Characterizing In-Kernel Observability of Latency-Sensitive Request-level Metrics with eBPF

Mohammadreza Rezvani*, Ali Jahanshahi†, Daniel Wong‡
*†Department of Computer Science and Engineering, ‡Department of Electrical and Computer Engineering
University of California, Riverside
Riverside, CA, USA
Email: *mrezv002@ucr.edu, †ajaha004@ucr.edu, ‡danwong@ucr.edu

***Abstract*—This paper explores a novel server observability approach using eBPF (extended Berkeley Packet Filter) for detailed request-level performance metrics of data center latency-sensitive applications. Utilizing eBPF system call tracing, we evaluate if syscall activity can reconstruct high-level application behaviors and bypass the need for direct userspace reporting of performance metrics. Through careful selection of eBPF events, we demonstrate that certain syscall statistics can provide robust insight into request-level metrics. In addition, we demonstrate that these metrics can also be robust to networking effects, such as packet loss. By demonstrating the ability for eBPF to provide request-level observability, we can potentially enable many non-intrusive, low-overhead use cases for feedback in system management runtime frameworks, such as resource allocation, scheduling, and power management.**

***Index Terms*—eBPF, performance, latency-sensitive**

I. Introduction

Modern data centers are characterized by dynamic and demanding environments, necessitating efficient management for a multitude of hosted applications. Achieving this efficiency is particularly challenging due to the lack of portable and non-invasive methods for capturing critical application-level metrics. To provide greater insight into the complex behavior of modern data centers, non-invasive tools for observability have become increasingly important for data center management and optimization. Observability tools aim to collect various telemetries and measurements of various data center components, such as metrics (bandwidth utilization, memory utilization, etc.), logs, or distributed traces of applications.

The emergence of eBPF (extended Berkeley Packet Filter) [1] has revolutionized observability and monitoring tools in the Linux ecosystem. eBPF allows the execution of custom programs in a secure, in-kernel virtual machine, thus providing a unique vantage point for monitoring system-level activities. Many observability tools have been built with eBPF to provide observability for security, networking activity, container activity, and monitoring of distributed infrastructures, such as Kubernetes. eBPF is able to transparently obtain various performance metrics and tracing mainly through analyzing network traffic activity and network protocol requests. Due to their approach of analyzing network-based activity, eBPF observability tools mainly target infrastructure and workloads in distributed or cloud-based environments. However, this powerful tool has not been explored for observability into more traditional server systems which would equally benefit from non-invasive observability tools.

Non-invasive observability tools for application instrumentation can have several major benefits. For example, many system management frameworks that control resource allocation, power management, or workload scheduling require direct feedback of the application's performance metrics which are typically reported by the application itself. However, this direct feedback requirement can introduce several limitations to many management frameworks. For example, power management frameworks, such as DVFS or sleep-states are carried out by drivers in the kernel. Many prior works that proposed advanced DVFS and/or sleep-state management [2]–[5] assume the availability of request-level metrics directly reporting to the power management runtime. However, in practice, it would be impractical to provide timely request-level inquiries to the driver as passing user-space application-level information to the driver would require significant overhead. Similarly, resource allocation management runtimes also rely on application-level feedback to guide various resource allocation techniques, such as core allocation and cache partitioning [6]–[8]. Thus, providing non-intrusive observability for applications to kernel-space management runtimes would enable significant new opportunities for practical implementations of advanced kernel-space management runtimes.

Currently, observability into application workloads running on a server system typically relies on traditional performance counters provided by the underlying server hardware. While existing eBPF tools effectively measure distributed workload metrics and traditional performance counters gauge underlying server hardware, a significant gap remains in capturing application-specific metrics needed by many system management runtimes. Applications typically fall into two categories: compute-intensive batch applications and user-facing latency-sensitive applications. For the former, traditional hardware performance counters like Instructions Per Cycle (IPC) effectively reflect performance. However, for latency-sensitive applications, the need for request-level metrics such as request per second (RPS) and request latency presents a unique challenge. Hardware metrics lack visibility into request boundaries, and there is often a poor correlation between these performance counter metrics and key performance indicators like RPS or tail latency.

In this paper, we characterize the ability of using eBPF to provide observability into application-level metrics of user-facing, latency-sensitive workloads. Our research specifically addresses the intricacies of user-facing, latency-sensitive services, which are governed by request-based metrics such as requests per second, latency, and tail latency percentiles, which are difficult to capture using traditional hardware performance counters. These services are inherently dependent on the request-response cycle, where the client's perceived performance is directly linked to server-side application behavior. Rather than monitoring network traffic activity as in existing eBPF observability tools, we aim to explore if eBPF can use the characterization of syscall activity to enable observability of application-level metrics.

This paper evaluates a novel observability methodology using eBPF for kernel-level tracing, thereby enabling the non-invasive capture of application-level metrics critical for optimizing the performance of latency-sensitive services. Significantly, no previous study has extensively explored the potential of syscall traces to yield meaningful information for performance optimization in such applications. Our methodology focuses on leveraging kernel-level activity to infer request metrics non-invasively, a critical advancement in understanding and optimizing user-facing application workloads. The key contributions of our research are:

- We present an eBPF-based approach for in-kernel observability of request-level metrics.
- We thoroughly evaluate the feasibility of utilizing eBPF system call activity for the observability of application-level metrics using a wide range of latency-sensitive workloads.
- We identify that eBPF-monitored syscall activity can provide strong observability of throughput and system saturation metrics.
- We discuss the implications and potential of enabling in-kernel observability of application-level metrics of latency-sensitive workloads.

In Section II we delve into related work, discussing the limitations of existing methods in estimating request-level metrics. Section III, explores the potential of using eBPF for monitoring critical metrics in latency-sensitive applications. The subsequent sections build upon this foundation, presenting a case study (Section IV), discussing challenges (Section V and concluding with the study's implications and potential for future research in application performance optimization (Section VI).

II. Related Work

We summarize below the related works that aim to estimate request-level metrics and their limitations.

Performance counters. The kernel already presents a rich set of hardware and software performance counters, such as utilization, IPC, cache misses, stalls, etc. These metrics have been demonstrated to have poor correlation with request-level metrics [9]–[13]. Due to their poor ability to accurately predict request-level metrics, performance counters present limitations for many system management runtimes. While performance metrics may be correlated to throughput, they are ineffective during QoS violations and latency-based metrics. For example, Seer [13] argues that utilization metrics have limitations in predicting QoS violation.

Queue pressure. Requests in user-facing workloads tend to flow through a series of queues during processing. For example, networking queues and potentially application-level request queues, such as between application stages in Memcached workloads [13]. Therefore, request-level metrics may be reflected in the queue pressure. Prior works, such as Seer [13], identified that queue depth is a strong indicator of predicting QoS violation. However, the major limitation here is that the networking queues and various software queues have to be instrumented, and that may require knowledge of the queuing structures in the program.

eBPF applications. eBPF can be a very efficient tool for monitoring the system behavior and detecting failures [14]–[16]. Aside from monitoring and observability applications, eBPF has been used as a lightweight virtual machine inside the kernel to run programs with low overhead [17], [18] and to provide light-weight serverless functions [19].

System Calls as a tool and a threat. Syscalls can provide a great deal of information regarding the behavior of an application. In this paper, we explore the use of syscalls to extract valuable performance information. However, syscall activity can also be used in malicious ways. Some studies aimed to reduce the attack surface in the operating system caused by system calls [20]–[23]; whereas other studies used modern machine learning approaches to detect anomalous activities [24]. As well as this, studies have been conducted to enhance the security of system calls by modifying the operating system architecture [25]. While security is a major issue, in this work, we mainly focus on the feasibility of utilizing syscall activity to characterize request-level metrics.

III. A Case for Request-level Observability with eBPF and Syscall Activity

We aim to explore the feasibility of utilizing eBPF for the observability of essential request-level metrics like request processing rate and latency. In this study, we examine latency-sensitive applications within cloud and data center environments, leveraging eBPF (extended Berkeley Packet Filter) for capturing and analyzing system call activities. The strength of our methodology lies in the non-invasive nature of eBPF, which allows us to collect in-depth observability without altering the normal operation of the application or causing significant overhead.

A. Monitoring with eBPF

eBPF revolutionizes kernel capability extension in Linux systems, allowing sandboxed programs to run within the kernel without altering its source code or adding kernel modules. This feature, with its remarkably low overhead demonstrated in prior works [14], [19], [26], has gained widespread adoption in industry applications [1]. Linux's extended Berkeley

Packet Filter (eBPF) feature, which has undergone extensive improvements particularly in the Linux Kernel versions 3.15+ and 4.15+, enables developers to run small, static programs attached to kernel functions (kprobes), kernel tracepoints, or userspace functions (uprobes), thus bringing eBPF to the forefront of kernel tracing and metric collection. Kprobes and tracepoints, integral to eBPF, have been part of the Linux kernel since the early 2000s, but recent advances have made writing more complex programs easier and more practical.

eBPF programs can be compiled by compilers like GCC and LLVM from classic C into bytecode, which is then injected into the kernel, ensuring that generated programs pass eBPF verification before being loaded. This verification enforces strict constraints such as fixed stack size, reduced instruction set, and prohibition of floating-point arithmetic and loops to ensure programs are verifiable in time and correctness, preventing kernel crashes or slowdowns. Importantly, eBPF supports shared data structures between user and kernel space, allowing for the exchange of information between programs and user processes. eBPF programs trigger upon specific system events like system calls or network events, executing attached instructions at predefined kernel hooks accessible through tools like BCC [27] and bpftrace [28]. This allows monitoring of various system aspects, including system calls, network activities, and other system events.

The effectiveness of eBPF as an interface for system-level tracing and monitoring lies in its ability to embed custom code at crucial kernel points, particularly at system call entry and exit points for this study. This enables comprehensive data collection, including process identifiers, system call types, timestamps, and other relevant metadata, thereby offering an in-depth view of application behaviors and their interactions with the system. eBPF's dynamic bytecode insertion into the Linux kernel, activated by specific events, positions it as a valuable tool across multiple domains, enhancing security, monitoring, and performance optimization without the limitations of traditional monitoring methods. eBPF has found applications in a wide range of areas, including TCP-Tuning, L-4 load balancing, and DDOS protection at Facebook [29], [30], and more broadly in cloud computing for security [31], [32], network optimization [33], [34], virtualization [35], [36], and monitoring [37]–[39], highlighting its versatility and effectiveness.

B. Motivating the Observability of Request-Level Metrics with System Calls

Most user-facing workloads are request-response workloads where servers process incoming requests from clients and then send a response back. These workloads interact with various sockets and queues through system calls (syscalls). Therefore, *our intuition* is that these system call activities can indirectly capture request activity intensity which can be used to observe request-level metrics. For example, as server load increases, more syscall activity would occur. Also, if QoS violation is occurring, our intuition is that there are more irregular activity patterns in syscalls which in turn will lead to more variance in the request latency, and that would similarly reflect in the syscall's timing properties.

```
PID_TGID: pid and tgid of the targeted application
// Hash map for looking up entry timestamp of each pid-tgid
BPF_HASH(start, u64, u64);
// Executed at the start of every syscall
TRACEPOINT_PROBE(raw_syscalls, sys_enter) {
  // Get pid_tgid of the application calling this syscall
  u64 pid_tgid = bpf_get_current_pid_tgid();
  if(pid_tgid != PID_TGID) return 0;  // Filter application
  if(args->id != 232) return 0;        // Filter epoll_wait
  u64 t = bpf_ktime_get_ns();            // Entry timestamp
  start.update(&pid_tgid, &t);               // Store start
  return 0;
}
// Executed at the exit of every syscall
TRACEPOINT_PROBE(raw_syscalls, sys_exit) {
  u64 pid_tgid = bpf_get_current_pid_tgid();
  if(pid_tgid != PID_TGID) return 0;
  if(args->id != 232) return 0;
  u64 start_ns = start.lookup(&pid_tgid);// Retrieve entry
  u64 end_ns = bpf_ktime_get_ns();         // Exit timestamp
  u64 duration = end_ns - start_ns;     // Latest duration
  /* Update metrics or stream data */
  return 0;
}
```

Listing 1. Example of eBPF probe that calculates the duration of system call `epoll_wait` with the id of 232. Built-in types/functions indicated in **bold**.

Collecting eBPF Events. In this paper, we collect and analyze eBPF event traces to evaluate the observability potential for request-level metrics. Initially, we streamed all available eBPF trace data to user space to explore potential correlations with request-level metrics. Subsequently, we leveraged eBPF capabilities to compute these metrics directly within the eBPF space in real-time.

In Listing 1, we show an example eBPF program snippet that measures the duration of `epoll_wait` following the syscall's completion. In section IV-C, we discuss the significance of this metric in accurately estimating request-level metrics. This eBPF program snippet places a probe at the beginning and end of a syscall event, filtering for the desired application and syscall, and records the timestamp for the event.

Request-oriented Syscalls. Due to the variety of system calls and their different applications across software, analyzing raw traces directly can be challenging and sometimes misleading without a thorough understanding of the application's architecture. Intuitively, the syscalls that we monitor should capture request-level behavior. For instance, in Figure 1(b), a variety of syscalls are observed during the setup and shutdown phases. However, these syscalls do not provide substantial information about the application's behavior during the active request processing phase. When examining request-response workloads, it is reasonable to view these tasks as black-boxes that interface with various sockets/file descriptors, such as networking and various libraries such as libevent, gRPC, etc. For example, in the case of memcached application, an incoming request is enqueued in a TCP socket, then read into the black-box application through libevent (which heavily uses the epoll syscall), then processed, and finally transmitted over a TCP connection via the send socket. By adopting this abstract perspective, we focus on system calls that interface

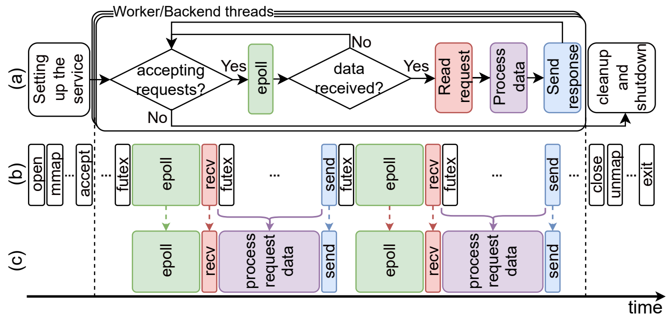

Fig. 1. (a) Example of an application providing a service to clients over network. Worker threads process the clients' request data and send the response (results) back to them. (b) The stream of application's system calls traced by eBPF. (c) Extracted subset of system calls used for observability of request-level metrics.

with networking during request processing. These syscalls manage network communication tasks, making them valuable for analyzing the application's performance and behavior.

In latency-sensitive applications, system calls like `send`, `recv`, and `accept` play crucial roles in managing network communication. Their frequency and functions are key to understanding application performance. For instance, as shown in Figure 1(b), `accept` is typically used in the setup phase, while `recv` and `send` are more prominent during the request processing phase, handling the majority of data transfers and communications. Analyzing these syscalls' occurrence and roles is vital for our study.

Challenges of reconstructing per-request syscall timelines. Our initial aim was to reconstruct the timeline of each request to derive observable metrics. However, this approach is feasible only in simple scenarios where a single thread handles all request-related activities.

In this simple scenario, the thread enters a waiting state for an incoming request, employing the `epoll` syscall. Once `epoll` returns, signaling the availability of a request at the specified file descriptor (typically a socket), the thread then proceeds to read the request using a syscall from the `recv` family. Following the processing of the request (which constitutes the service time), the application responds by sending back data through syscalls such as `send`. Subsequently, the application reverts to waiting for the next request, again utilizing `epoll`, thus establishing a continual cycle of request handling.

In Figure 1(c), a straightforward example of request processing reconstruction is showcased. In cases where a single thread is responsible for handling a request, the `recv` and `send` syscalls corresponding to that request can be paired. This matching allows for an accurate depiction of the application's state and key metrics like service time, providing a clear observability into the application's performance during request handling.

However, this methodology becomes complex with an increase in the number of threads, and almost impractical when requests are distributed across different threads and application components. Essentially, eBPF has no observability into request boundaries and cannot differentiate between syscalls of different requests. Therefore, we shifted our focus to deriving observability from broader syscall statistics.

Identifying System Calls of Interest. In our initial experiments, we realized that syscalls occurring infrequently and responsible for specific tasks – which might not directly correlate with every request processing activity – were not conducive to a generalized solution applicable across all applications. For instance, the `accept` syscall, while indicative of a client-server connection establishment, does not provide clear insights into how or for how long this connection is utilized in the context of application metrics.

Our focus thus shifted to syscalls directly involved in the transmission and reception of requests, pivotal in understanding application behavior. For receiving messages, common syscalls include `read`, `recvmsg`, `recvfrom`, and their variants, while sending messages often involves syscalls like `write`, `sendmsg`, `sendto`, and related functions. Additionally, polling syscalls such as `epoll`, `epoll_wait`, and `select`, typically used to await incoming requests, emerged as critical data sources in our analysis.

Observability Through Syscall Statistics. Dealing with raw syscall traces, even when focused on specific syscalls, presented challenges due to their sheer volume and potential for misleading interpretations. To mitigate this, we extracted higher-level statistics from these traces, enabling us to establish meaningful and explainable correlations between these statistics and the actual metrics of the application.

We found that syscall duration is particularly informative in scenarios where a syscall awaits an event, as observed with `epoll` family of syscalls. In contrast, for syscalls like `send` and `recv`, which perform defined tasks, the frequency of these calls offers better insight into application behavior. Rather than directly calculating call rates, we first measured the intervals between consecutive syscalls, or 'deltas', in a sorted trace. Analyzing these deltas not only aids in determining call rates at any given moment but also allows for a multifaceted analysis, leading to a more comprehensive understanding of the application's behavior and performance.

IV. Case Study

In our case study, we adopt a comprehensive approach to assess the efficacy of utilizing eBPF for the observability of request-level metrics of latency-critical applications in cloud and data center environments. Our analysis centers on gathering detailed system call data, which is then meticulously processed and analyzed. This analysis is not just limited to data collection; our primary goal is to establish realistic relationships between syscall timing statistics and actual application performance metrics, specifically throughput and latency, thus evaluating the observability of eBPF. By focusing on these key areas, we aim to accurately measure and optimize crucial performance indicators such as Requests Per Second (RPS) and tail latency, thereby gaining deeper insights into the application's load handling capability and overall responsiveness.

TABLE I
SYSTEM SPECIFICATION

	AMD	INTEL
CPU Model	AMD EPYC 7302	Intel Xeon CPU E5-2620
OS (Kernel)	Ubuntu 20.04.1 (5.15.0-52-generic)	Red Hat 4.8.5-36 (4.20.13-1.el7.elrepo)
Sockets	2	2
Cores/Socket	16	8
Threads/Core	2	1
Min/Max Frequency	1500/3000 MHz	1200/3000 MHz
L1 Inst/Data Cache	1/1 MB	32/32 KB
L2 Cache	16 MB	256 KB
L3 Cache	256 MB	20 MB
Memory	512 GB	128 GB
Disk	2 TB	2 TB

A. *Experimental Setup*

Our evaluation methodology consists of a diverse set of real-world applications in a controlled environment. This validation process includes comparing our eBPF-observed metrics with reported application-level metrics, where available, to assess the efficacy of request-level observability provided by eBPF. Additionally, we perform robustness checks to ensure that our methodology consistently produces reliable observability results across different workload types and system configurations.

Servers. Our experiments were performed on two high-end servers, an AMD-based server and Intel-based server, shown in Table I. We utilize two different processors to demonstrate that our technique is generalizable across hardware. We observe similar trends across both servers, showing us that as long as eBPF is supported, eBPF observability of request-level metrics will work on any underlying hardware.

Workloads. We thoroughly evaluate our findings across a wide range of latency-critical applications obtained from two distinct benchmark suites and a state-of-the-art ML inference server. This includes five applications from the tailbench latency-critical benchmark suite [40], specifically `img-dnn`, `xapian`, `silo`, `specjbb`, and `moses`. Furthermore, we evaluate the only two latency-critical applications from the CloudSuite benchmarks suite [41], `Data Caching` (Memcached) and `Web Search`. Additionally, we evaluate with the `Triton Inference Server`, an open-source deep-learning inference server developed by NVIDIA [42], [43], which supports both HTTP and gRPC protocols as its inference API, providing us with a robust comparison against different networking protocols and request queueing structure. To guarantee the precision of our results, we run all workloads inside docker containers. However, we purposely placed the docker containers of both the client and server on the same machine to control the quality impact of the network by using Linux's tc-netem tool [44]. The findings regarding these effects are further elaborated and discussed in section V. The RPS at which failures occurred for each benchmark in our high-end AMD server is as follows: `Img-dnn`=1950, `Xapian`=970, `Silo`=2100, `Specjbb`=3700, `Moses`=900, `Data Caching`=62000, `Web Search`= 420, and `Triton`=21 (both with HTTP and GRPC).

All workloads are capable of handling concurrent requests and can process multiple requests with multiple threads. They were selected to exhibit a wide range of request-handling software threading behavior. For example, Data Caching provides a straightforward and simple request-handling threading behavior where each thread consumes and processes a request. Web Search consists of two containers (and thus, two processes), the front-end and index search, where requests enter the front-end container process and are handled by the index search container process. Triton has dedicated threads that consume requests and dispatch them across other threads for processing.

The syscalls used by each application are as follows: in Tailbench, all applications use `recvfrom` and `sendto`, Data Caching employs `read` and `sendmsg`, Web Search utilizes `read` and `write`, Triton with GRPC protocol relies on `recvmsg` and `sendmsg` and lastly Triton with HTTP protocol makes use of `recvfrom` and `sendto`. All applications utilize `epoll_wait` as their polling mechanism; however, Tailbench uses a legacy syscall called `select`.

B. *Throughput Analysis: Observability of RPS*

In the context of user-facing, latency-sensitive workloads, throughput is a critical measure of performance. These workloads typically involve a client-server model where the client sends requests to the server, and the server processes these requests and responds. The efficiency of this interaction is quantified as throughput, measured in requests per second (RPS).

We propose a novel method to approximate RPS by monitoring the `send` syscall rate. This is based on the hypothesis that the frequency of response dispatches from the server (as captured by the `send` syscall) correlates strongly with the application's throughput. The formula for approximating RPS (RPS_{Obsv}) is as follows:

$$RPS_{Obsv} = \frac{r}{t_r^{\text{send}} - t_1^{\text{send}}} = \frac{1}{\overline{\Delta t^{\text{send}}}} \tag{1}$$

Here, r is the total count of `send` syscalls observed, and $\overline{\Delta t^{\text{send}}}$ represents the average interval between these syscalls. Our approach is particularly effective over extended periods (at least 2048 syscalls) where request distribution stabilizes. However, for very short observation windows, variations in request distribution can pose challenges.

Evaluation of Observed RPS. To assess the accuracy of observed Requests Per Second (RPS_{Obsv}), we conducted a comparative analysis with the actual Requests Per Second (RPS_{Real}) reported by the benchmarks. This comparison is illustrated in Figure 2, which includes two types of plots for each workload: a correlation plot and a residual plot.

In the correlation plots, we present the relationship between the normalized values of RPS_{Obsv} and RPS_{Real}. Here, the x-axis indicates the normalized RPS_{Obsv}, while the y-axis shows the normalized RPS_{Real}. These plots are instrumental in visualizing the degree of alignment between our estimated and actual RPS values.

The residual plots, on the other hand, provide insights into the error margin of our RPS_{Obsv} estimations at varying levels of RPS_{Real}. Residuals are calculated by comparing the RPS_{Obsv} against a linear regression model fitted between the two RPS variables. Essentially, a residual represents the difference between the observed RPS_{Real} and the predicted value based on our linear regression model.

Each green dot in the residual plots signifies an individual estimation based on syscall data, with ten such estimations plotted for each actual RPS level. These plots are crucial for understanding the nature of the errors in our estimations, demonstrating whether they are random or biased (consistently overestimating or underestimating).

Our findings reveal a strong positive correlation between RPS_{Obsv} and RPS_{Real} across all workloads. Most of the benchmarks exhibit a coefficient of determination (R^2) greater than 0.94. Notably, `WebSearch` had the lowest coefficient of 0.86, but even in this case, the results were supportive of RPS_{Obsv} being a valid estimate for RPS_{Real}.

The analysis of residuals further underscores the reliability of our estimation method. The scatter of residuals across different RPS levels indicates that the errors in our RPS_{Obsv} estimations are generally random and not systematically skewed in any one direction. This randomness in error suggests that our method of estimating RPS using syscall data is not inherently prone to consistent overestimation or underestimation, reinforcing the validity of RPS_{Obsv} as a proxy for actual RPS in various workloads.

C. Latency Analysis: Tail Latency Failure and Latency Slack

Latency analysis in our study involves two key aspects: identifying tail latency failures and understanding latency slack. Tail latency failure is crucial in evaluating service quality, while latency slack helps in determining the buffer before reaching critical latency thresholds which is used in many resource management tools.

However, this analysis is not without challenges. The primary challenge is that end-to-end latency encompasses client processing time, network transit times, and server processing time (service time). Since system calls do not capture client-side or network transit times, our focus is narrowed down to analyzing the service time within the server. Additionally, the threshold for tail latency failure is subjective and varies from one application to another.

Our strategy, therefore, pivots around identifying server saturation - a state where service times start increasing, leading to higher overall latency and potential tail latency failures. In scenarios where the server is the bottleneck, saturation emerges as a primary cause of latency issues. In such cases, analyzing syscall activities provides sufficient observability for decision-making regarding the server's status and resource management. This method allows for effective monitoring and timely interventions to maintain optimal server performance.

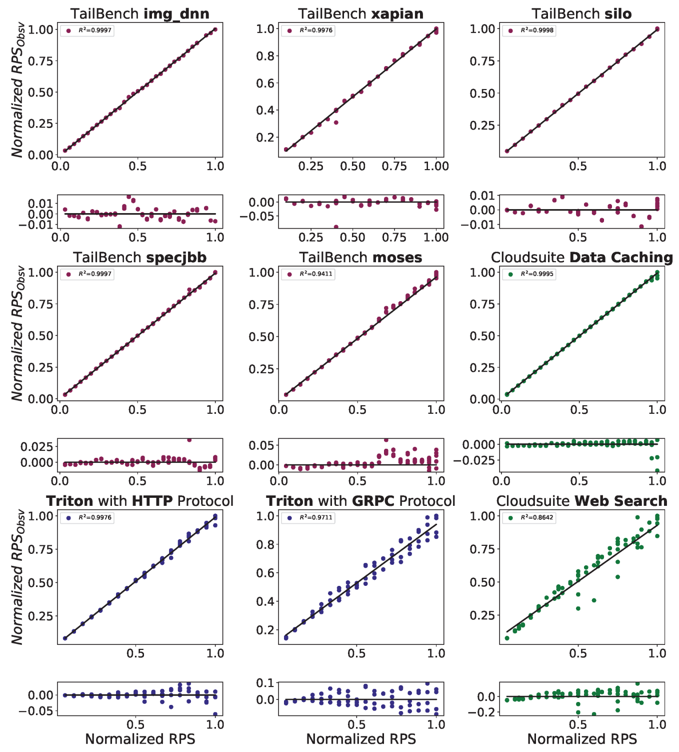

Fig. 2. Comparison of the workload's measured RPS and estimated observed RPS calculated using system calls. A standard linear regression is carried out to better illustrate the correlation of workload's reported RPS. Residual figures are shown to better capture the precision of linear regression. Eq. 1 provides a robust methodology to derive observability of throughput.

1) Observability of Server Saturation: Quality of Service (QoS) violations in data centers can arise from several factors, including request inefficiencies, context switch delays, and system saturation [13], [45]. We focus on the observability of saturation-induced QoS violations, hypothesizing that saturation will manifest as anomalous patterns in syscall timings, specifically for `recv` and `send` syscalls.

We posit that under saturation-based QoS failure, the system will experience some longer than usual delays, which will affect the inter-syscall times captured for `recv` ($\Delta t^{\mathtt{recv}}$) and `send` ($\Delta t^{\mathtt{send}}$) syscalls. Therefore, we monitor the *variance* of $\Delta t^{\mathtt{recv}}$ and $\Delta t^{\mathtt{send}}$. This unexpected increase in the variance of `recv` and `send` inter-syscall times indicates that incoming requests are not processing as normal.

An increase in the variance of inter-syscall times is a strong indicator of deviation from normal operation. We calculate this variance using the following formula:

$$var(\Delta t^{\mathtt{recv}}) = \overline{(\Delta t^{\mathtt{recv}})^2} - (\overline{\Delta t^{\mathtt{recv}}})^2 \quad (2)$$

An unexpected rise in this variance signals potential saturation, indicating that the server is struggling to keep up with incoming requests. This basic form of variance enables the detection of anomalies in the eBPF space within the kernel.

It's important to note that the observed behavior of increased variance as a result of saturation is predominantly seen in complex applications where saturation leads to contention.

In simpler, single-threaded applications, such contention is absent. Requests in these applications queue up and are processed sequentially and won't exhibit significant variance increases. However, this scenario is relatively rare. As detailed in Section III, even in such simple applications, the complete request handling timeline can be reconstructed from syscall timings. In multi-threaded applications, this approach can be extended, with each thread handling a segment of the request-response cycle. The most effective strategy, therefore, is to consider the application as a whole, aggregating all syscalls into a single trace. By analyzing the variance of time intervals (deltas) in this unified trace, we can effectively determine when the application surpasses its saturation point.

Evaluation of Saturation Observability. Figure 3 showcases the normalized variances calculated for the `send` syscall group in our benchmark studies. On this plot, the x-axis denotes the normalized Requests Per Second (RPS), while the y-axis illustrates the normalized variance derived from the Δt^{send} syscall family. It's important to note that different applications utilize various send syscalls, including `send`, `sendmsg`, and others, contributing to the diversity of our analysis.

A critical aspect of this plot is the vertical line representing the point where tail latency surpasses the predetermined Quality of Service (QoS) threshold. This point may not align with the application's saturation point and is defined by the tolerance of the service relied upon by the application. A significant observation from the plot is that the variance tends to increase as the QoS threshold is breached. This increase suggests that the application is struggling to process all incoming requests efficiently. This struggle leads to an accumulation of pending requests, potentially overloading the application's queue management system or causing increased contention among concurrent requests.

By monitoring the changes in variance alongside the trends in RPS_{Obsv}, we can effectively identify abnormal behaviors in the application, particularly those leading to QoS failures under saturation conditions. In the plot, after saturation, we observe a correlation where an increase in RPS_{Obsv} is often accompanied by a rise in variance.

2) Observability of Saturation Slack: Many management runtimes rely on knowledge of how far an application is from reaching its saturation point. Saturation in this context means the application is operating at its maximum capacity, and any additional incoming requests cannot be processed immediately but must wait in the queue. Understanding this saturation slack is vital for the observability of a latency slack.

Initially, we sought to identify specific system call patterns or indicators that could reliably signal approaching saturation. However, this proved challenging as no consistent syscall information or pattern could universally indicate application saturation across different workloads and configurations. This led us to shift our focus from trying to detect saturation directly to identifying signs of application idleness.

The logic behind this approach is that at the point of saturation, an application's idleness is at its minimum, as it is constantly busy processing requests. Conversely, below the saturation threshold, the application experiences periods of waiting for new requests to arrive. This distinction becomes crucial in understanding and measuring the saturation slack.

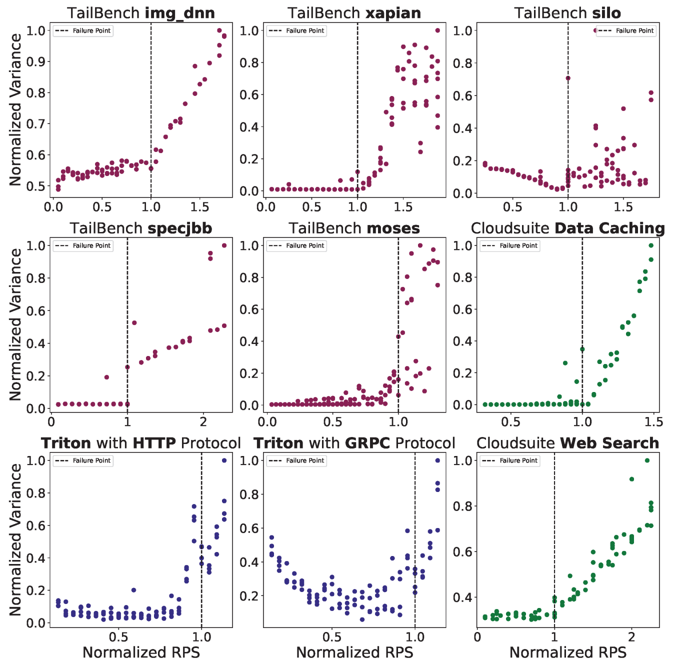

Fig. 3. An illustration of variance for inter-system call timing under varying RPS for system calls responsible for sending the response to the client. The variance increases as the application reaches its saturation therefore, it can be used as a reliable observability indicator for saturation detection.

We discovered that application idleness (a precursor to saturation) can be clearly discerned through syscall activities. Specifically, we focused on the duration of epoll family syscalls, such as `epoll`, `epoll_wait`, and `select`. These syscalls are commonly used in applications to wait for new incoming network events, such as client requests.

By monitoring and analyzing the duration of these epoll syscalls, we found a clear correlation: as the RPS increases, the idle time, as indicated by the duration of `epoll` syscalls, decreases. This inverse relationship provides a reliable measure of the application's idleness and, by extension, its proximity to saturation.

For instance, longer duration in `epoll` syscalls suggest lower request rates, indicating that the application operates well below its capacity. In contrast, shorter `epoll` syscall duration implies higher request rates, signaling that the application is nearing its saturation point. By quantifying this relationship, we can effectively gauge the saturation slack of the application.

Evaluation of Saturation Slack Observability. Figure 4 presents an analysis of how the duration of the `epoll` syscall group correlates with the measured Requests Per Second (RPS) in various applications. It's important to note that while most modern applications predominantly use the `epoll_wait` syscall, some older applications, like `Tailbench` included in our study, still rely on the `select`

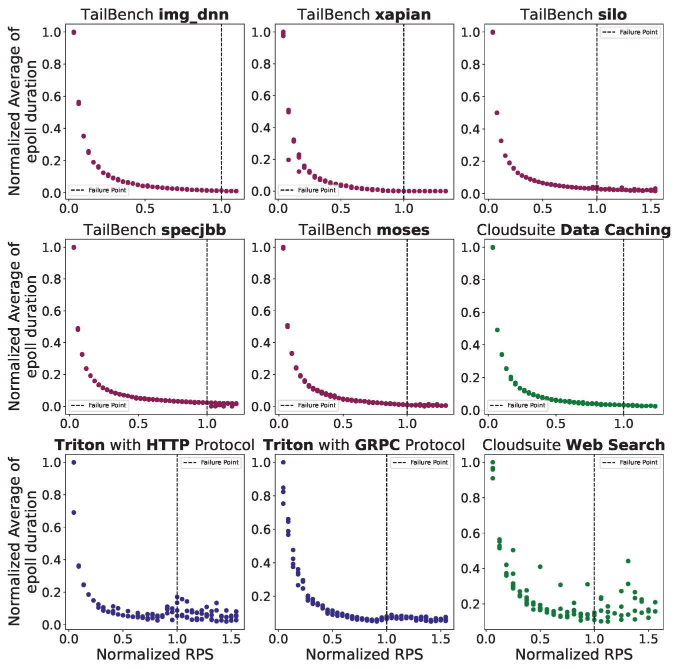

Fig. 4. Average of event polling duration under varying load. The vertical line indicates the point at which tail latency crosses the acceptable limit. epoll duration tends to stabilize under saturation and provides good observability of system saturation.

syscall. This diversity in syscall usage is reflected in our analysis.

In this chart, the x-axis indicates the actual RPS achieved by the application. A significant horizontal line is drawn across the chart, marking the point where Quality of Service (QoS) failure occurs, as reported by the client. This QoS failure point might differ from the actual saturation point of the application. On the y-axis, we display the normalized average duration of the `epoll` syscall group.

A notable trend observed in the chart is that the duration of the `epoll` syscall tends to decrease as the application approaches its saturation point. Upon reaching this saturation point, the duration typically stabilizes. This pattern is indicative of the changing behavior of the application under varying load conditions.

Furthermore, for certain workloads, such as the `web search` workload, an interesting phenomenon is observed post-saturation. Once the saturation point is breached, the application experiences an increase in idleness. This increase is attributed to heightened contention within the application and its queues. Essentially, as the application becomes overloaded and struggles to process incoming requests efficiently, it starts experiencing periods of inactivity at the application level. This scenario is often reflected in a decline in the achieved RPS, particularly after the QoS target is violated.

The insights gleaned from Figure 4 are crucial in understanding the dynamic behavior of applications under varying load conditions. By monitoring the `epoll` syscall duration in relation to the actual RPS, we can effectively gauge the application's performance and calculate a slack for saturation.

TABLE II
THE EFFECT OF THE NETWORK ON APPROXIMATED RPS

Network Config for RPS_Obsv (R^2)	**0ms Delay 0% Loss**	**10ms Delay 1% Loss**
TailBench **img-dnn**	0.9997	0.9998
TailBench **xapain**	0.9976	0.9964
TailBench **silo**	0.9998	0.9986
TailBench **specjbb**	0.9997	0.9996
TailBench **moses**	0.9411	0.9435
CloudSuite **Data Caching**	0.9995	0.9989
CloudSuite **Web Search**	0.8642	0.8573
Triton w/ **HTTP** Protocol	0.9976	0.9981
Triton w/ **GRPC** Protocol	0.9711	0.9703

V. CHALLENGES OF EBPF OBSERVABILITY

Our analysis of using eBPF observability of application-level metrics faced a series of challenges. As we discussed previously, since system call activity has no observability of request boundaries, eBPF can't have observability of individual request latency. Another notable challenge is the impact of network delays and loss and the intricacies introduced by various types of applications used in modern data centers. These factors complicate the accurate attribution of performance metrics to specific threads or processes within an application.

A. Impact of Network Delays and Packet Loss

Network-related issues, particularly delays and packet loss, significantly influence the performance of latency-critical applications. Delays refer to the time taken for requests to travel across the network, while loss denotes the failure of data packets to reach their destination. Therefore, we analyze the impact of networking quality on the efficacy of eBPF observability of application-level metrics.

We aimed to assess the impact of these network factors on the metrics analyzed in the previous sections. For this, we simulated different combinations of delays and losses on the loopback network interface, as the client and application were hosted on the same server.

Impact on Observed RPS. Our observations revealed that network delays, while impacting the latency perceived by the client, did not affect the number of requests processed by the server at any given time. Hence, our observed RPS remained unaffected. In scenarios with non-uniform delays, only the tail latency experienced fluctuations, but since our observed RPS relies on counting the number of syscalls within a specific duration, it remained consistent.

Loss, a common occurrence in TCP-based communications, can significantly impact tail latency due to retransmission. However, it did not markedly change the rate at which requests were processed, thus leaving the syscall count relatively stable. Consequently, the observed RPS still effectively mirrored the actual RPS, even in the presence of packet loss.

To evaluate this, we report the coefficient of determination (R^2) from linear regression (discussed in section IV-B) un-

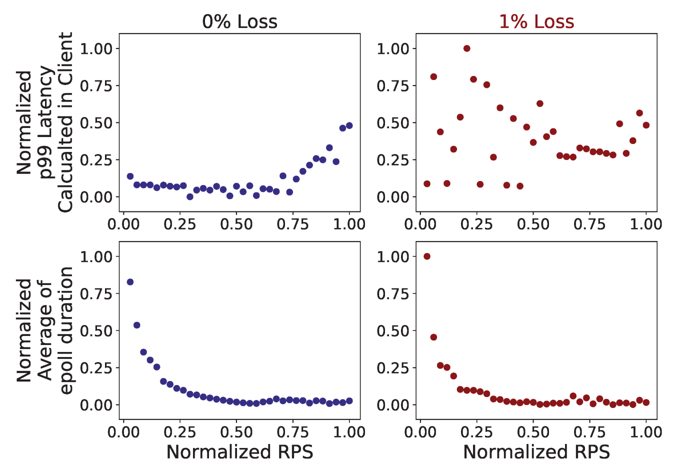

Fig. 5. Illustrating the impact of network loss on tail latency compared to a metric derived from system calls for Triton Inference server with GRPC communication protocol.

der various network configurations, as detailed in table II. The R^2 results demonstrate that different network settings, including those involving loss, which significantly impacts tail latency, do not notably affect the observed RPS. This finding underscores the robustness of an eBPF-derived observed RPS measurement across diverse network conditions.

Impact on Tail Latency. This experiment reinforces the understanding that system calls cannot fully capture actual latency values or the latency experienced by the client. External factors, particularly network elements, can significantly skew perceived latency. As illustrated in Fig 5(top row), even a small amount of network loss can substantially disturb tail latency. Yet, this disruption isn't mirrored in the metrics derived from syscalls (bottom row).

Fig 5 (top row) compares the 99th percentile latency for the `triton` workload with `GRPC` protocol under no loss and 1% loss scenarios. The bottom row of the figure, showing the normalized average duration of `epoll_wait`, indicates the application's idleness and saturation slack. While tail latency is significantly affected by network loss, this impact is not observable in the `epoll_wait` duration. Despite this, our focus on server saturation is justified, as the server's control over external factors like network conditions is limited and saturation metrics can still be effectively used in server management.

Impact on Saturation Observability. Our saturation detection method, based on observing abnormalities in syscall timing through variance, was not impacted by network delays. These delays added to the total request latency but did not alter the timing of syscall invocation or the point of system saturation. However, delays did increase the gap between the point of failure and saturation. Loss, while significantly affecting tail latency, also did not change the saturation point. Thus, saturation detection remains reliable even in the presence of network issues, although it cannot effectively be used interchangeably with failure detection.

Impact on Saturation Slack Observability. Similarly, network delays and loss created a discrepancy between the points of saturation and failure. Delays pushed the failure point further from the saturation point, while loss disrupted the tail latency failure analysis. As a result, saturation slack could not effectively replace latency slack, especially in scenarios with network loss. Additional network information is required to use saturation and failure metrics interchangeably and effectively.

B. *Complex, Multi-Stage Application.*

Application structures can vary significantly even going as far as distributed applications such as microservices, posing a challenge in generalizing our analysis approach. We evaluate the observability potential of a single-stage workload to evaluate the feasibility of this approach. For multi-stage workloads, like microservices, we would require eBPF observability of individual services in the microservice workload in order to then combine the request-level observability metrics together.

We aimed to identify metrics that would be broadly applicable across different application architectures, including those with complex multi-threading. While simple applications with straightforward threading structures allow for the reconstruction of request timelines as discussed in section III, more complicated applications require reliance on the broader metrics discussed in our analysis. However, we demonstrate that eBPF provides good request-level observability of even multi-threaded workloads.

C. *Async I/O interface (IO_uring)*

In the limitations of our approach, it's crucial to note that the effectiveness of syscall-based statistics is contingent on the presence of syscall activities. In scenarios where advanced I/O frameworks like IO_uring [46] are used, which bypass traditional syscalls for certain operations, our method may not yield useful insights as the receiving and sending of the request may not be observable by eBPF. Thus, further studies would be warranted to explore the ability for eBPF-based request-level observability of advanced I/O bypassing frameworks such as IO_uring.

VI. Implications of In-Kernel Observability of Application-Level Metrics

Our research opens several avenues for future applications leveraging kernel-level tracing to characterize application behavior, which we discuss below.

Blackbox Application Optimization. For third-party services lacking userspace access, eBPF kernel tracing offers a non-invasive way to understand and optimize resource demands and latency drivers. This enables the observability of various metrics without any modification to instrument workloads and provides a non-intrusive way to present additional feedback for optimization. For example, this methodology can pinpoint opportunities for improvements in scheduling, disk usage, or network configurations.

Resource Management. Efficient resource utilization techniques [6]–[9], [47]–[50] and power management tools [2]–[5],

[51]–[58] are crucial in data center operations; however, they all rely on constant client-side feedback in order to operate. Our methodology enables detailed monitoring of resource usage patterns at the system level, allowing for more efficient distribution and utilization of resources. This efficiency not only optimizes performance but also contributes to energy savings and reduced operational costs. More importantly, by relying on in-kernel observability, we can break the dependency on client-provided performance feedback and also enable kernel components (such as drivers) to be aware of application-level performance metrics.

Low overhead estimation. The metrics derived from our syscall-based methodology are straightforward and can be efficiently calculated within the eBPF space, leveraging the inherent advantages of eBPF, such as its low overhead [14], [19], [26]. This characteristic ensures that utilizing these metrics imposes a very low overhead on the system. By conducting calculations directly in the eBPF environment, we minimize the resource usage typically associated with data processing and analysis, making this approach particularly advantageous for performance-sensitive environments. This efficiency in computation aligns well with the goal of optimizing application performance without burdening the system. We measured the overhead of running the eBPF program on tail latency across our workloads and load levels. For all workloads, the median and upper quartile overhead remains significantly below 1% (typically below 0.5%). We will incorporate these values into the final paper version.

Predictive Provisioning. By analyzing the system-level behavior of applications, our approach facilitates predictive provisioning in data centers. By understanding the resource utilization patterns and performance characteristics of applications, data center operators can anticipate resource requirements and optimize resource allocation proactively. This predictive approach can lead to significant improvements in resource utilization efficiency and cost savings.

VII. Conclusion

In this paper, we introduced a methodology using eBPF for in-kernel observability of request-level metrics of latency-senstive workloads in cloud and datacenter environments. Our approach provided a non-invasive, detailed analysis of system call activities, offering insights into application behavior and optimization opportunities. We evaluate the feasibility of utilizing eBPF for observability of various performance metrics on a range of latency-sensitive workloads and demonstrate that eBPF can provide robust observability to a wide-range of metrics. We successfully navigated several challenges like network delays and multithreading, demonstrating the utility of eBPF in real-world scenarios. This study not only contributes to the field of performance analysis but also sets the stage for future advancements in efficient non-intrusive resource management and application optimization.

Acknowledgements

We would like to thank the anonymous reviewers for their invaluable comments and suggestions. This work is partly supported by the University of California, Riverside and the National Science Foundation under grants CNS-1955650, CNS-2047521, CCF-2324941, and CCF-2324940.

References

[1] "eBPF - introduction, tutorials & community resources," https://ebpf.io/, accessed: 2022-11-12.

[2] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 598–610. [Online]. Available: https://doi.org/10.1145/2830772.2830797

[3] C.-H. Chou, L. N. Bhuyan, and D. Wong, "μdpm: Dynamic power management for the microsecond era," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 120–132.

[4] C.-H. Chou, D. Wong, and L. N. Bhuyan, "Dynsleep: Fine-grained power management for a latency-critical data center application," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ser. ISLPED '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 212–217. [Online]. Available: https://doi.org/10.1145/2934583.2934616

[5] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 31–40.

[6] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 107–120. [Online]. Available: https://doi.org/10.1145/3297858.3304005

[7] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 57–68.

[8] R. B. Roy, T. Patel, and D. Tiwari, "Satori: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 292–305.

[9] C. Delimitrou and C. Kozyrakis, "Qos-aware scheduling in heterogeneous datacenters with paragon," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, dec 2013. [Online]. Available: https://doi.org/10.1145/2556583

[10] ——, "Hcloud: Resource-efficient provisioning in shared cloud systems," *SIGPLAN Not.*, vol. 51, no. 4, p. 473–488, mar 2016. [Online]. Available: https://doi.org/10.1145/2954679.2872365

[11] ——, "Bolt: I know what you did last summer... in the cloud," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, p. 599–613, apr 2017. [Online]. Available: https://doi.org/10.1145/3093337.3037703

[12] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 301–312.

[13] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 19–33. [Online]. Available: https://doi.org/10.1145/3297858.3304004

[14] J. Levin and T. A. Benson, "Viperprobe: Rethinking microservice observability with ebpf," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, 2020, pp. 1–8.

[15] X. Dong and Z. Liu, "Multi-dimensional detection of linux network congestion based on ebpf," in *2022 14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, 2022, pp. 925–930.

[16] K. Suo, Y. Zhao, W. Chen, and J. Rao, “vnettracer: Efficient and programmable packet tracing in virtualized networks,” in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 165–175.

[17] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, “A protocol-independent container network observability analysis system based on ebpf,” in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, 2020, pp. 697–702.

[18] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, “Ghost: Fast & flexible user-space delegation of linux scheduling,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 588–604. [Online]. Available: https://doi.org/10.1145/3477132.3483542

[19] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. K. Ramakrishnan, “Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 780–794. [Online]. Available: https://doi.org/10.1145/3544216.3544259

[20] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, “Temporal system call specialization for attack surface reduction,” in *USENIX Security Symposium*, 2020.

[21] A. Bulekov, R. Jahanshahi, and M. Egele, “Saphire: Sandboxing PHP applications with tailored system call allowlists,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2881–2898. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/bulekov

[22] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, “Confine: Automated system call policy generation for container attack surface reduction,” in *International Symposium on Recent Advances in Intrusion Detection*, 2020.

[23] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, “sysfilter: Automated system call filtering for commodity software,” in *International Symposium on Recent Advances in Intrusion Detection*, 2020.

[24] S. K. Peddoju, H. Upadhyay, J. Soni, and N. Prabakar, “Natural language processing based anomalous system call sequences detection with virtual memory introspection,” *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 5, 2020. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2020.0110559

[25] D. Skarlatos, Q. Chen, J. Chen, T. Xu, and J. Torrellas, “Draco: Architectural and operating system support for system call security,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 42–57.

[26] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. K. Ramakrishnan, and T. Wood, “Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 168–181. [Online]. Available: https://doi.org/10.1145/3472883.3487014

[27] “BCC - IO visor project,” https://www.iovisor.org/technology/bcc, Dec. 2016, accessed: 2022-11-12.

[28] “bpftrace: High-level tracing language for linux eBPF.”

[29] N. Shirokov and R. Dasineni, “Open-sourcing katran, a scalable network load balancer,” May 2018. [Online]. Available: https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

[30] H. Zhou, N. Shirokov, and M. Lau, “Xdp production usage: Ddos protection and l4lb,” *facebook*, 2017. [Online]. Available: https://netdevconf.info/2.1/slides/apr6/zhou-netdev-xdp-2017.pdf

[31] H. M. Demoulin, I. Pedisich, N. Vasilakis, V. Liu, B. T. Loo, and L. T. X. Phan, “Detecting asymmetric application-layer Denial-of-Service attacks In-Flight with FineLame,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 693–708. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/demoulin

[32] L. Deri, S. Sabella, and S. Mainardi, “Combining system visibility and security using ebpf,” in *Italian Conference on Cybersecurity*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:59616648

[33] P. Chaignon, K. Lazri, J. François, T. Delmas, and O. Festor, “Oko: Extending open vswitch with stateful filters,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3185467.3185496

[34] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, “Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications,” *ACM Comput. Surv.*, vol. 53, no. 1, feb 2020. [Online]. Available: https://doi.org/10.1145/3371038

[35] N. Amit and M. Wei, “The design and implementation of hyperupcalls,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 97–112. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/amit

[36] A. Bijlani and U. Ramachandran, “Extension framework for file systems in user space,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 121–134. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/bijlani

[37] I. Babrou, “Debugging linux issues with ebpf,” Oct 2018. [Online]. Available: https://www.usenix.org/conference/lisa18/presentation/babrou

[38] “Monitoring containerised application environments with eBPF,” https://www.ntop.org/ntop/monitoring-containerised-application-environments-with-ebpf/, May 2019, accessed: 2024-4-8.

[39] J. Perry, “Monitoring service architecture and health with bpf,” https://www.youtube.com/watch?v=J2NWvh3lgJI, 2019.

[40] H. Kasture and D. Sanchez, “Tailbench: a benchmark suite and evaluation methodology for latency-critical applications,” in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[41] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the clouds: a study of emerging scale-out workloads on modern hardware,” *SIGPLAN Not.*, vol. 47, no. 4, p. 37–48, mar 2012. [Online]. Available: https://doi.org/10.1145/2248487.2150982

[42] “server: The triton inference server provides an optimized cloud and edge inferencing solution.”

[43] “NVIDIA triton inference server,” https://developer.nvidia.com/nvidia-triton-inference-server, Mar. 2020, accessed: 2022-11-16.

[44] “tc-netem(8) — linux manual page,” https://man7.org/linux/man-pages/man8/tc-netem.8.html, accessed: 2023-12-18.

[45] Y. Zhang, D. Meisner, J. Mars, and L. Tang, “Treadmill: Attributing the source of tail latency through precise load testing and statistical inference,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16. IEEE Press, 2016, p. 456–468. [Online]. Available: https://doi.org/10.1109/ISCA.2016.47

[46] “io_uring - asynchronous i/o facility,” https://man.archlinux.org/man/io_uring.7.en, accessed: 2023-12-18.

[47] A. Jahanshahi, M. Rezvani, and D. Wong, “Wattwiser: Power & resource-efficient scheduling for multi-model multi-gpu inference servers,” in *2023 IEEE 14th International Green and Sustainable Computing Conference (IGSC)*, 2023.

[48] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, “Heracles: Improving resource efficiency at scale,” *SIGARCH Comput. Archit. News*, vol. 43, no. 3S, p. 450–462, jun 2015. [Online]. Available: https://doi.org/10.1145/2872887.2749475

[49] X. Wang, S. Chen, J. Setter, and J. F. Martínez, “Swap: Effective fine-grain management of shared last-level caches with minimum hardware support,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 121–132.

[50] H. Yang, A. Breslow, J. Mars, and L. Tang, “Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 607–618. [Online]. Available: https://doi.org/10.1145/2485922.2485974

[51] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo, “Ant-man: Towards agile power management in the microservice era,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

[52] D. Meisner, B. T. Gold, and T. F. Wenisch, “Powernap: Eliminating server idle power,” in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, p. 205–216. [Online]. Available: https://doi.org/10.1145/1508244.1508269

[53] D. Wong and M. Annavaram, “Knightshift: Scaling the energy proportionality wall through server-level heterogeneity,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 119–130.

[54] ——, “Implications of high energy proportional servers on cluster-wide energy proportionality,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 142–153.

[55] D. Wong, “Peak efficiency aware scheduling for highly energy proportional servers,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16. IEEE Press, 2016, p. 481–492. [Online]. Available: https://doi.org/10.1109/ISCA.2016.49

[56] L. Zhou, C.-H. Chou, L. N. Bhuyan, K. K. Ramakrishnan, and D. Wong, “Joint server and network energy saving in data centers for latency-sensitive applications,” in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 700–709.

[57] K. Kaffes, D. Sbirlea, Y. Lin, D. Lo, and C. Kozyrakis, “Leveraging application classes to save power in highly-utilized data centers,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 134–149. [Online]. Available: https://doi.org/10.1145/3419111.3421274

[58] L. Zhou, L. N. Bhuyan, and K. K. Ramakrishnan, “Gemini: Learning to manage cpu power for latency-critical search engines,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 637–349.