The Journal of Systems and Software 216 (2024) 112117

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Enhancing empirical software performance engineering research with kernel-level events: A comprehensive system tracing approach ☆

Morteza Noferesti *, Naser Ezzati-Jivan

Department of Computer Science, Brock University, St. Catharines, L2S 3A1, ON, Canada

ARTICLE INFO

Dataset link: https://github.com/mnoferestibrocku/dataset-repo

Keywords:
Empirical software engineering
Software performance engineering
Kernel-level tracing
Software reliability
Performance monitoring

ABSTRACT

Performance engineering is a proactive and systematic approach aimed at designing, building, and enhancing software systems to ensure their efficient and reliable operation. It involves observing and measuring the operational behavior of a software system without interference, assessing performance metrics like response times, throughput, and resource utilization. This entails delving into kernel-level events related to performance monitoring, which play a significant role in understanding system behavior and diagnosing performance-related issues. Kernel-level events offer insights into how both the operating system and hardware resources are utilized. This information empowers system administrators, developers, and performance analysts to optimize and troubleshoot the system effectively.

A critical aspect of performance analysis is root cause analysis, which involves delving deep into kernel-level events connected to performance monitoring. These events provide valuable insights into the utilization of operating system and hardware resources, equipping system administrators, developers, and performance analysts with tools to effectively troubleshoot and optimize the system. Our study introduces an innovative artifact that captures kernel-level events using Elasticsearch and Kibana, facilitating comprehensive performance analysis under diverse scenarios. By defining both Light-load and Heavy-load scenarios and simulating CPU, I/O, Network, and Memory noise, we offer researchers a realistic environment to explore innovative approaches to system performance enhancement.

The artifact comprises both kernel events and system calls, resulting in a cumulative count of 24,263,691 events. The proposed artifact can serve three distinct applications. The first application emphasizes performance analysis by utilizing kernel events for monitoring. The second application targets noise detection and root cause analysis, again using kernel events. Finally, the third application investigates software phase detection through monitoring at the kernel level. These applications demonstrate that through our artifact, researchers can effectively analyze performance, detect and address performance noise, and identify software phases, contributing to the advancement of performance engineering methodologies.

All the system configurations, scripts, and traces can be found in the artifact GitHub repository.[1]

1. Introduction and background

At the cutting edge of modern software development practices, performance engineering stands as a systematically proactive approach to designing, building, and optimizing software systems for efficient and reliable performance. According to a survey conducted by Dynatrace, a software intelligence company, 75% of respondents reported that performance issues had impacted their business in a year (Wert, 2018). This highlights the widespread impact of performance-related issues on organizations. It encompasses a range of activities such as analyzing, measuring, modeling, and enhancing performance attributes like response time, throughput, scalability, and reliability (Wert, 2018). These efforts are threaded throughout the software development lifecycle stages such as requirements gathering, design, implementation, testing, deployment, and monitoring (Woodside et al., 2007). By continuously improving system components and configurations, performance engineering ensures that potential performance bottlenecks are identified and addressed preemptively.

There are two primary approaches to analyzing software performance: model-based and measurement-based. Model-based techniques

☆ Editor: Professor Laurence Duchien.

* Corresponding author.
E-mail addresses: mnoferesti@brocku.ca (M. Noferesti), nezzati@brocku.ca (N. Ezzati-Jivan).

[1] URL: https://github.com/mnoferestibrocku/dataset-repo.

https://doi.org/10.1016/j.jss.2024.112117
Received 16 August 2023; Received in revised form 29 May 2024; Accepted 31 May 2024
Available online 3 June 2024

allow for early performance evaluation without fully implementing the entire system (Woodside et al., 2020). These techniques use models like Queueing Networks, Stochastic Petri Nets, or Queueing Petri Nets to directly represent performance aspects (Wert, 2018). However, there is a challenge in applying pure performance models to real-world software development projects due to the gap between theoretical models and practical implementation. Efforts are needed to connect these two aspects and effectively integrate performance models into actual software development processes (Woodside et al., 2020). Measurement-based software performance monitoring approaches refer to techniques that rely on observing and measuring the runtime behavior of a software system to assess its performance characteristics (Velez et al., 2020). These approaches collect performance-related data during the execution of software and analyze it to gain insights into its performance aspects. Unlike model-based approaches that rely on performance models, measurement-based approaches directly observe the system's behavior in real-world conditions (Loyola-González, 2019).

Measurement-based software performance monitoring can be further classified into white-box and black-box approaches. White-box monitoring involves embedding monitoring agents within the software system or instrumenting the source code to collect fine-grained performance data, such as function execution times, memory usage, network traffic, and other relevant metrics (Fan et al., 2021; Velez et al., 2020). White-box monitoring provides detailed visibility into the internal workings of the software and is effective in understanding the behavior of the application, particularly when the source code is accessible, small, or when known libraries are used (Velez et al., 2020).

Black-box monitoring, in contrast, adopts an external perspective and does not interfere with the execution of the software (Loyola-González, 2019; Ezzati-Jivan et al., 2020). It captures performance metrics at the system level, including response times, throughput, resource utilization, and other relevant indicators, without requiring any modifications to the source code (Ezzati-Jivan et al., 2020). This approach enables the identification of performance issues without disrupting the normal operation of the software. It is particularly valuable when dealing with proprietary software or third-party components where access to the source code is restricted or unavailable. By providing a non-intrusive means of performance analysis, black-box monitoring allows for comprehensive performance evaluation while ensuring minimal impact on the software's execution .

Kernel event tracing is a powerful technique used by black-box performance analysis approaches, to monitor and capture low-level events that occur within the kernel of an operating system (Woodside et al., 2020). This technique involves tracing and recording a wide range of events, such as interrupts, system calls, scheduler events, disk I/O operations, and network activities (Janecek et al., 2022; Chen et al., 2019). By collecting and analyzing these events, kernel event tracing enables the detection of performance problems and facilitates root cause analysis (Huang et al., 2022). It provides valuable insights into the inner workings of the kernel and helps identify the specific areas where performance issues may be originating from. While specific statistics on kernel event tracing adoption may be less readily available, kernel-level tracing tools like perf (Molnar, 2010) and SystemTap (Glatzmaier et al., 2005) are widely used in the Linux community. For example, SystemTap has been included in the Linux kernel since version 2.6.15, indicating its widespread adoption among Linux developers and administrators (Glatzmaier et al., 2005). By leveraging kernel event tracing, developers and administrators can gain a deeper understanding of the system's behavior, diagnose performance bottlenecks, and take targeted actions to optimize system performance.

Real-time applications present unique challenges in performance engineering and kernel-level event tracing due to their strict requirements for responsiveness and predictability. These applications operate under tight timing constraints, where any delays or inconsistencies in system performance can lead to failures. Studies have shown that software failures can cost businesses billions of dollars annually. For example, a report by the Consortium for IT Software Quality (CISQ) estimated that software failures cost the U.S. economy $2.8 trillion annually (Curtis et al., 2022). Efficient kernel-level event tracing is essential to avoid introducing overhead that could jeopardize meeting these timing requirements. Additionally, real-time applications often operate in resource-constrained environments, requiring careful management of CPU, memory, and I/O resources to prevent tracing activities from impacting performance.

Moreover, real-time applications demand deterministic behavior, meaning their execution must be predictable and consistent. Kernel-level event tracing must not introduce non-deterministic behavior, which could lead to unpredictable outcomes. Furthermore, tracing activities introduce latency to the system, which must be minimized to ensure timely responses in real-time applications. Effective synchronization between tracing activities and concurrent tasks is crucial to capture events accurately without causing interference. Balancing these challenges is essential for integrating kernel-level event tracing effectively into real-time systems, maintaining their responsiveness and reliability.

In the domain of black-box performance analysis, several methods aim to gather kernel-level events generated by software actions (Loyola-González, 2019). However, the absence of such a comprehensive artifact hinder the ability to apply diverse performance analysis approaches and compare their effectiveness (de Oliveira et al., 2022). The creation of such an artifact would be of immense value. It would enable the evaluation and benchmarking of different techniques, providing researchers and practitioners with insights into the strengths and limitations of various performance engineering approaches (Malallah et al., 2021). This, in turn, would facilitate well-informed decisions when choosing performance analysis methodologies. The development of such a comprehensive artifact holds significant potential for advancing performance analysis techniques, encouraging more robust and dependable performance optimizations in software systems.

In this paper, building upon our previous work (VanDonge and Ezzati-Jivan, 2022), we present a comprehensive dataset for software performance engineering that includes kernel-level events from a system running under different workloads and noise. We selected ELK[2] (Elasticsearch, Logstash, Kibana) " installed on a *Linux Ubuntu 22.04.2 LTS* operating system, which is widely used in practical scenarios. The ELK stack, which consists of Elasticsearch for indexing and searching data, Logstash for collecting and processing data, and Kibana for visualizing data, has gained popularity in academia for a variety of research purposes (Kathare et al., 2020). Further installation and system details can be found in Section 2.1.

The ELK stack was chosen as a use-case due to its unique combination of characteristics:

1. The distributed nature and intricate interactions between Elasticsearch, Logstash, and Kibana create a challenging environment for performance analysis (Bendechache et al., 2021). Traditional debugging methods often fall short in such complex systems, making it an ideal testbed for our approach.
2. The ELK stack has a high popularity among developers, system administrators, and data analysts (Dhulavvagol et al., 2020). By showing the effectiveness of our approach on such a widely used platform, we would like to make a positive impact on a large and diverse community of users.
3. The ELK stack has been studied in several research domains (Zmaranda et al., 2021; Ni et al., 2024), showing its importance in the field. Our work builds upon this existing research, leveraging the ELK stack as a representative example of complex, real-world software systems. We have now cited three of those papers in our revised manuscript.

[2] URL: https://www.elastic.co/what-is/elk-stack

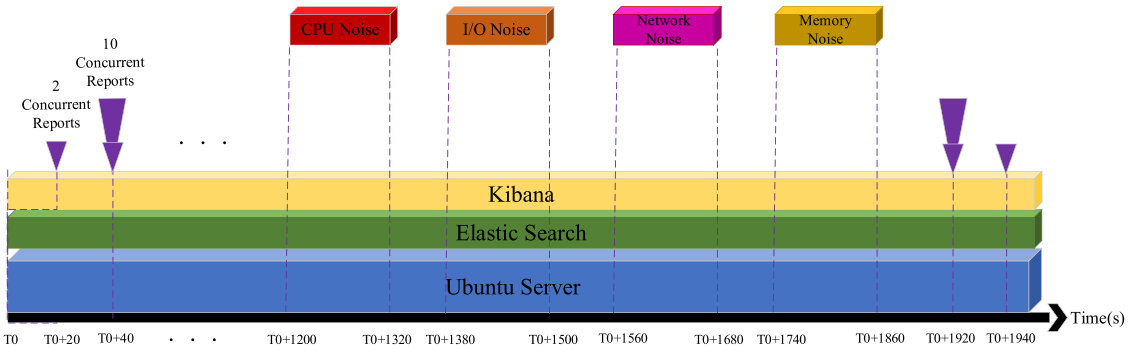


Fig. 1. The experiment scenario with Light-Load and Heavy-Load requests and multiple noises.

Elasticsearch provides powerful indexing and search capabilities, enabling efficient storage and retrieval of large volumes of log data by leveraging system resources such as hard drives for data storage (Wei et al., 2020) and CPUs for indexing and searching large datasets (Zmaranda et al., 2021). This resource utilization is essential for performance analysis research. The authors in Yang et al. (2022) highlight Elasticsearch as a highly scalable and robust search engine, specially designed for handling large datasets. In Ni et al. (2024), Elasticsearch is acknowledged as a powerful tool for semantic search, capable of conducting both text and vector searches simultaneously, catering to various application needs. Originally intended for searching structured and unstructured documents (Devins et al., 2022), Elasticsearch excels in managing large amounts of text and vector data, making it a valuable resource for Large Language Model Systems (LLMS) (Ni et al., 2024). Additionally, Elasticsearch queries are well-known for their speed and real-time retrieval capabilities, as supported by studies (Zehlike et al., 2020; Lee and Kwon, 2022).

The architecture of ELK utilizes various resources to effectively handle large volumes of data. Firstly, it leverages the I/O capabilities of hard drives to efficiently read and write data to secondary storage. Being a Java-based program, ELK also utilizes RAM to manage the Java Virtual Machine (JVM) environment, ensuring optimal performance and resource management. Additionally, ELK relies on network connectivity to transmit logs to the server side, facilitating real-time data ingestion and analysis (Lee and Kwon, 2022). Furthermore, the indexing process, which is integral to ELKś functionality, heavily utilizes the CPU to efficiently organize and search through the indexed data (Dhulavvagol et al., 2020). The diverse utilization of resources by ELK in high-usage scenarios makes it a suitable choice for our research in software performance engineering, enabling us to effectively analyze and optimize system performance under various conditions.

Fig. 1 gives an overview of the scenario used to create the system artifact. To assess the system's performance, we define two separate workloads:

1. Light-Load Scenario: This workload involves running two reports every 20 s. It is designed to simulate a situation with low processing demands, helping us evaluate the system's performance under minimal stress.
2. Heavy-Load Scenario: In this workload, we generate 10 reports every 40 s. This aims to simulate a scenario with more data and higher computational demands, putting increased processing pressure on the system.

We selected two workload scenarios to capture a broad spectrum of system behaviors while maintaining a manageable scope for detailed analysis. Scenario A represents high I/O operations, typical of database applications, whereas Scenario B emphasizes CPU-bound processes, common in computational tasks. This diversity aligns with common SE Perf practices, ensuring our results are relevant and diverse.

We chose 20 s and 40 s as reporting periods based on preliminary experiments that indicated these intervals effectively focus on transient noise events (and not the regular normal behavior which is not of interest in this experiment) without introducing server monitoring overhead. This balance is important for accurately diagnosing performance issues while maintaining system performance, aligning with best practices in the field.

In the workload scenarios, the shorter reporting period in the Light-Load scenario reflects a lower frequency of system interactions, representing scenarios with minimal processing requirements. Conversely, the longer reporting period in the Heavy-Load scenario signifies a higher frequency of system interactions, indicative of scenarios with more intensive computational demands. We also have overlapping periods while both scenarios are executed simultaneously. By varying the reporting periods, we create distinct workload scenarios that challenge the system in different ways, allowing us to evaluate its performance across a spectrum of processing loads.

The irregular and undesired fluctuations in system behavior, known as performance noise, can impact the expected dependability of a system. To make the artifact more realistic and replicate real-world situations, we introduce four types of noise into the system.

1. CPU noise: The CPU noise is initiated after 1200 s and lasts for 120 s (as shown in Fig. 1). This noise simulates conditions of high CPU utilization or intensive processing activities which can affect the system performance. It is used to evaluate the system's capability to manage resource-intensive tasks and sustain responsiveness under heavy CPU-intensive workloads.
2. I/O noise: The I/O noise is activated at 1380 s and continues for 120 s. It emulates increased input/output (I/O) operations or data transfer activities that can potentially impact disk or storage performance. This noise enables the assessment of the system's performance when dealing with high I/O loads and its ability to handle data-intensive operations.
3. Network noise: The network noise is introduced at 1560 s and persists for 120 s. It simulates network congestion, latency, or fluctuations in network connectivity, which can affect data transmission and communication between system components. By incorporating network noise, we can evaluate the system's resilience to network-related challenges and its ability to maintain effective data exchange under adverse network conditions.
4. Memory noise: The memory noise is activated at 1740 s and lasts for 120 s. It represents increased memory usage or memory-related issues that can impact system performance and stability. This noise helps assess the system's ability to handle memory-intensive tasks and its responsiveness in memory-constrained situations.

The specific durations and intensities for each type of noise are selected to emulate system disturbances and evaluate system resilience effectively. The choice of durations, such as 120 s for CPU, I/O,

Network, and Memory noise, is determined to provide ample time to induce noise and disrupt the system adequately. These durations are deemed sufficient to produce noticeable effects on system performance without excessively prolonging the evaluation process. Additionally, the intensities of the noise types are calibrated to represent varying levels of disruption, ranging from moderate to severe, thereby assessing the system's capability to maintain functionality under different stress levels.

The diversity of workload scenarios and types of noise selected for our research is underpinned by the need to comprehensively evaluate system performance under varying conditions. Firstly, the inclusion of two distinct workload scenarios, namely the Light-Load and Heavy-Load scenarios, enables us to assess system performance across a spectrum of processing demands. The Light-Load scenario simulates situations with minimal processing requirements, allowing us to evaluate baseline performance under low-stress conditions. Conversely, the Heavy-Load scenario introduces higher computational demands, providing insights into the system's response to increased processing pressures. By encompassing these contrasting scenarios, we ensure a thorough evaluation of system performance across different operational contexts.

Furthermore, the incorporation of four types of noise—CPU, I/O, Network, and Memory disruptions—adds another layer of complexity to our evaluation framework. Each type of noise represents a distinct system disturbance that can impact performance in real-world environments. CPU noise simulates high CPU utilization or intensive processing activities, I/O noise emulates increased input/output operations or data transfer activities, Network noise replicates network congestion or latency, and Memory noise represents heightened memory usage or memory-related issues. By introducing these diverse noise types, we aim to emulate realistic conditions and assess system resilience under various disruptive scenarios. This approach ensures that our evaluation captures a wide range of potential performance challenges, enabling us to comprehensively evaluate system behavior and responsiveness.

We present three separate applications that showcase the use-ability of our proposed artifact. Each application serves a distinct purpose in leveraging the proposed artifact for software performance engineering.

- Performance Analysis: This application focuses on utilizing kernel events to monitor and analyze system performance comprehensively. By leveraging kernel-level event tracing, researchers can gain insights into various performance metrics and identify potential bottlenecks within the system. Performance analysis is a fundamental aspect of software optimization, and this application provides researchers with the tools to assess system behavior under different workloads and noise conditions accurately.
- Noise Detection and Root Cause Analysis: In this application, the emphasis is on detecting and analyzing the impact of noise on system performance. By correlating kernel events with simulated noise patterns, researchers can pinpoint the root causes of performance degradation and identify areas for optimization. This application is crucial for understanding how different types of noise affect system behavior and for devising strategies to mitigate their impact effectively.
- Software Phase Detection: The third application explores the use of kernel-level event monitoring for detecting software phases. By analyzing patterns in kernel events, researchers can identify distinct phases in software execution, such as initialization, computation, and termination. Understanding software phases is essential for optimizing performance, as it allows researchers to focus optimization efforts on specific stages of program execution. Additionally, software phase detection can provide valuable insights into the underlying behavior of complex software systems, facilitating more informed decision-making in performance engineering tasks.

The selection of these three applications is driven by their significance in addressing key challenges in software performance engineering and their alignment with the capabilities of the proposed artifact. Each application offers unique insights into different aspects of system behavior and performance, contributing to a comprehensive understanding of software performance optimization strategies.

This enhanced artifact enables researchers to evaluate the system's performance, resilience, and effectiveness in handling disruptive conditions. By leveraging this artifact, they can gain valuable insights into optimizing system performance, identifying potential vulnerabilities, and developing robust strategies to mitigate the impact of disruptive conditions. All system configurations, scripts, and traces are available within the GitHub repository of the dataset.[3]

The paper makes the following contributions:

- Artifact Introduction: The paper presents a comprehensive artifact that includes kernel-level events collected from the ELK framework. Researchers can leverage these traces to validate their methods, identify strengths and weaknesses, and gain insights into the effectiveness of different techniques.
- Realistic Workloads and Noise: The artifact encompasses a range of complex, well-defined workloads imposed on the ELK framework, along with the introduction of various types of noise. This enhances the realism of the testing environment and enables the evaluation of performance engineering approaches under diverse conditions.
- Distinct Capabilities Demonstrated: The paper demonstrates the critical role of monitoring kernel-level events in significant performance engineering applications, namely performance analysis, noise detection and root cause analysis, and software phase detection. Each application exemplifies a distinct capability of the proposed artifact, offering valuable insights into various aspects of performance engineering research topics.

The rest of paper organized as follows: Section 2 covers the artifact details, including the system setup and specifics of kernel-level events. In Section 3, we explore three practical application of the artifact in performance engineering: performance monitoring, noise detection, and software phase detection. Section 4 review researches aiming to gather kernel-level events. Finally, Section 5 concludes the paper and discusses some future works.

2. Artifact details

This section describes an outline of the system setup, followed by a comprehensive breakdown of the ELK configuration. We will then explore the tracing technique used to capture kernel-level events and end by presenting statistical details related to the recorded artifact.

2.1. System setup

The installation script and all the steps are outlined in the GitHub repository of the artifact.[4] The experiments were carried out on a machine with specifications outlined in Table 1. The system featured an *Intel® Core™ i7* CPU, operating at a clock speed of 3.60 GHz, coupled with 2*16 GB of DDR5 RAM configured for data transfer at 4400 MHz. For storage, the system employed a DT01ACA100 Toshiba Desktop hard disk, boasting a 1TB capacity and spinning at 7200RPM. This hard disk connected via a SATA 6 Gbps interface and boasted a 32 MB cache. The system ran a fresh installation of *Linux Ubuntu 22.04.2 LTS*, containing default applications and devoid of any third-party software installations. For further detailed specifications of the

[3] URL: https://github.com/mnoferestibrocku/dataset-repo

[4] URL: https://github.com/mnoferestibrocku/dataset-repo/tree/main/Installation

Table 1
The specification of the system.

Name	*Specification*
CPU	*Intel® Core™ i7* 3.60 GHz
Memory	2*16 GB of DDR5 configured at 4400 MHz
Hard disk	1TB DT01ACA100 Toshiba Desktop
OS	*Linux Ubuntu 22.04.2 LTS*

system, as collected by the '*inxi -F*' command, you can refer to the dataset's GitHub repository.[5]

Subsequent to the OS installation, we proceeded to install OpenJDK version "18.0.2". The installations of Elasticsearch[6] and Kibana[7] were carried out following the provided guidelines. For data ingestion into Elasticsearch, we utilized the IoT and IIoT dataset offered by Ferrag et al. (2022). This dataset was generated through a dedicated IoT/IIoT testbed, encompassing an array of devices, sensors, protocols, and cloud/edge configurations. The dataset is composed of 2,219,202 events, with each event encompassing 71 features.

Within the IoT and IIoT dataset, we have identified and categorized 12 reports spread across two Kibana dashboards. The first dashboard, known as the heavy-load scenario, comprises ten reports, while the second one, named the light-load scenario, includes two reports. The light-load dashboard updates every 20 s, whereas the heavy-load dashboard refreshes every 40 s. These reports have been carefully designed to offer comprehensive visualizations and analyses of different aspects of the dataset. For more in-depth information and precise definitions of each report, please refer to our GitHub repository.[8]

To induce noise in our experiments, we employ *stress-ng*,[9] a utility designed to simulate various stressors on a system for testing and benchmarking purposes. The noise generation specifics are elaborated in Table 2. We initiate the stress-ng tool to create CPU noise by performing matrix multiplication with a matrix size of 256 × 256 with 6 workers for a duration of 120 s. For I/O noise, we trigger the stress-ng tool to generate I/O stress by employing a mix of different I/O operations with 6 concurrent workers for a duration of 120 s. Network noise is introduced using 6 workers that engage in various socket stress activities. This includes pairs of client/server processes executing rapid connect, send, and receive operations, as well as disconnects on the local host. The generation of memory noise entails allocating 4 GB per each set of 6 workers, who continually call mmap/munmap and write to the allocated memory.

2.2. Kernel-level event tracing

The interactions between the operating system and software generate various events and invoke system calls, shedding light on system performance. For instance, the `sched_switch` tracepoint shows when one thread is replaced by another thread on a CPU, offering insight into Linux's scheduling behavior. To capture these events, we employ LTTng (Linux Tracing Toolkit: next generation),[10] an open-source tracing tool. LTTng is known for its low overhead and scalability, suitable for tracing both the Linux kernel and user space applications. Each trace event is defined by attributes such as Timestamp, CPU, event type (e.g., system call, interrupt), event details (e.g., IP address), process ID (PID), and thread ID (TID). To collect the necessary attributes, LTTng utilizes tracepoints, which are hooks placed in the code that allows function probes to attach at runtime. When a tracepoint is encountered, the probe is executed within the caller's context, and control resumes to the caller after probe execution. By strategically placing tracepoints in Linux subsystems, essential runtime information can be extracted.

In the proposed artifact, all possible kernel tracepoints and their corresponding events are enabled, including the following:

- `sched_switch`: Records information when a thread is switched from running on one CPU to another.
- `irq_handler_entry` and `irq_handler_exit`: Record information when an interrupt handler starts and finishes execution.
- `block_rq_insert`: Records information when a request is inserted into the block I/O request queue.
- `block_rq_complete`: Records information when a block I/O request is completed.
- `netif_receive_skb`: Records information 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. The tracing configuration is implemented via script can be found at the dataset's GitHub repository.[11]

2.3. Event statistics

Throughout the experiment, we gathered both kernel events and system calls, resulting in a total of 24,263,691 events. This accumulation led to the creation of a trace data file of 924MB. The raw trace file is accessible in the dataset's GitHub repository.[12]

The distribution of these diverse events is visually depicted in Fig. 2. Notably, the `sched_switch` event emerged as the most prevalent, constituting 16.2% of the total occurrences. Additionally, distinct system calls such as `syscall_entry_futex` and `syscall_exit_futex` are also observed, providing valuable insights into the frequency and contexts in which the *futex* system calls are utilized. Analyzing patterns, high-frequency calls, and potential areas of significant time consumption can unveil performance bottlenecks and zones that might necessitate optimization.

The density of collected events throughout the experiment duration is depicted in Fig. 3. Over the course of the experiment, the count of kernel-level events fluctuated in response to the applied load or introduced noises. During the period of highest density, specifically in the midst of the *Memory Noise*, over 120,000 events were generated within a single second. Another notable duration observable in the figure pertains to the *Network Noise*, which led to the creation of more than 60,000 events per second during the noise's presence.

Beyond the type of kernel-level events, the count of events by itself offers insights into the system's behavior and can be analyzed to detect performance issues. For instance, a sudden surge in network connection events such as `netif_receive_skb`, even if it does not directly impact the overall execution time, could signify the existence of noise or even a potential denial-of-service attack (Ullah and Babar, 2019).

The sequence of events produced during the experiment possesses several attributes. Fig. 4 illustrates the events that materialized while the software executed. Each event is distinguished by its attributes, encompassing the Timestamp, CPU ID, event types (such as system calls and interrupts), Contents (like the IP address of a network connection),

[5] URL: https://github.com/mnoferestibrocku/dataset-repo/blob/main/system-info.txt
[6] URL: https://www.elastic.co/guide/en/elasticsearch/reference/8.8/install-elasticsearch.html
[7] URL: https://www.elastic.co/guide/en/kibana/8.8/install.html
[8] URL: https://github.com/mnoferestibrocku/dataset-repo/tree/main/Installation/Workloads
[9] URL: https://wiki.ubuntu.com/Kernel/Reference/stress-ng
[10] URL: https://lttng.org/
[11] URL: https://github.com/mnoferestibrocku/dataset-repo/tree/main/KernelTracing
[12] URL: https://github.com/mnoferestibrocku/dataset-repo/tree/main/Trace-RawData

Table 2
The noise commands.

Type	*Command*
CPU noise	stress-ng –matrix 6 –matrix-method prod –matrix-size 256 –timeout 120
I/O noise	stress-ng –iomix 6 –timeout 120
Network noise	stress-ng –sock 6 –timeout 120
Memory noise	stress-ng –vm 6 –vm-bytes 4G –timeout 120

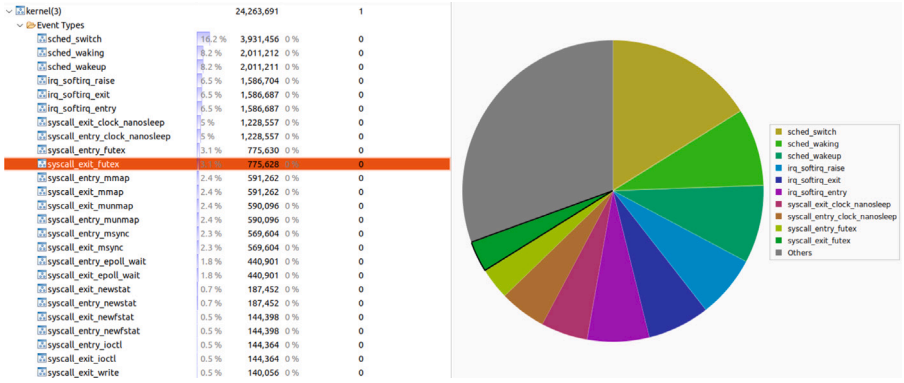


Fig. 2. Collected event statistic.

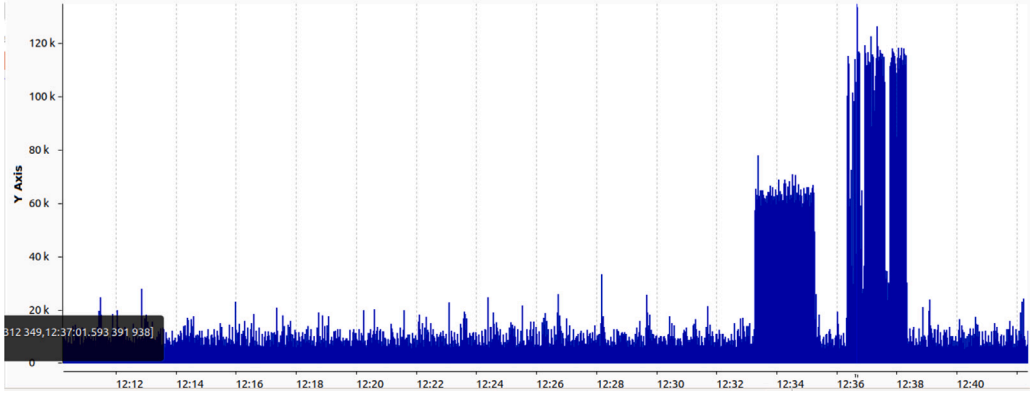


Fig. 3. Collected event density over the experiment.

Timestamp	CPU	Event type	Contents	TID	PID	Prio	Source
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>	<srch>	<srch>
12:17:04.839 615 711	11	sched_switch	prev_comm=swapper/11, prev_tid=0, prev_prio=	0		20	
12:17:04.839 631 812	11	syscall_exit_epoll_wait	ret=0, fds_length=0, overflow=0, fds=[], context.|	817	817	20	[fs/eventpoll.c:0]
12:17:04.839 701 048	11	syscall_entry_epoll_wait	epfd=13, maxevents=1024, timeout=0, context.p	817	817	20	[fs/eventpoll.c:0]
12:17:04.839 706 918	11	syscall_exit_epoll_wait	ret=0, fds_length=0, overflow=0, fds=[], context.|	817	817	20	[fs/eventpoll.c:0]
12:17:04.839 710 461	11	syscall_entry_epoll_wait	epfd=13, maxevents=1024, timeout=9, context.p	817	817	20	[fs/eventpoll.c:0]
12:17:04.839 726 750	11	sched_switch	prev_comm=node, prev_tid=817, prev_prio=20, p	817	817	20	
12:17:04.840 193 907	3	sched_waking	comm=VM Periodic Tas, tid=971, prio=20, target	0		20	
12:17:04.840 213 713	3	sched_wakeup	comm=VM Periodic Tas, tid=971, prio=20, target	0		20	
12:17:04.840 241 708	7	irq_handler_entry	irq=21, name=ahci[0000:00:0d.0], context.packet	0		20	

Fig. 4. Collected event attributes.

TID (Thread ID), PID (Process ID), Prio (denoting the thread's priority to which the event pertains), and Source (the source code of the process, if accessible).

This dataset allows for the examination of various performance concerns at the levels of the hardware, system or process. For instance, consider a situation where performance issues arise in *Hard Disk Drive (HDD)* I/O operations associated with a particular *process*. During operation, the software generates a substantial volume of events that include system calls and interrupts (Daoud and Dagenais, 2018). Each disk request is associated with multiple kernel events, representing different stages of its life cycle as illustrated in Fig. 5. Specifically, an *HDD* read/write operation initiates with the *block_getrq* event, which is succeeded by inserting the request into the waiting queue via the *block_rq_insert* event. Following this, the request is issued (*block_rq_issue*) to the driver for processing. If the disk successfully manages the request, the *block_rq_complete* event triggers, releasing the request data structure and awakening any blocked processes. It is evident that a single *HDD* operation triggers a range of events, underscoring the need for effective analysis of these events.

The events generated, along with their attributes and the time intervals between distinct events during these operations, offer crucial insights into the system's performance. For instance, in situations involving a slow Hard Disk Drive (HDD), a noteworthy time gap would manifest between the *block_rq_issue* and *block_rq_complete* events.

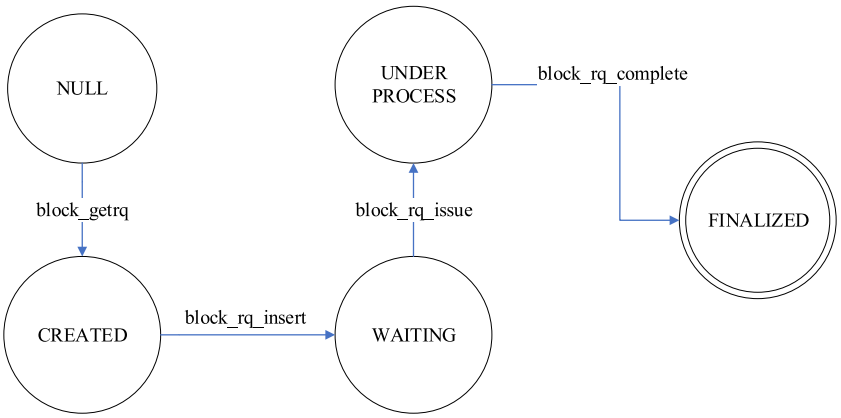


Fig. 5. Kernel events associated with a disk request's life cycle.

Conversely, a substantial time interval between the *block_rq_insert* and *block_rq_issue* events might point to a consistently busy "WAITING" queue within the driver, signaling a potential performance concern. Such an issue might occur if the software generates an overwhelming number of disk or file-system requests within a short time frame.

For a comprehensive performance analysis, it is crucial to deeply investigate diverse kernel events and their attributes, including timestamps and durations. This approach offers a holistic understanding of system operations and potential areas of slowdown. The artifact we introduce here plays a pivotal role in this endeavor. Packed with a multitude of kernel-level events and their corresponding data, it becomes an invaluable asset for identifying and addressing precise performance concerns. This tool empowers system administrators, developers, and performance analysts by unraveling the underlying causes of performance problems, enabling the development of efficient and reliable strategies to enhance system functionality.

3. Artifact applications

Kernel events linked to performance monitoring are essential for understanding system behavior and diagnosing performance-related issues. These events provide valuable insights into how both the operating system and hardware resources are used, enabling system administrators, developers, and performance analysts to optimize and troubleshoot the system.

In the upcoming section, we present three distinct use cases for the proposed artifact. Initially, in the subsection on performance monitoring, we delve into how kernel events contribute to performance analysis. Subsequently, we explore the utilization of kernel events for noise detection and root cause analysis. Lastly, we delve into the application of kernel-level event monitoring for detecting software phases. All application implementation codes are accessible through the GitHub repository.[13]

3.1. Performance monitoring

Performance engineering encapsulates a systematic approach directed at the design, development, testing, and enhancement of software, systems, and applications, all with the explicit aim of meeting specific performance objectives as cited by Wert (2018). This comprehensive framework encompasses a diverse set of techniques, methodologies, and tools, all orchestrated towards guaranteeing the seamless, dependable, and optimally responsive functioning of a product or system. This is achieved under a spectrum of conditions and workloads, ensuring that performance remains robust and reliable across varying scenarios.

The task of identifying and addressing performance bottlenecks while uncovering their underlying causes in multi-core systems is notably challenging. A spectrum of factors, ranging from coding errors and suboptimal database design to misconfigurations and system load, can contribute to performance irregularities. However, within multi-core systems, the core of performance issues often revolves around the complex interplay of multiple concurrent threads and their interactions. This complexity is further compounded by the need for these threads to share common resources, frequently leading to resource contention problems. In such scenarios, a significant portion of runtime is commonly spent not within the CPU itself, but rather in waiting for hardware resources and external elements like disk I/O, network operations, timers, or the completion of tasks by other threads, often linked to lock releases. The intricacies of synchronization and resource management inherent to multi-core systems amplify the complexity of identifying and effectively resolving performance bottlenecks.

A method commonly employed to assess the performance of multi-threaded applications revolves around the examination of the critical path, an approach extensively explored in various research endeavors (Janecek et al., 2022; Ezzati-Jivan et al., 2020). This application utilizes the algorithm provided by the open-source trace analysis software *Trace Compass 8.2.0*[14] to compute the critical paths of the relevant threads. The details of the implementation are outlined in the GitHub repository.[15] The critical path provides a visual representation of instances when a request encountered delays or was obstructed. Such paths prove invaluable for uncovering the origins and reasons behind latency occurrences. Additionally, the critical path serves as an indicator of the state of the subsequent thread's progress. For instance, a thread might be marked as in a `RUNNING` state when it is active on a CPU, or in a `TIMER` state if it is temporarily halted due to a timer interrupt. In scenarios where one thread hampers the progress of another, the actions of the new thread are also showcased within the critical path depiction.

The critical path of the Elasticsearch process is represented in Fig. 6. Within this illustration, the color green denotes the `RUNNING` state, while orange signifies the `PREEMPTED` state. Each arrow in the figure signifies a thread that has been obstructed by another thread. This critical path analysis leverages events such as `sched_switch`, `sched_wakeup`, `irq_handler`, `softirq`, and `hrtimer_expire`. Specifically, the `sched_switch` event unveils

[13] URL: https://github.com/mnoferestibrocku/dataset-repo/tree/main/Applications

[14] URL: https://wiki.eclipse.org/Trace_Compass/Development_Environment_Setup

[15] URL: https://github.com/mnoferestibrocku/dataset-repo/tree/main/Applications/Performance-Monitoring

Fig. 6. The critical path of Elasticsearch process where each arrow indicates a thread being blocked by another thread.

which threads are active on a CPU and their state upon being removed from it. The `sched_wakeup` event provides insight into why a thread is blocked. Meanwhile, the `irq_handler`, `softirq`, and `hrtimer_expire` tracepoints serve to monitor distinct interrupts transpiring across various CPUs.

Our dataset included detailed logs of CPU utilization over the experiment scenario. The critical path of the Elasticsearch process is represented in Fig. 6. We identified specific patterns in CPU critical path that correlated with certain applications and times of experiment. For instance, a distinct block period comes to the fore, illustrating a phase during which Elasticsearch encounters disk-related blocking. This interval is accompanied by a sequence of different events showcased in Fig. 7. Among these events, there is a sequence involving `sched_switch` occurrences, followed by futex and IRQ events, culminating in the appearance of the `block_rq_complete` event. This latter event signifies the completion of the blocked request, allowing the process to regain access to the CPU. Researchers often utilize these events to gauge scheduling efficiency. Leveraging this artifact, they can assess these aspects based on the behavior of Elasticsearch under various workloads, as demonstrated by this artifact. This insight allowed us to optimize the scheduling of the job, thereby reducing peak CPU load.

By examining the disk usage logs, we identified trends and potential issues with storage resources. The proposed ELK artifact helped us pinpoint periods of high disk I/O activity, which were traced back to specific operations running during the artifact scenario. In Fig. 7, a distinct block period is illustrated, showing a phase during which Elasticsearch encounters disk-related blocking. This interval is accompanied by a sequence of events showcased in the figure. The `block_rq_complete` event signifies the completion of the blocked request, allowing the process to regain access to the CPU. Researchers often use these events to evaluate disk usage efficiency.

Furthermore, in Fig. 8, the read and write throughput of the Disk is showcased through the time interval spanning from the receipt of a request `block_rq_issue` to its completion `block_rq_complete`. Although fluctuations in disk throughput are evident, it becomes apparent that the presence of noise significantly influences disk usage, leading to heightened utilization.

3.2. Noise detection

Performance analysis serves a dual purpose: it not only fine-tunes the utilization of system resources but also plays a crucial role in enhancing system reliability. The emergence of non-deterministic and undesired variations, referred to as performance noise, significantly impacts the anticipated or desired reliability of a system (de Oliveira et al., 2022; Rameshan et al., 2014). This noise can lead to decreased throughput, heightened latency, inefficient resource utilization, and an overall decline in system performance (Malallah et al., 2021). Therefore, the identification and mitigation of noise are crucial for optimizing system performance and ensuring the dependable execution of tasks (Iguchi and Yamada, 2022). By minimizing the disruptive effects of noise, system reliability can be enhanced, allowing tasks to effectively leverage available resources and achieve their objectives with greater dependability (Chen et al., 2020).

Conducting root cause analysis is critical for system administrators in ensuring the reliability of the system in the presence of noise. By identifying the underlying sources of noise, administrators can implement focused solutions to lessen its impact and improve system reliability (León et al., 2016). Different types of noise root causes, particularly in high-performance computing environments, call for specific actions to effectively address reliability concerns (Rameshan et al., 2014; Copik et al., 2021). For instance, managing CPU noises might involve optimizing the thread-to-core ratio to better allocate resources (Fan et al., 2021) and reduce performance variations. Similarly, addressing network noises could entail using redundancy mechanisms or load-balancing techniques to enhance data transmission reliability and reduce the likelihood of system failures. Proactively recognizing and tackling noise root causes empowers system administrators to enhance overall system reliability and ensure consistent, dependable operation.

Kernel event tracing is a technique used to monitor and capture low-level events occurring within the kernel of an operating system (Woodside et al., 2020). By investigating these events, kernel event tracing enables the identification of noise and supports root cause analysis. However, effectively analyzing a substantial number of kernel-level events poses a significant challenge when applying noise detection and root cause analysis (Janecek et al., 2022; Chen et al., 2019). Moreover, accurately processing complex events generated at the kernel level is essential for precisely pinpointing the root causes of the observed noises (Huang et al., 2022). The implementation details of the noise detection application are provided in the GitHub repository.[16]

Noises entail disruptions in performance that arise when a process encounters delays or interruptions while awaiting CPU or non-CPU resources like Disk I/O, Network I/O, or Memory. Observing kernel events can shed light on the characteristics of these noises and

[16] URL: https://github.com/mnoferestibrocku/dataset-repo/tree/main/Applications/Noise-Detection

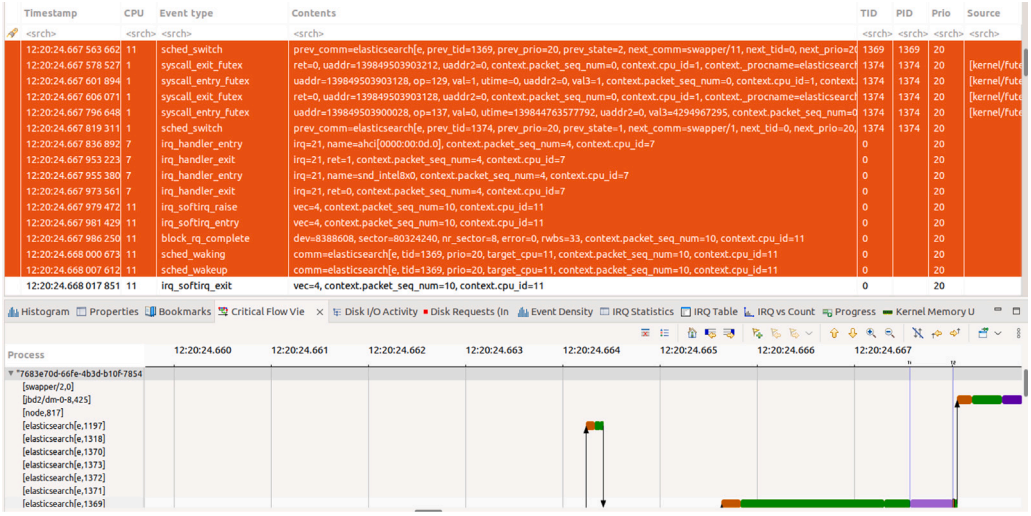


Fig. 7. The events generated during the block segment.

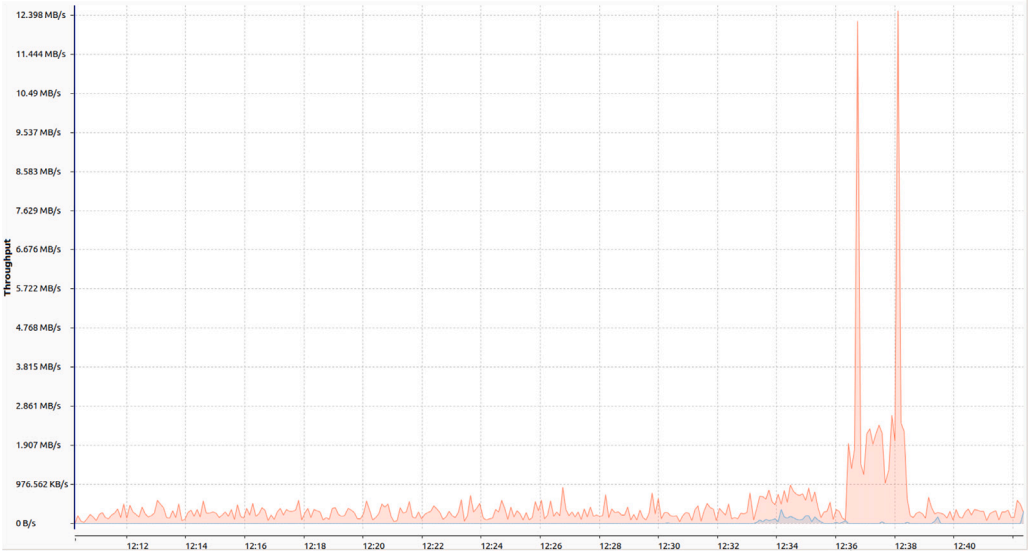


Fig. 8. The Disk read and write throughput.

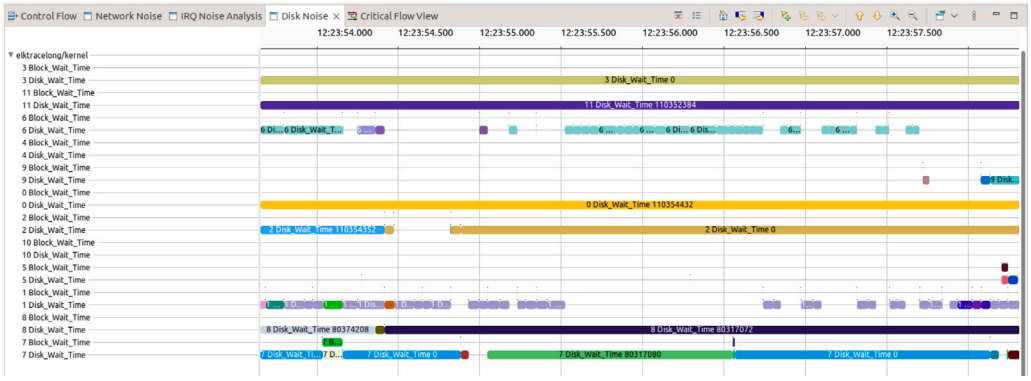
their underlying origins (Daoud and Dagenais, 2018). To illustrate, disk I/O operations trigger a sequence of events commencing with the `block_getrq` event, denoting the software's requirement for a particular disk block for I/O. This is followed by other events, including `block_rq_insert`, `block_rq_issue`, and `block_rq_complete`, collectively portraying the entire life cycle of the disk I/O operation. These events furnish valuable insights into distinct noise issues that may emerge during disk I/O operations. For instance, external noise affecting hardware utilization might lead to a noteworthy time interval between the `block_rq_issue` and `block_rq_complete` events. Conversely, if another process generates an excessive quantity of hard requests, a consistently busy queue within the driver could result in an elongated time span between the `block_rq_insert` and `block_rq_issue` events.

In the context of monitoring and analyzing noise-related issues in disk I/O performance, two key metrics can be defined: the *Block_Queue* and the *Disk_Queue*. The *Block_Queue* metric encompasses requests that have undergone the `block_rq_insert` event but have not yet encountered the subsequent `block_rq_issue` event. Similarly, the *Disk_Queue* metric includes requests that have experienced the `block_rq_issue` event but are still pending the `block_rq_complete` event. Through the ongoing observation of these metrics and the careful analysis of the associated events, various noise detection approaches can effectively identify and address issues pertaining to disk I/O performance, thus contributing valuable insights for conducting root cause analysis and enhancing system efficiency.

Fig. 9 provides a visual representation of the disk I/O-related metrics computed from the artifact, encompassing all 12 disks within the system. For a more detailed examination, one can zoom in on a specific time frame, as demonstrated in Fig. 9(b). This close-up view exposes a prolonged wait time for Disk 0. Such an observation prompts a comprehensive analysis to precisely identify the underlying origin of this noise. A deeper investigation into the extended wait duration for Disk 0 becomes crucial for the optimization of overall system performance and the assurance of dependable disk I/O operations.

The network traffic logs in our dataset were analyzed to detect unusual patterns that could indicate noise or inefficiencies. By closely observing specific metrics, the proposed artifact logs proved effective in identifying disruptions related to network activity. These disruptions encompass various types of network noise, each potentially impacting system performance and overall reliability. For example, network congestion might lead to increased latency and delayed data transfers,

(a) Visualization of the entire experiment, capturing the complete duration and all disks.



(b) Zooming in on a small period of the experiment to focus on specific details and events.

Fig. 9. Monitoring the artifact kernel events to detect disk I/O related noise and the root cause.

while packet loss incidents could result in incomplete or incorrect data transmission. Thoroughly analyzing relevant metrics is key to optimizing network performance and minimizing disruptions.

To elaborate on these metrics, consider that every send packet request within the network originates with the `net_dev_queue` event. This event signifies the point at which a network device queues a packet for transmission. In scenarios devoid of noise, we anticipate observing the `irq_softirq_entry` event near the `net_dev_queue` event, with matching Process ID (PID) and Thread ID (TID). The `irq_softirq_entry` tracepoint captures instances when a software interrupt or softirq is triggered, offering insights into the ongoing processing of the interrupt or softirq. Fundamentally, the emergence of the `irq_softirq_entry` event signifies the active transmission of a packet.

In cases where noise is present, however, the duration between the `net_dev_queue` and `irq_softirq_entry` events—referred to as Trs_Wait_Time—is extended, indicating potential delays or interruptions in the packet transmission process. By monitoring these intervals, we were able to detect and analyze network noise, leading to actionable insights for mitigating its impact. This capability to identify and address network disruptions demonstrates the practical value of our dataset and the effectiveness of Elasticsearch in network traffic monitoring and noise detection.

Similarly, concerning the reception of packets, the occurrence of the `net_if_receive_skb` event denotes the successful receipt of a network packet by the network interface card. Following this event is the `sched_waking` event, signifying the completion of the network request. The time interval between these events, termed Rcv_Wait_Time, can also serve as an indicator of network noise.

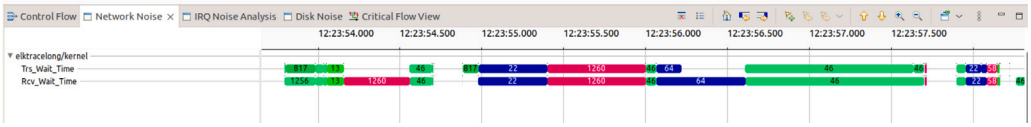
These metrics, encapsulating the entire timeline of the experiment, are depicted in Fig. 10. A more precise examination of specific time segments exposes instances such as Process ID 46 expending approximately 1 s on a receive packet—an interval significantly deviating from the norm, potentially indicating the presence of noise. This combination of metrics, along with various clear signs, creates a crucial basis in the suggested artifact framework. This foundation allows for the successful identification and thorough analysis of network noise and its root causes.

In our experiment, we introduced various noise types to emulate real-world challenges in the Elasticsearch environment, encompassing factors such as CPU contention, network congestion, disk latency, and memory constraints. This diversity of noise sources presents an opportunity to apply a range of noise detection approaches to our collected artifact. Each approach utilizes distinct techniques, algorithms, and heuristics to analyze performance data and identify patterns associated with different noise types and their root causes. Through this comprehensive analysis, we can gain insights into the effectiveness and accuracy of each approach under diverse noise conditions. The enriched artifact, featuring multiple noise scenarios, becomes a valuable dataset for comparative analysis, providing researchers and practitioners with a platform for benchmarking and evaluating noise detection algorithms in complex, real-world environments.

3.3. Software phase detection

Application phases correspond to specific intervals during software execution characterized by consistent behaviors and resource requirements (Criswell and Adegbija, 2020). Recognizing these phases

(a) Visualization depicting the entirety of the experiment.



(b) Zooming in on a specific timeframe within the experiment to closely examine the metric details.

Fig. 10. Monitoring the kernel events in the artifact to identify noise and root causes related to network send and receive operations.

enhances understanding of an application's performance traits and supports optimization. Previous research underscores the value of phase-aware analyses in adaptive computing tasks, like just-in-time compilation, thread-to-core assignment, and resource allocation (Criswell and Adegbija, 2020). These methods' efficacy underscores the potential of a phase-oriented approach in adaptive tracing.

Tracing is essential for analyzing, debugging, and monitoring running processes. The sequence of events generated during execution can reveal software phases. In Fig. 11, event sequences are segmented into windows, where each window covers specific amount of time. For each window, multiple features can be generated to show the perspective of the software's execution including the Software Behavior and Resource Utilization features. The System Behavior feature show how many times each system call was invoked during the execution window. how much time the thread used different resources to complete its task can be represented by the Resource Utilization feature. These two features, or any other features based on the artifact events can be defined to represent the windows. Basic code examples for feature extraction and clustering using K-means, SOM, and hierarchical clustering algorithms can be found in the GitHub repository.[17]

Phase identification approaches try to employ grouping (clustering) techniques to identify the software system's main execution phases including outlying behaviors of interest. For instance, clustering methods can apply on the Software Behavior feature to identify windows performing similar tasks. Utilizing the Resource Utilization features to the clustering methods, application phases can be find as intervals within a software's execution that exhibit similar behaviors and resource requirements.

The phase sequences are given to a prediction model such as LSTM model, which is trained to predict the upcoming phase. To account for dynamic behaviors, the model constantly compare a window's predicted label P'_{t+1}(outlier, or one of the phases) with its real phase P_{t+1}. If the number of incorrect predictions surpasses a threshold, the prediction model must be retrained. This prediction finds application in adaptive tracing or configuration. Adaptive tracing adjusts tracing levels according to phases, while system administrators alter configurations, especially in cloud environments, based on anticipated phases.

Various machine learning algorithms offer the capability to perform tasks like clustering, classification, or phase prediction by leveraging features extracted from the artifact. For instance, features such as the count of system calls within specific time intervals or the duration of these calls can be harnessed. These features, which are influenced by factors like system load and noise, empower machine learning algorithms to discern patterns and forecast distinct phases of system behavior.

[17] URL: https://github.com/mnoferestibrocku/dataset-repo/tree/main/Applications/Phase-Detection

4. Related works

Black-box performance engineering approaches aim to ensure software efficiency and reliability by observing and measuring software's operational behavior without interference, assessing diverse performance metrics.

In the realm of software engineering, the related work focuses on the pivotal role of reliability in ensuring the quality and success of software products (Grieco et al., 2020). Existing literature highlights the significance of reliability prediction models as valuable resources for managing software quality. However, despite the plethora of proposed models by practitioners, there exists a notable gap between the development of these models and their practical implementation in industrial settings (Sahu and Srivastava, 2020). Sahu and Srivastava in Sahu and Srivastava (2021) addresses the longstanding concern of software reliability, focusing on modeling and estimating methods to reduce failure rates. It highlights the challenges posed by nonlinear existing methods and proposes the use of soft computing techniques, such as fuzzy logic and neural networks, for reliability estimation and prediction. Through experimentation on large datasets, including the Apache server and MyLyn application software datasets, the authors demonstrate the effectiveness of these methods. They propose a hybrid neuro-fuzzy methodology to predict software reliability, tailored to the varying failure patterns observed in different datasets. This approach aids designers and developers in making informed decisions during reliability assessment, ultimately aiming to enhance software quality and reduce failures (Sahu et al., 2021).

The literature review underscores the pressing need for reliable prediction models and offers insights into best practices to enhance their usability (Sahu and Srivastava, 2018). Practitioners are urged to consider these insights in their efforts to reduce failure rates and enhance software reliability, ultimately contributing to the development of more robust and dependable software systems (Sahu et al., 2021). The proposed artifact is helpful in this area by adding more scenarios to test and evaluate different reliable prediction models. Different noise scenarios can demonstrate how the models are able to increase the reliability of the system. Our proposed artifact contains kernel-level events, prompting us to review certain approaches that utilize these events and how they evaluate their methodologies.

Desfossez et al. (2016) introduce a novel kernel-based approach that empowers developers and administrators to effectively trace latency issues in production environments and initiate actions upon detecting abnormal conditions. They quantified the impact of the latency tracker in terms of CPU usage using the scheduling latency module. Subsequently, they assessed the overhead in disk I/O through the block latency module. Finally, they evaluated the overall influence of the tracker, alongside varying configurations of LTTng, in a MySQL stress test that amalgamates all available resources.

The paper (Ezzati-Jivan et al., 2020) addresses the challenge of comprehending waiting dependencies among threads and hardware

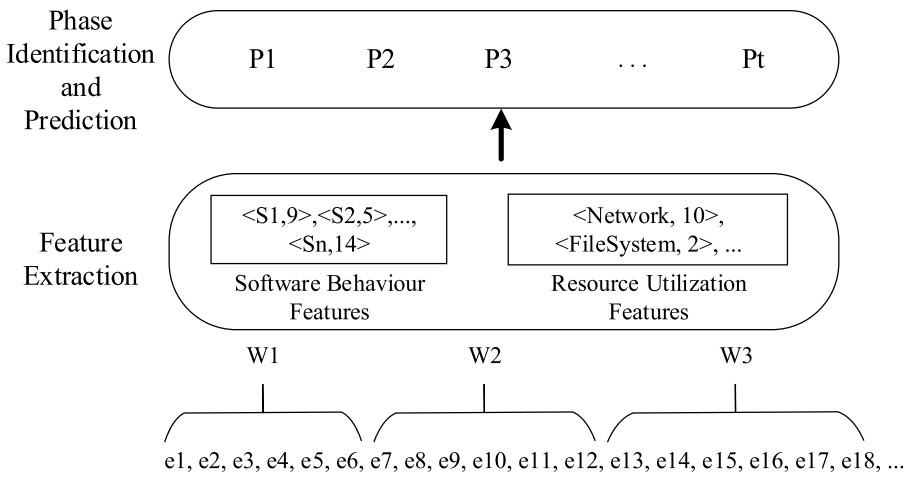


Fig. 11. Phase detection steps.

resources, aiming to enhance software performance by pinpointing bottlenecks stemming from system-level blocking dependencies. The authors adopt system-level tracing to extract a Waiting Dependency Graph (DepGraph) that visually represents the allocation of task execution across threads and resources. This approach aids developers and system administrators in dissecting the allocation of total execution time among interacting components, enabling the detection of bottlenecks and their potential origins. To evaluate the method, they utilize a web application written in PHP running on a server consisting of Apache web server, MariaDB database server, and PHP installed as an Apache module. During periodic evaluations, they notice increases in request response time. To validate their method, they generate three benchmark profiles using the sysbench tool, encompassing CPT test, IO test, and interleaving of sysbench CPU and IO tests. These profiles simulate multi-threaded applications like MySQL, MongoDB, and Apache, representative of various use cases for their method.

Programmers try to enhance application performance by addressing latency issues arising from bugs, design flaws, or external factors. While existing profilers offer limited insights, kernel tracers present comprehensive execution views for pinpointing performance problems. Rezazadeh et al. in Rezazadeh et al. (2020) propose an innovative approach that merges tracing data from both user and kernel spaces to systematically diagnose system performance. This method involves synchronizing and correlating data across different layers to create a unified model, facilitating comprehensive analysis of specific tasks. Unlike approaches constrained to kernel-level trace data, this method accommodates diverse locking mechanisms across both kernel and user levels. They capture traces from PHP requests in Apache, capturing thread activity and states, including the duration each thread spends waiting for lock contention.

The "Tracevizlab" repository,[18] comprises an array of labs and guides meticulously crafted to streamline the process of collecting traces from diverse sources, including the operating system, userspace applications, and cloud environments. These traces are subsequently visualized using specialized trace visualization tools. The primary tools in focus are LTTng for trace collection and Trace Compass for visualization, although certain labs may incorporate alternative tools. To expedite the learning curve, a set of sample traces such as "wget", "network experiment", "bug hunting", and a comparison involving the "package manager" are bundled together in a compressed file named "TraceCompassTutorialTraces.tar.gz" This resource allows users to bypass the need to reproduce experiments and generate traces. By providing both traces and labs, Tracevizlab serves as a comprehensive repository that covers the spectrum of tracing's utility—ranging from diagnosing performance issues to familiarizing users with various tracing tools and techniques for trace generation. Additionally, the repository aids in selecting and applying suitable trace visualizations to expedite the identification of performance-related problems. Furthermore, Tracevizlab facilitates a profound understanding of application and operating system behavior, as well as their underlying mechanisms, through the perspective of tracing techniques.

As various performance engineering approaches evaluate their techniques using their respective datasets, we have introduced a comprehensive artifact that encompasses all kernel-level events for the widely recognized and extensively implemented ELK framework. Diverse workloads are applied across varying time periods, and we have incorporated different types of noise to enhance the realism of the system. To our best knowledge, this marks the first instance of releasing kernel-level traces for intricate, precisely defined workloads within a noisy system. We firmly believe that this artifact holds the potential to serve diverse performance analysis methodologies, facilitating the evaluation and comparison of their outcomes. This resource provides a unique opportunity for researchers to thoroughly assess and contrast their approaches within a controlled yet authentically dynamic environment.

5. Conclusion and future work

Performance engineering is a systematic and forward-looking approach aimed at designing, constructing, and enhancing software systems to ensure their efficient and reliable functioning. Root cause analysis is a critical aspect of performance analysis, involving a thorough examination of kernel-level events associated with performance monitoring. To address the gap in comprehensive resources encompassing diverse scenarios and their corresponding traces, we have introduced an innovative artifact. This artifact captures kernel-level events using Elasticsearch and Kibana, well-regarded open-source tools known for robust log management and analysis. We have defined both Light-load and Heavy-load scenarios, comprising 2 and 10 reports, respectively, with periodic intervals of 20 s and 40 s. Additionally, we have enhanced the artifact's realism by simulating CPU, I/O, Network, and Memory noise, replicating disruptions that can impact performance and accuracy. The incorporation of different types of noise contributes authenticity to the artifact, creating a real-world-like environment for performance engineering.

We highlighted how the proposed artifact can be served in three distinct applications. The first is performance analysis, where kernel events are harnessed for monitoring. The second focuses on noise detection and root cause analysis using kernel events. Lastly, we delved into software phase detection through kernel-level event monitoring.

[18] URL: https://github.com/tracevizlab/tracevizlab

Our research contributes to advancing the field of performance engineering by offering researchers a comprehensive artifact to investigate the challenges of performance analysis and optimization in real-world scenarios.

While our artifact offers significant value in evaluating and optimizing system performance, it is essential to recognize its limitations. One limitation is the reliance on simulated noise scenarios, which may not fully capture the complexity and variability of real-world performance disruptions. Additionally, the scalability of our artifact to larger and more diverse systems could pose challenges, particularly in terms of resource utilization and data management. Furthermore, the generalizability of our findings may be limited by the specific characteristics of the ELK stack and the chosen workload scenarios. Future research could focus on addressing these limitations and refining our artifact to better align with the evolving needs of software performance engineering.

Leveraging the presented artifact, we identify potential areas for future work:

- Expanding Noise Simulation: There is potential for further advancement in noise simulation techniques to achieve a more accurate representation of real-world noise scenarios. This could entail delving into more sophisticated noise patterns, encompassing a broader range of system disturbances, and investigating the interplay between various types of noise. By refining our approach to noise simulation, we can better mimic the complexities of real-world environments, enabling researchers to conduct more comprehensive and realistic evaluations of system performance under diverse conditions.
- Expanding Performance Benchmarking: The artifact holds significant potential as a benchmarking tool to systematically evaluate diverse performance optimization techniques. Future endeavors could focus on leveraging the artifact to conduct thorough assessments of the efficacy of various strategies and methodologies in a controlled and reproducible setting. This entails subjecting the artifact to a range of performance optimization approaches and meticulously analyzing the outcomes to gauge their impact on system performance. By employing the artifact in benchmarking exercises, researchers can gain valuable insights into the comparative effectiveness of different optimization techniques, paving the way for informed decision-making and further advancements in performance engineering practices.
- Expanding to Cloud and Distributed Systems: A natural progression for the artifact would be its adaptation to assess the performance and noise characteristics of cloud-based and distributed systems. This extension could offer valuable insights into how these complex environments manage disruptions and resource allocation across distributed infrastructure. By incorporating cloud and distributed system architectures into the artifact's testing scenarios, researchers can explore how factors such as network latency, data replication, and dynamic workload allocation impact overall system performance and resilience. Furthermore, this expansion would enable the evaluation of performance engineering techniques in environments characterized by scalability, elasticity, and varying degrees of resource contention, providing a more comprehensive understanding of system behavior under diverse conditions.
- Exploring Operating System Kernels: The impact of differences in operating system (OS) kernels on performance analysis techniques and artifact applicability involves conducting comparative studies across various OS platforms, such as FreeBSD, Linux, and Windows NT. These studies aim to dissect how variations in kernel design influence the occurrence and characteristics of kernel-level events. Through systematic examination of these differences, researchers can tailor performance analysis methodologies to better suit each OS environment. Moreover, extending the artifact's support to multiple operating systems will facilitate evaluation in diverse environments, enhancing insights into kernel behavior and fostering the development of more resilient performance analysis tools.

These directions offer opportunities to build upon the foundation of the artifact and further advance the field of performance analysis and optimization.

CRediT authorship contribution statement

Morteza Noferesti: Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Conceptualization. **Naser Ezzati-Jivan:** Writing – review & editing, Validation, Supervision, Software, Resources, Project administration, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The artifact is thoughtfully provided under the MIT License within the GitHub repository of the artifact. You can access it through the following URL: https://github.com/mnoferestibrocku/dataset-repo.

References

Bendechache, M., Svorobej, S., Endo, P.T., Mihai, A., Lynn, T., 2021. Simulating and evaluating a real-world elasticsearch system using the RECAP DES simulator. Future Internet 13 (4), 83.

Chen, Y., Lu, Y., Shu, J., 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In: Proceedings of the Fourteenth EuroSys Conference 2019. pp. 1–14.

Chen, R., Zhang, S., Li, D., Zhang, Y., Guo, F., Meng, W., Pei, D., Zhang, Y., Chen, X., Liu, Y., 2020. Logtransfer: Cross-system log anomaly detection for software systems with transfer learning. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 37–47.

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

Criswell, K., Adegbija, T., 2020. A survey of phase classification techniques for characterizing variable application behavior. IEEE Trans. Parallel Distrib. Syst. 31 (1), 224–236. http://dx.doi.org/10.1109/TPDS.2019.2929781.

Curtis, B., Martin, R.A., Douziech, P.-E., 2022. Measuring the structural quality of software systems. Computer 55 (3), 87–90.

Daoud, H., Dagenais, M.R., 2018. Recovering disk storage metrics from low-level trace events. Softw. - Pract. Exp. 48 (5), 1019–1041.

de Oliveira, D.B., Casini, D., Cucinotta, T., 2022. Operating system noise in the linux kernel. IEEE Trans. Comput. 72 (1), 196–207.

Desfossez, J., Desnoyers, M., Dagenais, M.R., 2016. Runtime latency detection and analysis. Softw. - Pract. Exp. 46 (10), 1397–1409.

Devins, J., Tibshirani, J., Lin, J., 2022. Aligning the research and practice of building search applications: Elasticsearch and pyserini. In: Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining. pp. 1573–1576.

Dhulavvagol, P.M., Bhajantri, V.H., Totad, S., 2020. Performance analysis of distributed processing system using shard selection techniques on elasticsearch. Procedia Comput. Sci. 167, 1626–1635.

Ezzati-Jivan, N., Fournier, Q., Dagenais, M.R., Hamou-Lhadj, A., 2020. DepGraph: Localizing performance bottlenecks in multi-core applications using waiting dependency graphs and software tracing. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation. SCAM, pp. 149–159. http://dx.doi.org/10.1109/SCAM51674.2020.00022.

Fan, G., Chen, T., Yin, B., Chen, L., Wang, T., Wang, J., 2021. Static bound analysis of dynamically allocated resources for C programs. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 390–400.

Ferrag, M.A., Friha, O., Hamouda, D., Maglaras, L., Janicke, H., 2022. Edge-IIoTset: A new comprehensive realistic cyber security dataset of IoT and IIoT applications for centralized and federated learning. IEEE Access 10, 40281–40306.

Glatzmaier, G., Sievers, K., Magenheimer, D., et al., 2005. SystemTap: Instrumentation for the linux kernel. Linux J. 2005 (131).

Grieco, L.A., Boggia, G., Piro, G., Jararweh, Y., Campolo, C., 2020. Ad-hoc, mobile, and wireless networks. In: Proceedings of the 19th International Conference on Ad-Hoc Networks and Wireless, ADHOC-now. Springer, pp. 19–21.

Huang, Y., Chen, L., Luo, L., Xiao, K., Li, Y., 2022. Formal verification of the interrupt dispatch program of an embedded operating system. In: 2022 8th International Symposium on System Security, Safety, and Reliability. ISSSR, IEEE, pp. 104–112.

Iguchi, T., Yamada, H., 2022. Graceful ECC-uncorrectable error handling in the operating system kernel. In: 2022 IEEE 33rd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 109–120.

Janecek, M., Ezzati-Jivan, N., Hamou-Lhadj, A., 2022. Performance anomaly detection through sequence alignment of system-level traces. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. ICPC '22, Association for Computing Machinery, New York, NY, USA, pp. 264–274. http://dx.doi.org/10.1145/3524610.3527898.

Kathare, N., Reddy, O.V., Prabhu, V., 2020. A comprehensive study of elasticsearch. Int. J. Sci. Res. (IJSR).

Lee, J., Kwon, H.Y., 2022. TPC-C benchmarking for ElasticSearch. In: 2022 IEEE International Conference on Big Data and Smart Computing. BigComp, IEEE, pp. 171–174.

León, E.A., Karlin, I., Moody, A.T., 2016. System noise revisited: Enabling application scalability and reproducibility with SMT. In: 2016 IEEE International Parallel and Distributed Processing Symposium. IPDPS, IEEE, pp. 596–607.

Loyola-González, O., 2019. Black-box vs. White-box: Understanding their advantages and weaknesses from a practical point of view. IEEE Access 7, 154096–154113. http://dx.doi.org/10.1109/ACCESS.2019.2949286.

Malallah, H., Zeebaree, S.R., Zebari, R.R., Sadeeq, M.A., Ageed, Z.S., Ibrahim, I.M., Yasin, H.M., Merceedi, K.J., 2021. A comprehensive study of kernel (issues and concepts) in different operating systems. Asian J. Res. Comput. Sci. 8 (3), 16–31.

Molnar, I., 2010. Perf: Linux profiling with performance counters. Linux J. 2010 (191).

Ni, C., Wu, J., Wang, H., Lu, W., Zhang, C., 2024. Enhancing cloud-based large language model processing with elasticsearch and transformer models. arXiv preprint arXiv:2403.00807.

Rameshan, N., Navarro, L., Monte, E., Vlassov, V., 2014. Stay-away, protecting sensitive applications from performance interference. In: Proceedings of the 15th International Middleware Conference. pp. 301–312.

Rezazadeh, M., Ezzati-Jivan, N., Galea, E., Dagenais, M.R., 2020. Multi-level execution trace based lock contention analysis. In: 2020 IEEE International Symposium on Software Reliability Engineering Workshops. ISSREW, pp. 177–182. http://dx.doi.org/10.1109/ISSREW51248.2020.00068.

Sahu, K., Alzahrani, F.A., Srivastava, R., Kumar, R., 2021. Evaluating the impact of prediction techniques: Software reliability perspective. Comput., Mater. Continua 67 (2).

Sahu, K., Srivastava, R., 2018. Soft computing approach for prediction of software reliability. Neural Netw. 17, 19.

Sahu, K., Srivastava, R., 2020. Needs and importance of reliability prediction: An industrial perspective. Inf. Sci. Lett. 9 (1), 33–37.

Sahu, K., Srivastava, R., 2021. Predicting software bugs of newly and large datasets through a unified neuro-fuzzy approach: Reliability perspective. Adv. Math.: Sci. J. 10 (1), 543–555.

Ullah, F., Babar, M.A., 2019. Architectural tactics for big data cybersecurity analytics systems: A review. J. Syst. Softw. 151, 81–118.

VanDonge, R., Ezzati-Jivan, N., 2022. N-lane bridge performance antipattern analysis using system-level execution tracing. In: 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 83–93.

Velez, M., Jamshidi, P., Sattler, F., Siegmund, N., Apel, S., Kästner, C., 2020. ConfigCrusher: Towards white-box performance analysis for configurable systems. Autom. Softw. Eng. 27, http://dx.doi.org/10.1007/s10515-020-00273-8.

Wei, B., Dai, J., Deng, L., Huang, H., 2020. An optimization method for elasticsearch index shard number. In: 2020 16th International Conference on Computational Intelligence and Security. CIS, IEEE, pp. 191–195.

Wert, A., 2018. Performance Problem Diagnostics by Systematic Experimentation, vol. 20, KIT Scientific Publishing.

Woodside, M., Franks, G., Petriu, D.C., 2007. The future of software performance engineering. In: Future of Software Engineering. FOSE'07, IEEE, pp. 171–187.

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

Yang, W., Li, H., Li, Y., Zou, Y., Zhao, H., 2022. Design and implementation of intelligent warehouse platform based on elasticsearch. In: Proceedings of the 2022 6th International Conference on Software and E-Business. pp. 69–73.

Zehlike, M., Sühr, T., Castillo, C., Kitanovski, I., 2020. Fairsearch: A tool for fairness in ranked search results. In: Companion Proceedings of the Web Conference 2020. pp. 172–175.

Zmaranda, D.R., Moisi, C.I., Győrödi, C.A., Győrödi, R.Ş., Bandici, L., 2021. An analysis of the performance and configuration features of mysql document store and elasticsearch as an alternative backend in a data replication solution. Appl. Sci. 11 (24), 11590.

Morteza Noferesti is an accomplished professional specializing in software development, computer networks, and performance engineering. Currently a Postdoctoral Fellow at Brock University, Morteza's expertise lies in kernel-level tracing and software performance research. He has held various academic and industry roles, including positions as a University Lecturer and Senior Software Engineer. Morteza's academic journey includes earning a Ph.D. and an M.Sc. in Information Technology Engineering from Sharif University of Technology, along with a B.Sc. from Shiraz University of Technology. He has made significant contributions to academia through publications, presentations, and academic services, and is actively involved in mentoring students. Morteza possesses a diverse skill set, with proficiency in programming languages, databases, operating systems, containers, and network peripherals. Overall, he is a dedicated professional committed to advancing knowledge in software engineering and performance evaluation through his interdisciplinary expertise and passion for teaching and research.

Naser is an Assistant Professor in the Department of Computer Science at Brock University. His research focuses on automated software analysis and performance evaluation of complex software systems. In his research, Naser leads a dynamic team exploring various aspects of software engineering, including software analysis, debugging, performance evaluation, and energy efficiency. His work delves into operating system-level methods such as execution tracing and profiling to enhance software performance and reliability in IoT, cloud, and distributed systems.