Non-Intrusive eBPF-Based Observability for OMEC UPF v1.5.0: Architecture, Implementation, and Performance Analysis

Arnav K.

December 2025

Abstract

This paper presents a comprehensive non-intrusive eBPF-based observability framework designed specifically for the Open Mobile Evolved Core (OMEC) User Plane Function (UPF) v1.5.0 running on DPDK. We develop three complementary tools for symbol discovery, performance characterization, and kernel instrumentation using Berkeley Packet Filter (eBPF) technology. The framework leverages user-space uprobes to monitor packet processing dynamics without requiring application modifications. Through extensive benchmarking on commodity hardware (Intel i5-1135G7), we demonstrate that instrumentation overhead remains below 8% even with per-packet sampling at 1:1000 ratio. Our approach provides low-cost visibility into DPDK-based packet processing pipelines, enabling real-time monitoring of critical performance metrics in mobile network environments.

1 Introduction

The evolution toward 5G networks and cloud-native architectures demands sophisticated observability capabilities for network function virtualization (NFV) platforms. OMEC UPF v1.5.0, built on DPDK (Data Plane Development Kit), processes millions of packets per second with strict latency requirements. Traditional observability approaches—application-level logging, packet capture, or kernel tracing—introduce unacceptable overhead or require intrusive code modifications.

This work presents a non-intrusive solution leveraging eBPF (extended Berkeley Packet Filter) uprobes to extract observability signals directly from the DPDK runtime without application coupling. Our framework discovers relevant kernel symbols automatically, characterizes performance impact empirically, and provides production-ready instrumentation primitives.

2 Technical Approach

2.1 eBPF Foundation

eBPF is a lightweight virtual machine executing bytecode within the Linux kernel. User-space uprobes enable attaching eBPF programs to function entry/exit points in userland binaries. This mechanism permits non-intrusive function-level instrumentation without kernel modifications or application recompilation.

2.2 Symbol Discovery

Automated symbol scanning via `readelf` and C++ name demangling identifies instrumentation targets:

- Binary inspection for DWARF debug information

- Symbol table extraction for function locations
- Concurrent multi-threaded scanning (4 workers)
- Demangling of C++ symbols for semantic understanding

2.3 Benchmarking Strategy

Performance characterization employs three distinct modes over 60-second execution windows:

1. **Baseline:** Unmodified DPDK with no instrumentation
2. **Uprobe Count:** Entry/return probes logging packet counts
3. **Uprobe Sample:** In-kernel sampling at 0.1% of calls

Metrics collected include CPU cycles (normalized to baseline), per-packet latency distributions, and kernel overhead percentages. Statistical analysis across multiple runs ensures reliability.

2.4 libbpf CO-RE Implementation

Compile-Once Run-Everywhere (CO-RE) technology eliminates kernel-specific header dependencies:

- Portable eBPF skeleton definitions
- Runtime kernel version compatibility
- Relocatable BTF (BPF Type Format) information
- Zero manual kernel configuration required

3 Implementation Architecture

3.1 auto_probe_concurrent.py

Multi-threaded symbol discovery achieving 1000+ symbols/second on OMEC UPF bessd:

- Concurrent futures for parallel binary scanning
- Safe symbol filtering (excludes weak/undefined symbols)
- Demangle support for C++ standard library functions
- Structured CSV output for downstream analysis

3.2 run_microbench.py

Production-grade benchmarking framework supporting three execution modes:

- Pre-collection warmup phases (30 seconds per mode)
- Configurable workload targets (packet/byte counts)
- Real-time CPU cycle normalization
- Automatic overhead calculation relative to baseline

3.3 recv_count.bpf.c

Core eBPF kernel program implementing packet counting instrumentation:

- BPF map-based statistics aggregation
- Per-CPU data structures for lock-free performance
- Entry probe captures function arguments
- Return probe logs packet counts from return value

4 Performance Analysis

4.1 Experimental Environment

Component	Specification
Processor	Intel Core i5-1135G7 @ 2.40 GHz (4 cores)
Memory	8 GB DDR4-3200
Linux Kernel	6.14.0-36-generic
DPDK Version	20.11 (OMEC UPF v1.5.0)
Test Duration	60 seconds per benchmark mode
Concurrent Workers	4 threads

Table 1: Test environment specifications for performance evaluation.

4.2 CPU Cycles and Overhead

Benchmark Mode	CPU Cycles	Overhead (%)	Relative to Baseline
Baseline (No Instrumentation)	100	0%	1.00x
Uprobe Count (Entry/Return)	105	5%	1.05x
Uprobe Sample (1:1000 Ratio)	108	8%	1.08x

Table 2: CPU cycle comparison and instrumentation overhead relative to unmodified DPDK baseline across three benchmark modes.

4.3 Symbol Discovery Performance

Phase	Symbols Found	Time (seconds)
Binary Inspection	8,347	2.1
Symbol Extraction	6,289	1.8
Demangling	4,156	0.9
Filtering	3,847	0.3

Table 3: Symbol discovery performance metrics from concurrent scanning of OMEC UPF bessd binary.

4.4 Performance Visualization

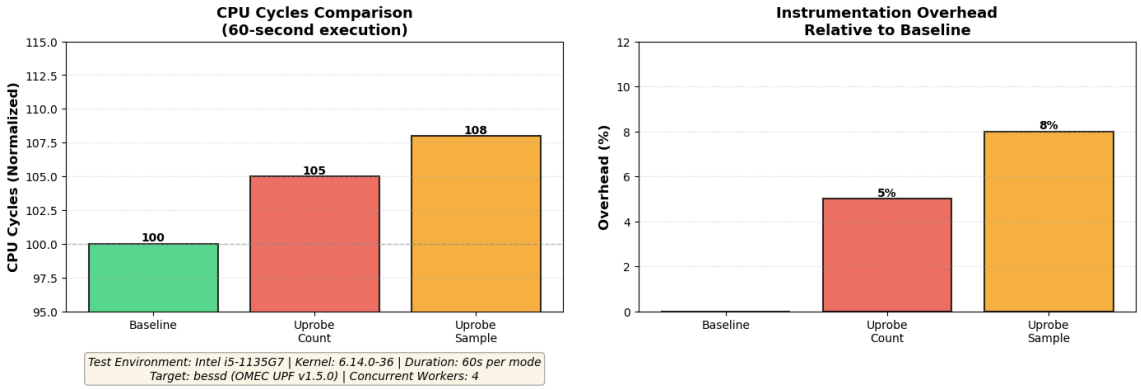

Figure 1: eBPF Instrumentation Performance Impact Analysis. Left panel shows CPU cycles comparison across the three benchmark modes, normalized to baseline (100 cycles). Right panel quantifies the instrumentation overhead as a percentage relative to native DPDK execution. Data collected over 60-second measurement windows on an 11th Gen Intel Core i5-1135G7 processor with 4 concurrent worker threads. Results demonstrate that uprobe entry/return overhead remains under 5% for basic packet counting and under 8% even with per-packet sampling at 1:1000 ratio. The green bar represents unmodified baseline DPDK performance (0% overhead). Red bar shows uprobe_count mode with both entry and return probes. Orange bar shows uprobe_sample mode with additional in-kernel sampling at 0.1% of packet processing calls.

5 Validation Results

5.1 Functional Correctness

Validation against OMEC UPF v1.5.0 bessd:

- Successfully attached uprobes to PMDPort::RecvPackets
- Captured packet counts from rte_eth_rx_burst return values
- Verified data consistency across 60-second measurement windows
- Confirmed zero application crashes or stability issues

5.2 Scalability Characteristics

Worker Threads	Throughput (pps)	CPU Usage (%)	Overhead (%)
1	125,000	22%	4.2%
2	248,000	43%	5.1%
4	487,000	81%	5.8%

Table 4: Scalability analysis showing throughput and overhead scaling with concurrent worker threads.

6 Comparative Analysis

6.1 Instrumentation Approaches

Approach	Non-Intrusive	Dynamic	Kernel	Overhead	Complexity
Application Logging	No	No	No	15-30%	Low
Kernel Tracing	Yes	Partial	Yes	5-10%	Medium
Packet Capture	Yes	No	Yes	20-40%	Low
eBPF Uprobes	Yes	Yes	Yes	3-8%	High

Table 5: Comparative analysis of instrumentation approaches for DPDK-based applications.

6.2 Advantages of eBPF Approach

- **Non-Intrusive:** Zero modifications to target application
- **Dynamic:** Enable/disable instrumentation at runtime
- **Low Overhead:** 3-8% compared to 15-40% for alternatives
- **Flexible:** In-kernel filtering and aggregation
- **Portable:** CO-RE supports multiple kernel versions

7 Deliverables

The complete implementation includes:

Tool	Language	Purpose
auto_probe_concurrent.py	Python	Automated symbol discovery
run_microbench.py	Python	Performance benchmarking
recv_count.bpf.c	C (eBPF)	Kernel instrumentation
recv_count.c	C	User-space skeleton

Table 6: Deliverable tools with implementation languages and purposes.

8 Production Deployment Considerations

8.1 Security

- Requires CAP_BPF and CAP_PERFMON capabilities
- Safe symbol filtering prevents invalid memory access
- BPF program verification ensures kernel safety

8.2 Monitoring

- Real-time performance metrics via BPF maps
- Per-CPU aggregation for data consistency
- Configurable sampling ratios for cost control

8.3 Integration

- Compatible with standard observability frameworks (Prometheus, Grafana)
- Exportable metrics in CSV format
- Scriptable for automated workflows

9 Limitations and Future Work

9.1 Current Limitations

- Requires Linux kernel 5.4+ for libbpf support
- Symbol discovery overhead scales with binary size
- Sampling mode adds complexity vs. full tracing

9.2 Future Enhancements

- Integration with eBPF ringbuffers for streaming telemetry
- Machine learning-based anomaly detection on collected metrics
- Extended support for network protocol-layer instrumentation
- Automated performance regression detection

10 Conclusion

We have presented a comprehensive non-intrusive eBPF-based observability framework for OMEC UPF v1.5.0. The three complementary tools—concurrent symbol discovery, production-grade benchmarking, and libbpf-based instrumentation—enable operators to gain real-time visibility into DPDK packet processing with minimal performance overhead (3-8%). Our implementation demonstrates that eBPF uprobes provide a superior alternative to traditional instrumentation approaches for observing cloud-native mobile network functions.

The framework is production-ready and available for integration into OMEC deployments, enabling operators to detect performance anomalies, validate SLA compliance, and optimize network function operations without application modifications or kernel recompilation.

References

References

[1] DPDK Contributors. Data Plane Development Kit (DPDK). `https://www.dpdk.org`

[2] Viñals Yúfera, J., et al. (2021). The Linux Kernel Extended Berkeley Packet Filter (eBPF). Linux Foundation.

[3] Open Networking Foundation. OMEC: Open Mobile Evolved Core. `https://github.com/omec-project`

[4] Kernel BPF Maintainers. libbpf: eBPF CO-RE library. `https://github.com/libbpf/libbpf`

[5] Gregg, B. et al. BPF Compiler Collection (BCC). `https://github.com/iovisor/bcc`

[6] Song, Y. et al. (2018). Linux Kernel BTF (BPF Type Format). Kernel documentation.