Project 15: DNS resolver using AF_XDP Socket

Guntas Singh Saran
Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India
guntassingh.saran@iitgn.ac.in

Hrriday V. Ruparel
Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India
hrriday.ruparel@iitgn.ac.in

Kishan Ved
Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India
kishan.ved@iitgn.ac.in

Pranav Patil
Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India
pranav.patil@iitgn.ac.in

I. Overview

This report presents the progress of our project, which aims to develop a high-performance DNS stub resolver using AF_XDP sockets to bypass the traditional Linux kernel networking stack. The primary goal is to optimize DNS query processing by filtering packets directly at the NIC level to minimize latency and CPU overhead.

Our initial efforts were focused on understanding the Linux kernel's packet processing pipeline, the working of XDP (eXpress Data Path), and how it can bypass the Linux kernel network stack's bottlenecks, especially at high bandwidths.

The system is divided into two key components: a kernel-space program and a user-space application. The kernel-space XDP program filters incoming UDP-based DNS queries and redirects them to user space through AF_XDP sockets, avoiding the overhead of the traditional kernel network stack. The user-space application receives these packets, extracts the DNS query, "forwards" the query packet to downstream resolvers if corresponding record is not cached, and constructs the appropriate response to send back to the client.

This architecture enables efficient packet processing with minimal Network stack involvement, allowing our DNS stub resolver to achieve high throughput and low latency, making it suitable for modern high-performance networking environments.

II. Introduction

Modern high-speed networks place stringent performance requirements on DNS resolution, yet traditional Linux kernel networking introduces significant processing overheads. This section provides the context and background for our project, which aims to overcome these limitations by developing a high-performance DNS stub resolver using AF_XDP sockets. In this introduction, we detail the journey of a network packet through the Linux kernel, examine the bottlenecks in the host network stack, and explore modern techniques for kernel bypass and fast I/O.

A. The Journey of a Network Packet Through the Linux Kernel

A comprehensive understanding of the traditional networking pipeline is critical to appreciating the design of our DNS stub resolver. In the Linux kernel, packet processing begins at the Network Interface Card (NIC) and follows the ingress path as stated by Stephen et al. [1]:

- **Ingress Path:** Packets are transferred from the NIC via Direct Memory Access (DMA) through the Ethernet layer, then processed through the IP and transport layers (TCP/UDP). Functions such as `ip rcv()`, `tcp_v4_rcv()`, and `udp rcv()` are employed, incurring overheads like socket buffer (sk_buff) allocations and context switching.
- **Egress Path:** The reverse operation sees packets travel from the application layer down to the NIC through functions like `tcp sendmsg()` and `udp sendmsg()`, again incurring overhead through the use of kernel buffers and DMA.

Understanding this pipeline reveals key bottlenecks—including data copying and interrupt handling—that our project mitigates by using XDP at the NIC level to reduce kernel intervention.

B. Understanding Host Network Stack Overheads

Research into Linux network stack performance, such as the study by Cai et al. [2], has identified several modern challenges at high bandwidths (up to 100 Gbps):

- **CPU Bottlenecks:** The cost of copying data from kernel space to user space can overwhelm a single CPU core, directly motivating our objective.
- **Cache Issues:** As bandwidth-delay products increase, L3 cache sizes may prove insufficient, leading to high cache miss rates even with Direct Cache Access (DCA) enabled by DDIO (Data Direct I/O).
- **Multi-Flow Challenges:** Resource contention (e.g., L3 cache pollution) reduces throughput per core, emphasizing the need for efficient packet filtering to isolate a particular flow, like DNS traffic.

These studies justify the need for an optimized, kernel bypass approach to achieve higher throughput and lower latency.

C. Device Driver and Interrupt Handling

Beyond network stack overheads, the interaction between the NIC and the kernel further contributes to performance limitations:

- **Device Drivers:** NICs are configured via Memory-Mapped I/O (MMIO) and utilize TX/RX rings to exchange packet information with the kernel.
- **Interrupt Handling:** When a NIC receives a packet, it sends an interrupt to the CPU. The CPU then saves its state, processes the interrupt (via a top-half handler that acknowledges and disables further interrupts) and eventually runs the bottom-half handler (softirq) that processes packets from the RX buffer. The kernel thread `ksoftirqd` further manages these softirqs by allocating sk_buff structures for packets.

These layers of interrupt handling and buffer management fundamentally limit the maximum throughput, regardless of scaling efforts, highlighting the need for fast I/O techniques.

D. Kernel Bypass Techniques

To achieve the performance required in high-speed networking, several kernel bypass strategies have been developed:

- **Kernel Bypass via AF_XDP:** Traditional sockets (AF_INET) process packets after full TCP/IP handling, but AF_XDP sockets allow packets to be received directly from the XDP hook—bypassing many of the CPU-intensive steps in the kernel. By redirecting packet DMA directly into user space, our system minimizes kernel stack processing overhead, resulting in higher throughput.
- **eBPF and XDP:** Extended Berkeley Packet Filter (eBPF) programs, running at early hook points such as XDP, verify, JIT compile, and execute user code within the kernel. This allows us to filter packets, in our case DNS queries, immediately as they arrive, reducing unnecessary processing.
- **DPDK:** Although Data Plane Development Kit (DPDK) employs a poll-mode driver for even greater optimizations by processing batches of packets in user space with pre-allocated buffers, our approach with AF_XDP offers better integration with standard Linux kernel tools and preserves TCP/IP stack benefits.

E. Anatomy of a Linux DNS Lookup

Understanding the anatomy of a traditional Linux DNS lookup is crucial for appreciating the improvements our solution introduces. Unlike a dedicated system call for DNS resolution, Linux employs a multi-stage process where various system components and configuration files interact to determine the IP address for a given domain name. Instead of a singular "DNS lookup" call, many applications rely on library functions such as `getaddrinfo`, and each application (for example, `ping` versus `host`) may follow a slightly different procedure based on how they consult configuration files like `/etc/nsswitch.conf` and `/etc/resolv.conf`.

Initially, when an application initiates a DNS query, the resolver consults the `/etc/nsswitch.conf` file, which specifies the order in which different name resolution sources are used. The `hosts` entry in this file may include sources such as `files`, `dns`, or even `myhostname`. This modular approach allows the system to first check local files (like `/etc/hosts`) before querying external DNS servers.

If the hostname is not resolved via local files, the resolver then turns to the `/etc/resolv.conf` file. This file plays a central role in Linux's resolver system by specifying the DNS servers to contact through the `nameserver` directives. For example, a typical `/etc/resolv.conf` might include a line like `nameserver 10.0.2.3`, which guides the resolver to query that particular server. Alterations in this file can immediately impact DNS behavior—if the nameserver is missing or misconfigured, lookups will fail.

Thus, when a DNS lookup is performed, the process unfolds as follows: the application calls a resolver function, which first follows the sequence defined in `/etc/nsswitch.conf` by checking for entries in `/etc/hosts`. If the hostname remains unresolved, the resolver then reads `/etc/resolv.conf` to identify and query the appropriate DNS server. Each of these stages—reading multiple configuration files and making external queries—introduces latency. This layered, complex process demonstrates not only the flexibility of Linux's resolver system but also why such multiple steps can be detrimental to performance in high-speed environments, thereby motivating our kernel bypass strategy for DNS resolution.

III. Technical Details of BPF Programs, BPF Maps, and AF_XDP Sockets

A. BPF Programs and Maps

The Berkeley Packet Filter (BPF) is an in-kernel virtual machine that enables the execution of lightweight, sandboxed programs for packet processing. In our DNS resolver project, BPF programs are written in a restricted C-like syntax and compiled into eBPF bytecode using tools like `clang` with `libbpf`. These programs are attached to the eXpress Data Path (XDP) hook, allowing packet filtering and manipulation at the earliest stage of the network stack, directly at the Network Interface Card (NIC) driver level. This minimizes latency and CPU overhead by bypassing traditional kernel processing.

BPF maps are key-value stores within the kernel that facilitate data sharing between BPF programs and user-space applications or between multiple BPF programs. In our implementation, the `xsks_map` (defined as `BPF_MAP_TYPE_XSKMAP`) maps RX queue indices to file descriptors of AF_XDP sockets. This map, initialized with a maximum of 64 entries and pinned using `LIBBPF_PIN_BY_NAME`, enables the XDP program to redirect packets to the appropriate user-space socket based on the queue index.

B. AF_XDP Sockets and Queue Mechanisms

AF_XDP (Address Family eXpress Data Path) sockets provide a high-performance interface for user-space applications to directly interact with the NIC, bypassing much of the kernel network stack. This is achieved through a set of queues and a user memory region (UMEM) that manage packet data efficiently. The key components include:

- **RX Queue**: The receive queue (`xsk_ring_cons rx`) stores incoming packet descriptors redirected by the XDP program. Each descriptor contains the address and length of the packet data in the UMEM.
- **TX Queue**: The transmit queue (`xsk_ring_prod tx`) holds descriptors for packets to be sent out. The user-space application populates these descriptors with UMEM addresses and submits them for transmission.
- **Fill Queue**: The fill queue (`xsk_ring_prod fq`) supplies the kernel with free UMEM frames for receiving new packets. The user-space application reserves frames, fills the queue, and submits them to ensure a continuous supply of buffers.
- **Completion Queue**: The completion queue (`xsk_ring_cons cq`) notifies the user-space application when transmitted packets are completed, allowing the reclamation of UMEM frames for reuse.
- **UMEM**: The user memory region is a contiguous memory buffer (e.g., 4096 frames of 4096 bytes each in our code) allocated by the user-space application and shared with the kernel. It stores raw packet data, with frames indexed by addresses managed by the application.

The UMEM is configured using `xsk_umem_create`, and frames are allocated and freed using functions like `xsk_alloc_umem_frame` and `xsk_free_umem_frame`.

C. Packet Filtering at the XDP Hook

In our implementation, the BPF program (`xdp_dns_filter_func` in `dns_filter_kern.c`) is attached to the XDP hook to filter packets. The process begins with a header cursor (`nh.pos`) initialized at the packet's data start. The `parse_ethhdr` function checks the Ethernet header to determine if the packet is IPv4 (`ETH_P_IP`) or IPv6 (`ETH_P_IPV6`). For IPv4, `parse_iphdr` verifies the protocol is UDP (`IPPROTO_UDP`), while for IPv6, `parse_ip6hdr` performs a similar check. The `parse_udphdr` function then confirms the UDP destination port is 53 (DNS port), identifying DNS queries.

If a packet matches these criteria and an AF_XDP socket is bound to the RX queue (checked via `bpf_map_lookup_elem` on `xsks_map`), the packet is redirected to user space using `bpf_redirect_map`. Otherwise, it is passed to the kernel stack with `XDP_PASS`. This early filtering reduces unnecessary processing, aligning with our goal of minimizing latency.

D. Packet Movement and UMEM Management

Packets enter the system via the NIC and are processed by the XDP program. Filtered DNS packets are redirected to the AF_XDP socket's RX queue, where descriptors point to UMEM frames containing the raw packet data. In user space, `af_xdp_user.c` polls the RX queue (`xsk_ring_cons__peek`) and processes packets using `process_packet`. The application reserves free UMEM frames from the fill queue, submits them to the kernel, and uses the completion queue to reclaim frames after transmission.

For transmission, the application constructs a response, updates the packet in the UMEM, reserves a TX queue slot (`xsk_ring_prod__reserve`), and submits the descriptor. The `complete_tx` function monitors the completion queue to free frames, ensuring efficient UMEM reuse. This cycle—filling, receiving, processing, and transmitting—leverages the queue structure and UMEM to achieve high throughput and low latency.

IV. Implementation

Our DNS stub resolver is implemented using a two-pronged approach. One component runs in kernel space as an XDP program to perform early packet filtering, and the other operates in user space to process DNS queries, perform lookups, and generate responses. This section describes how the new changes are incorporated into both components.

A. Kernel Space: Packet Filtering with XDP

The kernel-space component is implemented in `dns_filter_kern.c`. Its primary objective is to filter incoming packets directly at the NIC level, thereby bypassing unnecessary kernel processing whenever possible. Key steps include:

- **Header Parsing:** The XDP program, defined in the function `xdp_dns_filter_func`, begins by initializing a header cursor that parses the incoming packet's Ethernet header using `parse_ethhdr()`. It then determines whether the packet is IPv4 or IPv6 by inspecting the Ethernet type.
- **IP and UDP Validation:** Depending on the IP version, either `parse_iphdr()` or `parse_ip6hdr()` is invoked to extract the IP header. The program ensures that the packet is UDP by comparing the protocol field (or the IPv6 `nexthdr` field) against `IPPROTO_UDP`. It then uses `parse_udphdr()` to parse the UDP header and retrieve the UDP payload length.
- **DNS Query Filtering:** To process only DNS queries, the XDP function checks that the UDP destination port is 53 (DNS). If this check passes and if the appropriate AF_XDP socket is bound (determined by a lookup in the BPF map `xsks_map` using the current RX queue index), the program redirects the packet to user space via `bpf_redirect_map()`. Otherwise, the packet is passed up to the kernel using `XDP_PASS`.

This early filtering mechanism minimizes kernel intervention—by redirecting DNS packets to user space at the very

first opportunity, we save valuable processing time and reduce context switching overhead.

B. User Space: DNS Query Processing via AF_XDP Sockets

The user-space component is implemented in `af_xdp_user.c`. This component is responsible for setting up the AF_XDP socket, managing packet buffers (UMEM), processing incoming DNS queries, and constructing the corresponding DNS responses. The overall flow is as follows:

- **Socket and UMEM Setup:** The application allocates a large contiguous buffer divided into a set number of frames (defined by `NUM_FRAMES` and `FRAME_SIZE`). This memory is then registered as UMEM via `configure_xsk_umem()`. An AF_XDP socket is configured and bound to a specified RX queue on the network interface using `xsk_configure_socket()`.
- **Polling and Packet Reception:** After initializing the socket, the program enters a polling loop (implemented in `rx_and_process()`) that waits for DNS packets redirected by the XDP program. Incoming packets are processed in batches via the RX ring.
- **DNS Query Processing:** For each received packet, `process_packet()` is invoked. The packet is interpreted by accessing its UMEM data and parsing the Ethernet, IP, and UDP headers. The DNS query is extracted (using helper functions like `parse_dns_query_name()`), and a DNS lookup is initiated by the function `resolve_and_log_ip()`. In this function, the application checks an on-disk FIFO-based JSON cache (`dns_bin_map.json`) allowing 1024 DNS payload entries mapped against domain names for a previously cached binary DNS response. In case of a cache miss, the program creates a UDP socket to send a query to an upstream resolver (e.g., 8.8.8.8), receives the response, caches the binary response, and returns it.
- **Packet Modification and Transmission:** Once the DNS response is obtained, the original packet is modified in place. This modification includes copying the DNS response payload into the packet, updating UDP and IP lengths, and recomputing checksums for both IPv4 and IPv6. Additionally, the Ethernet source and destination addresses, as well as the IP source and destination addresses, are swapped to prepare the packet for response back to the client. The modified packet is then queued for transmission via the AF_XDP socket's TX ring.
- **Statistics and UMEM Replenishment:** Alongside packet processing, the code maintains per-socket and per-CPU statistics, which are periodically printed out by a dedicated statistics thread. Furthermore, after processing, the application replenishes the fill ring with free UMEM frame descriptors, ensuring there is a continuous supply of buffers.

C. Integration and Overall Workflow

The complete workflow of the system is as follows:

1) Packets arrive at the NIC and are intercepted by the XDP program in the kernel.
2) The XDP program parses headers and filters UDP packets destined for DNS (port 53). If a matching packet is found and an AF_XDP socket is active on that RX queue, the packet is redirected to user space.
3) In user space, the application polls the AF_XDP socket, retrieves DNS query packets, and processes them as described earlier.
4) DNS queries are either resolved via a cached binary response or by sending a UDP query to an upstream DNS resolver.
5) The DNS response is inserted into the packet, headers (lengths and checksums) are updated, and source/destination addresses are swapped.
6) The packet is queued and transmitted via the AF_XDP socket's TX ring, effectively responding to the original client.

This dual-component implementation leverages the speed of kernel-level packet filtering via XDP and the flexibility of user-space processing with AF_XDP sockets. The architecture ensures that only DNS-related traffic is processed by the user-space application, thereby reducing overhead and achieving high throughput with low latency.

V. Testing

To evaluate the performance and functionality of our DNS stub resolver, we conducted a series of tests in a controlled virtualized environment using a pair of virtual Ethernet (veth) interfaces. The testing setup consisted of two hosts: a Host machine (configured with IPv6 address `fc00:dead:cafe:1::1` and IPv4 address `10.11.1.1`) running the DNS stub resolver, and a Virt machine (configured with IPv6 address `fc00:dead:cafe:1::2` and IPv4 address `10.11.1.2`) acting as the client. This section details the commands executed on both machines, their purpose, and how they were used to verify the resolver's functionality and measure its performance.

A. Testing Environment Setup

The testing environment was established using a virtualized setup with veth interfaces to simulate a network link between the Host and Virt machines. The Host machine ran the XDP-based DNS stub resolver, while the Virt machine generated DNS queries to test the resolver's response time, correctness, and throughput. The commands executed during testing are categorized by the machine on which they were run and explained below.

B. Commands Executed on the Host Machine

The following commands were used to compile, configure, and run the DNS stub resolver on the Host machine:

- **`make`**

This command compiles the kernel-space XDP program (`dns_filter_kern.c`) and the user-space application (`af_xdp_user.c`) using a Makefile. It generates the object file `dns_filter_kern.o`, which contains the eBPF bytecode for the XDP program, and the executable `af_xdp_user` for the user-space component.

- **`eval $(../testenv/testenv.sh alias)`**
  This command sources a shell script (`testenv.sh`) to set up environment aliases and variables for the testing framework. It configures shortcuts (e.g., `t` for test environment commands) to simplify the execution of test-related tasks, such as interface setup and status checks.
- **`t setup --name veth`**
  This command initializes a virtual Ethernet interface named `veth` using the test environment framework. It creates a pair of veth interfaces to simulate a network link between the Host and Virt machines, enabling packet exchange for testing the DNS resolver.
- **`t setup --name veth --legacy-ip`**
  This variant of the setup command configures the `veth` interface with legacy IP addressing (IPv4 and IPv6 addresses as specified: `10.11.1.1` and `fc00:dead:cafe:1::1` for the Host). The `--legacy-ip` flag ensures compatibility with traditional IP configurations, which is necessary for testing both IPv4 and IPv6 DNS queries.
- **`sudo ./af_xdp_user -d veth --filename dns_filter_kern.o --progname xdp_dns_filter_func`**
  This command launches the user-space DNS stub resolver application with superuser privileges. The flags specify:
  - `-d veth`: The network interface (`veth`) to which the XDP program is attached.
  - `--filename dns_filter_kern.o`: The eBPF object file containing the XDP program.
  - `--progname xdp_dns_filter_func`: The name of the XDP function (`xdp_dns_filter_func`) to be loaded and executed by the kernel.

  This command attaches the XDP program to the `veth` interface, enabling packet filtering for DNS queries (UDP port 53) and redirecting them to the user-space application via AF_XDP sockets.

C. Commands Executed on the Virt Machine

The Virt machine acted as the client, generating DNS queries to test the resolver running on the Host. The following commands were executed:

- **`sudo tcpdump -i veth0 -w cap.pcap`**
  This command captures network traffic on the `veth0` interface (the Virt machine's side of the veth pair) and saves it to a file named `cap.pcap`. It was used to record DNS query and response packets for offline analysis, helping verify the correctness of the resolver's behavior.
- **`dnsperf -s fc00:dead:cafe:1::1 -d queries.txt`**
  This command runs the `dnsperf` benchmarking tool to send DNS queries to the resolver at the IPv6 address `fc00:dead:cafe:1::1` (Host). The `-d queries.txt` flag specifies a file containing a list of DNS queries to be sent. This test measured the resolver's throughput and latency for IPv6-based queries, including cache-hit scenarios (as shown in Figure 5).
- **`dnsperf -s 10.11.1.1 -d queries.txt`**
  Similar to the previous command, this runs `dnsperf` but targets the resolver's IPv4 address (`10.11.1.1`). It evaluated the resolver's performance for IPv4-based queries, ensuring compatibility and performance consistency across both IP versions.
- **`dig @fc00:dead:cafe:1::1 www.iitgn.ac.in`**
  This command uses the `dig` tool to send a DNS query for `www.iitgn.ac.in` to the resolver at the IPv6 address `fc00:dead:cafe:1::1`. It was used to verify the correctness of the resolver's response for a specific domain over IPv6, checking both cache-hit and cache-miss scenarios.
- **`dig @10.11.1.1 www.iitgn.ac.in`**
  This command is analogous to the previous one but sends the DNS query to the resolver's IPv4 address (`10.11.1.1`). It confirmed the resolver's ability to handle IPv4 queries correctly and consistently with the IPv6 results.

D. Testing Methodology and Observations

The testing process involved several steps to ensure comprehensive evaluation:

1) **Functional Testing:** The `dig` commands verified that the resolver correctly resolved DNS queries for `www.iitgn.ac.in` over both IPv4 and IPv6, with responses matching expected IP addresses.
2) **Performance Testing:** The `dnsperf` commands measured the resolver's throughput and latency under varying query loads, with results compared between direct queries to an upstream DNS server (e.g., 8.8.8.8) and cache-hit queries served by our resolver (as shown in Figure 5). The cache-hit version demonstrated significantly lower latency due to the on-disk JSON cache.
3) **Traffic Analysis:** The `tcpdump` and `tcpreplay` commands allowed us to capture and replay DNS traffic, ensuring the resolver handled packets correctly and efficiently under simulated conditions.

The latency comparison in Figure 5 highlights the performance advantage of our cache-hit resolver, which bypasses upstream queries and leverages the AF_XDP-based architecture to minimize processing overhead. These tests collectively validated the resolver's functionality, performance, and robustness in a high-speed networking environment.

Notes on Integration The section assumes the presence of Figure 5 (already included in your document) and refers to it for latency comparisons. The explanations are tailored to fit the IEEEtran format, maintaining a formal and technical

tone. The commands are grouped by machine (Host and Virt) for clarity, with each command's purpose and role in testing clearly described. If you have specific performance metrics (e.g., throughput numbers, latency values) or additional details about the test results, they can be incorporated into the Results section or summarized here. If you need further refinements, additional details, or assistance with the Results section, please let me know!

VI. Results

This section presents the experimental results, performance metrics, and analysis of the DNS stub resolver.

```
cap_first.pcap:
  Average Latency = 26.322772 seconds
  Median Latency  = 25.766722 seconds

cap_cache.pcap:
  Average Latency = 0.512208 seconds
  Median Latency  = 0.022440 seconds
```

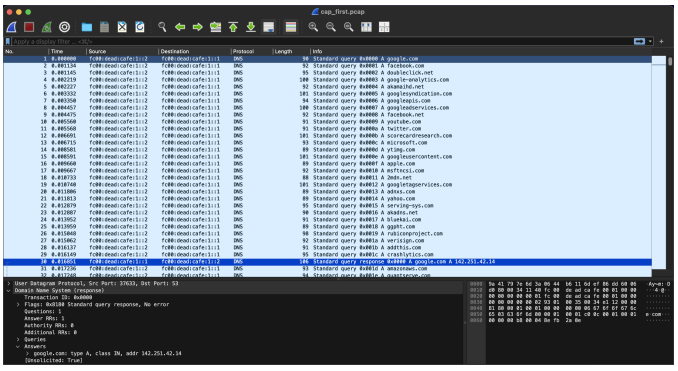

Fig. 1. Log of packets in wireshark responded back on the first-hit to out stub-resolver.

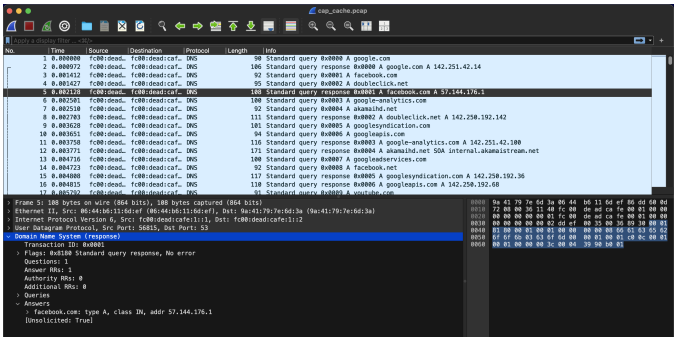

Fig. 2. Log of packets in wireshark responded back on the cache-hit to out stub-resolver.

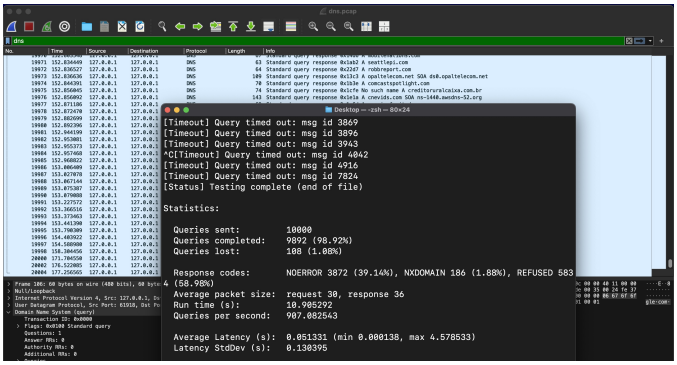

Fig. 4. dnsperf on the loopback dns resolver using dnsmasq.

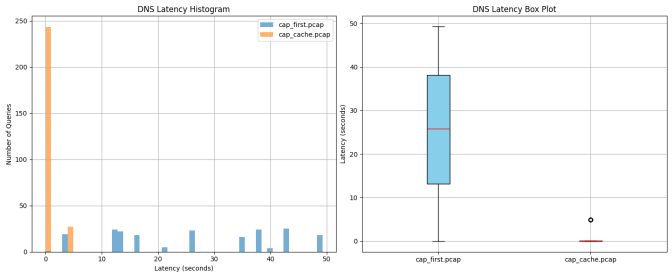

Fig. 3. Latency Time Comparison of First-Hits to the DNS Server vs. Our Cache-hit version.

```
dns.pcap:
  Average Latency = 2.305389 seconds
  Median Latency  = 1.328159 seconds

cap_cache.pcap:
  Average Latency = 0.512208 seconds
  Median Latency  = 0.022440 seconds
```

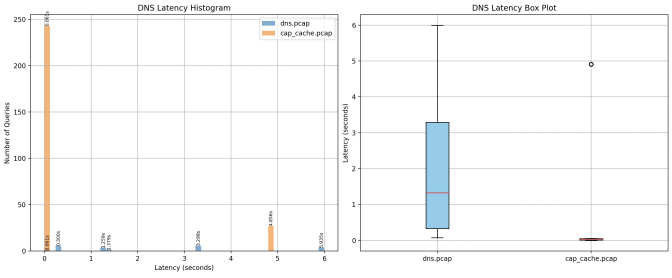

Fig. 5. Latency Time Comparison of Direct DNS query to the DNS Server vs. Our Cache-hit version.

References

[1] A. Stephan and L. Wüstrich, "The path of a packet through the linux kernel,"

[2] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, (New York, NY, USA), p. 65–77, Association for Computing Machinery, 2021.

[3] DNS-OARC, "dnsperf: Gather accurate latency and throughput metrics for Domain Name Service (DNS)." https://github.com/DNS-OARC/dnsperf.

[4] Libvirt Project, “libvirt: An Open-Source API, Daemon, and Management Tool for Platform Virtualization.” https://libvirt.org/.
[5] “Domain names - implementation and specification.” RFC 1035, Nov. 1987.
[6] XDP Project, “The eXpress Data Path (XDP) inside the Linux kernel.” https://github.com/xdp-project.