RS3Lab

École Polytechnique Fédérale de Lausanne

BPFNic: An exploration of receive-side scaling implemented in eBPF

by Ethan Graham

Bachelor Project Report

Approved by the Examining Committee:

Prof. Sanidhya Kashyap
Thesis Advisor

Kumar Kartikeya Dwivedi
Thesis Supervisor

EPFL IC IINFCOM RS3LAB
INN 240 (Bâtiment INN)
Station 14
CH-1015 Lausanne

June 6, 2024

The venn diagram between people who construct
`/proc` string formats and demons torturing people
with hot pokers in hell is a full circle.
— Kartikeya

Dedicated to RS3Lab servers six, eight, and nine.

Acknowledgments

I would like to thank Mari, who has put up with my rambling on about eBPF for the past three months, as well as the support she gives me on a daily basis.

I would like to thank my parents who have supported me this semester and throughout the entirety of my undergraduate studies at EPFL.

I would like to thank Yugesh Kothari who kindly answered my eBPF questions despite being a TA in an unrelated course.

And finally I would like to thank Kumar Kartikeya Dwivedi for supervising and guiding me while I made sense of this topic which I was unfamiliar with when I started working on it. I've learned a lot and for that I am grateful.

Lausanne, June 6, 2024 Ethan Graham

Abstract

Receive-side scaling (RSS) is a crucial technique for improving network performance in modern high-throughput systems by efficiently distributing incoming packet processing across multiple CPU cores. This load balancing is vital to fully utilize multi-core processors and prevent bottlenecks that can degrade system performance. Traditional dataplane operating systems, designed for high-performance packet processing, often employ static load balancing schemes or specialized hardware, which can lead to trade-offs in terms of flexibility, scalability, and ease of deployment. These systems may lack the adaptability required to optimize for diverse and dynamic traffic patterns, and their complexity can pose challenges for development and maintenance.

In contrast, the emergence of eBPF (extended Berkeley Packet Filter) as a programmable, high-performance packet processing framework offers a compelling alternative. eBPF allows for dynamic and fine-grained control over packet handling directly within the Linux kernel that can be controlled and monitored from user-space, enabling more responsive and adaptable RSS implementations. This programmability facilitates the development of custom load balancing algorithms that can adjust to real-time traffic conditions, offering improved efficiency and reduced overhead compared to traditional methods. Moreover, eBPF's ability to be deployed without significant changes to existing infrastructure makes it a versatile solution for a wide range of use cases. My thesis explores the advantages of eBPF-based receive-side scaling, highlighting how it addresses the limitations of conventional dataplane operating systems and demonstrating its potential for enhancing network performance in diverse environments.

Contents

Acknowledgments **1**

Abstract **2**

1 Introduction **5**

2 Background **6**
2.1 Definitions 7

3 Design **9**
3.1 Client Program 9
3.1.1 Request Format 9
3.1.2 Traffic Generation and Book-Keeping 10
3.1.3 Orchestrating the Benchmark 10
3.2 Server Program 11
3.2.1 Scheduler Program 11
3.2.2 Cpumap Program 12
3.3 Scheduling Policies 12
3.3.1 Bimodal: Round-robin with no core separation 12
3.3.2 Bimodal: Round-robin with core separation 12
3.3.3 Unimodal: Round-robin with dynamic core group 12

4 Implementation **13**
4.1 Client Suite 13
4.2 Server Suite 14
4.2.1 Live CPU Utilization Metric 14
4.2.2 Packet Filtering 16
4.2.3 Artificial Workload 16
4.2.4 IP Header Re-Computation 17
4.2.5 State in eBPF Programs 17
4.2.6 Dynamic Core Group Logic 18

5 Evaluation **19**
5.1 Bimodal Traffic . . . 19
5.1.1 Experiment Configuration . . . 19
5.1.2 Results and Observations . . . 20
5.1.3 Concluding Notes . . . 24
5.2 Unimodal Traffic . . . 25
5.2.1 Experiment Configuration . . . 25
5.2.2 Results and Observations . . . 25
5.2.3 Performance Mode versus Power-saver Mode . . . 26
5.2.4 Dynamic Round-Robin versus Static Round-Robin . . . 27

6 Related Work **29**
6.1 Dataplane Operating Systems . . . 29
6.2 eBPF . . . 30

7 Conclusion **31**

Bibliography **32**

Chapter 1

Introduction

The ever-growing demand for high-speed network processing presents a significant challenge for modern servers. Network interface cards (NICs) can often receive data at rates far exceeding a single CPU's ability to handle it, creating inefficiencies that lead to dropped packets, increased latency, and ultimately, a degraded user experience.

To address this challenge, a technique called receive-side scaling (RSS) has been employed. RSS distributes incoming traffic across multiple CPU cores, leveraging the processing power of parallel hardware and achieving better resource utilization. However, traditional implementations of RSS rely on static configuration or pre-defined flow hashing algorithms, which can become inefficient in dynamic network environments subject to different requests types, or bursty traffic.

Dataplane operating systems such as IX [1], ZygOs [3] and Shinjuku [2] offer high-performance RSS, but lack the the fine-grained control and dynamic adaptation capabilities needed for diverse network workloads.

This paper proposes a novel approach utilizing eBPF for dynamic and efficient receive side scaling. eBPF allows us to sandbox programs at strategic points within the Linux kernel's network processing pipeline. This fine-grained control enables us to design intelligent packet steering algorithms that can adapt to real-time network conditions, maximizing resource utilization and minimizing processing overhead.

The core contribution of this work is the development of a client-server suite for benchmarking the performance of various eBPF RSS policies against varying workloads. The code can be found in the project's GitHub repository: `https://github.com/rs3lab/bpfnic`

Chapter 2

Background

While dataplane operating systems [1] [2] [3] offer exceptional performance, they often require substantial modifications to the underlying infrastructure and may not integrate seamlessly with existing systems. This is where eBPF offers a compelling alternative. eBPF is a powerful technology within the Linux kernel that allows for the execution of custom bytecode in response to various events, including network packet processing at the driver-level.

eBPF RSS leverages this capability to distribute network traffic across multiple CPU cores, enhancing parallel processing and improving throughput. By being directly integrated into the Linux kernel, eBPF RSS provides several advantages over traditional dataplane OSs:

1. Seamless Integration: eBPF can be deployed without requiring significant changes to the existing system architecture, making it easier to adopt in a wide range of environments.
2. Flexibility: The programmable nature of eBPF allows for dynamic adjustments and fine-tuning of network processing policies, catering to specific application requirements.
3. Unified Management: As part of the Linux kernel, eBPF benefits from the extensive toolset and ecosystem available for Linux, simplifying management and monitoring tasks.

This paper aims to evaluate the performance of RSS policies implemented in eBPF through a series of benchmarks, namely how throughput affects tail latency on a μ second scale. Furthermore, I will highlight the added flexibility and ease of integration that eBPF RSS offers, making it a viable alternative for environments where modifying the existing infrastructure is impractical or undesirable.

The results of this study will provide valuable insights into the trade-offs between different RSS policies leveraging eBPF within the Linux kernel. Through comprehensive analysis, I aim

to contribute to the ongoing discourse on network performance optimization and offer practical guidelines for practitioners seeking to enhance their network processing capabilities.

2.1 Definitions

Definition 2.1 (NIC) *A Network Interface Card (NIC) is a hardware component that interfaces with the system's kernel to manage network communications, providing low-level control over data packet transmission and reception. It includes device drivers that integrate with the kernel's networking stack, facilitating efficient data processing and interrupt handling.*

Definition 2.2 (RSS) *Receive-Side Scaling (RSS) is a networking technique that distributes incoming network traffic across multiple CPU cores in a receiving system, improving the scalability and performance of network processing. By distributing the workload, RSS reduces the likelihood of bottlenecks and maximizes the utilization of available processing resources, especially in high-throughput and low-latency networking environments.*

Definition 2.3 (Queuing Delay) *The time a data packet spends waiting in a queue before it can be transmitted or processed in a network.*

Definition 2.4 (Service Time) *Service time is the duration it takes for a server to process a packet from the moment it starts handling it until the processing is complete. Happens after queuing delay by design.*

Definition 2.5 (Head-of-line Blocking) *Head-of-line blocking in networking refers to a situation where a queued packet at the head of a line (queue) delays the processing of subsequent packets, leading to increased latency and degraded performance for other packets waiting in the queue.*

Definition 2.6 (RTT) *Round-trip time (RTT) is the duration it takes for a packet to travel from a source to a destination and back again. Includes queuing delay and service time at both the source and destination by definition.*

Definition 2.7 (eBPF) *eBPF (extended Berkeley Packet Filter) is a powerful and flexible virtual machine within the Linux kernel that allows users to write custom programs in a restricted C-like language, which are then compiled into bytecode. This bytecode runs safely and efficiently within the kernel, enabling high-performance tasks such as network packet filtering, performance monitoring, and security enforcement. Writing in eBPF allows developers to execute user-defined logic directly within the kernel, providing deep observability and control without the need for kernel modifications or extensive performance overhead.*

Definition 2.8 (eBPF hook) *An eBPF hook is a designated point within the Linux kernel where an eBPF program can be attached and executed. These hooks allow eBPF programs to monitor or modify kernel behavior at specific events, such as system calls, network packets, or tracepoints, enabling custom, real-time processing and analysis of system activities.*

Definition 2.9 (XDP) *XDP (eXpress Data Path) is a high-performance, programmable network data path in the Linux kernel that allows for the processing of packets at the lowest level of the networking stack, directly in the network driver. It enables developers to write custom packet processing programs in eBPF, which can be loaded into the kernel and executed at various hook points, significantly reducing latency and increasing throughput. XDP is particularly useful for applications requiring high-speed packet filtering, load balancing, and DDOS protection.*

Definition 2.10 (Tail Latency) *Tail latency refers to the maximum or extreme end of the latency distribution in a system, representing the longest delays experienced by a small fraction of requests or operations. It highlights the outliers or worst-case scenarios in terms of response time, which can significantly impact user experience and system performance.*

Definition 2.11 ($M/G/k$ Queue) *A $M/G/k$ queue is a stochastic queuing model characterized by having Poisson arrivals, general service time distributions, and a finite number of servers. It is used to analyze systems where customers arrive according to a Poisson process, are served by a variable-duration service process, and there are a fixed number of servers available for service.*

Definition 2.12 ($M/G/k/N$ Queue) *A $M/G/k/N$ queue is a stochastic queuing model that extends the $M/G/k$ queue by incorporating a finite buffer size N to accommodate waiting customers, where excess arrivals beyond the buffer capacity are rejected. It is used to analyze systems with Poisson arrivals, general service time distributions, a fixed number of servers, and a maximum queue size, providing insights into performance metrics such as average waiting time and queue length.*

Chapter 3

Design

To effectively benchmark RSS policies running at μ-second latencies and at high throughputs, a reliable and high-performance benchmark suite was needed. This consists of a client-side and a server-side program running on two different systems communicating over a network. The client sends requests to the server, which are processed, and responded to.

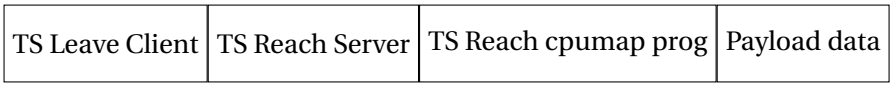
3.1 Client Program

The client-side program is tasked with generating high throughput traffic for the server, and doing any book-keeping necessary to allow for analysis of the numbers afterwards. Furthermore, it should also be able to generate different types of traffic simulating different workloads. We are most interested in queuing delay at the server, but the client program is also capable of tracking round-trip times.

3.1.1 Request Format

The first consideration is related to keeping state efficiently. A possible approach would have perhaps been to embed within a network packet some identifier, and then maintain some mapping of round-trip times by matching a packet's return time to when it was initially sent. Although this would provide additional information about dropped packets and the order in which packets are processed by the server, it was deemed unnecessary in the scope of this project.

Instead, a stateless approach was opted for. A benchmark packet consists of 25 bytes of data that is sent to the server, processed, and then sent back. It consists of three eight-byte timestamps and an additional one byte of payload data used for embedding information for the server.



This solution allows us to efficiently infer both the round-trip time and queuing delay of a packet, and embed an intended processing time for the server.

3.1.2 Traffic Generation and Book-Keeping

UDP was the choice of protocol given that reliability was not necessary: if packets are dropped, then the client program should know about it, but should not try and recover. This could perhaps be problematic in a production environment as it offloads reliability of communication from the kernel to the user-space application.

Although several layers of abstraction were built on top of this, at its essence the client program is running two threads per UDP socket; one for sending all outgoing traffic, and one for handling all incoming traffic. No data races occur between the two threads operating on the same UDP socket; under the hood, they are operating on different queues.

These two threads are wrapped by a `Client` object, which also handles the number-related book-keeping for all returning packets, which it does by maintaining histograms for the desired metrics. Histograms were chosen for this as the sheer volume of traffic would cause the size of the output data to be too large.

We can increase the throughput of the client program by increasing the number of `Client` objects. The increase is theoretically linear in the number of `Clients`, and at a count of 32, a throughput of approximately 6 million Rps can be generated and sent, which is more than sufficient for benchmarking the server.

3.1.3 Orchestrating the Benchmark

To be able to manage an arbitrary number of `Client` objects, a `Benchmark` object was defined for coordinating the duration of a benchmark cycle, and write any necessary results out for future processing.

Concretely, it manages a vector of `Client` objects, runs them for a set duration, merges all of their histograms together, and writes the result out in a comma-separated format allowing for easy data processing.

3.2 Server Program

The server system hosts the eBPF scheduling policies that are being tested, thus the server program must handle the lifecycle of any attached eBPF programs, as well as providing any user-space data needed for their correct execution.

At a high level, the server-side program is running one scheduler sub-program, and `num_cpus` sub-programs on the target CPUs simulating an artificial workload. These subprograms are implemented in eBPF, and execute in the kernel. They can communicate with a user-space program after being attached via eBPF maps, although this is not strictly necessary if the program is self-contained after launch.

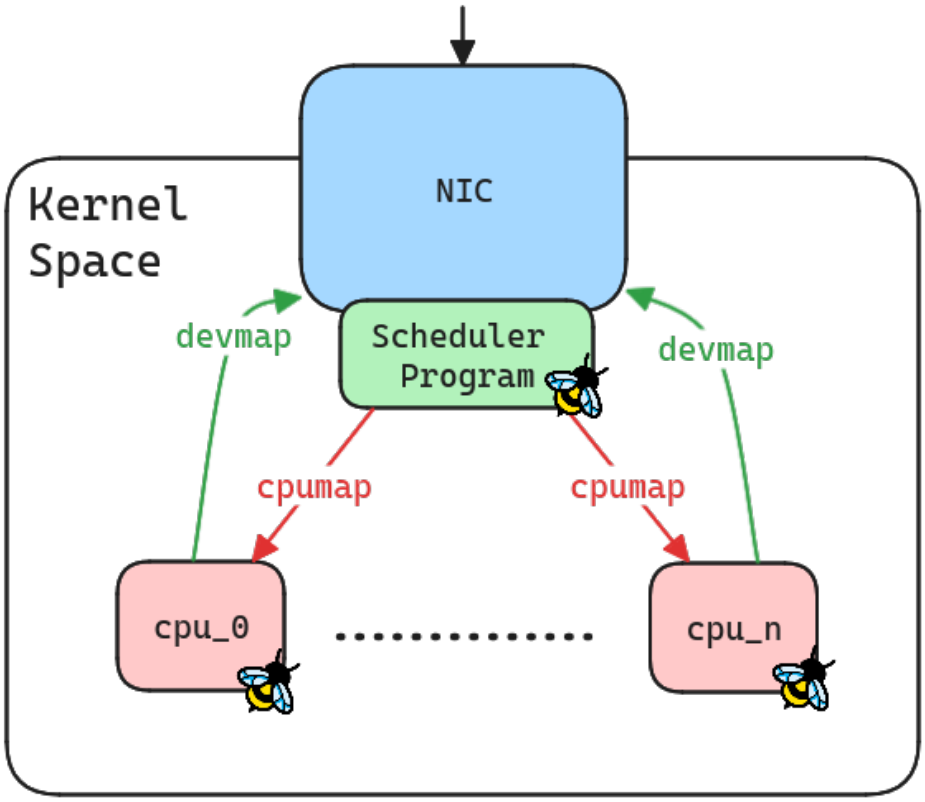


Figure 3.1: Design of the server-side program, an *eBee* representing an eBPF hook.

3.2.1 Scheduler Program

This program is executed by whichever CPU handles the NIC interrupt for a given packet at the driver level, and redirects the request to a target CPU based on some scheduling policy. See 3.3.

3.2.2 Cpumap Program

This eBPF program is executed by the target CPU by a cpumap kthread. Consists of an artificial workload, and redirection back to the client.

3.3 Scheduling Policies

The tested scheduling policies can be divided into two categories

1. Bimodal: Requests with long artificial workload and short artificial workload
2. Unimodal: All requests have the same artificial workload duration

3.3.1 Bimodal: Round-robin with no core separation

Given a set of n cores, this policy redirects received packet $i+1$ to the $(i+1\ mod\ n)^{th}$ available core.

3.3.2 Bimodal: Round-robin with core separation

Applies the round-robin policy to two separate core groups: one reserved for long requests, one reserved for short requests. At packet arrival, the scheduler program inspects the packet payload to decide which core group it will redirect the packet to.

3.3.3 Unimodal: Round-robin with dynamic core group

Applies the round-robin policy to a core group that varies in size based on some metric. For example, if average queuing delay exceeds some threshold value, the core group will allocate an extra core to the core group. Currently, this policy is only capable of growing the core group, but future work could consist in implementing the logic for shrinking it as well and would not require much modification.

Chapter 4

Implementation

The language of choice for both the client and server userspace programs was C++, which was chosen for performance reasons, as well as for interoperability with linux header files and libbpf which was needed for managing loaded eBPF programs. Another possibility would have been Rust which is gaining traction in systems programming and satisfies the requirements. The eBPF programs are written in constrained C code.

4.1 Client Suite

3.1 touches on the high-level design of the client program, and mentions that each `Client` object is running two threads, one for sending and one for receiving packets respectively, and was managed by a `Benchmark` abstraction. Two different types of client class were defined: one which generates bimodal traffic with a 9:1 short requests over long requests ratio, and one which generates traffic at a capped throughput set by the managing benchmark class. Both inherit from a base class `AbstractClient` that provides an interface that they both implement as well as defining some shared methods.

```
1  class AbstractClient {
2   public:
3    //...
4    void recvLoop() {
5      while (!stopFlag) {
6        numReceivedPackets++;
7        if (recvAndProcessPacket() != Err::NoError) std::cerr << "Invalid
             packet format...\n";
8
9        // helpful for testing functionality at low throughput
10       if (testMode) std::this_thread::sleep_for(std::chrono::seconds(1));
11     }
```

```
  }

  virtual void sendLoop() = 0;
  //...
};

class AbstractBenchmark {
 public:
  //...
  void launchClients() {
    for (auto& client : clients) {
      client->start();
      threadPool.enqueue([&client] { client->sendLoop(); });
      threadPool.enqueue([&client] { client->recvLoop(); });
    }
  }
  //...
};
```

Separating sending logic and receiving logic was key in achieving a high throughput; the `Benchmark` program maintains threadpool, and enqueues the loops. The number of threads allocated to the threadpool is hardcoded to be $2 \times$ `num_clients` to ensure that there is at least one thread available for every enqueued loop for every client. Once the duration of the benchmark is over, it will set a stop flag in the client classes, merge their histograms, and write the result out. Note here that `sendLoop()` is a pure virtual function; this is because the capped throughput client class and bimodal client class exhibit different behavior when generating traffic despite having the same behavior for processing incoming packets.

Since all traffic is being sent over UDP, everything is done on a best-effort basis without emphasis on reliability. Outgoing and incoming packets are counted and logged to `stdout`, giving us an exact count of how many packets were dropped. However, the root cause of dropped packets is not investigated further by the program. A tool such as `xdp-monitor` on the server can be used for this, and compared to the number of dropped packets on the client side to determine which packets are dropped by which system.

4.2 Server Suite

4.2.1 Live CPU Utilization Metric

This class searches through the `/proc` directory to compute the per-window CPU utilization of the cpumap kthreads associated to the cpumap programs. The window is defined by the caller - i.e. everytime CPU utilization is probed, it will either count CPU utilization since the process was

launched, or CPU utilization since the last probe. The high-level idea is to find all `/proc/[pid]` subdirectories that contain substring `cpumap` in their names as read from the `status` file, and then compute their CPU utilization with the following expression:

$$\texttt{cpu\_utilization} = \frac{\text{Process clock ticks in kernel mode since last probe}}{\text{Total clock ticks since last probe or since } \texttt{starttime}}$$

Concretely, the computation of CPU utilization is implemented as follows:

```
// @return cpu utilization of a process with pid ‘pid‘ since the last probe
    as a fraction between [0, 1]
std::optional<double> computeCpuUtilization(int pid) {
  auto statContentsOpt = readStat(pid);
  if (statContentsOpt == std::nullopt) return std::nullopt;
  StatContents statContents = statContentsOpt.value();

  // get memoized data for this process or insert it
  MemoizedData memoizedData;
  auto itValue = memoizedDataMap.find(pid);
  if (itValue != memoizedDataMap.end())
    memoizedData = itValue->second;
  else {
    MemoizedData defaultVal = {.cyclesCounted = 0, .prevProbe =
        statContents.starttime};
    memoizedDataMap.insert(std::make_pair(pid, defaultVal));
    memoizedData = defaultVal;
  }

  auto currUptime = getUptimeTicks();
  if (currUptime == std::nullopt) return std::nullopt;

  long activeTicksInWindow = statContents.stime -
      memoizedData.cyclesCounted;
  long totalTicksInWindow = currUptime.value() - memoizedData.prevProbe;

  memoizedData.prevProbe = currUptime.value();
  memoizedData.cyclesCounted = statContents.stime;
  memoizedDataMap[pid] = memoizedData;

  if (activeTicksInWindow > totalTicksInWindow) {
    std::cout << "\033[33mWarning: active ticks > total ticks! pid = " <<
        pid << "\033[0m" << std::endl;
    return 1.0;
  }

  return ((double)activeTicksInWindow / (double)totalTicksInWindow);
}
```

This can be leveraged by the server user-space program as a direct metric of CPU utilization, and is more reactive than running `ps` or `htop or htop`. It is always displayed by the program, and can also be used, for example, by the dynamic core group policy as a metric for growing the core group size (this was implemented, but will not be discussed further in this paper as it was observed that a threshold latency worked better as a metric).

All information is read directly from the `/proc` pseudo-filesystem, which is held in-memory despite appearing as a normal directory. Every file in this directory is plaintext, which made parsing it quite a challenge.

4.2.2 Packet Filtering

The eBPF redirection policies are attached directly to the NIC, and operate at the driver level. If care isn't taken, any and all packets will be processed by the benchmark suite, causing an unresponsive server until the program is unloaded. To address this, all requests are processed by a function `bpfnic_benchmark_parse_and_timestamp_packet()` that returns `false` if the request is unrelated to the benchmark program, otherwise it timestamps the packet in-place and returns `true`. A request is determined to be benchmark-related by comparing the destination port number from the request's UDP header to the value set in user-space at program launch. The UDP header is read from the raw packet data.

```
if (bpf_ntohs(udphdr->dest) != *port /*set in user-space, read from BPF map*/)
    return false;

packet = (struct packet *)(udphdr + 1);
if (packet + 1 > data_end) // eBPF requires bounds-checking for every access
    return false;

packet->reach_server_timestamp = bpf_ktime_get_ns();
```

A limitation of this approach is the case that the client-side program has a UDP socket active with the same port number as the server benchmark. This will cause the packet to ping-pong between the cpumap programs and the scheduler program, as the eBPF hook is executed for both incoming packets and outgoing packets. This did not cause any issues during my testing.

4.2.3 Artificial Workload

The artificial workload is simulated by looping for a number of iterations that takes roughly as long the desired workload.

```
// bpfnic.bpf.c
int iters = 0;
```

```
if (packet->data == REQ_SHORT)
    iters = SHORT_REQ_ITERS;
else if (packet->data == REQ_LONG)
    iters = LONG_REQ_ITERS;
else
    iters = SHORT_REQ_ITERS;

bpf_loop(iters, _empty_loop_func, NULL, 0);
```

4.2.4 IP Header Re-Computation

When the server program redirects packets back to the client program, it needs to recompute the IP header checksum for it not to be dropped by the client's networking stack on arrival. The computation needs to be done manually because the eBPF hook runs before the kernel's networking stack, and is done using the raw IP header contained in the packet. Normally, the packet's UDP header would also need to be recomputed, but setting it to 0 sufficed for the benchmark and didn't cause any packets to be dropped on arrival at the client.

```
// recomputes an ipv4 checksum.
   https://en.wikipedia.org/wiki/Internet_checksum
static void __always_inline recompute_iphdr_csum(struct iphdr *iphdr)
{
    iphdr->check = 0;

    u16 *short_words = (u16*)iphdr;
    u32 csum = 0;

    #pragma unroll // required by eBPF verifier
    for (int i = 0; i < 10; i++)
        csum += short_words[i];

    csum = (csum & 0xFFFF) + (csum >> 16);
    iphdr->check = (u16)(~csum);
}
```

4.2.5 State in eBPF Programs

Maintaing state between executions of the eBPF programs, and sharing state between the eBPF programs and the user-space program, was done using BPF maps which is a persistent storage mechanism. The following example shows how they were used for maintaing iterators for the two queues in the core-separated round-robin scheduling policy.

```
// 0: short request iterator
// 1: long request iterator
```

```
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __type(key, __u32);
    __type(value, __u32);
    __uint(max_entries, 2);
} cpu_iter_core_separated SEC(".maps");

// 0: number of cpus reserved for short requests
// 1: number of cpus reserved for long requests
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, __u32);
    __type(value, __u32);
    __uint(max_entries, 2);
} cpu_count_core_separated SEC(".maps");
```

4.2.6 Dynamic Core Group Logic

Like with the static round-robin case, the available CPUs and the cpumap array are set before attaching the eBPF program to the network interface. What differs here, and what allows the core-group to grow dynamically, is simply by adjusting the count in the userspace program once a threshold queuing delay is exceeded. This allows the same round-robin program to be used in both the static and the dynamic case with only minor changes to the user-space part of the program.

```
// set in userspace program
if (avgQueuingDelay > THRESHOLD_LATENCY_NANOS && cpusCount <
    availCpus.size()) {
  cpusCount++;
  bpf_map_update_elem(countFd, &key0, &cpusCount, 0);
}
```

```
// accessed in eBPF program
cpu_iterator = bpf_map_lookup_elem(&cpu_iter, &key0);
*cpu_iterator += 1;
    if (*cpu_iterator >= *cpu_count)
        *cpu_iterator = 0;
```

Chapter 5

Evaluation

5.1 Bimodal Traffic

There are two different scheduling policies that we want to evaluate for the bimodal case.

- Round-robin over all cores with no core-separation
- Core-separated round-robin: reserve some subset of CPU cores for long requests and the rest for short requests.

The metric that we are interested in is the 99th percentile queuing delay and in particular how this metric changes when faced with increasing traffic. What we expect to see is that the respective scheduling policies will cope up to a certain point before queuing delay skyrockets. The throughput at which this happens is what we're looking for.

5.1.1 Experiment Configuration

- Server kernel version: Linux `6.5.0-14-generic`
- eBPF redirection program listening on some UDP port number and redirecting requests to remote CPUs based on the chosen redirection policy via eBPF cpumap. Loaded on the system's NIC.
- eBPF cpumap programs that run a $6\mu s$ / $60\mu s$ artificial worload for short / long requests respectively, redirecting packets back to the client after completion via devmap.
- 16 provisioned CPUs in total.

- Client program sends bimodal requests to the server at a rate approximately $225'000Rps \times$ *number of clients*, increasing the number of clients periodically and thus increasing the throughput. The client program maintains a histogram of round-trip times and queuing delays.
- Traffic consists of 90% short requests and 10% long request distributed uniformly.

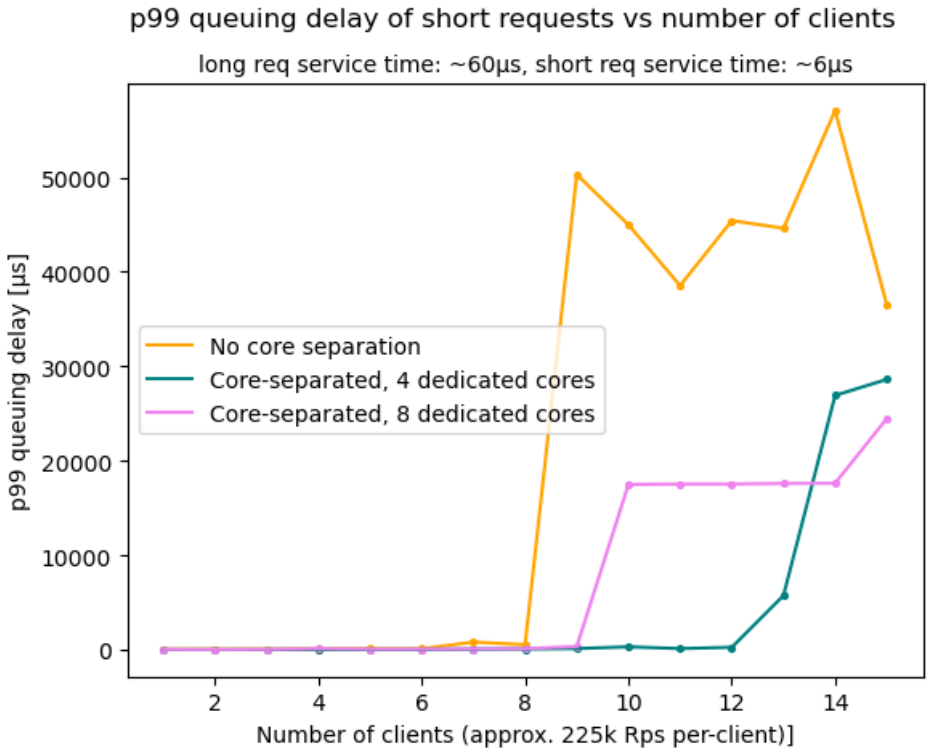


Figure 5.1: 99th percentile queuing delay of short requests over number of clients for the three tested bimodal scheduling policies.

5.1.2 Results and Observations

Firstly, an important consideration to bear in mind is that measurements after the queuing delay spikes are subject to high variance. By this point, the target CPUs are fully saturated, and their queues *(fixed size of 4096, maximum allowed size for a cpumap)* full as they struggle to keep up with load. Most requests will be dropped at this point in time.

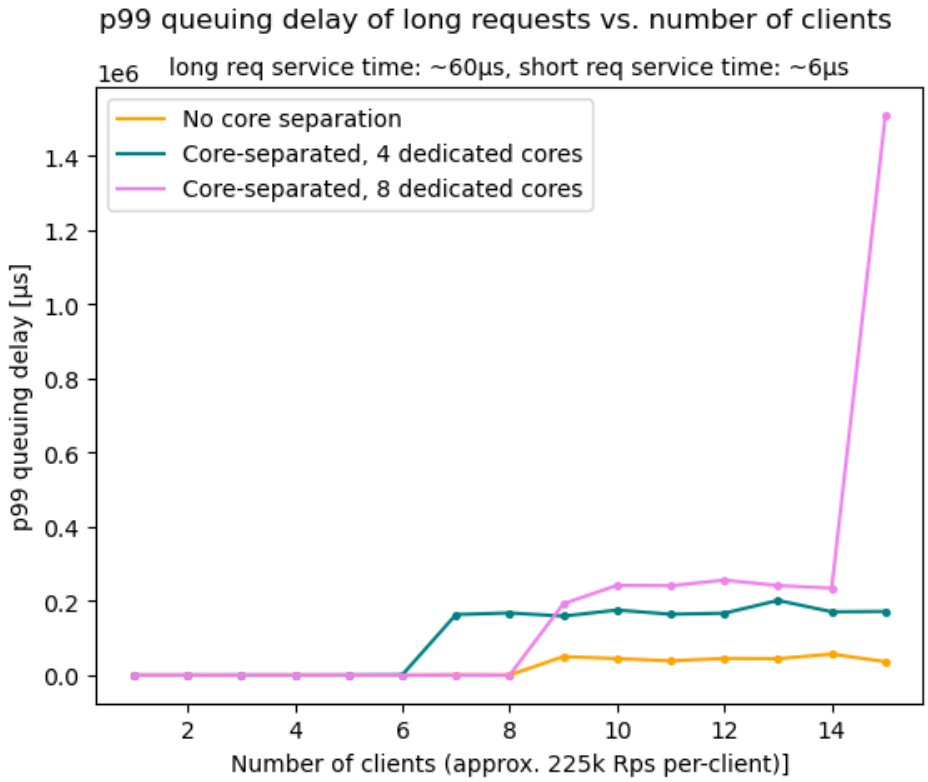


Figure 5.2: 99th percentile queuing delay of long requests over number of clients for the three tested bimodal scheduling policies.

Short Requests

Let's start by analyzing the behavior of 99th percentile queuing delay for the short requests. What we observe is that the core separated redirection policies can sustain a higher maximum throughput before queuing delay explodes. This is in line with expected behavior. Indeed, in the non-core-separated case, a short request can arrive at a given CPU right after a long request causing it to be stuck in line behind it - an archetypal case of head-of-line blocking.

We also observe that between the two redirection policies leveraging segregated core-groups, the one with less cores reserved for long requests *(and thus more for short requests)* has a higher sustained maximum throughput with respect to short requests.

Long Requests

The first observation that we can make is that the core-separated cases sustain a lower maximum throughput than the non-core-separated case, most noticeably the case with four reserved cores for

long requests. As throughput increases, sending all long requests to a small subset of cores results in massive head-of-line blocking. This head-of-line blocking is mitigated in the case that we reserve more cores for long requests.

1. The p99 queuing delay plateaus after the spike. This is because the queues are full, thus stabilizing in a state where any additional requests are dropped causing a static queue size.
2. The p99 queuing delay post-spike is greater in the core-separated (queue consisting of 4096 long requests) case than in the non-core-separated case (queue consisting of 4096 requests that are a mixture of long and short).

Naturally, since long and short requests share queues in the non-core-separated case, the queuing delay for both request types is equal.

Source of packet drops

It was mentioned before that the packet drops and observed queuing delay plateau was related to the target CPUs saturating and their queues reaching full capacity. This can be confirmed by observing the incoming Rps from the client (counted by the scheduler) versus the outgoing Rps (counted by the target CPUs).

What we observe in Figure 5.3 is that the target CPUs are saturating before the scheduler program being run on NIC interrupt, indicating that the cpumap programs become a bottleneck before the scheduler.

Mathematical Analysis

With the insight that the server can be modeled as a $M/C/k/N$ queue with C indicating a constant service time, we can rationalize some of the observations with queuing theory.

We can model the non-core-separated case as a $M/C/k/N$ queue with arrival rate $\lambda =$ `throughput` and departure rate μ such that

$$\mu = (p_L \cdot \texttt{srv\_long} + p_S \cdot \texttt{srv\_short})^{-1} \cdot k$$

$$N = \texttt{num\_cores} \cdot \texttt{queue\_size}$$

$$k = \texttt{num\_cores}$$

With `srv` indicating service time and p_L / p_S representing probability of a request having a long

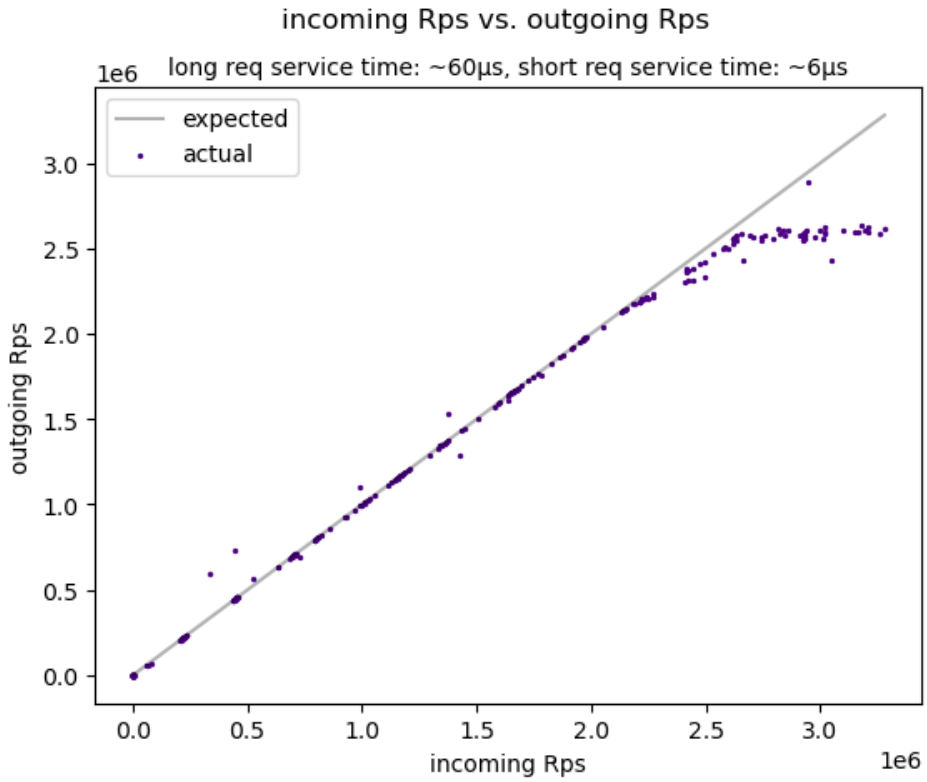


Figure 5.3: Incoming Rps versus outgoing Rps for a bimodal traffic distribution. Core separated round-robin policy with 16 total cores, 8 reserved for long requests.

/ short service time, contraining that $p_L + p_S = 1$ since the traffic distribution is bimodal. We also constrain that `srv_long` > `srv_short`.

The core-separated case can be thought of as two separate queues; one for the long requests and one for the short requests

$$M/C_L/k_L/N_L \ and \ M/C_S/k_S/N_S$$

With

$$\lambda_L = \lambda \cdot p_L, \ \lambda_S = \lambda \cdot p_S$$

$$\mu_L = \texttt{srv\_long}^{-1} \cdot k_S, \ \mu_S = \texttt{srv\_short}^{-1} \cdot k_L$$

$$k_L = \texttt{long\_reserved\_cpus}, \ k_S = \texttt{short\_reserved\_cpus}$$

$$N_L = k_L \cdot \texttt{queue\_size}, \ N_S = k_S \cdot \texttt{queue\_size},$$

In a birth-death process, we reach an unstable state (unbounded growth) as soon as $\lambda > \mu$ i.e. when the arrival rate exceeds the departure rate. Concretely, since our queues have a fixed size, the point where $\lambda > \mu$ is the point where the queues fill up and that packets are dropped. This is the plateau that we see after the spike in all of the plots. $\lambda_{max} = \mu$ is the theoretical maximum sustainable throughput. This approximation omits any other external factors such as CPU utilization by other processes, or scheduling-related overhead.

Setting $\mu = \lambda$ and $\mu_S = \lambda_S = p_S \cdot \lambda$ we can solve for k_S to find the point at which the maximum sustainable throughput of short requests is the same in both the core-separated and non-core-separated cases.

$$p_S^{-1}\mu_S = \mu \Leftrightarrow p_S^{-1} \cdot \texttt{srv\_short}^{-1} \cdot k_S = (p_L \cdot \texttt{srv\_long} + p_S \cdot \texttt{srv\_short})^{-1} \cdot k$$
$$\ldots \Leftrightarrow k_S = \frac{p_S \cdot \texttt{srv\_short}}{p_L \cdot \texttt{srv\_long} + p_S \cdot \texttt{srv\_short}} \cdot k$$

Evaluating with the tested values $p_S = 9 \cdot p_L$, $\texttt{srv\_long} = 10 \cdot \texttt{srv\_short}$, we can compute

$$k_S \simeq 0.47 \cdot k$$

Which tells us concretely that the queuing delay of short requests is stable for longer in the core-separated case than in the non-core-separated case as long as *at least* around 47% of the cores are reserved for short requests. We see this to hold empirically - the case where we reserve 8 cores for short requests only slightly outperforms the non-core-separated case in terms of maximum sustained throughput, with the case reserving 12 cores outperforming the both of them.

5.1.3 Concluding Notes

What we can conclude is that there is a very concrete trade-off between sustaining a high maximum throughput of long requests or a high maximum throughput of short requests. When we segregate core groups for long and short requests, we can sustain a much higher throughput of short requests before experiencing head-of-line blocking at the cost of greatly amplifying head-of-line blocking for long requests. When we have one core group for all requests, the resulting queuing delay roughly becomes a weighted average of the core-separated queuing delays of the long-reserved and short-reserved core groups, increasing the maximum sustained throughput of the long requests and reducing that of the short requests.

5.2 Unimodal Traffic

The goal of this experiment is to see how a dynamically-sized core-group reacts with respect to increasing traffic from the client. The metric of interest is queuing delay as it was in the bimodal case, with the addition of 50th percentile queuing delay which was ignored in 5.1.

5.2.1 Experiment Configuration

- Server kernel version: Linux `6.5.0-14-generic`
- Client sends packets with a fixed 3μs service time, starting at 25'000 requests-per-second (Rps) and increasing throughput periodically at a stride of 25'000 Rps.
- Server starts with a core group consisting of a single core and expands whenever average queuing delay across cores exceeds a threshold of 20μs. There is a cooldown period between adding cores so that a spike in average queuing delay does not cause cores to be added successively within too short a timeframe.
- Round-robin redirection policy to available cores.

5.2.2 Results and Observations

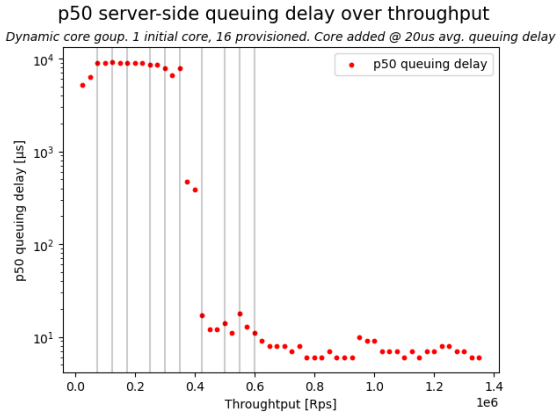


Figure 5.4: 50th percentile queuing delay over throughput in Rps. Core additions marked with vertical lines

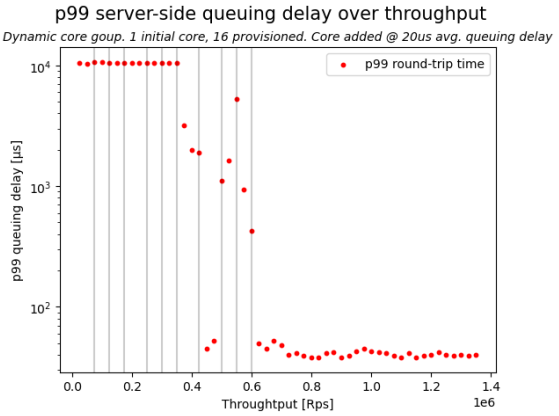


Figure 5.5: 99th percentile queuing delay over throughput in Rps. Core additions marked with vertical lines

What we observe very clearly is an initial struggle to keep up with client demands, even at a low throughput, and an eventual and very sudden stabilization after enough cores have been added. From this point on, even after more than doubling the client throughput, we do not stray from this stable state and never add more cores to the core group.

The stabilzation of queuing delay and apparent uncorrelation between it and throughput seems to imply the existence of some warmup period for the cpumap kthreads, whose initial CPU utilization remains very low (< 10%) despite the high queuing delay observed. This is very likely scheduling related, which we can confirm by keeping the cpumap kthreads busier by sending high-throughput bimodal traffic (with long and short requests as in 5.1). What we observe in this case near instant stabilization, and high CPU utlization across all available cores.

Observation: keep the workers busy
If the cpumap kthreads are not kept busy enough, they run the risk of being put to sleep. Waking them up again causes huge overhead in terms of queuing delay. This is a caveat of the Linux kernel's scheduler which was designed to operate at millisecond timescale, not microsecond as we target here. The kthreads do however perform well under very high pressure from the client. This is analogous to redirecting to worker threads in a traditional server setup.

5.2.3 Performance Mode versus Power-saver Mode

An observation that was made through trial, error, and a little bit of frustration, was the impact of power-saver mode on the benchmark.

Initially, unbeknownst to me, the server was configured in power-saver mode which underclocks all CPUs. Configuring the CPUs to run in performance mode yielded huge decreases in both 50th and 99th percentile queuing delays.

Observation: CPU clock speed has a noticeable impact on queuing delay
It may seem almost trivial, but this is indeed a trade-off that must be considered.

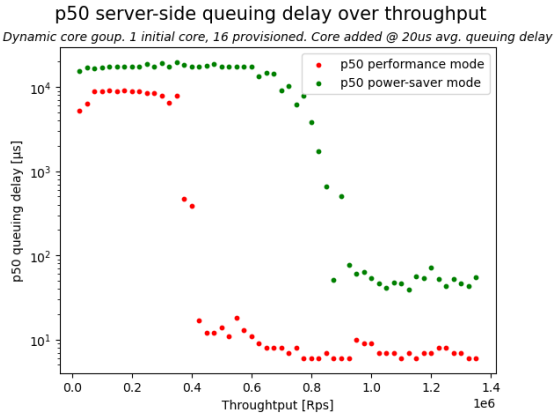


Figure 5.6: 50th percentile queuing delays of performance and power-saver modes over throughput in Rps. log-scaled y-axis

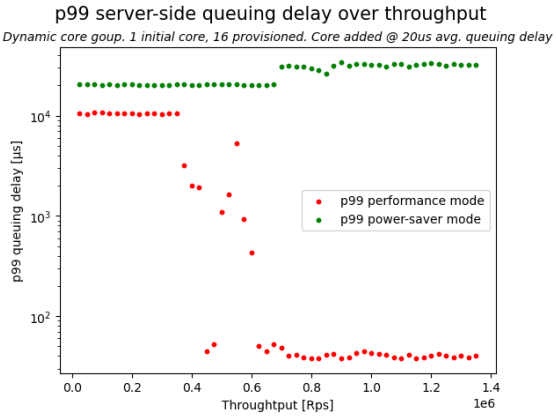


Figure 5.7: 99th percentile queuing delays of performance and power-saver modes over throughput in Rps. log-scaled y-axis

We observe that in both the performance configuration and power-saver configuration, the 50th percentile queuing delay stabilizes. However, the 99th percentile queuing delay does not stabilize when the cores are underclocked. This is an interesting phenomenon, with the 50th percentile queuing delay dropping and the 99th percentile queuing delay increasing.

5.2.4 Dynamic Round-Robin versus Static Round-Robin

What we expect to see is that the maximum sustained throughput of the dynamic program is equivalent to that of the static round-robin program. This would imply that there is little down-side to using the dynamic round-robin program.

The client-side of this experiment was configured identically to 5.1, with the aim of testing how the round-robin redirection policy with a dynamically sized core-group compares to the round-robin policy with a fixed core-group. What we observe is near-identical maximum sustained throughputs.

Observation: starting with a small core-group and growing it when queuing delay starts rising performs as well as starting with a full-sized core group
The implication of this is that starting with a small core-group and adding cores when client demand increases is a viable and more efficient alternative to over-provisioning cores.

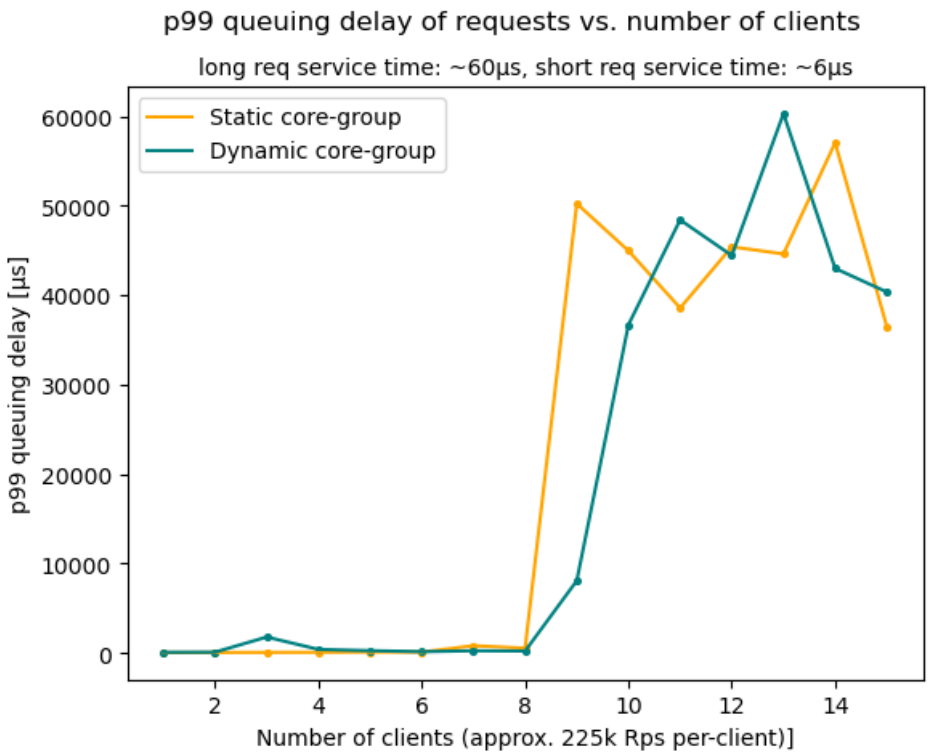


Figure 5.8: 99th percentile queuing delay of static and dynamic round-robin redirection policies given a bimodal service time distribution vs number of clients.

Chapter 6

Related Work

6.1 Dataplane Operating Systems

There has been notable work done in the field of dataplane operating systems, which are OSs designed for handling a high throughput of requests at μ second tail latencies.

IX [1] uses hardware virtualization to offload networking from the kernel, and opts for a distributed FCFS model and network I/O polling instead of interrupts. It employs NICs with RSS using flow consistent hashing for distributing packets amongst hardware queues. It achieves huge performance gains over Linux equipped with specialized user-space networking stacks.

The major limitation of IX is its flexibility. For one, given that it re-implements certain kernel functionality, it requires that user applications be implemented using the `libix` API for compatibility.

ZygOS [3], with the additional insight that single-queue FCFS models have a higher theoretical upper bound in terms of sustainable throughput than multi-queue models, achieves better performance than IX for requests with longer service times, but falls short for requests with shorter service times, such as `memcached` workloads.

Shinjuku [2] differs from IX and ZygOS by implementing a preemptive approach that is practical at the μ second scale, deemphasizing RSS in favor of a centralized and scheduled approach. It can thus avoid the common pitfall that is head-of-line blocking by preempting worker threads that spend too long processing a given request (placing the packet at the tail of the queue), and in doing so, it outperforms both IX and ZygOS. Like IX, it requires a certain level of care from the user when it comes to developping for Shinjuku, for example the use of a `call_safe(fn)` API that disables interrupts which would cause interference.

These implementations all achieve performance increases over the Linux kernel thanks to their specialized nature. An eBPF approximation of the stratgies that they employ will not likely out-perform them. In my implementations, a noteworthy bottleneck that is not considered is the preemptive overhead related to the handling of NIC interrupts, which despite being more power efficient, will always underperform network I/O polling.

Where the aforementioned dataplane operating systems excel in terms of raw performance, they lack in terms of flexibility and ease of deployment.

6.2 eBPF

HEELS [4] extends off-the-shelf load balancers with eBPF programs, reducing costs without any kernel modification. At its essence, HEELS is tackles the same problem as I did in this project - balancing and maximizing resource utilization. It does so at the granularity of TCP connections, however, which is coarser than the RSS that was implemented here.

Chapter 7

Conclusion

In this project, I set out to implement and benchmark eBPF receive-side scaling policies capable of handling high throughput at μs scale latencies. My findings show that a RSS implemented in eBPF can comfortably cope with high demand in a client-server environment, and in fact that it can sustain a greater throughput than the underlying application logic being redirected to (simulated in my case by an artificial workload of fixed length). Future work on eBPF receive-side scaling is promising thanks to how easy it is to hot-swap redirection policies in without any kernel modifications, or the need for a specialized dataplane operating system.

My findings also showcase a limitation in handling low-latency workloads on a general-purpose operating system like Linux, which is likely related to scheduling that is optimized for millisecond latencies as opposed to microseconds.

The redirection policies that I implemented in the context of this project are simple. Possible future work could focus on replicating policies implemented in dataplane operating systems such as IX, [1], ZygOS [3], and Shinjuku [2], as a means to more accurately compare performance for real-world workloads.

In conclusion, there is compelling evidence suggesting that eBPF is a viable solution for implementing performant and flexible receive-side scaling for high-throughput and low-latency workloads.

Bibliography

[1] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. "IX: A Protected Dataplane Operating System for High Throughput and Low Latency". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. October 6–8, 2014, Broomfield, CO.* 2014.

[2] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. "Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency". In: *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19). February 26–28, 2019, Boston, MA, USA.* 2019.

[3] George Prekas, Marios Kogias, and Edouard Bugnion. "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks". In: *Proceedings of SOSP '17. ACM, New York, NY, USA.* 2017.

[4] Rui Yang and Marios Kogias. "HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme". In: *eBPF '23, September 10, 2023, NY, USA.* 2023.