Performance Profiling Tool for Linux using eBPF (Extended Berkeley Packet Filter)

Sriram Srinivasan*, Nikhilesh Myanapuri†, Srivathsa Vamsi Chaturvedula‡, Ganesh Koleti§, Sameer Kulkarni¶
Computer Networks Course, Indian Institute of Technology Gandhinagar (IITGN)
Gandhinagar, Gujarat, India
*sriram.srinivasan@iitgn.ac.in (22110258), †nikhilesh.myanapuri@iitgn.ac.in (22110162)
‡vamsi.srivathsa@iitgn.ac.in (22110260), §eswar.koleti@iitgn.ac.in (22110123)
¶sameergk@iitgn.ac.in (Course Instructor)

***Abstract*—Modern applications demand high performance and efficiency. Understanding their behavior at the system level is crucial for optimization and troubleshooting. Traditional monitoring tools often introduce significant overhead or lack the necessary granularity. This project presents the design and implementation of a high-fidelity, low-overhead profiling tool leveraging the power of eBPF (extended Berkeley Packet Filter) within the Linux kernel. The tool collects a diverse range of performance metrics from applications, including system call latencies, network packet transmission times within the kernel, CPU scheduling behavior, memory usage patterns (page faults, kernel allocations), I/O statistics, and page cache interactions. The collected data is processed by a user-space Python application featuring a graphical user interface (GUI) built with PySide6, which allows users to select specific metrics, configure monitoring parameters, and initiate profiling. The application employs worker threads to manage the execution of individual eBPF metric scripts (developed using the BCC framework) and incorporates robustness features like data buffering, periodic transmission, and retries. Performance data is transmitted via HTTP POST requests in JSON format to a remote Flask server for logging and potential further analysis. This report details the system architecture, the implementation of the core eBPF metric probes, the design of the user-space application, and the data transmission mechanism, demonstrating the effectiveness of eBPF for granular performance monitoring.**

***Index Terms*—eBPF, Network Monitoring, Performance Analysis, Linux Kernel, System Tracing, BCC.**

I. Introduction

A. Problem Statement

Optimizing the performance of applications is critical in today's distributed systems. Developers and system administrators require tools that can provide deep insights into application behavior, including interactions with the operating system kernel, network stack, memory subsystem, and I/O devices. Traditional tools like system-level profilers (`perf`, `top`) or network sniffers (`tcpdump`) offer valuable information but may lack the ability to correlate events across different subsystems efficiently or may introduce unacceptable performance overhead, altering the very behavior they are trying to measure.

B. Motivation and eBPF

The emergence of eBPF (extended Berkeley Packet Filter) technology in the Linux kernel provides a powerful and safe mechanism for in-kernel monitoring and observability. eBPF allows custom, event-driven programs to run directly within the kernel's context without requiring kernel module loading or kernel recompilation. Its JIT (Just-In-Time) compilation and in-kernel verifier ensure safety and efficiency. This makes eBPF particularly well-suited for building high-performance profiling tools that can capture fine-grained data with minimal overhead. Our motivation was to leverage eBPF to create a flexible and comprehensive profiling tool.

C. Project Goals and Scope

The primary goal of this project was to develop an eBPF-based profiling tool capable of:

1) Collecting diverse performance metrics to monitor applications using eBPF probes. Metrics include:
   - System Call Latencies and Counts
   - TCP Connection Latency
   - Network Packet Transmission Time
   - CPU Usage and Scheduling Behavior (Run queue latency, context switches)
   - Memory Usage (Page Faults, Kernel Allocations)
   - Page Cache Hit/Miss Ratios
   - Block I/O Statistics per Process
2) Providing a user-friendly Graphical User Interface (GUI) for selecting metrics, configuring parameters (like PID filtering), and controlling the monitoring process.
3) Executing individual eBPF metric scripts reliably in the background.
4) Transmitting the collected performance data to a remote server for centralized logging or analysis.
5) Implementing basic robustness mechanisms for data transmission.

The scope includes the development of the eBPF scripts using the BCC framework, the Python-based GUI application, the worker mechanism, and a simple remote server endpoint. Advanced server-side analysis and visualization are considered outside the current scope.

II. Background and Related Work

A. Performance Monitoring Techniques

Performance analysis relies on various techniques. Profiling typically involves sampling the program counter or specific

events to understand where time is spent. Tracing involves recording specific events (like function calls, system calls, or network packets) as they occur, often providing detailed causal information. Tools like `gprof`, `perf`, and `valgrind` represent different approaches to profiling and tracing, each with trade-offs in terms of overhead, granularity, and ease of use.

B. eBPF Fundamentals

eBPF revolutionizes Linux tracing and monitoring. It allows sandboxed, event-driven programs to run within the kernel. Key concepts include:

- **Hooks:** Points in the kernel where eBPF programs can be attached (e.g., kprobes, kretprobes, tracepoints, socket filters, TC classifiers).
- **Maps:** Key-value stores residing in kernel memory, used for communication between eBPF programs and userspace, or between different eBPF programs. Common types include hash maps, arrays, histograms, and perf/ring buffers for event streaming.
- **Helper Functions:** A predefined set of kernel functions callable from eBPF programs (e.g., `bpf_ktime_get_ns()`, `bpf_get_current_pid_tgid()`, `bpf_perf_event_output()`).
- **Verifier:** Ensures eBPF programs are safe to run by checking for unbounded loops, out-of-bounds memory access, and illegal instructions before loading.

eBPF's main advantages are its safety, low overhead (compared to traditional kernel modules or context-switching tools like `strace`), and programmability, allowing for custom, targeted metric collection.

C. BCC Framework

The BPF Compiler Collection (BCC) is a toolkit for creating kernel tracing and manipulation programs, primarily using eBPF. It simplifies eBPF development by embedding BPF C code within Python (or Lua) scripts. BCC handles the compilation of the C code to BPF bytecode using LLVM, loading the bytecode into the kernel, creating maps, and attaching programs to hooks. Its Python interface provides easy access to BPF maps and perf/ring buffers, making it ideal for rapid prototyping and developing complex userspace control logic, which led to its selection for this project.

D. Existing Tools

Numerous performance tools exist, including standard Linux utilities (`top`, `vmstat`, `iostat`, `strace`, `ss`), kernel tracing infrastructure (`perf`, `ftrace`), and a rich ecosystem of BCC/eBPF tools (many developed by Brendan Gregg, such as `opensnoop`, `execsnoop`, `biosnoop`). While these tools are powerful, our project aimed to create an integrated application that combines multiple relevant metrics, provides a GUI for easier control, and includes built-in remote data forwarding, offering a tailored solution for networked application analysis within the scope of this course project.

III. System Design and Architecture

A. Overview

The system consists of a user-space control application with a GUI, a set of independent eBPF metric scripts, worker threads to manage script execution, and a simple remote server for data ingestion. The user selects a metric script and provides parameters via the GUI. The application launches the script in a separate worker thread, which executes the BCC Python script. The BCC script loads and attaches the eBPF C program to relevant kernel hooks. The eBPF program collects data and pushes it to userspace (via maps or perf buffers). The worker thread reads this data from the BCC script's output, parses it, and sends it back to the main GUI thread. The main GUI thread buffers the data and periodically transmits it to the configured remote server.

B. Components

- **3.2.1. eBPF Metric Scripts (Kernel Probes):** These are Python scripts (`metrics/*.py`) utilizing the BCC framework. Each script contains embedded BPF C code designed to attach to specific kernel tracepoints or kprobes to capture raw performance data (timestamps, counts, identifiers). They use BPF maps or perf buffers to communicate results back to the Python userspace portion of the script, which typically formats and prints the data to standard output.
- **3.2.2. User-Space Control Application (GUI):** Implemented in Python (`main_app.py`) using the PySide6 library (`ui_main_window.py`). It provides the user interface for selecting scripts, entering arguments dynamically based on the selected script's configuration, setting the remote server URL, starting/stopping monitoring, and displaying status logs.
- **3.2.3. Worker Threads:** Implemented in `worker.py`. Each time monitoring starts, a new `Worker` object is created and run in a dedicated thread. This worker launches the selected BCC script as a subprocess, captures its standard output and standard error, parses the output based on the script's configuration, and emits signals (`output_ready`, `error_occurred`, `process_finished`) to communicate back to the main GUI thread using Qt's signal/slot mechanism. This prevents the GUI from freezing during script execution.
- **3.2.4. Data Transmission Module:** Integrated within `main_app.py`. It uses the `requests` library to send buffered data as JSON payloads via HTTP POST requests to the remote server. It includes timers for periodic sending, a backup timer, and logic for retrying failed transmissions.
- **3.2.5. Remote Server:** A simple Flask web server (`server.py`) listening for POST requests on a specific endpoint (`/data`). It receives the JSON payload, logs the incoming data (including metadata like source script and batch ID), and tracks sequence numbers to detect potential data loss.

C. Data Flow

1) Kernel events trigger attached eBPF C programs.
2) eBPF programs collect data (e.g., timestamps, counters) and store it in BPF maps or send it via perf buffers.
3) The Python part of the BCC script reads data from maps/buffers, processes/formats it, and prints it to `stdout`.
4) The `Worker` thread captures the `stdout` lines from the BCC script subprocess.
5) The `Worker` parses the line (using the script's configured `output_parser`) and emits the parsed data via a Qt signal to the `MainWindow`.
6) The `MainWindow` receives the data and appends it to a `deque` buffer.
7) Periodically (or based on buffer size/backup timer), the `MainWindow` takes data from the buffer, packages it into a JSON payload (with metadata), and sends it via HTTP POST to the remote server.
8) The remote Flask server receives the request, logs the data, and returns a success/error response.
9) If the transmission fails, the payload is stored for later retry attempts.

D. Technology Stack

- **Kernel Probing:** eBPF
- **eBPF Scripting:** BCC (BPF Compiler Collection) framework
- **Core Languages:** Python 3, C (for BPF programs)
- **GUI:** PySide6 (Qt for Python)
- **HTTP Client:** `requests` library (Python)
- **Remote Server:** Flask framework (Python)
- **Operating System:** Linux (requires appropriate kernel version and headers for eBPF/BCC)

IV. Implementation Details

This section details the implementation of the major components, with a strong emphasis on the eBPF metric scripts.

A. User-Space Application (`main_app.py`, `ui_main_window.py`)

The main application orchestrates the entire process. The UI provides widgets for script selection (`QComboBox`), dynamic argument input (`QFormLayout` populated with `QLineEdit`), server URL configuration (`QLineEdit`), control buttons (`QPushButton`), and logging (`QTextEdit`).

`main_app.py` handles the application logic:

- **Initialization:** Populates the script dropdown from `config.py`, sets up timers (`QTimer`) for data sending and retries, initializes the output buffer (`deque`).
- **Dynamic Argument UI:** The `update_argument_fields` slot clears and repopulates the argument form layout whenever the selected script changes, based on the definitions in `config.py`.
- **Monitoring Control:** `start_monitoring` retrieves arguments, validates required fields, creates and starts a `Worker` instance, and updates the UI state. `stop_monitoring` signals the worker to terminate its subprocess, stops timers, attempts a final data send, and resets the UI.
- **Data Handling:** The `handle_output` slot receives parsed data from the worker and adds it to the `output_buffer`. `send_data_to_server` packages buffered data into JSON and POSTs it. `retry_failed_transmissions` attempts to resend previously failed batches.
- **Error Handling:** `handle_error` logs errors from the worker. `handle_finish` cleans up resources when a worker thread completes. Robustness features like increased buffer size, sequence numbers, and backup timers were added to enhance reliability.

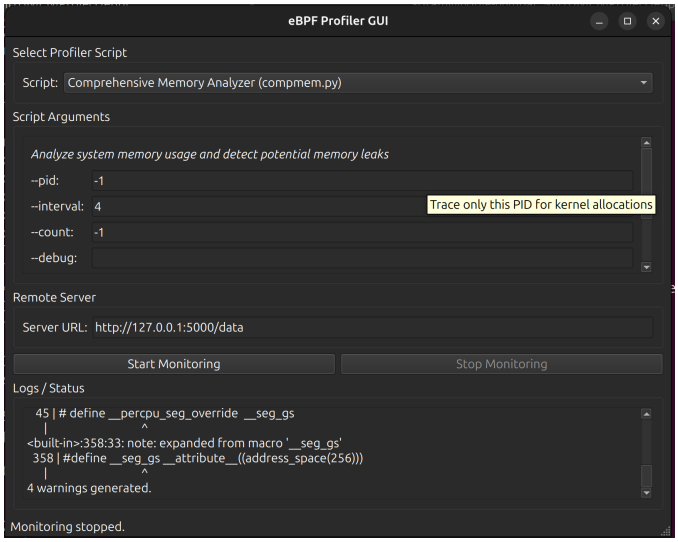

Fig. 1. User Interface

B. Worker Thread (`worker.py`)

The `Worker` class runs a BCC script in isolation.

- **Subprocess Management:** It uses `subprocess.Popen` to execute the target Python BCC script (`python3 metrics/script_name.py <arg> <value> ...`). `stdout` and `stderr` are piped for capture.
- **Output Capture:** It uses a non-blocking `select.select` loop on the subprocess's `stdout` to read output lines as they become available without blocking the worker thread.
- **Parsing:** Each captured line is passed to the script-specific `output_parser` lambda function defined in `config.py` to structure the raw output line (e.g., into a dictionary) before emitting it via the `output_ready` signal.
- **Termination:** The `stop` method attempts graceful termination (`terminate()`) followed by a forced kill (`kill()`) if necessary, ensuring the subprocess is cleaned up. It joins the thread to wait for completion and emits the `process_finished` signal.

C. Configuration (`config.py`)

This file acts as a central registry for the available eBPF metric scripts. It defines a list (`AVAILABLE_SCRIPTS`) where each entry is a dictionary containing:

- `name`: User-friendly name for the script (shown in the GUI).
- `file`: The path to the BCC Python script.
- `description`: A brief explanation displayed in the GUI.
- `args`: A list defining the command-line arguments the script accepts, including name, type (for basic validation), requirement status, default value, and help text (used for tooltips). This allows the GUI to dynamically generate the correct input fields.
- `output_parser`: A lambda function specific to the script, responsible for converting a single line of the script's `stdout` into a structured format (typically a dictionary) suitable for buffering and JSON serialization.

D. Remote Server (`server.py`)

The server is a minimal Flask application providing a single `/data` endpoint that accepts HTTP POST requests.

- **Data Reception:** It expects a JSON payload containing `timestamp`, `source_script`, `batch_id`, `sequence`, and `metrics` (a list of parsed data points).
- **Logging:** It logs received data batches to both console and a file (`server_data.log`), including metadata and individual metrics for inspection.
- **Sequence Tracking:** It maintains a dictionary (`sequence_tracker`) to store the last received sequence number for each source script, logging a warning if a gap is detected, indicating potential data loss during transmission.
- **Response:** It returns a JSON response indicating success or failure.

E. eBPF Metric Scripts (`metrics/`)

1) General Approach: Most scripts in the `metrics/` directory follow the BCC pattern:

1) **Python Userspace:** Handles argument parsing, defines and loads the BPF C program text, attaches probes (kprobes/tracepoints) specified in the C code, interacts with BPF maps or perf buffers, processes the retrieved kernel data, and prints formatted output to `stdout`.
2) **BPF C Code (Embedded):** Contains the logic executed in the kernel. It includes necessary headers, defines data structures (for map keys/values), declares BPF maps, and implements functions attached to kernel hooks. These functions capture event data (timestamps, PIDs, relevant arguments), perform calculations, and store/submit results using maps or perf buffers.

2) System Call Analysis (`sysnest.py`):

- **Goal:** Summarize system call counts and latencies, allowing grouping by process or syscall, and showing nested details (top syscalls per process or top processes per syscall).
- **eBPF Probes:** `raw_syscalls:sys_enter` and `raw_syscalls:sys_exit` tracepoints. These provide stable hooks for capturing entry and exit of nearly all system calls.
- **eBPF Logic & Maps:**
  - `BPF_HASH(start, pid_tgid, timestamp_ns)`: On `sys_enter`, stores the entry timestamp (`bpf_ktime_get_ns()`) keyed by the thread ID.
  - `BPF_HASH(data, key_t{pid, syscall_nr}, data_t{count, total_ns})`: On `sys_exit`, retrieves the start time from `start`, calculates duration, deletes the `start` entry, and updates the count and total nanoseconds in the `data` map for the specific (PID, syscall) combination using `lock_xadd` for safe concurrent updates.
  - `key_t`: Contains PID and syscall number. `data_t`: Holds aggregate count and total duration in nanoseconds.
- **Userspace Interaction:** Periodically reads and clears the `data` map. It then aggregates this raw data based on the `group_by` argument (either process-first or syscall-first), sorts the results, and prints the hierarchical summary. Uses `syscall_name()` from BCC utils to resolve syscall numbers to names.
- **Output:** Prints tables grouped by process (showing top syscalls within each) or by syscall (showing top processes calling each), including counts and average latencies (calculated from `total_ns` / `count`) in microseconds or milliseconds.

3) TCP Connection Latency (`tcpconnlat-bpfcc.py`):

- **Goal:** Measure the time taken for a TCP active connection to complete, specifically the duration between the initiation of the connection (`tcp_v[46]_connect`) and the transition to the `ESTABLISHED` state (`tcp_rcv_state_process`).
- **eBPF Probes:** Kprobes on `tcp_v4_connect`, `tcp_v6_connect` (entry points) and `tcp_rcv_state_process` (state change handler where SYN-ACK is likely processed, moving state from `TCP_SYN_SENT`).
- **eBPF Logic & Maps:**
  - `BPF_HASH(start, struct sock *, info_t{ts, pid, task})`: On `connect`, stores the start timestamp and process info, keyed by the socket pointer (`struct sock *`).
  - `BPF_PERF_OUTPUT(ipv4_events)` / `BPF_PERF_OUTPUT(ipv6_events)`: On `tcp_rcv_state_process`, if the state was `TCP_SYN_SENT`, it looks up the start info using the socket pointer. It calculates the duration,

```
Tracing syscalls, grouping by syscall, printing top 10 i
^C[22:51:27]
SYSCALL              TOT COUNT        TIME (us)
-----------------------------------------------
futex                    20299    258074820.348
  3259   code                1300     42932426.758
  3332   code                  86     25312599.758
  10074  chrome              1706     22584954.596
  45847  chrome               234     21887116.073
  3765   chrome               286     13833297.536

epoll_wait                3176    127010275.182
  3321   code                  28     14630902.584
  3168   code                 114     13897925.805
  3829   chrome               172     12859378.846
  3765   chrome               454      7869687.472
  3259   code                 472      7662420.284

poll                      2979     98667050.876
  1915   gnome-shell          735     21393978.605
  3275   code                 356     14228235.722
  3827   chrome              1027     12739288.596
  3168   code                 293      7652365.895
  3332   code                  14      7007496.261

clock_nanosleep              4     14674990.865
  46860  sysnest.py             1      7674260.339
  18460  cachetop-bpfcc         1      5000589.113
  46906  [gone]                 1      1000085.419
  46876  [gone]                 1      1000055.994

ppoll                        8     12676645.606
  46834  sudo                   5      7674070.962
  18458  sudo                   3      5002574.644
```

Fig. 2. Tracing latencies grouped by system calls

extracts connection details (IP addresses, ports) from the socket structure, populates a data structure (`ipv4_data_t` or `ipv6_data_t`), and submits it to the corresponding perf buffer for userspace processing. The `start` entry is deleted.

- **Userspace Interaction:** Opens the perf buffers (`ipv4_events`, `ipv6_events`) and polls them. When an event arrives, the callback function (`print_ipv4_event` or `print_ipv6_event`) formats the data (using `inet_ntop` for addresses) and prints the connection details and latency.
- **Output:** Prints one line per completed TCP connection, showing PID, command, IP version, source/destination addresses, source/destination ports (optional LPORT), and the connection latency in milliseconds.

4) Network Packet Kernel Latency (`netpack.py`):

- **Goal:** Measures the time a network packet (`sk_buff`) spends between kernel queueing (`net_dev_queue`) and being handed off to the NIC driver (`net_dev_xmit`), indicating kernel transmit path latency.
- **eBPF Probes:** Utilizes stable tracepoints: `net:net_dev_queue` (packet enqueued) and `net:net_dev_xmit` (packet handed to driver).

TIME(s)	PID	COMM	IP	SADDR	LPORT	DADDR	DPORT	LAT(ms)
0.000	3321	code	4	10.7.5.170	47078	140.82.113.21	443	255.59
2.356	3829	Chrome_Child	4	10.7.5.170	56264	64.90.53.70	443	266.96
2.603	3829	Chrome_Child	4	10.7.5.170	56280	64.90.53.70	443	263.38
10.626	3829	Chrome_Child	4	10.7.5.170	43610	199.232.45.140	443	62.66
10.830	3829	Chrome_Child	4	10.7.5.170	43622	199.232.45.140	443	63.31
11.215	3829	Chrome_Child	4	10.7.5.170	43638	199.232.45.140	443	61.87
11.777	3829	Chrome_Child	4	10.7.5.170	43646	199.232.45.140	443	65.23
12.137	3829	Chrome_Child	4	10.7.5.170	37312	142.250.70.68	443	13.72
12.369	3829	Chrome_Child	4	10.7.5.170	37324	142.250.70.68	443	13.22
12.370	3829	Chrome_Child	4	10.7.5.170	37334	142.250.70.68	443	13.72
12.370	3829	Chrome_Child	4	10.7.5.170	37346	142.250.70.68	443	13.98
12.472	3829	Chrome_Child	4	10.7.5.170	33558	142.250.194.106	443	36.12
12.494	3829	Chrome_Child	4	10.7.5.170	54958	23.203.78.16	443	82.95
12.554	3829	Chrome_Child	4	10.7.5.170	56008	74.125.24.84	443	72.56
12.563	3829	Chrome_Child	4	10.7.5.170	56010	74.125.24.84	443	77.18

Fig. 3. TCP Connection Latencies

- **eBPF Logic & Maps:**
  - `BPF_HASH(startmap, skbaddr, timestamp_ns)`: Stores the enqueue timestamp (`bpf_ktime_get_ns()`) keyed by the packet's SKB address (`skbaddr`).
  - `BPF_PERF_OUTPUT(events)`: Sends latency results and packet info (length, device name) to userspace.
- **Core Logic:**
  - On `net_dev_queue`: Records the packet's entry time in `startmap`.
  - On `net_dev_xmit`: Looks up the packet's `skbaddr` in `startmap`.
  - If found: Retrieves the start time, calculates latency (`now - start_time`), removes the entry from `startmap`, and populates a `data_t` struct (timestamp, latency, length, device name) for submission to the `events` perf buffer.
  - If not found (e.g., missed start event), the packet is ignored.
- **Userspace Interaction:**
  - Python initializes BPF, defines a callback (`print_event`) for processing events.
  - Opens the `events` perf buffer, linking it to the callback.
  - Enters a loop polling the buffer (`b.perf_buffer_poll()`).
  - The callback receives event data, formats it (decodes name, converts ns to us), and prints the output line.
- **Output Explanation:** Prints one line per successfully measured packet, showing the Packet Length (bytes), and the calculated kernel Latency (microseconds).

5) Page Fault Monitoring (`pagefault.py`):

- **Goal:** Count minor (memory-resident) and major (disk-backed) page faults per process.
- **eBPF Probes:** Tracepoint `exceptions:page_fault_user`. This tracepoint provides information about faults occurring in user mode, including an `error_code` indicating fault type.
- **eBPF Logic & Maps:**
  - `BPF_HASH(page_faults, key_t{pid, comm}, val_t{minor, major})`: On `page_fault_user`, creates a key with the

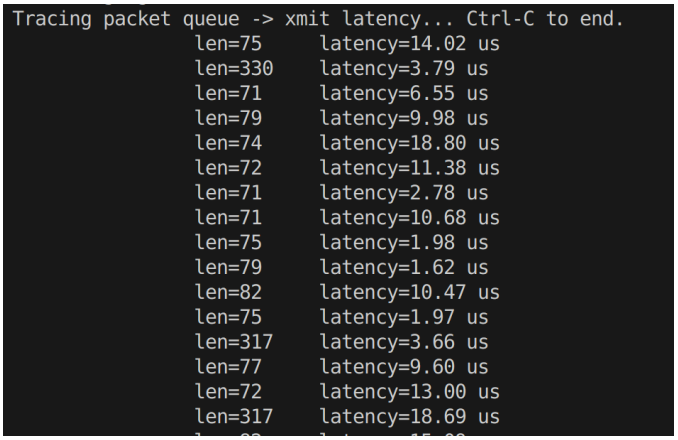

Fig. 4. Network Packet Transmission Latencies Example Output/Visualization

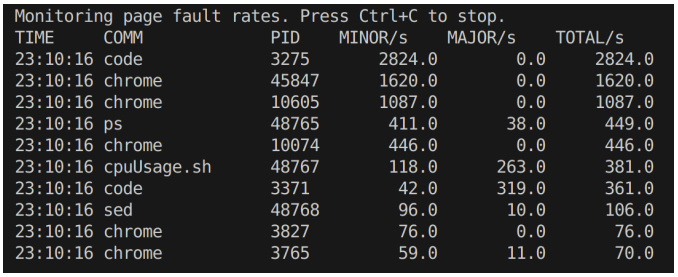

Fig. 5. Monitoring Page Fault Rates

current PID and command name. It looks up or initializes the corresponding entry in the hash map. Based on the `error_code` argument (checking bit 1 for major fault), it increments either the `minor` or `major` counter in the `val_t` structure.

- **Userspace Interaction:** Periodically reads the `page_faults` map. It calculates the delta (change since the last interval) for minor and major faults for each process, stores the current cumulative counts for the next interval, sorts processes by total faults in the interval, and prints the fault rates (delta/interval).
- **Output:** Prints a table showing the command, PID, and the calculated minor, major, and total page fault rates per second for the top faulting processes during each interval.

6) CPU Scheduling and Runtime Analysis (`cpuschrun.py`):

- **Goal:** This script aims to provide a combined view of CPU scheduling behavior and runtime characteristics. It focuses on measuring how long processes spend waiting in the run queue before getting CPU time (run queue latency) and how long their execution bursts (run time) last. It presents this information both as distributions and, optionally, as per-process summaries including a calculated CPU utilization percentage and priority score.
- **eBPF Probes:** The script primarily relies on two scheduler tracepoints:
  - `raw_syscalls:sched_switch`: Fired whenever the kernel switches the currently running task on a CPU. It provides information about the previously running task (`prev`) and the newly scheduled task (`next`).
  - `raw_syscalls:sched_wakeup`: Fired when a task transitions from a sleeping state to a runnable state (i.e., it's added to the run queue).
- **eBPF Logic & Maps:**
  - `BPF_HASH(start, struct proc_key{pid, tgid, cpu}, timestamp_ns)`: This map is central to tracking state durations. It stores the timestamp (`bpf_ktime_get_ns()`) marking the beginning of a state (either waiting in the queue or running on the CPU). The key includes PID, TGID, and CPU to handle threads correctly.
  - `BPF_HASH(metrics, pid, struct proc_metrics{...})`: Aggregates statistics per PID. The `proc_metrics` struct stores total run time, total queue time, counts of run/queue events, and a calculated priority score. Updates are intended to be atomic (implicitly via map operations or ideally using `lock_xadd` if high contention is expected, although not explicitly shown in the snippet).
  - `BPF_HISTOGRAM(run_dist)`, `BPF_HISTOGRAM(queue_dist)`: These maps store logarithmic histograms (power-of-2) of run durations and queue latencies, respectively, measured in microseconds.
- **Core Logic:**
  - On `sched_wakeup`: The timestamp when the task becomes runnable is stored in the `start` map, keyed by the task's identifiers. This marks the beginning of its time spent waiting in the run queue.
  - On `sched_switch`:
    - *Previous Task:* The script looks up the start time for the `prev` task from the `start` map (this timestamp marks when it started running). The difference between the current time and the start time gives the task's run duration. This duration is added to the `run_time` in the `metrics` map, the `run_count` is incremented, and the duration (converted to microseconds) is added to the `run_dist` histogram. The `start` entry for the `prev` task is deleted.
    - *Next Task:* The script looks up the start time for the `next` task (this timestamp marks when it became runnable via `sched_wakeup`). The difference between the current time and this start time gives the task's queue latency. This latency is added to the `queue_time` in the `metrics` map, the `queue_count` is incremented, and the latency (converted to microseconds) is added to

the `queue_dist` histogram. The `start` entry (representing queue start time) is deleted.
    - Finally, the current time is stored in the `start` map for the `next` task, marking the beginning of its run time.
  - *Priority Calculation:* The `update_metrics` helper calculates a simple priority score based on the ratio of run time to total time (run + queue), aiming to prioritize processes that utilize the CPU effectively rather than spending excessive time waiting.
- **Userspace Interaction:** The Python script (`monitor_cpu`) runs in a loop controlled by the `interval` and `count` arguments. In each loop:
  - It reads the `metrics` map.
  - If `-per-process` is enabled, it processes this data: fetches command names from `/proc`, calculates CPU percentages, converts nanosecond times to milliseconds, sorts the processes (default by priority), and prints the formatted table (`print_process_metrics`).
  - It reads the `run_dist` and `queue_dist` histogram maps and prints their contents in a human-readable log2 format (`print_histograms`).
  - Crucially, it clears the `metrics`, `run_dist`, and `queue_dist` maps to ensure the next iteration reports statistics only for that specific interval.
- **Output Explanation:** The script can produce two main outputs per interval:
  - Per-Process Table (if `-P`): Shows PID, Command Name, CPU Usage Percentage (calculated as run_time / (run_time + queue_time)), Total Run Time (ms), Total Wait/Queue Time (ms), and the calculated Priority Score for the interval.
  - Histograms: Displays distributions of CPU run times and (optionally, if `-r`) run queue latencies in microseconds, using power-of-2 buckets.

7) Scheduler Parameter Monitoring (`schparm.py`):

- **Goal:** This script focuses on capturing and reporting the static scheduling parameters (policy, priority, nice value) of processes, along with basic runtime and context switch counts, specifically at the moment a process exits.
- **eBPF Probes:**
  - `kprobe:do_exit`: Attaches to the kernel function responsible for process termination (`do_exit`). This is the primary trigger for collecting and reporting data.
  - `tracepoint:sched:sched_switch`: Used passively in the background to track context switches and estimate runtime.
- **eBPF Logic & Maps:**
  - `BPF_HASH(start_time, pid, timestamp_ns)`: On `sched_switch`, this map stores the timestamp when the previous task

PID	COMM	CPU%	RUN(ms)	WAIT(ms)	PRIO
3279	Thread<02>	99.91	44.1879	0.0403	99
3288	Thread<11>	99.79	43.3803	0.0915	99
3289	Thread<12>	99.82	44.0308	0.0797	99
3282	Thread<05>	99.88	44.1049	0.0542	99
3281	Thread<04>	99.86	44.1337	0.0605	99
3277	Thread<00>	99.76	43.8288	0.1065	99
3283	Thread<06>	99.57	38.8605	0.1671	99
3290	Thread<13>	99.83	43.9302	0.0741	99
49285	ServiceWorker t	99.76	6.4296	0.0155	99
3292	Thread<15>	99.89	43.9340	0.0502	99
3284	Thread<07>	99.86	44.1881	0.0631	99
10077	ThreadPoolForeg	99.66	5.5696	0.0192	99
3280	Thread<03>	99.90	44.1740	0.0464	99
3285	Thread<08>	99.86	44.1039	0.0639	99
3827	chrome	98.23	6.2103	0.1120	98
3821	ThreadPoolSingl	98.83	0.5584	0.0066	98
40656	chrome	98.46	2.9986	0.0468	98
3275	code	98.52	2.3750	0.0356	98
3301	VizCompositorTh	97.49	15.5392	0.3999	97
3168	code	97.82	1.3007	0.0290	97
3259	code	97.05	12.5875	0.3827	97
973	systemd-oomd	96.19	0.4644	0.0184	96
40662	chrome	96.18	0.8226	0.0326	96
3829	chrome	96.25	0.7273	0.0284	96
1915	gnome-shell	96.08	2.9291	0.1195	96

Fig. 6. CPU Usage (Wait time, Run time) process-wise

```
CPU Runtime Distribution (microseconds):
     usecs               : count     distribution
         0 -> 1          : 0       |                                        |
         2 -> 3          : 575     |**************                          |
         4 -> 7          : 1543    |****************************************|
         8 -> 15         : 883     |**********************                  |
        16 -> 31         : 1090    |****************************            |
        32 -> 63         : 511     |*************                           |
        64 -> 127        : 573     |**************                          |
       128 -> 255        : 434     |***********                             |
       256 -> 511        : 369     |*********                               |
       512 -> 1023       : 316     |********                                |
      1024 -> 2047       : 218     |*****                                   |
      2048 -> 4095       : 180     |****                                    |
      4096 -> 8191       : 264     |******                                  |
      8192 -> 16383      : 200     |*****                                   |
     16384 -> 32767      : 105     |**                                      |
     32768 -> 65535      : 82      |**                                      |
     65536 -> 131071     : 69      |*                                       |
    131072 -> 262143     : 45      |*                                       |
    262144 -> 524287     : 22      |                                        |
    524288 -> 1048575    : 11      |                                        |
   1048576 -> 2097151    : 1       |                                        |
```

Fig. 7. CPU Runtime Distribution

```
Run Queue Latency Distribution (microseconds):
     usecs               : count     distribution
         0 -> 1          : 1230    |****************************************|
         2 -> 3          : 845     |***************************             |
         4 -> 7          : 655     |*********************                   |
         8 -> 15         : 273     |********                                |
        16 -> 31         : 674     |**********************                  |
        32 -> 63         : 51      |*                                       |
        64 -> 127        : 2       |                                        |
       128 -> 255        : 1       |                                        |
       256 -> 511        : 6       |                                        |
       512 -> 1023       : 1       |                                        |
      1024 -> 2047       : 0       |                                        |
      2048 -> 4095       : 0       |                                        |
      4096 -> 8191       : 0       |                                        |
      8192 -> 16383      : 2       |                                        |
```

Fig. 8. Run Queue Latency Distribution

(prev_pid) was scheduled out. This timestamp is later used in do_exit to calculate a measure of runtime.
  - BPF_HASH(vol_switches, pid, count), BPF_HASH(nonvol_switches, pid, count): Also updated during sched_switch. Based on the prev_state of the task being scheduled out, these maps count voluntary (task blocked, e.g., waiting for I/O) and non-voluntary (task preempted while still runnable) context switches, respectively, keyed by the PID of the task being switched out.
  - BPF_PERF_OUTPUT(sched_events): A perf buffer used to send the final collected data structure (sched_data_t) from the kernel (specifically from the do_exit probe) to userspace.
- **Core Logic:**
  - On sched_switch: Updates start_time for the prev_pid with the current timestamp. Increments either vol_switches or nonvol_switches for prev_pid based on its state.
  - On do_exit (kprobe trace_sched_process_exit):
    * Retrieves the current task's task_struct.
    * Extracts PID, PPID, current CPU, and command name.
    * Reads scheduling parameters directly from the task_struct: prio (internal kernel priority), static_prio (used to calculate the nice value relative to the default priority 120), and policy (scheduling policy identifier).
    * Looks up the timestamp in start_time (representing the last time it was scheduled out) and calculates runtime as the difference between the current time and that timestamp. (Note: This runtime might not represent the total lifetime runtime but rather time since last deschedule).
    * Looks up the counts from vol_switches and nonvol_switches.
    * Populates the sched_data_t structure with all collected information.
    * Submits the sched_data_t structure to the sched_events perf buffer.
    * Deletes the exiting PID's entries from start_time, vol_switches, and nonvol_switches.
- **Userspace Interaction:**
  - The Python code defines a ctypes.Structure (SchedData) that mirrors the sched_data_t struct in the BPF code.
  - It defines a callback function (print_event) to process data received from the perf buffer.
  - It opens the sched_events perf buffer, associating it with the print_event callback.

```
Monitoring process scheduling parameters. Press Ctrl+C to stop.
COMM                 PID    PPID   CPU  PRIO   NICE   POLICY     RUNTIME(ms)  VOL    NONVOL
which                49663  49662  10   120    0      NORMAL     0.00         0      0
sh                   49662  3371   4    120    0      NORMAL     0.64         2      0
ps                   49665  49664  19   120    0      NORMAL     0.00         0      0
sh                   49664  3371   11   120    0      NORMAL     14.45        3      0
sed                  49668  49667  18   120    0      NORMAL     0.00         0      0
cat                  49669  49667  18   120    0      NORMAL     0.00         0      0
cat                  49670  49667  18   120    0      NORMAL     0.00         0      0
cat                  49671  49667  18   120    0      NORMAL     0.00         0      0
cat                  49672  49667  18   120    0      NORMAL     0.00         0      0
cat                  49673  49667  18   120    0      NORMAL     0.00         0      0
cat                  49674  49667  18   120    0      NORMAL     0.00         0      0
cat                  49675  49667  6    120    0      NORMAL     0.00         0      0
cat                  49676  49667  6    120    0      NORMAL     0.00         0      0
cat                  49677  49667  7    120    0      NORMAL     0.00         0      0
cat                  49678  49667  7    120    0      NORMAL     0.00         0      0
cat                  49679  49667  7    120    0      NORMAL     0.00         0      0
cat                  49680  49667  7    120    0      NORMAL     0.00         0      0
cat                  49681  49667  7    120    0      NORMAL     0.00         0      0
```

Fig. 9. Monitoring Scheduling Parameters

  - The main loop continuously polls the perf buffer (b.perf_buffer_poll()). Polling waits for events to arrive from the kernel.
  - When an event arrives (i.e., a process exits and data is submitted from do_exit), the print_event callback is invoked. It casts the raw data to the SchedData structure, looks up the policy name from the POLICY_NAMES dictionary, converts the runtime to milliseconds, and prints the formatted line containing all the scheduling parameters and statistics for the exited process.
- **Output Explanation:** The script prints one line of output only when a traced process exits. Each line contains the Command Name, PID, Parent PID (PPID), CPU it last ran on, Kernel Priority (prio), Nice Value, Scheduling Policy Name (e.g., NORMAL, FIFO), Runtime (in milliseconds, representing time since last deschedule before exit), and the total counts of Voluntary and Non-Voluntary context switches accumulated during its lifetime (as tracked by the script).

8) Memory Analysis (compmem.py):

- **Goal:** Track kernel memory allocations (kmalloc) and deallocations (kfree) to estimate per-process kernel memory usage and identify potential leaks by tracking unfreed allocations associated with their allocation stack traces. Also integrates user-space memory info (RSS - Resident Set Size, VmSize) from /proc.
- **eBPF Probes:** Raw Tracepoints kmem:kmalloc and kmem:kfree. (Raw tracepoints are preferred for stability over kprobes on kmalloc / kfree).
- **eBPF Logic & Maps:**
  - BPF_HASH(allocs, alloc_address, alloc_info_t{size, timestamp_ns, stack_id}): On kmalloc success, stores allocation size, timestamp, and stack trace ID (from BPF_STACK_TRACE), keyed by the allocated memory address.
  - BPF_HASH(proc_mem, pid, proc_mem_info_t{total_allocated, active_allocations}): Atomically updates the total kernel bytes allocated and the count of

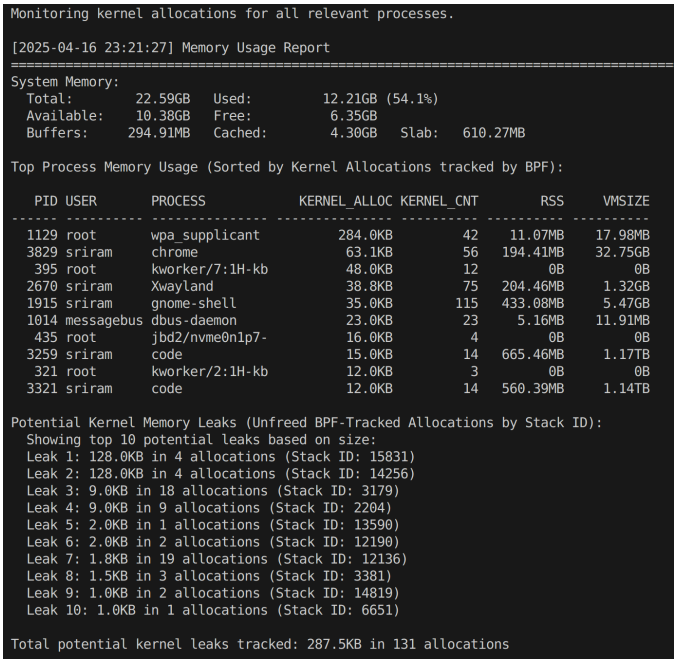

Fig. 10. Memory usage Analysis

active allocations for the specific PID on `kmalloc` and `kfree`.
  - `BPF_STACK_TRACE(stack_traces)`: Stores kernel stack traces captured during allocation.
  - `BPF_HASH(combined_allocs, stack_id, combined_alloc_info_t{total_size, number_of_allocs})`: Tracks the total size and count of currently unfreed allocations, aggregated by their allocation stack trace ID. Updated on both `kmalloc` (increment) and `kfree` (decrement).
  - On `kfree`, looks up the allocation in `allocs`. If found, deletes the entry, updates `proc_mem`, and updates `combined_allocs`.
- **Userspace Interaction:** Periodically reads the `proc_mem` map to get kernel allocation stats per process. It also reads `/proc/[pid]/status` to get user-space RSS and VmSize. It reads the `combined_allocs` map and uses the `stack_traces` map to potentially display leak sources. It prints system memory overview, a table of top processes sorted by kernel memory, and a summary of potential leaks based on the `combined_allocs` map.
- **Output:** Prints system memory summary, a table listing top processes with their kernel allocation size/count and user RSS/VmSize, and a summary of potential leaks aggregated by kernel stack trace ID.

9) Page Cache Analysis (`cacheraw.py`):

- **Goal:** Analyze page cache hit/miss ratios per process for read and write operations.
- **eBPF Probes:** Kprobes on `add_to_page_cache_lru` (read miss/write

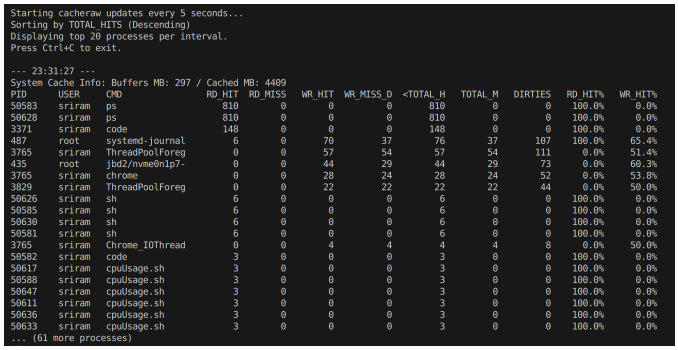

Fig. 11. Cache Access Analysis

allocate), `mark_page_accessed` (read hit), `mark_buffer_dirty` (write hit to buffer cache), and either `folio_account_dirtied`, `account_page_dirtied` (kprobes) or `writeback:writeback_dirty_folio/page` (tracepoints) for pages marked dirty by writes.
- **eBPF Logic & Maps:**
  - `BPF_HASH(counts, key_t{nf_type, pid, uid, comm})`: Uses a single map where the key includes an enum (`nf_type`) identifying which probe was hit, along with PID, UID, and command. The value is simply a counter (incremented atomically).
- **Userspace Interaction:** Periodically reads and clears the `counts` map. It aggregates the raw counts from different `nf_type` values for each process (PID, UID, Comm) into meaningful categories (Read Hits, Read Misses, Write Hits, Writes Dirtying Pages). It calculates derived metrics like total hits/misses and hit percentages. Sorts the results and prints a table. Also reads `/proc/meminfo` for system-wide cache/buffer stats.
- **Output:** Prints system cache/buffer sizes, followed by a table listing processes sorted by a chosen metric (e.g., total hits, read misses), showing raw hit/miss counts, derived totals, and hit percentages for reads and writes.

10) Block I/O Analysis (`biopwise.py`):

- **Goal:** Summarize block device I/O activity (bytes, operations) per process, distinguishing between read/write and sync/async operations.
- **eBPF Probes:** Tracepoint `block:block_rq_issue`. This captures requests as they are issued to the block layer.
- **eBPF Logic & Maps:**
  - `BPF_HASH(counts, key_t{dev, slot, comm}, val_t{bytes, ios, sync_ios, async_ios})`: On `block_rq_issue`, creates a key containing the device number (`dev`), command name (`comm`), and a `slot` encoding both the PID (`pid << 1`) and the operation type (`is_write`). It parses the `rwbs` flags to determine if it's a read/write and if it's synchronous. It updates the

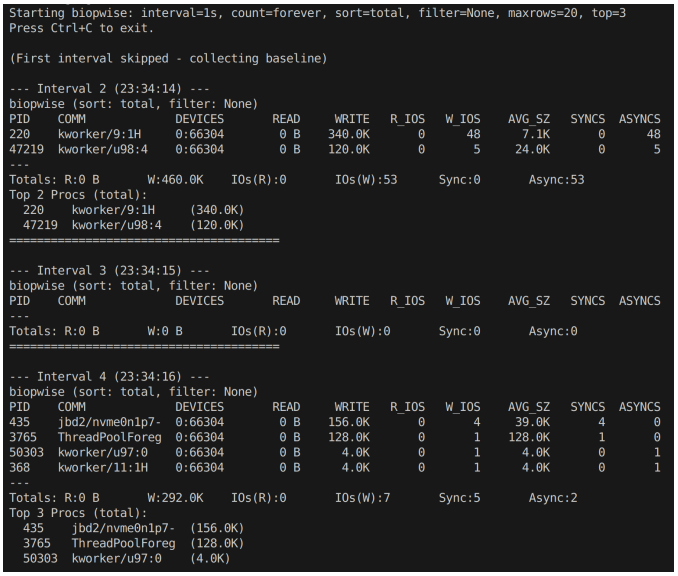

Fig. 12. Block Input/Output Process Wise Analysis

corresponding counters (`bytes`, `ios`, `sync_ios`, `async_ios`) in the map using `lock_xadd`.

- **Userspace Interaction:** Periodically reads the `counts` map. It calculates the delta in bytes and IOs for each process compared to the previous interval. It aggregates these deltas per process (PID, Comm), calculates average I/O size, determines the devices involved, sorts the processes based on the chosen criteria (e.g., total bytes, reads, writes), and prints the per-process statistics for the interval. It also prints overall totals and a summary of the top N processes. Uses `/proc/partitions` to map device numbers to names.
- **Output:** Prints a table showing PID, command, devices used, read/write bytes and IO counts, average I/O size, and sync/async IO counts for the interval. Also includes overall totals and a summary of top processes.

V. Data Transmission and Robustness

Recognizing that network conditions can be unreliable and data volume potentially high, several features were implemented in the user-space application (`main_app.py`) and server (`server.py`) to improve robustness.

A. Protocol and Payload

HTTP was chosen for its ubiquity and simplicity. Data is sent via POST requests to the configured server URL (`/data`). The payload is structured as JSON, containing:

- `timestamp`: Unix timestamp of when the batch was created.
- `source_script`: The name of the eBPF script that generated the data.
- `batch_id`: A unique identifier for the transmission batch.
- `sequence`: A monotonically increasing number for batches from a specific script run, allowing the server to detect gaps.
- `metrics`: A list containing the individual data points collected from the eBPF script's output during the collection interval, each typically being a dictionary representing a parsed line.

B. Data Buffering and Timers

- **Buffering:** A `collections.deque(maxlen=1000)` is used in `main_app.py` to buffer parsed data points received from the worker thread. This allows data collection to continue even if the network is temporarily unavailable and smooths out transmission bursts. The `maxlen` prevents unbounded memory growth if the server remains unreachable.
- **Periodic Sending Timer (`send_timer`):** A `QTimer` triggers `send_data_to_server` every 2 seconds to send accumulated data from the buffer.
- **Backup Timer (`backup_timer`):** Another `QTimer` triggers a forced send attempt every 10 seconds (`force_send_data`), ensuring data is sent even if the buffer doesn't fill rapidly or the regular timer somehow fails.
- **Retry Timer (`retry_timer`):** A separate `QTimer` periodically calls `retry_failed_transmissions` (e.g., every 15 seconds) to attempt resending batches that previously failed.

C. Error Handling and Retries

The `send_data_to_server` function wraps the `requests.post` call in a `try...except` block to catch `requests.exceptions.RequestException` (covering connection errors, timeouts, etc.) and other potential exceptions. If a transmission fails, the entire JSON payload is appended to a `failed_transmissions` list. The `retry_failed_transmissions` function iterates through this list, attempting to resend each failed payload. Successfully resent payloads are removed from the list.

D. Sequence Tracking

To help identify potential data loss (e.g., if the client crashes or network issues persist beyond retry attempts), the client includes a `batch_id` and a `sequence` number in each payload. The `server.py` script maintains a dictionary (`sequence_tracker`) mapping each `source_script` to the last received sequence number. Upon receiving a new batch, it compares the incoming `sequence` number with the expected next number and logs a warning if a gap is detected.

VI. Challenges

A. Challenges Encountered

- **Kernel Dependencies:** eBPF features and tracepoint/kprobe availability are kernel-version specific. Ensuring scripts worked required testing against the target kernel version and having correct kernel headers installed.

- **eBPF Verifier:** Writing correct and efficient BPF C code that passes the verifier requires understanding its constraints (e.g., bounded loops, limited stack size).
- **Permissions:** Running eBPF requires root privileges, meaning the entire GUI application had to be run with `sudo`, which is not ideal from a security perspective.
- **Output Parsing:** Reliably parsing the `stdout` of diverse BCC scripts required careful definition of `output_parser` functions in `config.py`. Scripts outputting structured data (like JSON) would simplify this.
- **UI Responsiveness:** Using worker threads was essential to prevent the GUI from freezing, requiring careful use of Qt signals/slots for inter-thread communication.
- **Data Volume:** High-frequency events (e.g., packet tracing, syscalls on busy systems) can generate substantial data, stressing the buffering and transmission mechanisms.

B. Achievements

- Successfully implemented a functional eBPF-based profiling tool with a GUI.
- Developed and integrated multiple eBPF scripts covering a wide range of relevant performance metrics.
- Implemented a system for dynamically configuring and launching monitoring tasks based on user input.
- Created a data pipeline for collecting, buffering, and transmitting performance data to a remote server.
- Incorporated robustness features like retries and sequence tracking into the data transmission process.
- Gained practical experience with eBPF, BCC, kernel tracing concepts, and GUI development.

VII. Conclusion and Future Work

A. Summary

This project successfully demonstrated the application of eBPF technology for building a versatile performance profiling tool to monitor applications running on Linux Systems. By leveraging kernel-level probes via BCC, the tool gathers granular data across system calls, networking, CPU scheduling, memory, and I/O with potentially lower overhead than traditional methods. The Python GUI provides a user-friendly interface for managing the monitoring process, and the system includes mechanisms for robustly transmitting collected data to a remote server. The implementation provided valuable insights into the practicalities of eBPF development and system-level performance analysis.

B. Future Enhancements

Several avenues exist for future improvement:

- **Richer Visualization:** Integrate plotting libraries (e.g., `pyqtgraph`) into the GUI to visualize metrics over time directly.
- **Lower Overhead Backend:** Transition from BCC Python scripts to using `libbpf` with CO-RE (Compile Once - Run Everywhere) for potentially lower overhead and easier distribution.
- **More Metrics:** Add probes for other subsystems (e.g., filesystem activity, specific application-level events using USDT probes if available).
- **Advanced Server Analysis:** Develop more sophisticated server-side logic for storing data in a time-series database (e.g., InfluxDB, Prometheus) and providing querying and alerting capabilities.

References

[1] ebpf.io, "What is eBPF?". [Online]. Available: https://ebpf.io/what-is-ebpf/#documentation

[2] B. Gregg, "Learn eBPF Tracing: Tutorial and Examples," Brendan Gregg's Blog, Jan. 01, 2019. [Online]. Available: https://www.brendangregg.com/blog/2019-01-01/learn-ebpf-tracing.html

[3] Red Hat, "BCC tools brings dynamic kernel tracing to Red Hat Enterprise Linux 8.1," Red Hat Blog, Nov. 05, 2019. [Online]. Available: https://www.redhat.com/en/blog/bcc-tools-brings-dynamic-kernel-tracing-red-hat-enterprise-linux-81

[4] Cilium Documentation, "BPF and XDP Reference Guide". [Online]. Available: https://docs.cilium.io/en/stable/reference/bpf/

[5] iovisor/bcc Project, GitHub Repository. [Online]. Available: https://github.com/iovisor/bcc