PASD: A Performance Analysis Approach Through the Statistical Debugging of Kernel Events

Mohammed Adib Khan
Department of Computer Science
Brock University
St. Catharines, Canada
ak19qp@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—**Dynamic performance analysis plays a crucial role in optimizing systems and identifying performance bottlenecks. Traditional software debugging methods frequently encounter difficulties when trying to pinpoint performance problems in complex software settings. This is often because performance issues remain hidden during the code execution within debugging tools or under certain run-time circumstances, making them challenging to identify and address. This paper introduces PASD (Performance Analysis through Statistical Debugging), a dynamic performance analysis approach based on statistical debugging of kernel-level trace events. Importantly, this approach requires no application code instrumentation and purely utilizes operating system kernel trace events for analysis. PASD collects kernel trace events generated during software execution and utilizes heuristics to analyze their performance issues and the root-causes. Through statistical debugging techniques, PASD identifies the most important functions correlated with performance problems. It notably does so without disrupting the software's normal functions and ensuring that any issues are detected in the software's typical operating conditions, thus avoiding additional complexity in the debugging process. We have conducted two empirical studies to assess the effectiveness of PASD on performance issues in the Firefox web browser as well as the 'ls' tool (a common utility in Unix-like systems). Our experiments demonstrate that PASD successfully identifies performance issues and their causes in software without prior knowledge of the architecture or source code instrumentation. By providing an overview of software behavior through the kernel-level, our proposed method can aid developers and testers in quickly pinpointing performance problems in the source code. This, in turn, can result in improved software quality, increased user satisfaction, and the prevention of critical system failures.**

Index Terms—**Dynamic Analysis, Software Performance Debugging, Statistical Debugging, Kernel Tracing**

I. Introduction

Performance debugging, the process of identifying and diagnosing bottlenecks within software systems, has become increasingly vital in today's computing landscape [1]. As software architectures grow more complex and diverse deployment environments become the norm, performance issues can greatly undermine user experience and overall system efficiency. For instance, a major e-commerce platform may experience a slowdown during peak shopping periods [2], leading to longer page load times. This seemingly minor issue can have a significant impact on user experience, potentially leading to customer dissatisfaction and loss of revenue. In a more critical context, consider a healthcare system where a delay in processing patient data could result in delayed diagnoses or treatment, with potentially life-threatening consequences. These examples underscore the importance of effective performance debugging in maintaining the efficiency, reliability, and user satisfaction of software systems. However, as software architectures become increasingly complex and diverse, traditional debugging techniques often fail to identify and resolve performance issues [3].

Performance issues pose a unique set of challenges in comparison to conventional functional bugs [4]. Unlike functional bugs, which typically manifest as distinct errors or incorrect output, performance issues tend to be subtle, elusive, and dependent on specific runtime conditions or code execution paths. These issues often remain concealed during code execution within debugging tools or when the software is run under normal operating conditions [5]. This hidden nature makes performance issues notoriously difficult to identify and resolve, creating a critical need for more advanced and nuanced debugging approaches [1].

While a variety of solutions have been proposed to address performance debugging challenges, they often prove to be either insufficient or overly complex in practice. For instance, static analysis methods [6], although capable of detecting potential performance issues, lack the ability to capture the dynamic behavior of a system during runtime [7]. On the other hand, pure dynamic analysis methods, while more comprehensive, are often associated with high overhead, making them less suitable for systems running in production [8], [9]. Instrumentation-based techniques provide fine-grained details about program execution but can significantly alter the system's behavior and performance [10]. Similarly, machine learning-based approaches for performance debugging, despite their potential to uncover subtle, non-linear relationships in data, face scalability issues and the risk of overfitting [11]. In response to these challenges, our work seeks to combine the benefits of statistical debugging with the richness of kernel-level event data to offer a more effective approach for performance debugging.

Performance debugging through kernel events is a methodology that involves analyzing and optimizing software system performance by capturing and studying events generated at the kernel level during runtime. These events, including system calls, interrupts, context switches, and memory allocations, provide crucial insights into the behavior and resource utilization of the system [12]. By monitoring and collecting kernel events, performance debuggers gain a deep understanding of the software's execution flow, resource usage patterns, and potential performance bottlenecks. This approach offers a comprehensive view of the system's interaction with the underlying operating system, enabling developers to diagnose and resolve performance issues more effectively [5].

Kernel-level debugging presents two prominent challenges: the generation of a large amount of data and the inherent complexity of the events involved. Debugging at the kernel level results in a significant volume of data due to the multitude of events occurring during software execution [12]. This data overload poses difficulties in terms of storage, processing, and analysis. To effectively handle this data, efficient techniques such as event filtering, aggregation, and compression are necessary. Furthermore, the complexity of kernel-level events adds another layer of complexity to the debugging process. These events are intricately linked to low-level system operations, hardware interactions, and resource management. The diverse range of events, including interrupts, system calls, and memory allocations, further complicates the analysis process [11]. By efficiently managing a large amount of data and comprehending the complexities of the events, developers can successfully identify and resolve performance bottlenecks and bugs in software systems at the kernel level.

In this paper, we present PASD (**P**erformance **A**nalysis through **S**tatistical **D**ebugging), a dynamic kernel event analysis approach aims to identify functions that are correlated with performance issues. PASD collects kernel-level events generated during software execution and applies statistical debugging techniques to estimate the performance of various functions. By analyzing the collected data, PASD can identify the most significant functions related to performance problems without interfering with the normal execution process. Conventionally, statistical debugging is predominantly implemented in static analyses at the source code level [5], [13]. However, we have taken a different approach by applying these techniques to the data derived from kernel level events, thereby enriching the landscape of potential insights.

We provide four main contributions in this paper:

1) We formally define the framework for function performance debugging using kernel-level event analysis, a novel methodology that enhances the understanding of system behavior at the kernel level.
2) We propose a new approach for function performance debugging that significantly extends existing methodologies by capturing and efficiently analyzing a large volume of kernel-level events.
3) We introduce the use of statistical debugging techniques to analyze the software trace logs and identify the most important functions associated with performance issues.
4) We demonstrate the effectiveness of our approach through real-world case studies, including two performance bugs in the Mozilla Firefox web browser and one performance bug of the 'ls' command in Linux.

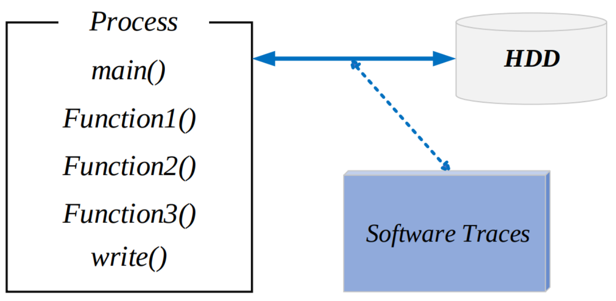


Fig. 1. An example: performance issues in *HDD* I/O Operations for *Process*.

The rest of this paper is organized as follows: Function performance debugging framework is defined in Section II. The details of PASD are described in Section III. Section IV evaluates the effectiveness of PASD in debugging multiple bugs in Firefox through monitoring kernel-level traces. Related works are reviewed in Section V. Finally, Section VI concludes the paper and discusses some future works.

II. The Formulation of Function Performance Debugging

This section starts by presenting an example that illustrates how function performance debugging can be approached using dynamic analysis through kernel events. The purpose of this example is to provide a clear and accessible understanding of the fundamental principles involved. Building upon this example, we then offer a formal description of the problem, delving into the specific details and complexities associated with formulating performance debugging within the context of dynamic analysis at the kernel level.

Consider a scenario where performance issues in *Hard Disk Drive (HDD)* I/O operations are reported for a specific *process*. This process comprises three distinct functions, namely *function1*, *function2*, and *function3*, which interact with the *HDD* through the "*write*" function. The overall depiction of the example is presented in Figure 1. The primary objective is to determine the extent of association between these functions and the reported performance issues.

During runtime, the software generates a significant number of events, which are collected as *Software Traces*. Kernel tracing enables the collection of information from the operating system, including system calls, interrupts, and function calls without instrumenting the software itself. Each disk request is associated with multiple kernel events. Specifically, an *HDD* read/write operation begins with the *block_getrq* event, followed by the insertion of the request into the waiting queue using the *block_rq_insert* event. Subsequently, the request is issued (*block_rq_issue*) to the driver for processing. If the disk successfully processes the request, the *block_rq_complete* event is triggered, releasing the request data structure and awakening any blocked processes. It is evident that a single

HDD operation generates various events, necessitating efficient analysis of these events.

The generated events, their attributes and the time intervals between different events during these operations can provide crucial insights into the system's performance. For instance, in the case of a slow HDD, there will be a significant time interval between the *block_rq_issue* and *block_rq_complete* events. On the other hand, a significant time interval between the *block_rq_insert* and *block_rq_issue* events could indicate a consistently busy driver's "WAITING" queue, suggesting a potential performance issue (e.g., when the software generates an excessive number of disk/file-system requests in a short amount of time). Therefore, performance analysis needs to consider different kernel event types and their attributes, such as timestamps, duration, etc., to effectively identify and address specific performance issues.

One of the key challenges in performance debugging is identifying the functions/methods responsible for the problem. In the given example, all the functions call the *write* function, which operates with the *HDD*. Let us consider that *Function1* calls the *write* function 10 times, and the time interval between *block_rq_insert* and *block_rq_issue* is around 5ms. In contrast, when *Function2* calls the same function, the time interval exceeds 100ms. This stark difference indicates an evident problem in *Function2*. Yet, pinpointing the problematic function is not always straightforward, especially in scenarios involving multi-threading and workload-related performance issues.

A process is composed of a multiple function sequences where each generate kernel events. The call stack $Call-Stack^i = \langle F^{main}, F^1, F^2, ..., F^i \rangle$, represents the sequence of function calls leading up to function F^i. The call stack begins with the main function, F^{main}, and continues with subsequent functions until it reaches the target function F^i. During the execution of $Call-Stack^i$ distinct kernel events are captured denoting by $\langle e_1^i, e_2^i, ..., e_l^i \rangle$.

Metrics are quantitative measures that capture various aspects of the events and their relationships within the process. They provide a way to quantify the relationship between events based on their specific characteristics or properties. Let us consider a process P, which is represented as a set of call stacks denoted by $P = \{Call-Stack^i | 1 \leq i \leq n\}$. One metric could be defined as the duration between $syscall\_read\_entry$ and $syscall\_read\_exit$ events, which represents the duration of a read operation during the runtime of $Call-Stack^i$. Performance issues occur when certain metrics, which capture event characteristics, exceed predefined thresholds. By continuously monitoring these metrics during runtime, deviations from expected behavior can be detected, signaling potential functions that may be responsible for the performance problems.

Function performance debugging is a process that involves monitoring the execution of function paths within a process P and analyzing the associated event sequences $Call-Stack^i = \langle e_1^i, e_2^i, ..., e_l^i \rangle$ using a set of metrics $Metrics$. This process indicates deviations from expected performance behavior, prompting further investigation and identification of the specific functions responsible for the performance problems. In the upcoming section, we introduce PASD (Performance Analysis through Statistical Debugging), a function performance debugging approach that dynamically analyzes kernel-level events. PASD provides a comprehensive framework for detecting and diagnosing performance issues, enabling efficient optimization and fine-tuning of system performance.

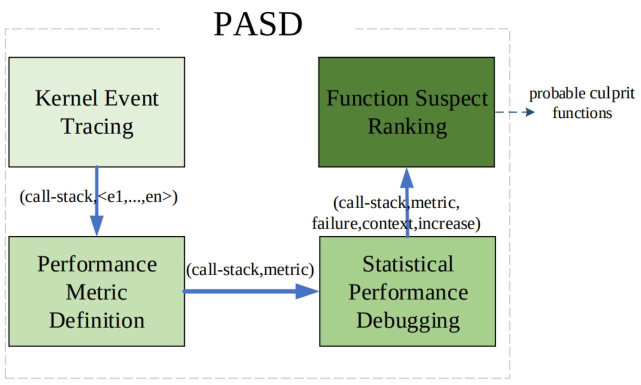

PASD
Kernel Event Tracing
(call-stack,<e1,...,en>)
Performance Metric Definition
(call-stack,metric)
Statistical Performance Debugging
(call-stack,metric, failure,context,increase)
Function Suspect Ranking
probable culprit functions

Fig. 2. The architecture of PASD.

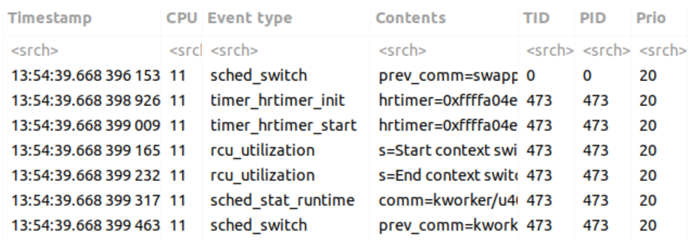
III. PASD: Performance Analysis Through Statistical Debugging

This section provides an in-depth overview of PASD, a dynamic performance analysis approach that leverages statistical debugging of kernel events. The PASD architecture is illustrated in Figure 2. It comprises four main modules: *Kernel Event Tracing*, *Performance Metric Definition*, *Statistical Performance Debugging*, and *Function Suspect Ranking*. The *Kernel Event Tracing* module is responsible for collecting kernel-level events and managing event traces related to the same execution. This module operates as a black box, collecting events without modifying the software code or interfering with process behavior. The *Performance Metric Definition* module defines relevant metrics for performance analysis. The *Statistical Performance Debugging* module analyzes the kernel event traces and performance metrics using statistical debugging techniques to rank functions associated with performance issues. The resulting ranked function list is further prioritized by the *Function Suspect Ranking* module to identify the most correlated functions to the performance problem.

A. Kernel Event Tracing module

The execution of a process generates several kernel events [14], [15]. As a black-box approach, PASD captures these events from the kernel. Considering the volume and variety of events created at the kernel level, *Kernel Event Tracing* module should be able to consider this event stream of data and filter the required events.

PASD models a process as a set of call stacks where each generates a sequential sequence of events $\langle e_1^i, e_2^i, ..., e_l^i \rangle$. Figure 3 depicts the events generated during the execution



Timestamp	CPU	Event type	Contents	TID	PID	Prio
<srch>	<srcl	<srch>	<srch>	<srch>	<srch>	<srch>
13:54:39.668 396 153	11	sched_switch	prev_comm=swapp	0	0	20
13:54:39.668 398 926	11	timer_hrtimer_init	hrtimer=0xffffa04e	473	473	20
13:54:39.668 399 009	11	timer_hrtimer_start	hrtimer=0xffffa04e	473	473	20
13:54:39.668 399 165	11	rcu_utilization	s=Start context swi	473	473	20
13:54:39.668 399 232	11	rcu_utilization	s=End context swit	473	473	20
13:54:39.668 399 317	11	sched_stat_runtime	comm=kworker/u4	473	473	20
13:54:39.668 399 463	11	sched_switch	prev_comm=kwork	473	473	20

Fig. 3. *Kernel Event Tracing* module collects the event stream.

of the software. Each event is characterized by its attributes, including Timestamp, CPU ID, process ID numbers, serial numbers of TUs, event types (e.g., system calls and interrupts), and event details (e.g., IP address of a network connection). The *Kernel Event Tracing* module facilitates the inclusion of call stack data in the recorded events, enhancing their contextual information. The call stack provides a real-time blueprint of a running program, revealing valuable insights into the sequence of function invocations. To ensure accurate pairing, it is important to track the $process\_id$ and $thread\_id$ to avoid mixing the entry and exit events of different processes or threads.

Kernel Event Tracing module collects events by enabling the specific tracepoints of the corresponding Linux kernel subsystem(s). The following list identifies enabled tracepoints and explains the events that they record.

- syscalls (e.g., `futex`): Shows the occurrence of a system call and provide insights into the context of a thread's blocked state.
- `sched_switch`: Indicates that one thread was removed from running on a CPU, and another thread was scheduled on to that CPU to start running.
- `sched_wakeup`: Indicates that a previously blocked thread has become runnable.
- `irq_handler`: Indicates the occurrence of a hardware interrupt.
- `softirq`: Indicates the occurrence of a software interrupt.
- `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 *Performance Metric Definition* module can extract the necessary metrics for the performance statistical debugging.

B. Performance Metric Definition module

The *Performance Metric Definition* module is responsible for analyzing the event stream, establishing relationships between events, and extracting relevant metrics. By understanding the order of events and identifying causal patterns, this module enables effective statistical performance debugging. It calculates metrics such as blocking queue length and CPU waiting times, providing valuable insights into the system's behavior and performance characteristics. Acting as a bridge between raw event data and meaningful performance insights, the *Performance Metric Definition* module tries to identify the performance issues.

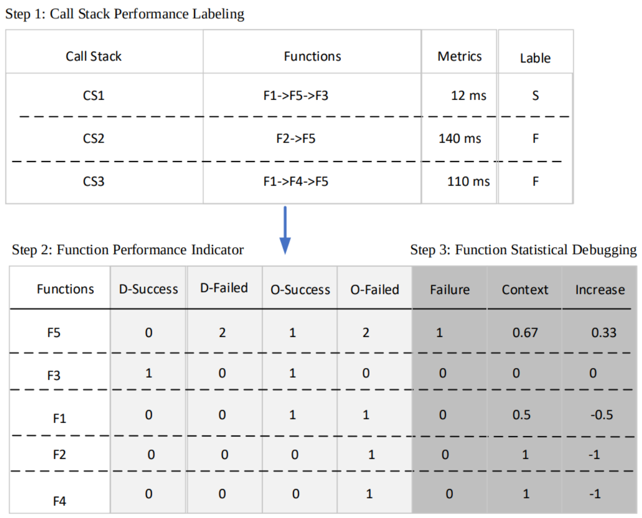
kernel-level events and their attributes could be used to define performance metrics. For instance, the *Performance Metric Definition* module constructs performance metrics by assessing the duration between `block_rq_insert` to `block_rq_issue`, and from `block_rq_issue` to `block_rq_complete`. The first metric potentially reveals the degree of congestion in the waiting queue, providing a useful parameter of system load. On the other hand, the second metric determines the time consumed by a function to carry out a read or write operation to disk, thereby shedding light on system IO performance. The *Performance Metric Definition* module defines a set of performance metrics, including but not limited to the followings:

- The duration between all system call entry and exit events such as `system_call_read_entry` and `system_call_read_exit`.
- The duration between `irq_handler_entry` and `irq_handler_exit` events, monitoring the functions interrupt processing time.
- The duration between `block_rq_insert`, `block_rq_issue`, and `block_rq_complete` events, monitoring the functions disk I/O processing time.
- The duration between `irq_softirq_entry` and `net_dev_queue` events, monitoring the functions network sending packet processing time.
- The duration between `sched_waking` and `net_if_receive_skb` events, monitoring the functions network receiving packet processing time.

C. Statistical Performance Debugging module

This module receives the call stacks and associated metrics as input. The call stacks provide information about the sequence of function calls, while the metrics contain performance-related data such as CPU processing time, disk response time, or network usage. It then processes them in three steps: *Call Stack Performance Labeling*, Function Performance Indicator and *Function Statistical Debugging*. Figure 4 demonstrates an example illustrating the three steps involved in statistical performance debugging. In the following subsections, we will provide a detailed explanation of the steps involved in the example of statistical performance debugging.

1) Call Stack Performance Labeling: PASD continuously monitors the execution of call stacks and identifies performance issues when one or more metrics exceed predefined threshold values. Three threshold values are used to identify the performance issue in a call stack execution: $Pruning_{THR}$, $SUCCESS_{THR}$, and $FAILED_{THR}$. If the observed duration for a metric is below $Pruning_{THR}$, it is disregarded as



Step 1: Call Stack Performance Labeling

Call Stack	Functions	Metrics	Lable
CS1	F1->F5->F3	12 ms	S
CS2	F2->F5	140 ms	F
CS3	F1->F4->F5	110 ms	F

Step 2: Function Performance Indicator

Step 3: Function Statistical Debugging

Functions	D-Success	D-Failed	O-Success	O-Failed	Failure	Context	Increase
F5	0	2	1	2	1	0.67	0.33
F3	1	0	1	0	0	0	0
F1	0	0	1	1	0	0.5	-0.5
F2	0	0	0	1	0	1	-1
F4	0	0	0	1	0	1	-1

Fig. 4. Example: Three Steps of Statistical Performance Debugging

it falls within an acceptable range. If the duration is above $Pruning_{THR}$ but below $SUCCESS_{THR}$, the execution of $Call_{S}tack$ is categorized as a successful execution. Durations falling between $SUCCESS_{THR}$ and $FAILED_{THR}$ are considered ambiguous and are overlooked. However, any duration exceeding $FAILED_{THR}$ is labeled as a failure execution, indicating a performance issue.

Methods of setting thresholds to partition between normal and failure could be obtained by observing performance data over an extended period. From our case studies we have found using the summation of standard deviation and mean provides meaningful results, where values above the threshold indicate failure and values below denote success. For finer tuning, thresholds such as mean minus twice the standard deviation can be employed to filter out events with negligible performance impact. Assuming a metric $metric$ as a statistical random variable, PASD defines the threshold using Equations 1 Where $mean[]$ and $std[]$ represent the average and standard deviation functions, respectively. As depicted in Figure 4, after the completion of the first step, each call stack is assigned a label indicating whether it represents a failure or success.

$$\begin{aligned} Pruning_{THR} &= mean[metric] - 2std[metric] \\ SUCCESS_{THR} &= mean[metric] + std[metric] \\ FAILED_{THR} &= mean[metric] + 2std[metric] \end{aligned} \quad (1)$$

2) Function Performance Indicator: In this step, for each unique function, four indicators are calculated: D-Success, D-Failed, O-Success, and O-Failed. The indicators are defined as follows:

- D-Success: refers to the number of successful executions in which the function is the last function in the call stack.
- D-Failed: accounts for the number of failure executions where the function is the last function in the call stack.
- O-Success: indicates the number of successful executions during which the function is observed.
- O-Failed: indicates the number of failure executions during which the function is observed.

Considering the example shown in Figure 4, five unique functions are observed during the execution. For each function, four indicators are calculated. For instance, function F5 is not the last function in the successful execution (D-Success=0), but it is the last function in two failed executions (O-Failed=2). It is observed during one successful execution (O-Success=1), and it is also observed during the two failed executions (O-Failed=2). These indicators provide valuable insights into the behavior of each function and their occurrences in successful and failed executions.

3) Function Statistical Debugging: As part of our empirical assessment process, we have adopted the predicate ranking method proposed in the Cooperative Bug Isolation (CBI) research [16]. In the realm of statistical debugging [13], [17], a predicate is typically defined as a condition or property within the source code. This predicate is used to pinpoint the presence of a failure or bug in a program, leveraging the runtime information that has been gathered. However, PASD redefines this traditional concept of predicates, applying it at the function level in its statistical debugging methodology for call stacks. These function-level predicates may be present or absent in successful and failed executions.

Following the statistical debugging techniques outlined in [16], [17], three metrics named *Failure*, *Context*, and *Increase* are calculated for the function:

$$Failure = \frac{\text{D-Failed}}{\text{D-Failed} + \text{D-Success}} \quad (2)$$

$$Context = \frac{\text{O-Failed}}{\text{O-Failed} + \text{O-Success}} \quad (3)$$

$$Increase = Failure - Context \quad (4)$$

The *Increase* metric indicates the increasing correlation and active involvement of a function in relation to failure executions. A high increase value suggests that the function holds true in a significant number of failure executions, actively participating in scenarios that lead to failures and making it a potential culprit. Furthermore, during successful executions, the function should be observed sparingly. This contrasting behavior reinforces the function's correlation with failures. When a function is rarely observed during successful executions but frequently observed during failed executions, it strengthens its association with failures and highlights its significance as a potential root cause. By considering these conditions, we establish a ranking based on the *Increase* metric, allowing us to identify and prioritize functions that exhibit a strong correlation with failures. This approach increases the likelihood of pinpointing the functions responsible for performance issues.

D. Function Suspect Ranking module

The ranked list which is generated by *Statistical Performance Debugging* module, is then subjected to pruning by the *Function Suspect Ranking* module. The pruning strategy employed in this module can vary depending on the available debugging resources.

There are two general approaches to pruning the ranked list. The first approach involves selecting a specific percentage of functions from the top of the list for further investigation. For example, this could be set at 15% of the functions in the list. This ensures that the most significant functions, in terms of their impact on the performance issue, are only prioritized for analysis while discarding functions with lesser impacts to save on processing overhead.

The second approach to pruning is based on a threshold criterion. Functions are only investigated if their associated increase value exceeds a predefined threshold. This threshold acts as a filter, allowing only functions that are deemed sufficiently important to be included in the investigation. Both approaches to pruning the ranked list aim to optimize the utilization of debugging resources by focusing on the most promising functions. This ensures that the investigation efforts are directed towards functions that are most likely to contribute to the resolution of the performance problem.

IV. Evaluation

In the upcoming section, we will evaluate the effectiveness of the PASD method in identifying performance issues within open-source applications. We have undertaken three experiments to examine the functionality of our method. Two of these experiments involve assessing PASD's performance in detecting issues within Firefox. The reason for selecting Firefox is because it is an extremely sophisticated multi-threaded application which emulates as a complex enterprise software with hundreds of thousands of lines of codes and functions which is extremely difficult to debug or select points of interest or functions to log or trace for performance related issues. Also, given that it is open-source, we should also be able to verify our findings. The final experiment applies PASD to the Linux 'ls' tool from the GNU Core Utilities library. We have conducted three case studies focusing on performance-related bug reports within the BugZilla repository of the mentioned applications:

- Bug 1637586 [18]: The discernment of explicit functions within the Firefox source code that precipitates performance impediments concerning animation rendering.
- Bug 1565019 [19]: The detection of a series of functions in Firefox inducing CPU resource saturation and subsequently attenuating performance.
- Bug 1290036 [20] & 467508 [21]: Slow performance of the 'ls' command for directories with huge numbers of files and folders.

The PASD method was able to successfully identify the actual functions responsible for the performance issues in all the above case studies. In the subsequent sections, we will provide a detailed explanation of the case studies performed.

All data and source code for our case studies can be accessed via our GitHub repository[1].

A. Case Study 1: Firefox CSS Animation Rendering Bug

In the forthcoming case study, we will examine a performance bug reported for Firefox within Mozilla's Bugzilla repository [18]. According to the report, Firefox encounters severe performance degradation during the rendering of CSS animations; a task handled quite competently by its contemporaneous browser counterparts. Despite the initial reporting of this performance bug dating back several years, the issue persistently remains unresolved. The report identifies functions within the *GFX Web Renderer* class as harboring the malfeasant functions responsible for this issue. Our proposed methodology has successfully corroborated this assertion by detecting the functions from the same culpable class.

1) Study Setup: In the context of this case study, we have reproduced the bug reported in the Bugzilla report [18]. The 3D CSS animation [22] was repeatedly loaded in Firefox to unveil all potential functions correlated with a system call for performing this task. The Firefox browser employed for this investigation was compiled from the source with the 'perf' flag enabled, thus enabling Just-In-Time (JIT) profiling following the Firefox documentation [23] by Mozilla.

We utilized the Linux 'Perf' tool [24] to capture the call stacks. With JIT profiling [25] engaged, 'Perf' was capable of recording the semantically meaningful versions of function names within the call stack. Perf was subsequently hooked to the Firefox Process ID (PID) to only collect Firefox-related call stack information. It was run using the command "Perf record -g -a -e 'syscalls:sys_*' -p[Firefox PID]" which states that only events related to system calls should be monitored and stored across all CPUs. This helps to reduce overhead drastically as we are only monitoring for events which are only system call related and that too for a specific PID.

A sufficient number of events was recorded by performing the experiment multiple times over a period of 30 seconds. The collected data amounts to a total of 78.6MB, containing 38418 call stack information. This data is considered substantial and provides sufficient data for conducting statistical performance debugging analysis. The timestamps of the system call entries and exits were recorded along with the call stack data. Unique functions from the call stacks during system call entries were recorded, and the latencies were computed by determining the difference between the timestamps of the respective system call entries and exits.

2) Statistical Debugging and Function Prioritization: After collecting the trace events, the performance metric is defined, which in this case is the system call duration, calculated from the timestamp difference between entry and exit of the respective system calls. Based on this metric and the call stack data provided by Perf, the failure or success label is assigned to the call stack. In this case study, 1316 unique functions are identified. For each function, statistical debugging metrics

[1]https://github.com/ak19qp/SCAM2023

TABLE I
CASE STUDY 1 CULPRIT FUNCTIONS

Functions	Failure	Context	Increase
...	...	...	...
getifaddrs_internal	0.024	0.008	0.016
start_thread	0.002	0.002	0
pthread_cond_signal@@GLIBC_2.3.2	0.504	0.504	0
...	...	...	...
mozilla :: layers :: WebRenderCommandBuilder :: PushItemAsImage	0	1	-1
mozilla :: layers :: WebRenderCommandBuilder :: CreateWebRenderCommandsFromDisplayList	0	1	-1
mozilla :: layers :: WebRenderCommandBuilder :: BuildWebRenderCommands	0	1	-1
mozilla :: layers :: WebRenderLayerManager :: EndTransactionWithoutLayer	0	1	-1
...	...	...	...

(Failure, Context, and Increase) are calculated. The Increase value of the functions are then used to sort and rank the functions. A cut-off value is then applied, disregarding anything below the top 15% of the list. This reveals the functions that are more directly implicated in failures, with the most significant ones appearing at the very top of the list. Table I highlights the culprit functions responsible for the performance bugs found in the ranked cut-off list.

3) Discussion: Through our case study, we have recognized the substantial value of applying statistical debugging techniques, specifically using runtime data of kernel events such as system call waiting time, to uncover critical functions that impact the performance of CSS animation rendering in Firefox. Table I presents the results of the proposed PASD, which identifies three crucial Firefox functions in the **WebRenderCommandBuilder** class, matching the findings reported in Bugzilla [18]. This external validation enhances the credibility and reliability of our findings, ensuring internal consistency.

It is important to note that in statistical debugging, 'Increase' values can occasionally be negative, as demonstrated in Table I. However, we do not exclude functions that exhibit a negative 'Increase' value, since this metric is only employed for the purpose of ranking potential functions, not for any other analytical process.

Given that our analysis is based solely on runtime function calls and does not have access to the code branches (as defined by original definitions of predicates), it is feasible to observe negative 'Increase' values, even for functions that are the root cause of issues. this arises due to two main reasons outlined below:

- *Interactions with other predicates:* The negative 'Increase' value of a specific predicate may be influenced by interactions with other predicates or conditions in the code.
- *Hidden dependencies:* Certain predicates may have hidden dependencies or indirect effects on failures. These dependencies may not be immediately apparent and can result in a negative 'Increase' value.

Therefore, irrespective of the 'Increase' value being positive, zero, or negative, we do not directly use it to discard candidate functions. Instead, we employ it to only sort and rank the list of candidate functions.

B. Case Study 2: Firefox Tripadvisor CPU Exhaustion Bug

In this case study, we analyze a performance bug reported in Bugzilla in 2019 [19], which pertains to an extraordinary elevation in CPU usage while loading the website *tripadvisor.ca* in the Firefox browser. The report indicates that the main thread monopolizes CPU resources for a protracted duration. In our experimental environment, designed to replicate this bug, we experienced system-wide crashes on multiple occasions, attributable to resource exhaustion.

1) Study Setup: In an approach similar to the first case study, the experimental setup entailed compiling Firefox from the source with the 'Perf' tag enabled. 'Perf' was employed to capture call stack data associated with system call events hooked to the Firefox PID. The website *tripadvisor.ca* was loaded recurrently to accumulate sufficient call stack data.

The call stack data collected in this study were analyzed to identify unique function names and their associated wait times for different system calls. The task of the experiment was carried out multiple times over 110 seconds. A total amount of 1.23 GB of data containing 398297 call stacks was collected, sufficient to deliver meaningful insights into the wait time of unique functions during system calls.

2) Statistical Debugging and Function Prioritization: Implementing statistical debugging on the collated data for individual functions, and then ranking them by the 'Increase' value followed by a cut-off of 15% of the list, reveals three significant function calls across multiple addresses in the list. This implies their execution by distinct threads. These functions are "gethostbyaddr_r@@GLIBC_2.2.5", "pthread_cond_signal@@GLIBC_2.3.2", and "getifaddrs_internal".

The function gethostbyaddr_r is entrusted with reverse Domain Name System (DNS) resolution and is a thread-safe variant of the gethostbyaddr function. The function pthread_cond_signal, an integral part of the POSIX threads (pthreads) library, provides a parallel programming interface grounded in threads for Unix-like operating systems. This function is employed for thread synchronization, facilitating threads to await the fulfillment of a specific condition. Lastly, the getifaddrs_internal function retrieves data regarding the network interfaces available on the system, such as their IP addresses, network masks, and interface names. It allocates memory for a linked list of struct ifaddrs structures, each encapsulating information about an interface.

When the functions are sorted based on the highest standard deviation and mean, "std::panicking::try" and "std::panic::catch_unwind" appear prominently on the list. Along with these, multiple function calls from the class "mozilla::net::nsSocketTransportService", responsible for network layer communications, are also found towards the top of the list. This analysis provides insights into the functions potentially contributing to the observed performance issues.

TABLE II
CASE STUDY 2 CULPRIT FUNCTIONS

Functions	Failure	Context	Increase
...	...	...	...
gethostbyaddr_r@@GLIBC_2.2.5	1	0.3963	0.6037
pthread_cond_signal@@GLIBC_2.3.2	0.6	0.6	0.0
...	...	...	...
mozilla :: net ::nsSocketTransportService :: DoPollIteration	0	0.1806	-0.1806
mozilla :: net :: nsSocketTransportService :: OnDispatchedEvent	0	1	-1
...	...	...	...

3) Discussion: Upon meticulous examination of our results, it becomes apparent that the performance issue is intrinsically tied to network layer communication. The website *tripadvisor.ca* operates with a substantial assortment of interconnected websites and URLs, which it references to populate its content.

Table II shows the culprit functions from the sorted ranked list of candidate functions. Our analysis, derived from statistical debugging, indicates that the parent thread commissions numerous child threads to retrieve various data from different URLs during the loading of tripadvisor.ca. Given that pthread_cond_signal exhibits an extended system call wait time concurrent with the network-related functions, it can be inferred that a failure occurs at the code level. This inference is supported by the activation of std::panicking::try and std::panic::catch_unwind.

We postulate that, amidst this failure, the catch block perpetually generates new threads to reattempt the same task. It is during this period that the CPU usage escalates dramatically. This relentless cycle of creating and terminating threads to retry the same task persists for an extended duration causing performance bottleneck and CPU resource exhaustion.

Our analysis gains further validation from the bug report, which attributes the occurrence of this issue to a dependency on off-main-thread (OMT) networking. The report further indicates an extensive array of processes transpiring on the main thread, with a majority triggered by initiating network requests. This correlation corroborates the findings from our method, bolstering our analysis that network-related functions play a significant role in this performance issue.

C. Case Study 3: ”ls” Slow Performance in Large Directories

In this case study, we investigated a performance bug that arises from the usage of the 'ls' command in Linux, which is a component of the GNU Core Utilities package [26]. The 'ls' command is a commonly used command-line utility in Unix-like operating systems. Its purpose is to display a list of files and directories within a specified directory or the current working directory. When executed without any arguments, 'ls' will present the contents of the current directory, providing a concise listing that includes the names of files and directories.

Reports on various online discussion platforms [27], [28] as well as RedHat BugZilla reports [20], [21] have indicated that 'ls' exhibits significant slowness when dealing with directories containing a large number of files and folders. Furthermore, it is also slow when handling a large number of top-level entries.

TABLE III
CASE STUDY 3 CULPRIT FUNCTIONS

Functions	Failure	Context	Increase
__GI___statfs	1	0.5	0.5
...	...	...	...
fstatat64	1	0.909	0.091
read	0.168	0.119	0.049
...	...	...	...
get_link_name	0	0.033	-0.033
do_lstat	0	0.112	-0.112
...	...	...	...
print_color_indicator	0	0.86	-0.86
...	...	...	...

This issue specifically arises when color coding is enabled for the 'ls' command.

1) Study Setup: For this case study, we began by installing the libc6-dbg package using the command 'sudo apt-get install libc6-dbg'. This ensured that the libc compiler had debugger information available for the compiled program. We also installed the build-essential package in the same manner to facilitate the building process. Subsequently, we downloaded the source code of the GNU Core Utilities package and proceeded to extract and compile it with debugger information enabled.

Once the necessary preparations were complete, we utilized a bash script to generate a substantial number of random files and folders, along with numerous top-level directories. We employed the 'Perf' tool to execute the 'ls' command and record call stacks whenever system calls were made by 'ls'. The command used for this purpose was ”perf record -g -e 'syscalls:sys_*' /[coreutils directory]/src/ls –color=always /[directory to inquire]”. Additionally, we also repeated the call stack recording process with Perf while disabling the color parameter of 'ls' and collect 276,412 call stack data. To obtain a comprehensive dataset for statistical analysis, we executed 'ls' through Perf multiple times under various loads and stress conditions.

2) Statistical Debugging and Function Prioritization: The proposed PASD approach analyzes the collected call stack data to identify unique functions. The unique functions were then sorted based on their 'Increase' value and ranked accordingly. Table III illustrates the culprit functions identified from the sorted ranked list. The results of this case study emphasize the significance of not disregarding functions with negative 'Increase' values. While __GI___statfs is not directly part of the ls class's code, it is the top-ranked function in the list. On the other hand, do_lstat is a direct function from the ls class and exhibits a negative 'Increase' value. It is worth noting that ls executes statfs through do_lstat, and the issue lies not in statfs itself, but rather in how do_lstat calls statfs.

3) Discussion: Upon analyzing the results from the ranked list of functions, we can observe that __GI___statfs appears as the top candidate. This finding is consistent with the information provided in the answer from [28], which states that individual statfs calls are made to gather information about the type of file, permissions, and file capabilities for the purpose of setting colors accordingly. This aligns with our observations

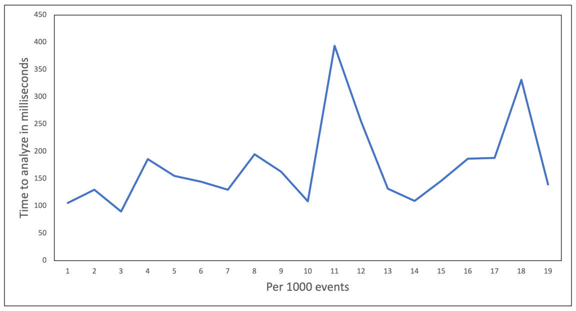


Fig. 5. Overhead: Case Study 1

from the ranked list.

Furthermore, upon conducting further analysis, we discovered two significant functions: do_lstat and print_color_indicator, as indicated in Table III. Upon reviewing the ls source code, we found that do_lstat is the function responsible for calling the statfs function and determining whether color codings are required, utilizing the print_color_indicator function. When color coding is enabled, do_lstat needs to retrieve additional metadata related to each files and folders, which ultimately contributes to the performance problem. This discovery identifies the actual cause of the performance issue.

The findings from [20], [27] further validates our method, as they also mentioned that disabling color coding for 'ls' significantly improves its performance which aligns with our method's findings.

D. Computational Overhead

The computational overhead imposed by the method proposed in this study hinges largely on the resources employed by the 'Perf' tool for the surveillance and recording of call stack data. Importantly, as the process of statistical debugging is carried out offline, it is devoid of any imposition of overhead on the production system during its active run-time operation.

In the monitoring phase of the call stacks, 'Perf' is employed, resulting in a marginal escalation in CPU utilization overhead. Figure 5, Figure 6 and Figure 7 show the overhead for Case Study 1, 2, and 3 respectively while analyzing the events offline. For Case Study 1, it took an average of 173 milliseconds to analyze 1000 events, with the peak being at 393 milliseconds. For Case Study 2, the average was 330 milliseconds to analyze per 1000 events, with the peak being 2499 milliseconds. Finally, for Case Study 3, the average was 54 milliseconds to analyze per 1000 events, with the peak being 206 milliseconds.

Given that data collection takes only a fraction of the total CPU utilization, it can be considered remarkably efficient. This, coupled with the benefit of the method not interfering with live operations, further underscores the applicability of our approach in a production environment. Even in scenarios of extensive surveillance and analysis, our method remains unobtrusive and resource-conservative, making it an effective tool for production systems seeking to diagnose and address

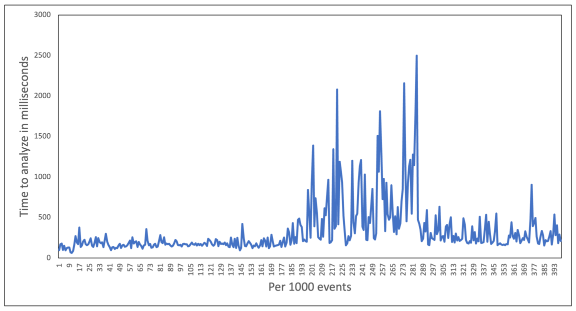


Fig. 6. Overhead: Case Study 2

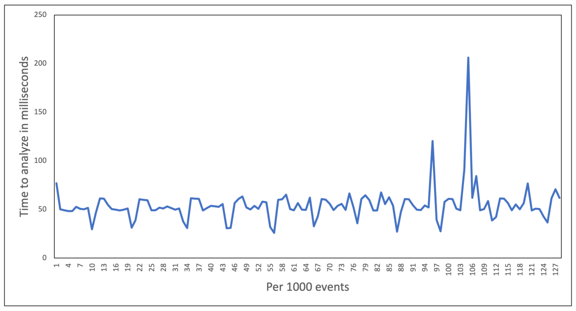


Fig. 7. Overhead: Case Study 3

performance issues with minimal disturbance to their operational efficiency.

V. Related works

Performance debugging in large-scale software systems presents a significant challenge, as underscored by numerous studies [4]–[6], [9], [29], [30]. These studies explore a range of methods to tackle this issue, such as the use of call stack traces, mining techniques, extraction of performance metrics from trace data, and the application of statistical debugging.

Han *et al.* [4] introduced StackMine, a novel approach that uses call stack traces and mining techniques to identify impactful performance bugs. However, this approach does not provide specific function names associated with the bugs.

Ezzati-Jivan *et al.* [9] proposed a solution for identifying performance degradation root causes in distributed systems by extracting performance metrics from trace data. However, it lacks function-level information for pinpointing specific functions related to the detected bugs.

Guoliang Li *et al.* [6] conducted a comprehensive study highlighting the significant impact of inefficient code sequences on performance degradation and resource waste. This work, based on static analysis, is effective in detecting potential performance issues but lacks the ability to capture the dynamic behavior of a system during runtime.

Performance analysis using statistical debugging approaches has been a topic of extensive research over the past decade [5], [29], [30]. Song and Lu [5] demonstrated the effectiveness of statistical debugging in diagnosing performance issues. However, their approach, primarily based on static analysis, falls short in accounting for performance issues that arise during runtime under varying workloads and different contextual situations. This limitation stems from the inability of static analysis to monitor and understand the system's real-time operations, leading to potential missed bugs that only manifest under specific runtime conditions.

Zhiqiang Zuo has made significant contributions to the field of statistical debugging with two notable works [29], [30]. In his 2014 paper [29], Zuo introduced a hierarchical instrumentation technique that allows for selective instrumentation, thereby enhancing the efficiency of debugging while maintaining its effectiveness. This approach was applied to both in-house and cooperative debugging, demonstrating significant improvements in debugging efficiency. However, it did not specifically address the dynamic behavior of systems during runtime or provide function-level information for debugging.

In a later work in 2023 [30], Zuo and his colleagues introduced a systematic pruning technique for statistical debugging that requires only partial instrumentation, thereby reducing the overhead. Despite this improvement, their approach still did not consider the dynamic runtime behavior of systems or pinpoint the functions causing these issues.

These studies highlight a common limitation across existing solutions: the lack of function-level information, insufficient consideration for dynamic behavior during runtime, and the reliance on costly instrumentation. Our proposed PASD method aims to address these shortcomings by integrating statistical debugging with kernel-level event data, offering a more effective and targeted approach for performance debugging at the function level, without the need for extensive instrumentation.

VI. Conclusion and Future Work

Performance debugging poses significant challenges due to the elusive nature of performance bugs and the potential interference caused by traditional debuggers. To address these challenges, this paper introduced PASD as a novel approach for Performance Analysis through Statistical Debugging of kernel events. PASD combines the power of kernel event tracing and statistical debugging techniques to effectively diagnose performance-related issues. The PASD framework operates by collecting a selected list of kernel trace events, extracting relevant performance metrics, analyzing these metrics using statistical debugging methods, and ultimately ranking the functions associated with the identified performance issues. A key aspect of PASD revolves around the identification of function invocations that hinder performance by analyzing different metrics defined over kernel-level events. These metrics serve as reliable indicators of operational performance, enabling PASD to accurately pinpoint performance-inhibiting function invocations.

Through rigorous and comprehensive analysis conducted on Firefox, a multi-threaded browser application, and 'ls' of the GNU Core Utilities library package, we have provided empirical evidence of the effectiveness of our approach. Three distinct case studies, each highlighting different performance issues, further underscored the ability of our method to identify function invocations that contribute to performance degradation. This was accomplished by analyzing the latency of unique functions during system calls and utilizing statistical debugging techniques to identify the functions that correlated with performance failures.

Looking forward, our plans for future work are focused on evolving and enhancing the capabilities of our method. This involves refining our approach to minimize the need for manual intervention and increasing the level of automation, thereby improving efficiency and user-friendliness. Additionally, we intend to explore the potential of utilizing machine learning models to predict system performance and dynamically configure the tracing level. This proactive and preventive approach to system performance management holds promise for more effective and efficient performance analysis.

Additionally, we purpose extending the applicability of our method to include a broader range of software systems. This encompasses database management systems, server applications, parallel processing system and other complex software systems where performance is critical.

References

[1] P.-F. Denys, Q. Fournier, and M. R. Dagenais, "Distributed computation of the critical path from execution traces," *Software: Practice and Experience*, 2023.

[2] N. Statt, "Amazon's website crashed as soon as Prime Day began," 7 2018. [Online]. Available: https://www.theverge.com/2018/7/16/17577654/amazon-prime-day-website-down-deals-service-disruption

[3] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "On debugging the performance of configurable software systems: Developer needs and tailored tool support," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1571–1583.

[4] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 145–155.

[5] L. Song and S. Lu, "Statistical debugging for real-world performance problems," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 561–578, 2014.

[6] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 77–88. [Online]. Available: https://doi.org/10.1145/2254064.2254075

[7] K. Meng and B. Norris, "Mira: A framework for static performance analysis," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 103–113.

[8] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006, pp. 84–88.

[9] N. Ezzati-Jivan, H. Daoud, and M. R. Dagenais, "Debugging of performance degradation in distributed requests handling using multilevel trace analysis," *Wireless Communications and Mobile Computing*, vol. 2021, pp. 1–17, 2021.

[10] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 156–166.

[11] I. Kohyarnejadfard, D. Aloise, M. R. Dagenais, and M. Shakeri, "A framework for detecting system performance anomalies using tracing data analysis," *Entropy*, vol. 23, no. 8, p. 1011, 2021.

[12] J. Rhee, H. Zhang, N. Arora, G. Jiang, and K. Yoshihira, "Software system performance debugging with kernel events feature guidance," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–5.

[13] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang, "Combining spectrum-based fault localization and statistical debugging: An empirical study," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 502–514.

[14] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, ser. O'Reilly Series. O'Reilly, 2002. [Online]. Available: https://books.google.ca/books?id=9yIEji1UheIC

[15] N. Ezzati-Jivan and M. R. Dagenais, "A stateful approach to generate synthetic events from kernel traces," *Advances in Software Engineering*, vol. 2012, pp. 6–6, 2012.

[16] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," 6 2005. [Online]. Available: https://doi.org/10.1145/1065010.1065014

[17] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 1105–1112. [Online]. Available: https://doi.org/10.1145/1143844.1143983

[18] J. Muizelaar, "Bug 1637586," 5 2020. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1637586

[19] M. Stange, "1565019 - High CPU usage in parent process main thread when loading tripadvisor.ca, for over 700ms," 7 2019. [Online]. Available: 'https://bugzilla.mozilla.org/show_bug.cgi?id=1565019'

[20] M. Hergaarden, "1290036 – [RFE] Slow 'ls' performance and directory listing within Samba," 12 2015. [Online]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=1290036

[21] J. Antill, "467508 – ls slower, due to capabilities," 10 2008. [Online]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=467508

[22] I. SuperHi, "Looping squares." [Online]. Available: https://looping-squares.superhi.com/

[23] Mozilla, "Jit profiling with perf — firefox source docs documentation." [Online]. Available: https://firefox-source-docs.mozilla.org/performance/jit_profiling_with_perf.html

[24] "Perf: Linux profiling and performance monitoring," https://perf.wiki.kernel.org/, accessed on 20-06-2023.

[25] J. Jones and E. Smith, "Profiling and optimization techniques for just-in-time compilation," in *Proceedings of the International Conference on Compilation Techniques*, 2022, pp. 123–135.

[26] Coreutils, "GitHub - coreutils/coreutils: upstream mirror." [Online]. Available: https://github.com/coreutils/coreutils

[27] "Why might 'ls –color=always' be slow for a small directory?" [Online]. Available: https://serverfault.com/questions/316951/why-might-ls-color-always-be-slow-for-a-small-directory

[28] "'ls' command very slow." [Online]. Available: https://superuser.com/questions/1345268/ls-command-very-slow

[29] Z. Zuo, "Efficient statistical debugging via hierarchical instrumentation," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 457–460.

[30] Z. Zuo, X. Niu, S. Zhang, L. Fang, S. C. Khoo, S. Lu, C. Sun, and G. H. Xu, "Toward more efficient statistical debugging with abstraction refinement," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–38, 2023.