Perforator Demystified: An Empirical Study of Yandex Perforator's Low Overhead Mechanisms

Chirag Modi
Computer Science and Engineering
23210031
IIT Gandhinagar
chirag.modi@iitgn.ac.in

Mithil Pechimuthu
Computer Science and Engineering
21110129
IIT Gandhinagar
pechimuthumithil@iitgn.ac.in

Naman Dharmani
Computer Science and Engineering
21110136
IIT Gandhinagar
dharmaninaman@iitgn.ac.in

***Abstract*—This project presents an empirical study of Yandex Perforator, a modern continuous profiling tool that leverages eBPF (Extended Berkeley Packet Filter) to deliver low-overhead performance insights in cloud-native environments. We deploy Perforator in both standalone and Kubernetes-based environments to benchmark its capabilities. By profiling CPU-intensive and TCP server and DPDK based DNS applications, we validate Perforator's ability to detect bottlenecks and improve performance, effectively supporting real-time optimisation and observability.**

***Index Terms*—Continuous Profiling, eBPF, Perforator, Yandex, Perf**

I. Introduction

Modern computing hardware is complex, and its complexity is not ceasing to grow. It makes computing faster, but the simple mental models of computers we follow are not applicable now. Due to this, a developer is puzzled while trying to identify why a program is using many resources or is performing poorly. Couple this with the impact of poor-performing and over-utilising programs on cost while being scaled on cloud-based computing platforms. In such scenarios, the ability to profile the program is invaluable to the developer. Profiling allows us to perform performance analysis and isolate and analyse bottlenecks. However, we can not always simulate the live traffic that the program will face. Also, we do not always have the opportunity to stop the program profile, optimise it, and then put it back into production. Hence, monitoring the program on the cloud while being exposed to live traffic is preferred. We can add specific hooks into the source code to profile the program, but these intrusive techniques can add overhead to the program's performance. They may cause degradation beyond the limits that we can tolerate. Thus, we use the other alternative, sampling-based continuous profiling with low overhead while being descriptive enough to help developers pinpoint the bottlenecks. It must be emphasised that in cloud-based environments, it is necessary to manage profiling overhead, as any unnecessary profiling overhead can cost millions of dollars in additional resources because of these data centres' massive scale of operation.

Through profiling, we can answer the following questions [10] [11]:

- What are the hottest processes, routines, or code regions?
- How does performance differ between software versions?
- Which locks are most contended?
- Which processes are memory hogs?
- How much money would optimizing this function save?

Yandex [11], a Russian technology company that provides Internet-related products and services, claim that through profiling, they regularly achieve tens of per cent performance improvements in Yandex's largest services, such as the Ad Delivery System and Yandex Search.

II. Background

Effective performance profiling has long been a critical component in optimizing computing systems. As applications have grown increasingly complex and distributed, the methods for analyzing their performance have evolved dramatically. This section explores the historical context of profiling methodologies, the evolution toward continuous profiling, and the emergence of eBPF as a revolutionary technology in this domain.

Profiling methodologies have historically fallen into two main categories [11]:

- Instrumentation-based: These intrusive profilers modify the program by injecting markers into the beginning and end of functions, to measure execution time. This can be easily done on single systems, but if the program scales across multiple systems, then this is difficult and has substantial overheads.
- Sampling-based. In contrast, these periodically pause program execution and sample its state. To get a statistically significant picture of the program's execution, we need to collect a large number of these states.

Sampling profilers have hence become a trademark when profiling large systems with low overhead. Some of the most common sampling profile methods and tools are Poor Man's Profiler [7], Tracy [6], Parca [4], Google-Wide Profiling [10] and Pyroscope [5]. Table I compares *Perforator* with the tools mentioned above, while also highlighting the key features.

1) Poor Man's Profiler (PMP): The Poor Man's Sampling Profiler (PMSP) represents one of the simplest implementations of sampling-based profiling. It typically involves using a debugger to attach to a running process, extract stack traces

at regular intervals, and then detach—all without significantly impacting the application's performance [2]. While rudimentary compared to modern profiling solutions, PMSP established the fundamental principles that inform contemporary sampling-based profilers.

2) Linux Perf: The Linux Performance Events (perf) subsystem provides low-level CPU profiling capabilities. It operates through the *perf_events* API, which allows applications to trigger event-based sampling. Perf implements event-based profiling by configuring hardware performance counters to trigger interrupts when they overflow. When a counter reaches its threshold, a software interrupt is generated, which the operating system handles by collecting relevant performance data [9]. This mechanism allows perf to collect detailed information about CPU usage, cache misses, branch predictions, and other hardware events with minimal overhead.

3) Continuous Profiling: As distributed computing environments became more prevalent, the need for continuous performance monitoring across entire data centers grew increasingly important. In response to this need, Google introduced Google-Wide Profiling (GWP), a continuous profiling infrastructure for data centers [10]. Unlike traditional profiling, which typically occurs on-demand in response to specific performance issues, GWP operates continuously, collecting performance data across thousands of machines and applications. This approach provides a comprehensive view of system performance over time, enabling both immediate issue detection and long-term optimization.

However, continuous profiling also presents significant challenges, particularly in terms of data volume and processing. Collecting profiles from thousands of machines generates enormous amounts of data that must be efficiently stored, processed, and analyzed [10]. GWP addresses this challenge through efficient symbolization processes and specialized storage systems designed to handle profile data at scale.

4) Extended Berkeley Packet Filter (eBPF): eBPF evolved from the Berkeley Packet Filter (BPF), which was initially designed for network packet filtering. At its core, eBPF allows developers to write programs that run directly in kernel space without modifying the kernel source code or loading additional kernel modules [8]. These programs are executed in a sandboxed environment that ensures they cannot compromise system stability or security. eBPF programs can be attached to various kernel hooks, enabling them to collect data about system calls, network traffic, process execution, and other kernel activities . This capability provides unprecedented visibility into system operations without requiring invasive instrumentation or significant overhead.

Traditional observability solutions often rely on heavy agents running in user space, which can introduce significant overhead and provide limited visibility into kernel-level operations [13]. eBPF transforms this paradigm by enabling lightweight, kernel-level monitoring that provides comprehensive visibility with minimal performance impact. Linux allows eBPF programs to run when the Performance Monitoring Unit

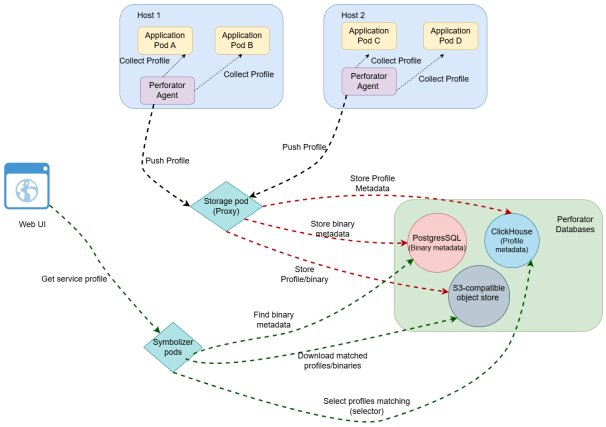


Fig. 1: Perforator Architecture

(PMU) triggers an interrupt, and it provides helper functions for reading memory in both user and kernel space [11].

Perforator leverages eBPF technology to monitor and analyze server performance efficiently, without requiring source code modifications. It works by providing continuous, real-time monitoring and analysis of servers and applications, helping businesses track and optimize resource-intensive code sections.

One of Perforator's standout features is its support for Profile-Guided Optimization (PGO), which further enhances its ability to optimize application performance [11]. This capability, combined with eBPF's low-overhead data collection, positions Perforator as a powerful tool for continuous profiling in production environments particularly cloud-native environments which provide comprehensive visibility across containerized, dynamically scheduled workloads.

III. Perforator Overview

Perforator [1] is mainly designed for large data centeres which collect profiles with negligible overhead. It can be deployed both as standalone executables or inside the Kubernetes cluster. The detailed architecture is explained in the following section:

A. Architecture

Figure 1 shows the architecture of the perforator deployed on the Kubernetes cluster. Besides the application that we want to profile, there are some pods of perforators that are interconnected with each other to perform specific tasks related to profiling.

Agent: The perforator agent pod is a core component of architecture. The agent runs on each cluster node, collects the profiles using eBPF, aggregates, compresses, and sends them to the storage via gRPC in the pprof-compatible format. As per Figure 2, the agent connects to the kubelet to identify running pods on a node. Each CPU core triggers a perf event (like 10M clock cycles or 1K major page faults) to the kernel, and then the kernel triggers the eBPF program. The eBPF program collects info about the program/thread running on that core, such as process/thread name, process/thread

TABLE I: Comparison of Perforator with Other Profilers

Tool	Key Features	Supported Languages	Overhead
Perforator	Cluster-wide continuous profiling using eBPF; kernel and userspace stack collection; flamegraphs; PGO support	C++, Rust, C, Go, Python, Java	∼512MB RAM; <1% CPU
Tracy	Real-time profiling with high-resolution timing; remote telemetry; GPU profiling support; integrates with various languages	C, C++, Lua, Rust, Zig, C#	2.25 nanoseconds per instrumentation call
GWP	Sampling-based detection of heap-use-after-free and buffer overflows in production; minimal overhead; probabilistic detection	C, C++	negligible overhead like Perforator
Poor Man's Profiler (PMP)	Minimalist sampling profiler using GDB to capture stack traces; no code modification required; suitable for quick diagnostics	Any language with GDB support	Low; depends on sampling frequency
Pyroscope	Continuous profiling with support for various profile types (CPU, memory, etc.); integrates with Grafana for visualization; supports multiple languages	Go, Java, .NET, Ruby, Python, Rust, Node.js	∼50MB RAM per pod.

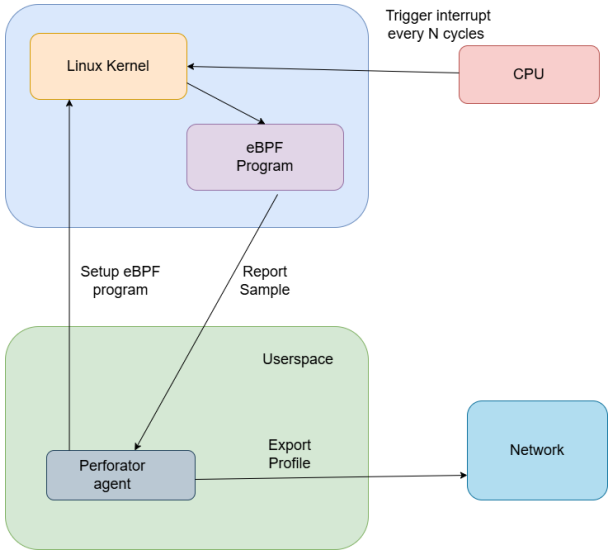


Fig. 2: Perforator Agent

ID, userspace and kernel space call stack, and so on. This aggregate data is a sample sent to the agent in userspace via eBPF perfbuf API. The agent collects memory samples from the eBPF program and periodically sends them to the storage services over gRPC. By default, an agent will send a profile for each pod every minute. That profile consists of samples of one workload over one minute. To generate a human-readable profile, we symbolize to map addresses of executable instructions to source code lines. To do this efficiently, the agent uploads the executable binary files to S3 storage via storage pods as a proxy. That binary is processed later to generate symbolized profiles. The metadata of binary is stored in the PostgreSQL database.

Storage Proxy: This simple gRPC server acts as a proxy and redirects upload requests to database services such as click house,postgreSQL, and S3. These are stateless, so we can run hundreds of storage pods in our cluster.

Symbolizer: Symbolizer is the leading user-facing service, which allows users to list profiles or services and build merge profiles by combining the atomic profiles that occur at every minute. Also, it does the symbolization process and presents a human-readable profile to the user.

Database services: Perforator uses three different storages.

1) S3 Compatible Storage: It stores raw atomic profiles, binaries. For local setup or testing purpose , we can use any S3 emulated storage such as minio.

2) Click-house Database: The clickhouse database contains entry for each atomic profile send by agent. This allows users to quickly find the interesting profiles using filters.

3) PostgresSQL Database: PostgresSQL database contains metadata for every executable binary uploaded through agent.

B. Deployment Models

Perforator can be used to profile the host machine using two different models, depending on your monitoring needs.

Standalone CLI binary: With this model, each component of Perforator can be executed as a standalone binary. These components are located in the perforator/cmd directory and can be built using the yatool build system. To capture a profile for a specific process over a given duration, you can use the Perforator agent, which is located at perforator/cmd/cli. This allows you to generate and publish profiles locally. However, please note that these profiles are temporary and are only retained for the duration of the recording. For continuous profiling and long-term storage, it's necessary to persist the data in a database, as described in the previous section.

Kubernetes Deployment: Perforator can be deployed within a Kubernetes cluster by running all its services as individual pods. It uses Helm as the package manager to simplify deployment within the cluster. For testing and development purposes, Perforator provides an easy setup where both its core components and database services run

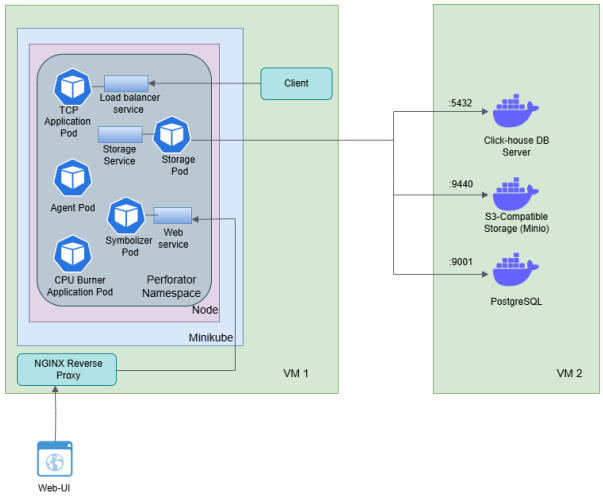


Fig. 3: Experiment Setup

inside the same cluster. This eliminates the need to configure external database connections. However, for production deployments, it's recommended to decouple the database services from the cluster and configure their endpoints explicitly within Perforator. As of the latest update, Perforator also supports profiling across multiple clusters.

IV. Experiment

The primary goal of our experiment is to evaluate the accuracy and effectiveness of Perforator in comparison to other profilers. We collected profiling data from various applications using both standalone and cluster-based deployment models. The detailed experimental procedures and methodology are outlined in the sections below.

Experiment Setup: We conducted our experiments on a system equipped with an Intel® Xeon® CPU E5-2630 v3 @ 2.40GHz, running two Linux virtual machines (VMs). As shown in Figure 3, the first VM hosted the Perforator cluster, which included all core components and services. We used Minikube to set up the cluster, deploying it within an isolated Kubernetes namespace named perforator. The second VM was dedicated to database services required by Perforator. It ran ClickHouse, PostgreSQL, and MinIO (an S3-compatible object storage service) as Docker containers. We then configured the Perforator cluster to connect to these external database endpoints. The complete deployment process is documented in the README file of our repository. Using this setup, we profiled two applications, as illustrated in Figure 3.

A. CPU Burner Application

We profiled a C++ application containing a function named inefficient_calc_sum(), which performs a summation of vector elements using a large for loop. The complete source code for this application is available in our repository. As illustrated in Figure 3, the application was executed inside a Kubernetes pod labeled as the "CPU Burner Application" pod. Using the flamegraph generated by Perforator, we identified that the inefficient_calc_sum() function consumed a significant amount of CPU resources—approximately 11.4 billion cycles. To optimize the function, we replaced the for loop with a direct mathematical equation that yields equivalent results. Upon re-profiling with Perforator, the optimized version of the function became so efficient that it did not appear in the flamegraph—likely because Perforator's sampling interval was not sufficient to capture any samples for it. Both the original and optimized flamegraphs are shown in Figure 4.

B. TCP server application

As shown in Figure 3, we profiled a simple TCP server application written in C++. This server accepts connections from clients, receives a string, and responds with the reversed version of that string. To profile this application, we containerized the C++ code and deployed it as a pod within the Kubernetes cluster. Our focus was primarily on analyzing the RX path, including both kernel-level and user-space library functions involved in handling incoming data. To simulate traffic, we ran a client-side script outside the cluster that periodically sent requests to the TCP server pod. Figure 5 presents the flamegraph of the application, illustrating the CPU cycles consumed by various functions involved in both the RX and TX paths of the TCP communication. The complete source code for this application is available in the repository.

C. KDNS - DNS resolver with DPDK

Using the standalone deployment model, we profiled the DNS resolver application kdns, which utilizes DPDK [3] as its packet I/O library. DPDK (Data Plane Development Kit) is a kernel-bypass technology that allows packets to be forwarded directly from the NIC to userspace, bypassing the traditional kernel network stack. Running a DPDK-based application inside a container can be complex, primarily due to the need to bind the NIC directly to the DPDK process. For this reason, we opted for a standalone deployment model for profiling kdns. The steps for deploying kdns and setting up standalone monitoring are detailed in the repository's README file.As part of our experiment, we profiled the kdns application in two scenarios: 1.With DPDK (kernel-bypass), 2. Without DPDK (using the kernel network stack). Using flamegraphs, we analyzed the CPU cycle distribution across RX and TX path functions in both cases. As shown in Figure 6: Figure 6.a presents the flamegraph of the DNS resolver without DPDK, where kernel functions are clearly visible in the stack trace. Figure 6.b shows the flamegraph with DPDK enabled, in which only userspace functions and DPDK APIs appear, with no involvement from kernel functions.

V. Challenges with Perforator

Despite its robust design, Perforator faces challenges in stack unwinding [11], particularly when dealing with handwritten assembly code lacking proper DWARF CFI annotations and binaries compiled with the `-fno-asynchronous-unwind-tables` flag, which disables `.eh_frame` generation. The `.eh_frame` section that encodes the steps to reconstruct the parent stack frame

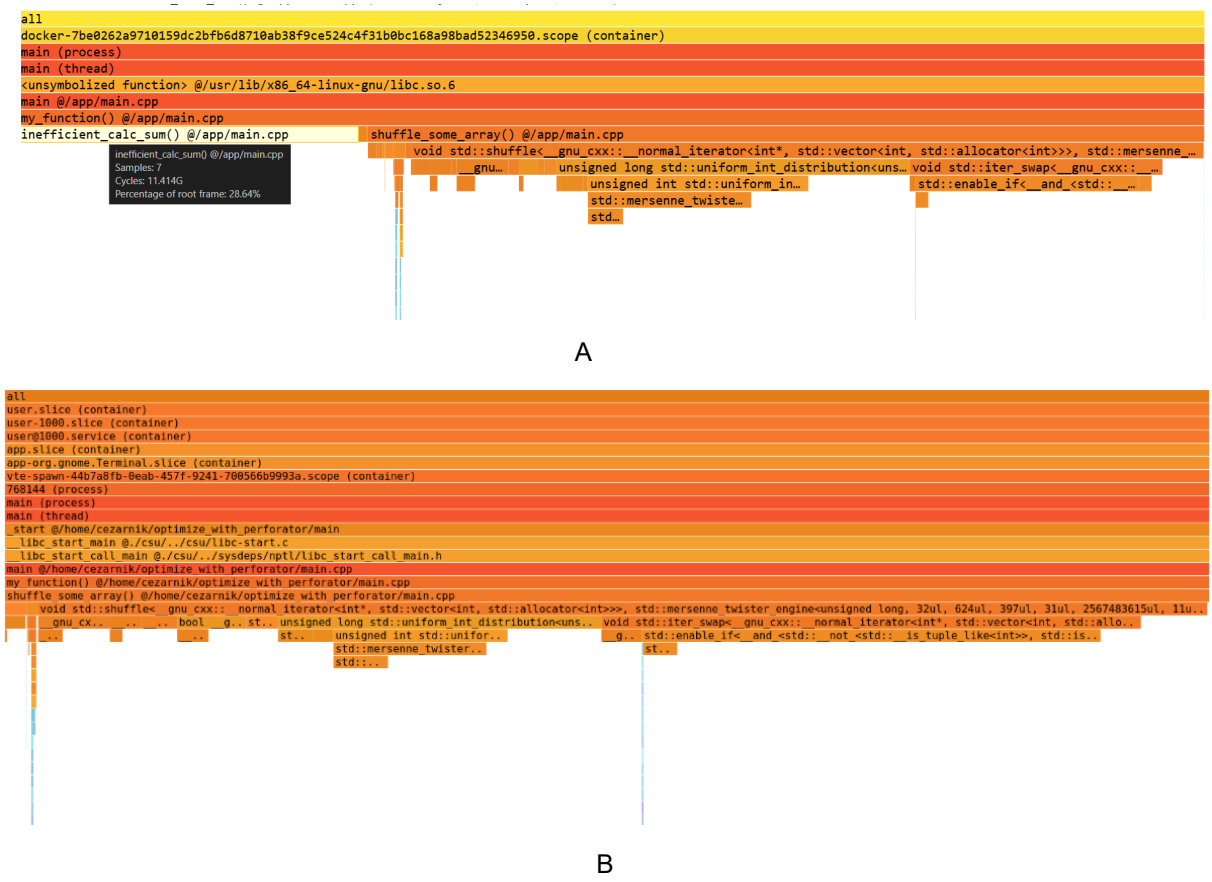


Fig. 4: Flame graphs for (A) Inefficient CPU burning program and (B) after optimizing it.

from any instruction in the program, even in programs without exceptions or where exceptions are disabled (for example, in C). Absence of `.eh_frame` hinders accurate stack trace reconstruction. Additionally, lowering function exclusion thresholds can result in excessively large profiles, sometimes reaching gigabytes, leading to performance issues in rendering and interaction due to the heavy DOM load in SVG-based visualizations. System-wise, Perforator requires `CAP_SYS_ADMIN` privileges and is limited to x86-64 Linux platforms with kernel version 5.4 or later, restricting its deployment in certain environments. While it supports multi-threaded and multi-process applications, accurately attributing CPU usage in complex concurrent scenarios remains challenging. Furthermore, the agent may consume several gigabytes of anonymous memory to store unwind tables on large hosts, though efforts are underway to optimize memory usage [11].

VI. Future Work

While Perforator provides a robust tool for continuous profiling, many aspects can still be explored. Currently, Perforator employs event-based sampling to capture performance insights with minimal overhead. This is conceptually similar to head-based sampling, as mentioned in the OpenTelemetry documentation [12], where the decision to record is made early (typically at the event ingestion point), ensuring that only a subset of events are profiled or traced from the outset. This model effectively keeps system overhead low while still generating actionable data. Google's Wide Profiling (GWP) samples in two dimensions:

- Machine-level sampling – Only a small subset of machines in the fleet run profiling at any given time.
- Event-level sampling – In those selected machines, event-based sampling is applied to determine which runtime events are profiled.

This dual-dimension approach helps balance fidelity and overhead at scale. Sampling in just one dimension leads to trade-offs: either the system uses too many resources (if all machines are profiled with standard event sampling), or the profiles become too sparse (if event sampling is applied uniformly at a low rate). By adjusting sampling rates per event type and spreading the load across the fleet, GWP achieves meaningful, granular data with very low interference—a system successfully deployed across nearly all machines at Google for years. Something like this can also be done for Perforator to even further reduce it's overheads.

VII. Conclusion

This project demonstrated the effectiveness of Perforator as a modern, low-overhead profiler for cloud-native applications.

Fig. 5: TCP Server Application

Its seamless integration with Kubernetes, ability to capture meaningful stack traces via eBPF, and visual insights through flamegraphs make it a valuable asset for developers and system administrators. The experiments highlight its practical use in real-world scenarios and its role in driving performance gains with minimal disruption. As profiling becomes more integral to DevOps and performance engineering, tools like Perforator are poised to play a critical role in observability at scale.

References

[1] Perforator. URL: https://perforator.tech/docs/en/.

[2] A Poor Man's Profiler with PowerShell and CDB, 8 2012. URL: https://www.leeholmes.com/a-poor-mans-profiler-with-powershell-and-cdb/.

[3] Data plane development kit. http://dpdk.org/, 2014. [ONLINE].

[4] Parca Contributors. Parca: Continuous profiling for cloud-native environments. Available at: https://www.parca.dev.

[5] Pyroscope Contributors. Pyroscope: Continuous profiling for modern applications. Available at: https://pyroscope.io.

[6] given=Wolfpld given i=W. Github - wolfpld/tracy: Frame profiler. URL: https://github.com/wolfpld/tracy.

[7] family=Callaghan given=Mark and family=Mituzas given=Domas. Poor man's profiler. URL: https://poormansprofiler.org/.

[8] Christopher Lentricchia. Next-Generation Observability with eBPF - Isovalent, 3 2024. URL: https://isovalent.com/blog/post/next-generation-observability-with-ebpf/.

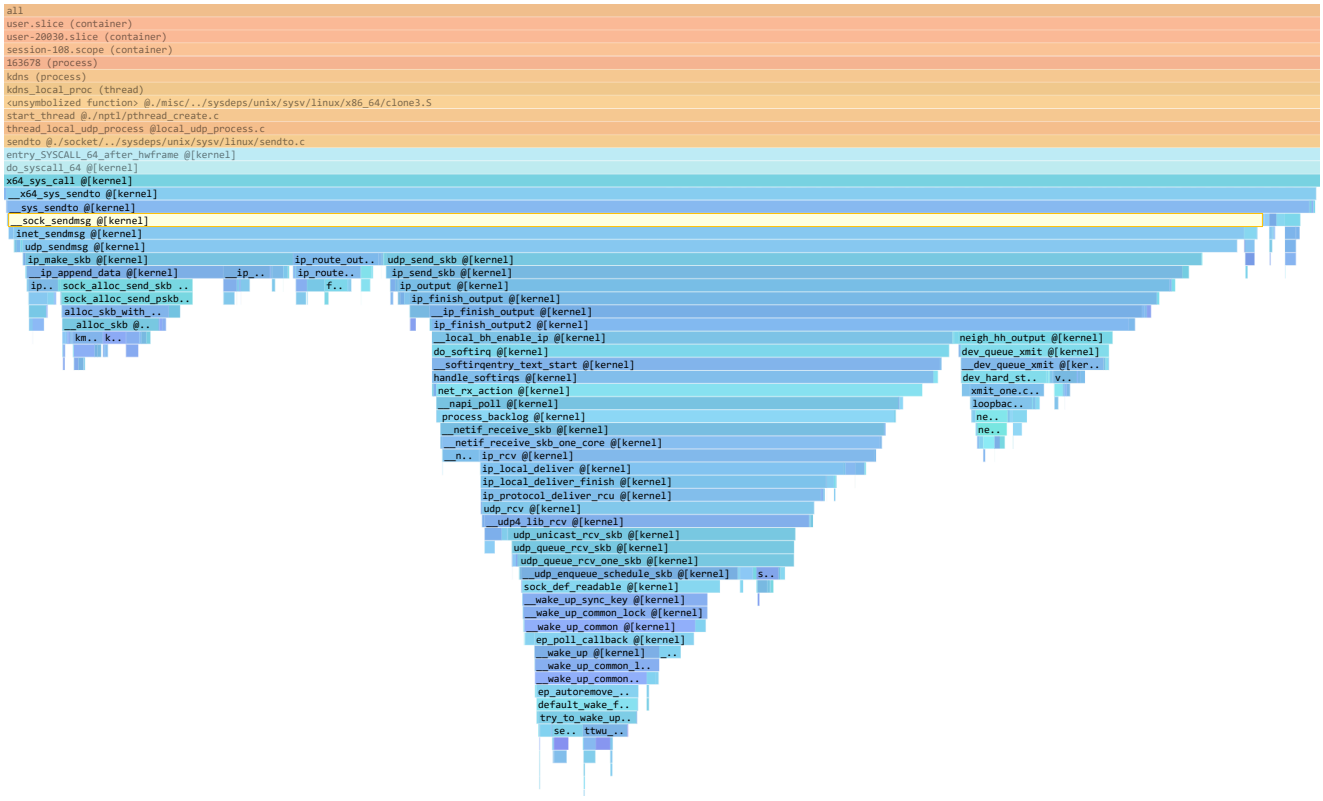
[9] Linux perf wiki Contributors. perf: Linux profiling with performance counters. URL: https://perfwiki.github.io/main/.

[10] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010. URL: http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68.

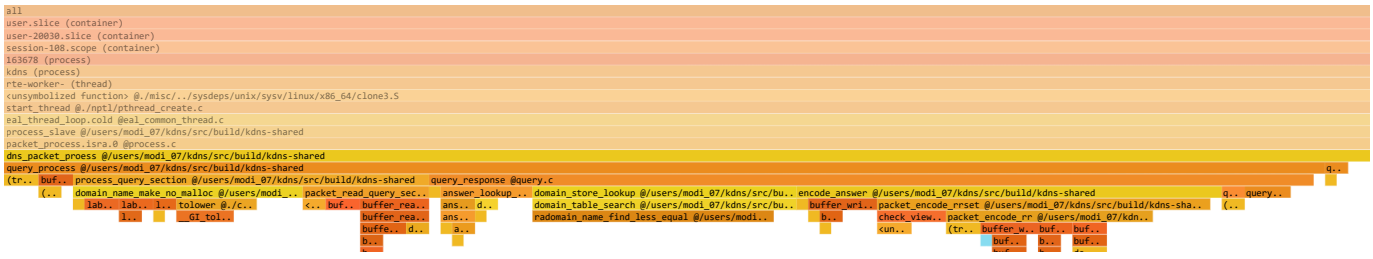
[11] Sergey Skvortsov. Yandex's High-Performance profiler is now open source. 3 2025. URL: https://medium.com/yandex/yandexs-high-performance-profiler-is-now-open-source-95e291df9d18.

[12] Open Telemetry. Sampling. URL: https://opentelemetry.io/docs/concepts/sampling/.

[13] Aviv Zohari. EBPF observability, 4 2025. URL: https://www.groundcover.com/ebpf/ebpf-observability.



(a)



(b)

Fig. 6: Flame graphs for A) DNS resolver without DPDK B) DNS resolver with DPDK