool widely used for measuring system performance under different workloads and stress testing scenarios. Sysbench was configured to utilize maximum CPU resources for calculating the first 10,000 prime numbers. By running nine instances of Sysbench concurrently, we intentionally challenged the CPU scheduler, resulting in a noisy trace that accurately represented the impact of CPU noise on system behavior.

The duration between `sched_switch` and `sched_wakeup` events in *CPU Noise* test case is depicted in Fig. 12. Due to the short duration of the entire test case (approximately 2.5 seconds), each bar representing the duration appears very small, almost like a dot. To provide a clearer visualization of the events, we created a diagram in Fig. 13 that illustrates the number of `sched_switch` events associated with the main Python code in both the noisy and noise-free environments. As depicted, the number of events in the noisy environment is noticeably higher compared to the noise-free environment. By monitoring this metric, the proposed approach effectively detects the presence of noise in the CPU Noise test case scenario. The increased number of `sched_switch` events indicates a higher frequency of CPU context switches, which can be attributed to the additional workload generated by the noisy environment. This analysis

TABLE II: Noisy Trace

block_rq_insert count	block_rq_issue	block_rq_complete	Total Disk I/O Noise	Average Noise
253	253	253	2609.915	10.3159
137	137	137	1644.336	12.0025
128	128	128	634.261	4.9552
139	139	139	3479.797	25.0345

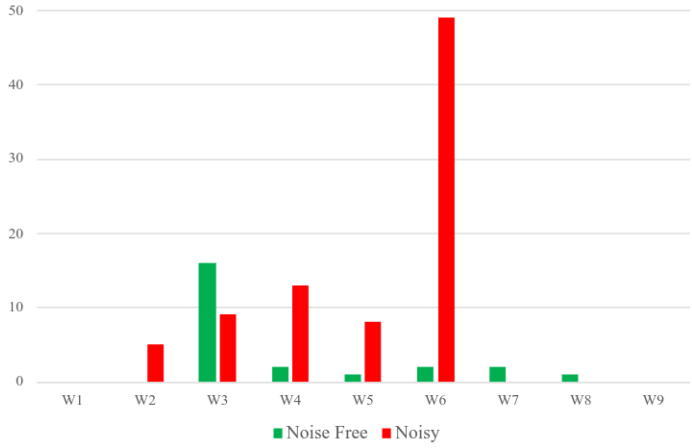


Fig. 13: The number of `sched_switch` events in each 500ms window (W) of the running process in the *CPU Noise* test case, with and without the presence of noise.

enables the proposed approach to identify and quantify the impact of CPU noise on the system's performance.

E. IRQ Noise Test-case

The evaluation of the system's response to various types of noise serves as a crucial step in understanding modern operating systems' potential challenges and reliability. The evaluation section aims to assess the impact of various types of noise on system behavior and performance, based on the methodology outlined in the previous section. We conducted a comprehensive analysis of the system by tracing it under both noise-free and noisy conditions, allowing us to compare and evaluate the effects of different noise sources on key system components. To evaluate the impact of IRQ noise, we followed the methodology outlined in the previous section. Firstly, a noise-free trace was obtained by tracing a *wget* function call while the OSNOISE_WORKLOAD feature was disabled. This ensured that no additional IRQs were introduced into the system. Then, to generate the noisy trace, we turned the OSNOISE_WORKLOAD feature to introduce additional IRQs into the system. We then analyzed and compared these traces in TraceCompass. We wrote a Javascript program to determine the time between our main focuses of how the noise impacts our program. We focus on three main metrics: the total amount of time spent in an IRQ, the total amount of events, and the average time each event was. We did this by iterating through the events in TraceCompass and when we come upon an IRQ event, either irq_handler_entry or irq_handler_exit pushed them into a 2d array in pairs. The following results in

TABLE III: Noisy Trace

Total	Number of Events	Average
17771	36	483.638
21192	237	89.418
7755	67	115.746
11029	81	136.16
13020	208	62.596

Table III show the times taken in IRQs during this execution. The time is recorded in microseconds.

V. Conclusion and Future Works

Noise detection and root cause analysis are critical for configuring systems and ensuring reliable application performance. This study presents an approach for noise detection and root cause analysis using system-level execution tracing. By monitoring system-level events, our proposed approach detects a wide range of performance noises. It gathers system-level events, maintains traces, analyzes the root causes of different noises, and visualizes the metrics through a four-module architecture. Experimental results confirm the high effectiveness, accuracy, and efficiency of our proposed approach in detecting various noises with different root causes by monitoring kernel-level traces. This approach provides valuable insights into system performance by identifying the sources of noise and their impact on the reliability of the application.

For future work, we have identified three main directions to further enhance our approach. Firstly, we aim to extend the noise metrics by analyzing additional root causes. This involves monitoring events related to CPU management, memory management, and external device management. Secondly, we plan to explore noise detection in virtualized environments. Monitoring the impact of noise on the host system and co-existing VMs solely through kernel event processing is of great interest due to the widespread use of containers and virtual machines in data centers. Lastly, we intend to develop a plugin for TraceCompass specifically focused on noise detection. TraceCompass is a widely-used tool for performance monitoring and analysis. By integrating our noise detection capabilities into TraceCompass, we can empower system administrators with enhanced noise monitoring and analysis features. This plugin will enable administrators to effectively monitor and analyze noise root causes, providing valuable insights for system optimization.

References

[1] D. B. de Oliveira, D. Casini, and T. Cucinotta, "Operating system noise in the linux kernel," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 196–207, 2022.

[2] N. Rameshan, L. Navarro, E. Monte, and V. Vlassov, "Stay-away, protecting sensitive applications from performance interference," in *Proceedings of the 15th International Middleware Conference*, 2014, pp. 301–312.

[3] H. Akkan, M. Lang, and L. M. Liebrock, "Stepping towards noiseless linux environment," in *Proceedings of the 2nd international workshop on runtime and operating systems for supercomputers*, 2012, pp. 1–7.

[4] G. Casale, S. Kraft, and D. Krishnamurthy, "A model of storage i/o performance interference in virtualized systems," in *2011 31st International Conference on Distributed Computing Systems Workshops*. IEEE, 2011, pp. 34–39.

[5] C. Lameter, "Shoot first and stop the os noise," in *Linux Symposium*. Citeseer, 2009, p. 159.

[6] H. Akkan, M. Lang, and L. Liebrock, "Understanding and isolating the noise in the linux kernel," *The International journal of high performance computing applications*, vol. 27, no. 2, pp. 136–146, 2013.

[7] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of os noise," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 852–863.

[8] M. Copik, A. Calotoiu, T. Grosser, N. Wicki, F. Wolf, and T. Hoefler, "Extracting clean performance models from tainted programs," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 403–417.

[9] E. A. León, I. Karlin, and A. T. Moody, "System noise revisited: Enabling application scalability and reproducibility with smt," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 596–607.

[10] H. Malallah, S. R. Zeebaree, R. R. Zebari, M. A. Sadeeq, Z. S. Ageed, I. M. Ibrahim, H. M. Yasin, and K. J. Merceedi, "A comprehensive study of kernel (issues and concepts) in different operating systems," *Asian Journal of Research in Computer Science*, vol. 8, no. 3, pp. 16–31, 2021.

[11] M. Woodside, S. Tjandra, and G. Seyoum, "Issues arising in using kernel traces to make a performance model," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 11–15.

[12] M. Janecek, N. Ezzati-Jivan, and A. Hamou-Lhadj, "Performance anomaly detection through sequence alignment of system-level traces," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 264–274. [Online]. Available: https://doi.org/10.1145/3524610.3527898

[13] N. M. Gonzalez, A. Morari, and F. Checconi, "Jitter-trace: a low-overhead os noise tracing tool based on linux perf," in *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, 2017, pp. 1–8.

[14] "Lttng," *https://lttng.org/*, 2018.

[15] H. Daoud and M. R. Dagenais, "Recovering disk storage metrics from low-level trace events," *Software: Practice and experience*, vol. 48, no. 5, pp. 1019–1041, 2018.

[16] "Trace compass," *https://projects.eclipse.org*, 2015.

[17] Free Software Foundation, "Grep," Software, 2009, version 2.5.4. [Online]. Available: https://www.gnu.org/software/grep/

[18] "Sysbench," https://github.com/akopytov/sysbench, accessed: May 26, 2023.