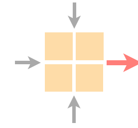
ETH zürich



Networked Systems
ETH Zürich — seit 2015

Understanding the performance of eBPF-based applications

Semester Thesis

Author: Xinge Xie

Tutor: Laurin Brandner and Georgia Fragkouli

Supervisor: Prof. Dr. Laurent Vanbever

October 2024 to January 2025

Abstract

The rise of modern computing demands high-performance, low-latency systems capable of handling complex workloads. Extended Berkeley Packet Filter (eBPF) technology offers a promising solution by allowing user-space applications to inject custom programs into the kernel, optimizing data processing and reducing overhead. This thesis investigates the performance characteristics of eBPF-based applications, focusing on scalability, resource utilization, and interference in multi-core environments. By designing and benchmarking a system under various configurations, we analyze factors such as the number of eBPF programs, map types, CPU allocation, and memory access patterns. The findings provide actionable insights into the design and optimization of eBPF-based systems for high-performance network applications.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Task and Goals . . . 1
1.3 Overview . . . 1

2 Background and Related Work **2**
2.1 Background . . . 2
2.1.1 Extended Berkeley Packet Filter . . . 2
2.1.2 eBPF Hooks . . . 2
2.1.3 eBPF Maps . . . 3
2.1.4 Cache Coherence . . . 3
2.2 Related Work . . . 4

3 Design **5**
3.1 Benchmark System Architecture . . . 5
3.2 eBPF Program Chain . . . 6
3.3 eBPF Program Interference . . . 7
3.3.1 Scaling with eBPF Programs and Maps . . . 7
3.3.2 Scaling with CPU Cores . . . 8

4 Evaluation **9**
4.1 Benchmark Setup . . . 9
4.2 eBPF Program Triggering . . . 9
4.3 Benchmark Results . . . 10
4.3.1 Scaling with eBPF Programs . . . 10
4.3.2 Scaling with eBPF Map Size . . . 12
4.3.3 Scaling with CPU Cores . . . 13

5 Outlook **16**

6 Summary **17**

References **18**

Chapter 1

Introduction

1.1 Motivation

The rise of modern computing demands high-performance, low-latency systems capable of efficiently handling complex workloads. The Extended Berkeley Packet Filter (eBPF) enables user-space applications to inject custom programs into the kernel for optimized data processing, avoiding expensive user-kernel space switches and bypassing slow kernel code. While eBPF offers flexibility and performance benefits, understanding its impact on system performance under different configurations is crucial for harnessing its full potential.

1.2 Task and Goals

This thesis investigates the performance characteristics of eBPF-based applications, focusing on their scalability, resource utilization, and interference in multi-core environments. By developing a benchmark system, we aim to analyze how various factors, such as the number of eBPF programs, map types, and CPU allocation, influence application performance. The ultimate goal is to provide actionable insights for designing efficient eBPF systems.

1.3 Overview

The main sections are as follows:

- **Section 2.1**: Describes eBPF's evolution, three hook points to attach eBPF programs during network packet processing, and the characteristics of eBPF maps.
- **Section 3.1**: Explains the design of the benchmark system used to evaluate eBPF programs.
- **Section 3.2**: Details how eBPF program chains can be used to scale with the number of eBPF programs attached.
- **Section 3.3**: Describes how we design benchmarks to run eBPF programs simultaneously and explores different configuration options and their potential impacts on performance.
- **Section 4.2**: Evaluates the unavoidable performance impacts of adopting eBPF technology.
- **Section 4.3**: Presents the evaluation results, focusing on the performance and scalability of eBPF programs under various configurations.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Extended Berkeley Packet Filter

The Berkeley Packet Filter (BPF) is an in-kernel virtual machine for packet filtering, developed in 1992 for UNIX[10]. Its primary purpose is packet filtering to improve the performance of network monitoring applications, such as `tcpdump`[13].

eBPF is an extended form of the BPF virtual machine, introduced in 2013, with a network-specific architecture designed to be a general-purpose filtering system[12]. It enables user-space applications to inject code into the kernel at runtime without recompiling the kernel or installing optional kernel modules[13]. eBPF programs can be written in restricted C and compiled into bytecode using the LLVM Clang compiler. The bytecode is then loaded using the `bpf()` system call.

In a quest to improve application performance, developers use eBPF to offload (parts of) applications to the kernel, avoiding expensive user-kernel space switches and bypassing slow kernel code.

2.1.2 eBPF Hooks

eBPF allows the execution of bytecode at specific locations within the kernel, known as hook points. These hook points significantly expand the functionality of eBPF beyond its original design[11]. Once eBPF programs are attached to specific hook points and registered for certain events. These programs execute whenever an event for which they are registered occurs.

In computer networking, hooks are used to attach eBPF programs and intercept packets during their processing in the operating system. Packets entering the OS are processed by several layers in the kernel: the socket layer, TCP stack, Netfilter, Traffic Control (TC), the eXpress Data Path (XDP), and the NIC. Packets destined for a userspace application traverse all these layers[15], where they can be intercepted to enable packet mangling and filtering.

Traffic Control

Traffic Control (TC) operates at the Linux link layer to manage and manipulate the transmission of packets[8]. When attaching eBPF programs to TC, a specific network interface and attach point (ingress, egress, or custom) must be selected. Multiple eBPF programs can be attached with different priorities using `libbpf`[3] or similar wrapper libraries. These programs are triggered and executed in order of priority. The common return values can be as follows:

- `TC_ACT_OK`: Terminate the packet processing pipeline and allows the packet to proceed, even if lower-priority eBPF programs exist.
- `TC_ACT_SHOT`: Terminate the packet processing pipeline and drops the packet, even if lower-priority eBPF programs exist.
- `TC_ACT_PIPE`: Iterate the packet to another lower-priority eBPF program, if available.

eXpress Data Path

eXpress Data Path (XDP)[7] programs can intercept packets directly from the NIC driver, potentially before the Linux socket buffer (`skb`) is allocated. This allows for earlier packet drops compared to TC. However, `libbpf` currently does not support attaching multiple XDP programs, though this feature is under development.

Socket Message

Socket message programs are invoked for every `sendmsg` or `sendfile` system call, operating at a layer above TC and XDP. To trigger an eBPF program attached to a socket message, a Socket Operations (Socket Ops) program must first be attached to cGroups. This setup allows the program to adjust per-connection settings or record socket information. Once recorded, the socket can trigger the associated socket message eBPF program.

2.1.3 eBPF Maps

eBPF programs operate under certain resource constraints, such as a maximum stack size of 512 bytes and a limit of 4096 instructions. These restrictions ensure that eBPF programs do not consume excessive resources, preserving the stability of the system. For scenarios requiring more storage or communication between components, eBPF maps provide an effective mechanism for interaction between eBPF programs (kernel space) and user space.

Maps come in various types, with the generic type being the `ARRAY` map. Similar to a standard array, an `ARRAY` map has numeric keys starting at 0 and incrementing sequentially. A specialized variant, the `PERCPU` map, maintains a separate array for each logical CPU. When an eBPF program accesses a `PERCPU` map through a helper function, it implicitly accesses the array associated with the CPU on which the program is currently running. Since preemption is disabled during eBPF program execution, no other programs can concurrently access the same memory. This design guarantees the absence of race conditions, enhancing performance by eliminating congestion and synchronization overhead. However, this improvement comes at the cost of a larger memory footprint.

When defining an `ARRAY` or `PERCPU` map, the key size must always be 4 bytes, representing a 32-bit unsigned integer, while the value size is essentially unrestricted and fixed.

2.1.4 Cache Coherence

In a shared-memory multiprocessor system where each processor has its own private cache, multiple copies of shared data can exist across the system. Cache coherence is the discipline that ensures changes to the values of shared data are propagated throughout the system in a timely and consistent manner [14].

Modern CPUs typically implement the MESI-like (Modified, Exclusive, Shared, Invalid) protocol to maintain cache coherence. When two processes running on separate CPU cores attempt to update the same memory address simultaneously, the cache coherence protocol ensures consistency

by invalidating the cache line in one core when it is updated in the other. Ultimately, the updated value is resolved in the shared last-level cache (LLC). While the changes may not propagate to main memory immediately, this process increases the number of references to the last-level cache, potentially impacting performance.

2.2 Related Work

There has been significant research aimed at accelerating network packet processing. The Data Plane Development Kit (DPDK) [1] enables offloading TCP packet processing from the operating system kernel to processes running in user space. This offloading achieves higher packet throughput compared to the interrupt-driven processing provided by the kernel. However, it requires CPU pooling in user space, which can introduce additional overhead.

Smart Network Interface Cards (SmartNICs) represent another emerging trend, where parts of network processing are offloaded to hardware, such as FPGAs in the Azure cloud [6]. This approach provides high scalability and throughput while conserving valuable cloud CPU resources. Nevertheless, developing such a software and hardware co-designed system is challenging due to its complexity.

In eBPF-based applications, network acceleration is achieved by avoiding user-kernel space switches and bypassing slow kernel code. Writing eBPF programs is generally easier than developing SmartNIC solutions but still presents difficulties. Prior work [9] has examined how the data structures used by eBPF programs affect performance. However, other factors that may influence performance have been largely overlooked, such as which parts of applications are offloaded, how frequently the code paths are triggered, and how many eBPF programs are running concurrently.

Chapter 3

Design

3.1 Benchmark System Architecture

To evaluate how eBPF programs affect the performance of network packet processing, we focus primarily on the end-to-end latency of network packets and the increase in computing resource usage and interference when introducing additional eBPF programs.

In a typical client-server model, network latency ranges from tens to hundreds of milliseconds, which is significantly higher than the time required to load and execute a single eBPF program for processing a packet. Consequently, our primary focus is on low-latency, high-load environments, such as distributed systems connected via high-speed networks. To simplify the evaluation, we implement an echo process where the server responds with the same payload it receives. This echo process runs locally alongside the benchmark suite. To ensure precise measurement, we isolate the CPU resources using system control groups. This setup dedicates a specific CPU core to the operating system and unrelated processes, while assigning separate CPU cores for the HTTP load generator and the echo server. This isolation minimizes interference and ensures a more accurate analysis of the system's performance.

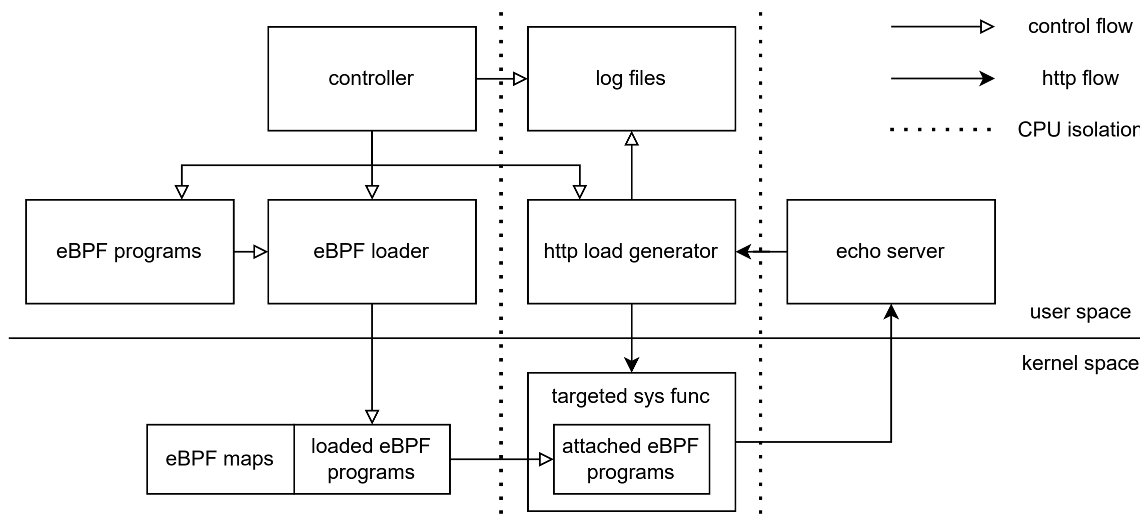


Figure 3.1: Benchmark System Architecture

As illustrated in the figure 3.1, the eBPF programs encompass various functionalities associated with different system hook points and eBPF maps. Configuration parameters, such as map type and size, can be customized during compilation. To load eBPF programs from user space to kernel space, we utilize `libbpf-rs` [4], a Rust wrapper for `libbpf`. The loader facilitates specifying attachment points, detaching programs, and monitoring the status of eBPF programs and their maps.

For HTTP load generation, we employ `k6` [2], which can produce targeted HTTP request flows and collect a variety of performance metrics as output.

The controller serves as the entry point of the benchmark system. It determines the eBPF programs to load and attach, configures parameters, and specifies the HTTP load patterns. Additionally, it manages resource allocation, such as assigning CPU cores to the HTTP load generator and the echo server. Finally, the controller clears the eBPF programs and processes the log files, which serve as inputs for subsequent visualization.

3.2 eBPF Program Chain

Triggering even an empty eBPF program introduces a slight latency [9] due to CPU time consumption. However, this latency is generally negligible compared to the network stack latency, as eBPF is designed for high-performance processing. To better understand how eBPF programs impact performance, we amplify the effect by triggering the programs multiple times.

A single HTTP request can trigger not just one eBPF program, but multiple eBPF programs. While different eBPF programs can be attached to various hook points, this approach does not scale well, as each hook point requires a corresponding eBPF program. However, certain special hook points allow the attachment of multiple eBPF programs, creating a program chain.

As part of Linux's packet management and filtering mechanisms, Traffic Control (TC) operates at the link layer [5], and the TC is bound to the network interface. In our design, the client and server run on the same machine, using the *loopback* interface, which is a virtual network interface for internal communication. TC provides three attach points: ingress, egress, and custom attach points. In this work, we focus on the TC ingress point.

The return value of an eBPF program attached to TC determines whether the packet is sent to the upper network layer, dropped, or forwarded to another eBPF program. The return value `TC_ACT_PIPE` allows the triggering of another eBPF program. The order of execution in a TC program chain is determined by the priority number assigned when the eBPF program is attached.

It is worth noting that packets unrelated to the benchmark system may also trigger the eBPF programs attached to TC. To handle this, we use port-based filtering to identify and exclude these unrelated packets, returning `TC_ACT_OK` to send them directly to the upper network layer without invoking the eBPF program chain. Additionally, these unrelated packets are processed on reserved CPU cores, as described in Section 3.1, ensuring they do not affect the benchmark results.

Using the eBPF program loader, we can either load multiple eBPF programs or attach a single loaded program multiple times to a TC hook point. These two configurations, illustrated in Figure 3.2, have distinct implications. Each loaded TC eBPF program has its own eBPF map, while multiple attachments of the same loaded program share the same map at runtime. The length of the eBPF program chain can extend to thousands of programs or even longer, constrained only by the available kernel space. These configurations influence performance differently depending on the map access pattern and the HTTP load pattern.

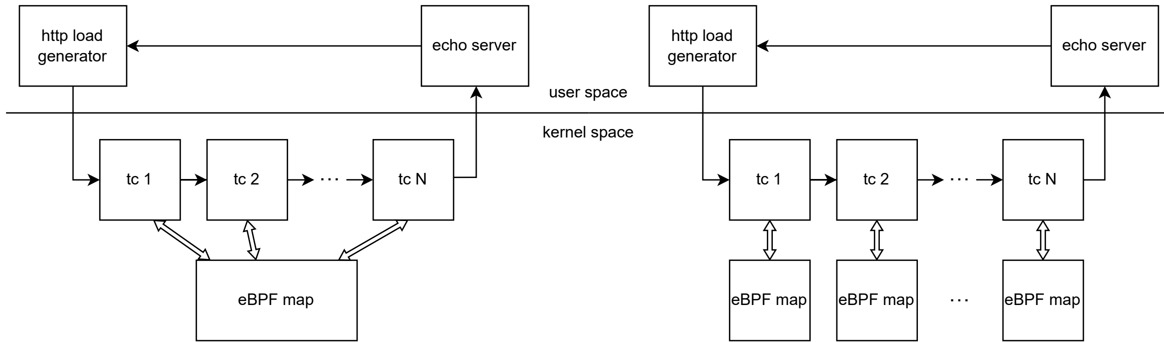


Figure 3.2: eBPF program in TC: Load once and attach multiple times vs. load and attach multiple times.

3.3 eBPF Program Interference

eBPF applications often run in multi-core environments. To understand the performance of such applications, we should also benchmark scenarios where multiple eBPF instances run simultaneously. This allows us to study their interactions, evaluate how they affect each other, and analyze the relationship between computing resources and performance.

3.3.1 Scaling with eBPF Programs and Maps

Consider multiple processes triggering the TC chains, as shown in Figure 3.3. In the configuration where an eBPF program is loaded once and attached N times, all eBPF programs in a chain share the same eBPF map. When packet B follows packet A and their processing is scheduled on different CPU cores, multiple instances of the eBPF chain may run simultaneously in kernel space, accessing the shared map. If an eBPF program updates this shared map, it can introduce write conflicts, leading to invalid cache lines, increased cache misses, and ultimately added latency to packet processing. As the number of eBPF programs in the chain increases, the total number of map entries remains fixed, but the frequency of accesses increases, exacerbating the write conflict. However, in read-intensive access patterns, performance benefits can be achieved by leveraging the ”hot map” effect.

In contrast, the configuration where the same TC eBPF program is loaded and attached N

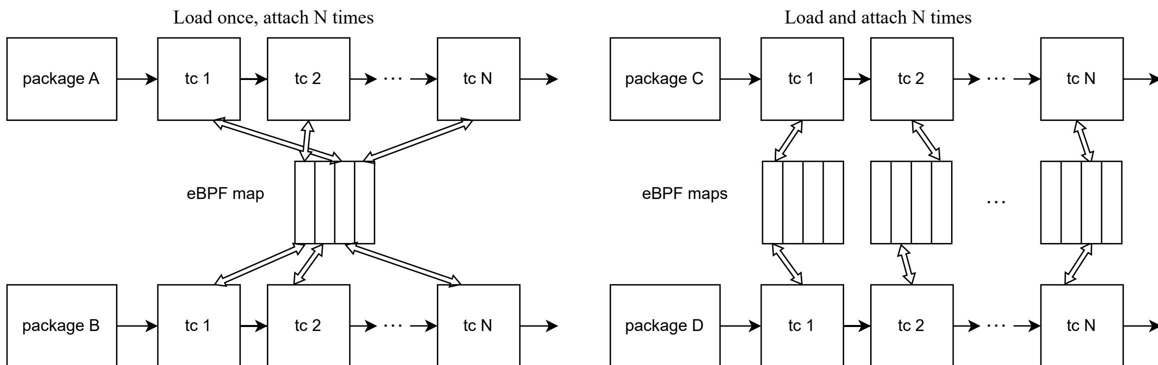


Figure 3.3: eBPF map sharing and access patterns.

times creates a scenario where each eBPF program in the chain has its own eBPF map. Despite this, map sharing may still occur when packets are processed simultaneously. For instance, if multiple packets trigger instances of the same eBPF program (with identical IDs), these instances may share a map. Unlike the previous configuration, the total number of maps now scales with the chain length, increasing the total number of map entries. While this alleviates the aggravation of write conflicts with a growing chain length, the increased number of maps consumes more kernel space and introduces additional cache misses.

To address this problem, the eBPF community provides per-CPU maps, which maintain a separate array for each logical CPU. When an eBPF program accesses the map, it implicitly uses the array assigned to the CPU it is currently running on. This guarantees that no other program can concurrently access the same memory, improving performance. However, this approach comes at the cost of increased memory usage, as the number of maps scales with the number of logical CPUs rather than being a single map.

3.3.2 Scaling with CPU Cores

Increasing the number of CPU cores allows a web server to process more requests simultaneously, thereby increasing throughput. However, it also results in more eBPF programs running concurrently, which may introduce interference and prevent throughput from scaling linearly with the number of CPU cores.

To investigate this behavior, we enable the controller to dynamically adjust the CPU core allocation for the HTTP load generator. Additionally, we monitor the CPU usage of the echo server to ensure it does not fully utilize the CPU resources assigned to it. This guarantees that the echo server does not become a bottleneck, allowing us to focus on analyzing the performance of the HTTP load generator under varying CPU core allocations.

Chapter 4

Evaluation

4.1 Benchmark Setup

We set up a 10-core virtual machine on a dual-socket server equipped with Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz, with each socket containing 12 cores. The virtual machine operates on a single socket, exclusively utilizing its cores and remaining isolated from other users. The server's cache sizes are 32KB per core for L1, 256KB per core for L2, and 30MB shared for L3.

We use *k6* [2] as the HTTP load generator. It sends HTTP requests based on a given configuration and collects metrics. Specifically, we use the `http_req_duration` metric to measure the total round-trip latency of requests.

The echo server, running on separate cores, responds with the same payload it receives. As a result, an attached eBPF program is triggered twice for each request: once for the HTTP request and once for the HTTP response. Additionally, we implement a filtering mechanism in the eBPF program to process target package selectively.

4.2 eBPF Program Triggering

In this section, we evaluate the performance impact of triggering an empty eBPF program. This program performs no operations and returns immediately upon being triggered. Note that the triggering process contributes only a small portion of the overall latency compared to typical network latency. To ensure accurate results, we minimize and stabilize the original HTTP round-trip latency.

For this experiment, we allocate one core each to *k6* and the echo server, with a maximum of one HTTP connection to eliminate context switching and reduce variability. Additionally, *k6* sends HTTP GET requests, which have no payload and a very short total length, enabling the echo server to respond quickly.

We measure the round-trip latency under three different eBPF trigger points: `tc`, `xdp`, and `sk_msg`. These results are compared with the baseline scenario where no eBPF program is attached. The eBPF program is triggered twice per HTTP round trip: once for the request and once for the response. For `sk_msg`, an additional step involves calling `sockops` to store the socket in a `sockmap`.

Figure 4.1 shows the average, median, and 95th percentile latencies of HTTP round trips with and without attached eBPF programs. The trigger times for `tc` and `xdp` are similar. However, `sk_msg` has a longer trigger time, as it involves calling additional `sockops` functions in HTTP connections. Nonetheless, the triggering times, measured in microseconds, are negligible compared to typical network latencies.

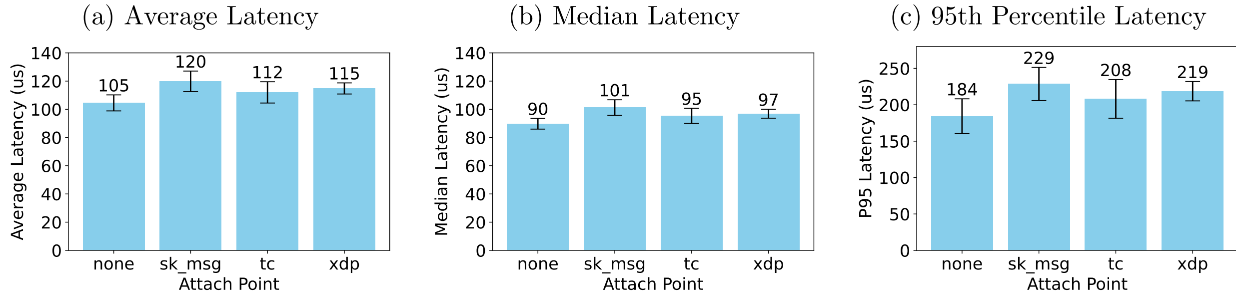


Figure 4.1: HTTP round-trip latency across different eBPF attach points.

4.3 Benchmark Results

In this section, we evaluate the performance and scalability of eBPF-based applications under different configurations. We use *k6* as the load generator, maintaining 100 connections to send HTTP POST requests with a 200-byte random string payload. The total number of packets sent remains constant across experiments.

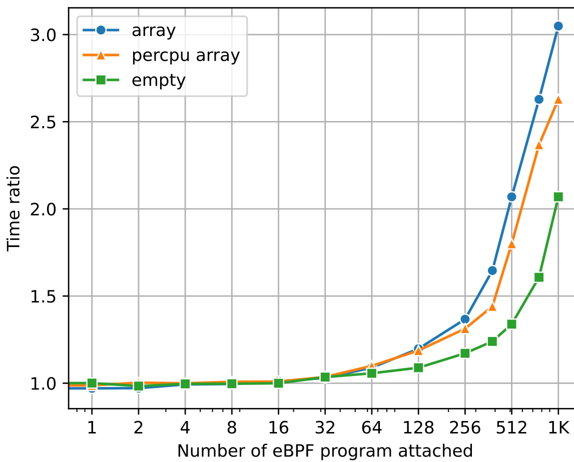
4.3.1 Scaling with eBPF Programs

As mentioned in Section 3.2, we can trigger an eBPF program chain at the `tc` attach point. Here, we assess how the length of the eBPF program chain affects performance. The eBPF programs in the chain are identical except for their triggering order. Each program uses a map with 1024 entries, where each entry is 256 bytes. The map type can be an `ARRAY` or a `PERCPU_ARRAY`, the latter maintaining a separate array for each logical CPU. We allocate five CPU cores for the HTTP load generator in experiments.

The eBPF program initially filters packets on the loopback interface, processing only HTTP requests from the load generator. It stores the HTTP payload into a randomly selected entry in the eBPF map. As a result, the eBPF program runs predominantly in the kernel space of the HTTP load generator.

We measure the HTTP round-trip latency and normalize it against the baseline latency (without any eBPF program attached). To identify performance bottlenecks, we compare the results with an "empty" eBPF chain that triggers but does not update the eBPF map. Additionally, we use *Perf* to measure L3 cache misses—specifically, those that miss the L3 cache and are served from DRAM.

In Figure 4.2, we observe that latency increases as more eBPF programs are attached. This is due to the additional CPU time required for packet filtering and processing in the eBPF chain. Comparing the eBPF chain that updates an entry in a `PERCPU_ARRAY` map with one that only filters, we find that both have similar L3 cache miss rates. However, the latency is higher for the map-update case, attributable to the overhead of packet reads and map update function calls. When using a normal `ARRAY` map instead of a `PERCPU_ARRAY`, updates from multiple connections on different cores to the same map location invalidate cache lines, leading to increased L3 references, cache misses and higher latency.



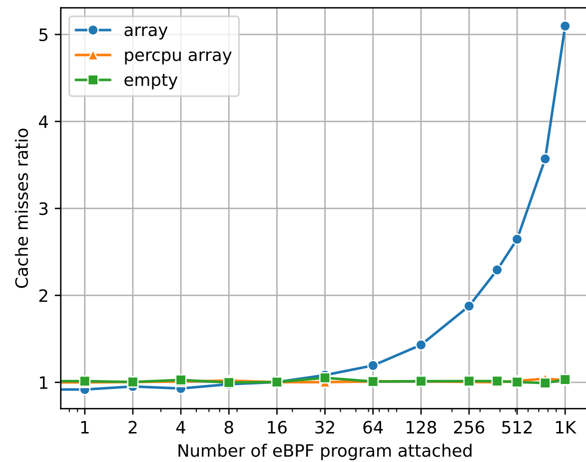
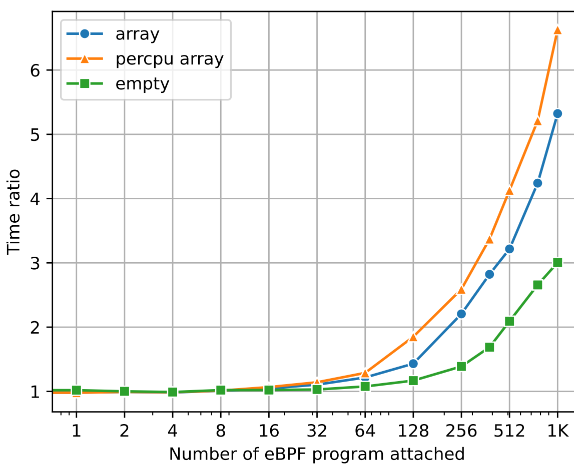


Figure 4.2: Configuration 1: Load Once, Attach N Times



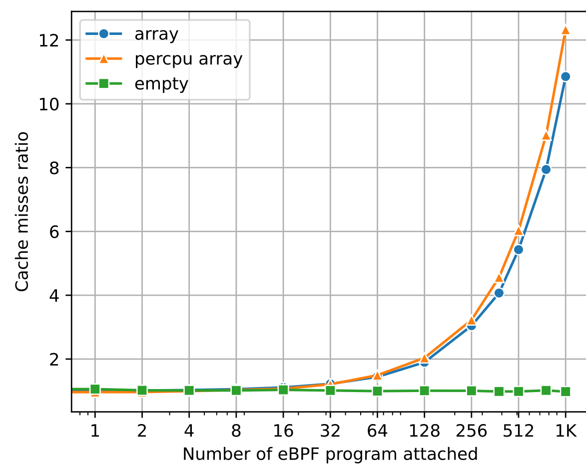


Figure 4.3: Configuration 2: Load and Attach N Times

In Figure 4.3, we analyze the case where each eBPF program is loaded and attached repeatedly. In this configuration, maps are not shared among eBPF program instances. Although write conflicts still exist when updating `ARRAY` maps, the total map size scales with the number of eBPF programs. Thus, the severity of write conflicts does not worsen with the number of programs. However, the total map size increases with the number of eBPF programs, leading to more L3 cache capacity misses, contributing to higher latency. This effect is even more pronounced with `PERCPU_ARRAY` maps, as they require additional space to maintain separate arrays for each logical CPU core.

4.3.2 Scaling with eBPF Map Size

In the previous benchmark, we observed a trade-off between write conflicts and total map size, which are influenced by the memory access range, access patterns, and update frequency. In this section, we fix the update frequency and evaluate how round-trip latency and cache misses scale with map size. Here, we load a single eBPF program and attach it 64 times, allowing all instances to share the same map. Instead of performing one update per eBPF program instance, we perform 10 updates per instance with two different access patterns:

- **Write 10x1:** Each eBPF program instance updates 10 random, different entries in the map.
- **Write 1x10:** Each eBPF program instance updates one random entry 10 times.

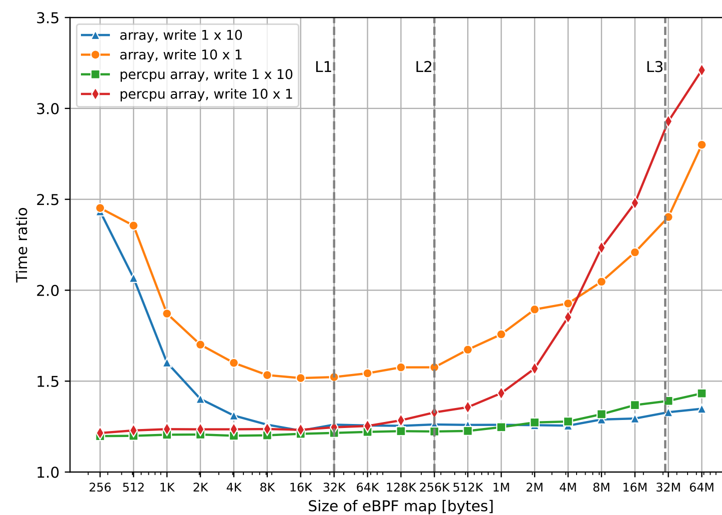


Figure 4.4: Normalized Average HTTP Request Latency

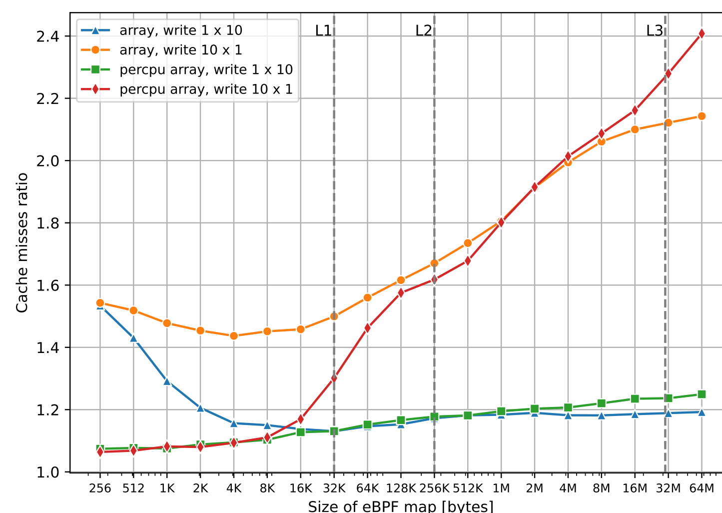


Figure 4.5: Normalized L1d Cache Miss

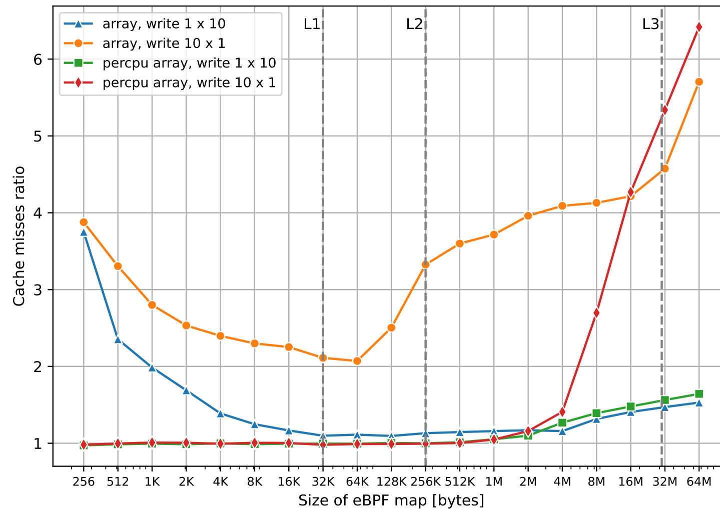


Figure 4.6: Normalized L3 Cache Miss

We normalize the results against a baseline configuration with no eBPF program attached. Figure 4.4 shows the normalized average round-trip latency, Figure 4.5 shows the L1 data cache misses, and Figure 4.6 shows the L3 cache misses as the size of the eBPF map scales.

For the `ARRAY` map: - Both access patterns start with high latency due to small map sizes causing more contention among eBPF program instances. As the map size increases, the memory access range grows, reducing write conflicts and subsequently lowering latency. - The **Write 10x1** access pattern, which accesses more memory addresses, incurs more capacity misses as the map size increases. This leads to a rapid latency increase for large map sizes. - The **Write 1x10** access pattern access less memory addresses. Consequently, it results in fewer cache misses and slower latency growth.

For the `PERCPU_ARRAY` map: - It performs well for small map sizes, as there are no write conflicts. However, latency starts to increase as the map size exceeds the L1 cache size and grows rapidly when the map size surpasses the L2 cache size. - For large map sizes, `PERCPU_ARRAY` maps require more memory space because they allocate separate arrays for each logical CPU. This leads to more capacity misses and significantly higher latency compared to `ARRAY` maps.

If we configure `PERCPU_ARRAY` maps to have the same total size as `ARRAY` maps while considering the number of cores, it requires setting significantly fewer entries for `PERCPU_ARRAY` maps in an eBPF program before compilation. In this way, `PERCPU_ARRAY` maps can offer better performance by eliminating write conflicts. However, they may require additional functionality to handle distributed data across cores, introducing extra complexity.

Overall, for workloads with working sets larger than the L1 cache, `ARRAY` maps may be more reasonable to use despite the existence of write conflicts.

4.3.3 Scaling with CPU Cores

The write conflicts and memory space requirements of `PERCPU` maps are related to the number of CPU cores allocated to eBPF-based applications. To investigate this relationship, we conducted a benchmark where we scaled the number of CPU cores. In this benchmark, a single eBPF program was loaded and attached 64 times at the `tc` hook point, with 1 to 6 CPU cores allocated to the HTTP load generator.

We measured the throughput normalized against the throughput without any eBPF program attached and running on a single core. Figures 4.7a and 4.7c show results for `ARRAY` maps, while Figures 4.7b and 4.7d show results for `PERCPU` maps.

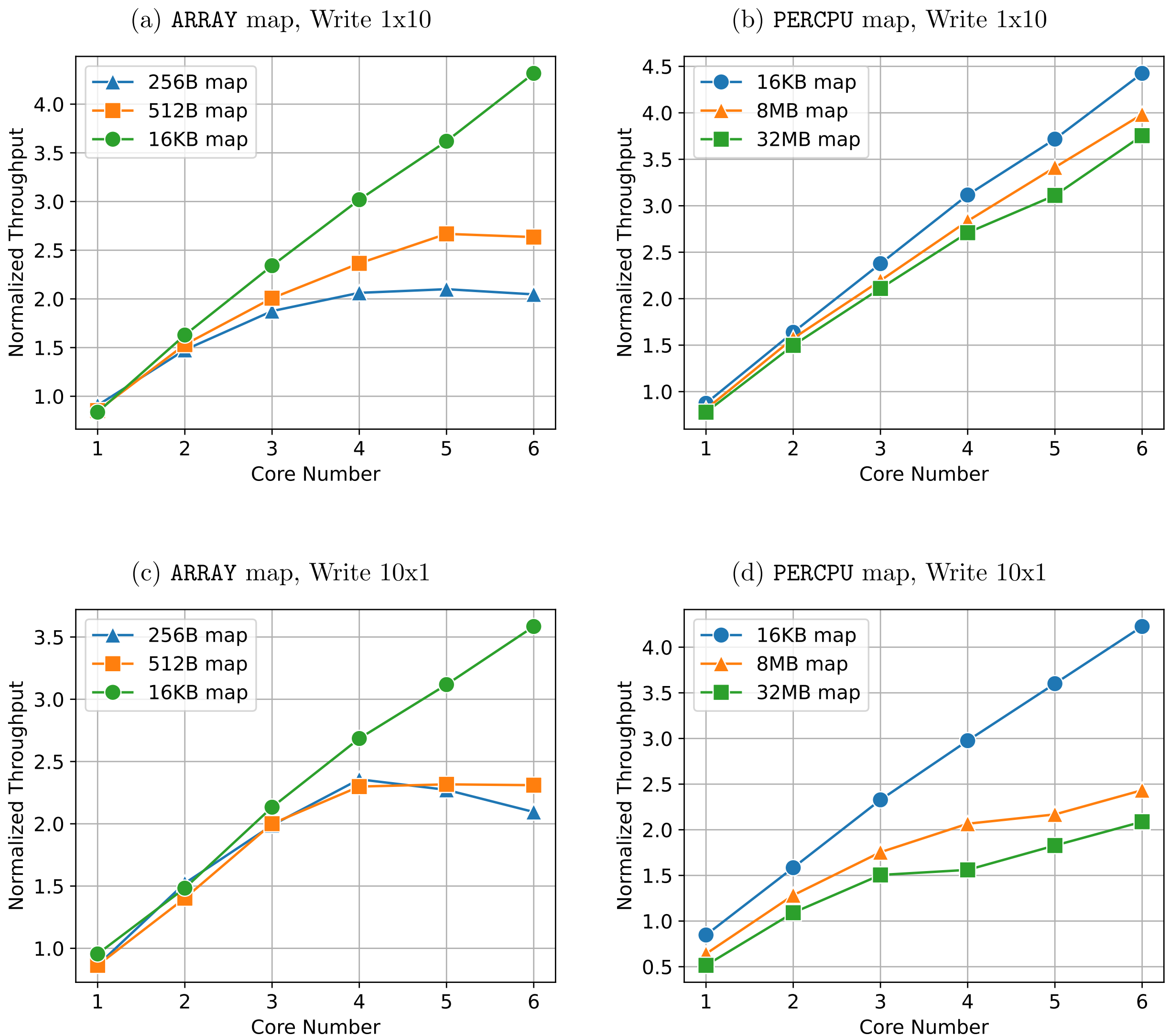


Figure 4.7: Normalized throughput scaling with CPU cores

For `ARRAY` maps: - In **Write 1x10** pattern (Figure 4.7a) and **Write 10x1** (Figure 4.7c) pattern, the throughput scales well when the map size is large enough but still smaller than the L1 cache size. However, for small map sizes, write conflicts worsen as the number of cores increases, even causing throughput to decrease with additional cores.

For `PERCPU` maps: - In **Write 1x10** pattern (Figure 4.7b) and **Write 10x1** (Figure 4.7d) pattern, small map sizes show better scalability. When the map size is smaller than the L1 cache, it achieves near-linear scalability with the number of cores. In the **Write 10x1** pattern (Figure 4.7d), which accesses more memory addresses, scalability becomes more sensitive to map size. This is reflected in the larger gaps between lines representing different map sizes.

Overall, the choice between `ARRAY` and `PERCPU` maps depends on the access pattern, map size, and the number of cores. While `PERCPU` maps avoid write conflicts, their memory overhead can reduce performance and scalability in cases with large memory access range and high memory access diversity.

Chapter 5

Outlook

Evaluating an eBPF system is a complex task. Numerous factors can affect the performance of eBPF-based applications, making it challenging to achieve optimal results. In this thesis, we evaluated various factors, including attach points, the number of eBPF programs, eBPF map types, map sizes, access patterns, and the number of CPU cores. When deciding to adopt eBPF technology to accelerate a network application, it is essential to consider the specific use case, performance targets, computing resources, budget constraints, and to choose the appropriate approach for designing eBPF programs.

Regarding future work, further benchmarking could be conducted across a broader range of eBPF map types and configurations. For instance, we could compare performance under conditions where the total map sizes remain constant while accounting for the number of CPU cores. Additionally, exploring the simultaneous execution of eBPF programs attached at different points and analyzing their potential interference could offer valuable insights for optimizing eBPF-based systems. Furthermore, we could extend the evaluation to different network applications, each with varying performance targets, to better understand the adaptability and effectiveness of eBPF in diverse scenarios.

Chapter 6

Summary

This thesis explores the performance of eBPF-based applications in multi-core and networked environments. We developed a benchmark system to evaluate how configurations such as the number of eBPF programs, eBPF map types, CPU allocations, and memory access patterns impact performance. The study reveals trade-offs in latency, computing resources, and scalability, providing valuable guidelines for designing efficient eBPF applications.

Bibliography

[1] Dpdk. `https://github.com/DPDK/dpdk`.

[2] k6. `https://github.com/grafana/k6`.

[3] libbpf. `https://github.com/libbpf/libbpf`.

[4] libbpf-rs. `https://github.com/libbpf/libbpf-rs`.

[5] Almesberger, W., et al. Linux network traffic control—implementation overview, 1999.

[6] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 51–66.

[7] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. The express data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2018), CoNEXT '18, Association for Computing Machinery, p. 54–66.

[8] Hubert, B., et al. Linux advanced routing & traffic control howto. *Netherlabs BV 1* (2002), 99–107.

[9] Liu, C., Tak, B., and Wang, L. Understanding performance of ebpf maps. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions* (New York, NY, USA, 2024), eBPF '24, Association for Computing Machinery, p. 9–15.

[10] McCanne, S., and Jacobson, V. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter* (1993), vol. 46, Citeseer, pp. 259–270.

[11] Messina, A. *Analysis and Testing of eBPF Attack Surfaces.* PhD thesis, Politecnico di Torino, 2024.

[12] Miano, S., Bertrone, M., Risso, F., Tumolo, M., and Bernal, M. V. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)* (2018), pp. 1–8.

[13] Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K., and Carle, G. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)* (2018), vol. 1, IEEE, pp. 209–217.

[14] Thomadakis, M. E. The architecture of the nehalem processor and nehalem-ep smp platforms. *Resource 3*, 2 (2011), 30–32.

[15] Vieira, M. A., Castanho, M. S., Pacífico, R. D., Santos, E. R., Júnior, E. P. C., and Vieira, L. F. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR) 53*, 1 (2020), 1–36.