Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Accelerating data centre communication patterns using eBPF

Author:
Alex Constantin-Gómez

Supervisor:
Marios Kogias

Second Marker:
Peter Pietzuch

June 16, 2023

Abstract

We explore the use of eBPF (extended Berkeley Packet Filter) as a solution to improve the communication efficiency in highly-distributed data center environments, focusing on minimising the overhead caused by user-kernel context switches and excessive traversals of the kernel networking stack. By leveraging eBPF, which allows user-defined code execution inside the Linux kernel, it becomes possible to move logic from the application in user-space into the kernel and achieve significant performance benefits.

The work presented focuses on accelerating scatter-gather workloads, which involve extensive communication between a coordinator machine and multiple worker nodes. We first explore the feasibility of using eBPF to accelerate this communication pattern, and then propose an eBPF-enabled scatter-gather network primitive called *sgbpf* available as a library for network applications.

Our experiments show that *sgbpf* outperforms the standard Linux-native I/O APIs (including state-of-the-art interfaces such as `epoll` and `io_uring`) by at least 42% both in terms of latency and throughput for fan-outs of all sizes, varying from as low as 10 nodes all the way up to over 1000 workers.

Acknowledgements

I am deeply grateful to my supervisor, Marios Kogias, for their invaluable guidance and support throughout this project. Having undertaken this project and learnt about eBPF, I will be a keen follower of this technology and will be very interested to see how it continues to grow and applied to other projects in the future.

I would also like to express my appreciation to my friends and family, especially my mum and dad, whose support and belief in me have been a driving force and a constant source of motivation.

Contents

Chapter 1

Introduction

1.1 Motivation

With the huge increase in demand for data storage and processing in the last decade, data centres have become reliant on handling client requests and performing computation in a highly-distributed manner, using hundreds or thousands of machines connected over a network. As such, the communication between these machines is critical to the successful operation of data centres.

The applications running in such data centres typically follow complex scatter-gather patterns, where a single coordinator machine needs to communicate with hundreds of worker nodes, collect responses from them, and produce an answer for the client. However, during these exchanges, there may be a significant amount of user-kernel crossings and unnecessary traversals of the kernel networking stack, which can reduce performance as they induce direct and indirect costs. In recent years, high-performance network cards have made the user-kernel transition overhead the bottleneck for I/O-heavy applications. In addition, most modern general-purpose kernel networking stacks are designed for a small number of persistent connections with a high establishment overhead; on the other hand, scatter-gather workloads in data centre applications typically make use of a large number of short-lived connections (or even connectionless data exchanges using protocols like UDP). Hence, the kernel networking stack incurs a large per-packet overhead and can limit performance for large fan-outs. As a consequence, there is a need for more efficient communication techniques.

One technology that has recently emerged as a potential solution to this problem is eBPF (extended Berkeley Packet Filter), which enables the execution of user-defined code inside the Linux kernel without modifying the kernel nor compromising on safety and performance. Users can attach programs to different points inside the kernel which are executed in an event-driven manner. eBPF has recently become an alternative approach to implementing high-throughput and low-latency networking applications, potentially allowing packets to bypass the kernel and thus leading to significant performance benefits.

Although user-space networking is a well-known kernel-bypass technique for implementing high-performance network applications, these solutions are often too radical for simple scenarios, as they typically require developers to build their own networking stack in user-space and sacrifice CPU resources by busing polling the network card. This can lead to a large amount of wasted energy during periods of low I/O loads, which is undesirable in data centres. This project will focus solely on using eBPF to accelerate such applications, therefore retaining the benefits of the Linux kernel networking stack: security, isolation, reliability and load-aware CPU scaling without the need for busy polling.

1.2 Objectives and challenges

The aim of this project is to explore the feasibility of using eBPF to accelerate certain network communication patterns which typically incur a large number of user-kernel context switches. Specifically, this project focuses on accelerating scatter-gather workloads. The primary goal is to

minimise the overhead resulting from communication across the user-kernel boundary and traversing the in-kernel network stack, which contributes towards the total network bottleneck. As such, this involves identifying and minimising context switches, data copies and system calls (which are generally all coupled together). This will require a solid understanding of the existing I/O APIs offered by the Linux kernel and evaluating the performance of these standard APIs in comparison with an eBPF-enabled alternative.

As part of this work, one specific objective of this project will be to investigate the possibility of moving entirely or partially the aggregation logic into the kernel using eBPF. This logic typically takes place in the user-space application when gathering the incoming responses from the worker nodes to produce a final result which is returned to the client. This implies that the individual responses from the workers are copied from the kernel into the application, contributing to the previously mentioned user-kernel communication overhead. If this logic can be moved into the kernel using eBPF, then this overhead can be minimised to single user-kernel crossing which returns the final aggregated result of the scatter-gather operation.
The main challenge here is to understand the limitations of eBPF and what kind of operations are permitted inside the kernel. This is because all eBPF programs are statically verified before being loaded into the kernel to guarantee safe execution (as described in section 2.2.3).

As a result of the aforementioned investigations, the overall objective of this project will be to implement an eBPF-enabled scatter-gather network primitive and evaluate its performance against standard implementations of a scatter-gather operation. Ideally, this novel implementation should achieve higher throughput and lower latency compared with the standard baseline implementations. Because these scatter-gather workloads do not typically require a connection-oriented communication, we will be focusing on an implementing a solution for UDP-based applications. If time allows, we will also explore the feasibility of implementing a similar system for TCP-based communications.
The challenge here is to apply all the knowledge and relevant techniques discovered during the exploration phase of the project to design, build and benchmark a fully-functioning and optimised scatter-gather primitive, which should serve as a foundation for accelerating other high-level network communication patterns using eBPF.

1.3 Contributions

In this project we propose ***sgbpf***, an eBPF-accelerated implementation of a scatter-gather network primitive for UDP-based communication. It outperforms the alternative standard Linux I/O APIs both in terms of latency and throughput under load for fan-outs of all sizes, according to our benchmarks described in chapter 5. It is made available to application developers through a C++ library which can easily be integrated into network applications. It makes no underlying assumptions of the application threading model nor event loop structure, and relies only on Linux-native mechanisms readily available in modern kernels.

The library also **supports in-kernel packet aggregation**, whereby the application developer can express custom aggregation logic (such as numeric reduction operations) in an eBPF program which is loaded into the kernel at runtime and executes whenever a response is received from the workers. This feature means that individual response packets can be dropped early and the final aggregated result is accessible to the user-space application with a single system call, avoiding a network stack traversal per packet, as explained in section 4.4.2.

While this project focuses on accelerating scatter-gather communication, the work presented can be used as a technical blueprint for accelerating different communication patterns with eBPF, as it **introduces generic design patterns** which can be reused in other settings to improve performance using eBPF, such as the one described in section 4.3.1.

Finally, this dissertation describes a high-level design of a TCP-based scatter-gather system with eBPF. The proposed design described section 6.2.2 takes advantage of the existing functions in the TCP/IP networking stack but minimises user-kernel crossings by running the eBPF programs in the socket layer.

Chapter 2

Background

2.1 Data centre communication patterns

Data centres have become the backbone of many modern computing systems, providing the necessary infrastructure to support the growing demands of cloud computing, big data, web services and data storage. These facilities house large numbers of servers and storage devices, which are connected to a network that allows for the efficient communication and transfer of data between them.

In data centres, the servers connected to each other are often used to build large scalable distributed systems. In these, communication patterns between the various nodes play a crucial role in determining the overall performance of the system. For example, in a data centre, network communication patterns between servers can greatly impact the performance of applications, such as distributed databases or distributed file systems. The ability to effectively manage and optimise these communication patterns is therefore crucial for ensuring the efficient operation of distributed systems in data centres. The performance of these distributed applications directly impact the ability of the system to handle large amounts of data and requests in a timely manner. Poor performance can lead to delays, bottlenecks, and increased costs, all of which can negatively impact the overall effectiveness of the system. As such, it is essential to understand and optimise network communication patterns in order to ensure the optimal performance of data centres and distributed systems.

Depending on the application, the communication pattern and network workload will vary on a per-case basis. In many cases, communication is based on the request-response model, where multiple machines exchange messages back and forth. In some cases, it is sufficient to send a message without requiring a response. This section of the report will introduce the two most common high-level patterns found in large-scale distributed systems: the **scatter-gather gather pattern** (based on the request-response model) and the **publisher-subscriber pattern** (fan-out communication without responses).

2.1.1 Scatter-gather pattern

The *scatter-gather* or *fan-out/fan-in* pattern refers to the scenario where computation is distributed and executed concurrently among multiple workers, each of which produces a result and sends it back to the coordinator node, which aggregates them into a final output.
Although this pattern can apply to a multi-threaded application running on a single machine, it is more commonly found in distributed computing applications where the worker nodes are connected over a network and communicate through messages, rather than shared memory. Therefore, this is a recurring communication pattern found in data centres, where hundreds or thousands of worker nodes are involved. Some examples of distributed applications that utilise this pattern are:

- **Web search engines**: these often involve a frontend server which handles user's requests, and contacts thousands of backend servers to query a distributed search index. These results are aggregated and presented to the user.

- **Data analytics frameworks**: these typically involve thousands of parallel workers to compute a response to a OLAP-style query. Examples are Spark and BigQuery [1].
- **Distributed graph processing**: each vertex in a large-scale distributed graph invokes a scatter-gather operation on its neighbours iteratively to compute data. Examples of such frameworks are Pregel [2] and X-Stream [3].
- **Consensus protocols**: one way of achieving fault-tolerance in distributed systems is through consensus protocols such as Raft or Paxos. These algorithms make heavy use of broadcasting primitives, which can be seen as an instantiation of the scatter-gather pattern.
- **Network file systems**: cluster-based storage systems such as a NFS software stripes data in blocks over multiple file servers and should support parallel data transfers from multiple nodes to retrieve this data. This design resembles a scatter-gather workload.
- **Distributed machine learning**: the training process in ML frameworks will scatter batches of inputs and model parameters to worker nodes, perform local training, and then gather the updated gradients on the coordinator node. Pytorch is one of many ML frameworks that adopts this scatter-gather design [4].

The scatter-gather pattern has become ubiquitous in data centres and a key design in large-scale distributed data processing systems. As a consequence, most applications treat scatter-gather as a primitive operation and invoke it as if it were a call to any other service. The two phases are described below.

Scatter phase

The application invokes the primitive by providing a list of the remote worker addresses and ports, and the query itself. The query sent to all of the nodes is identical or similar. It is then sent out to all the workers. Depending on the setup, this can be achieved through a single multicast socket or multiple individual sockets (one socket per worker). While multicast networking is more efficient, it is only a feasible option for connectionless protocols such as UDP (and requires more configuration on the network layer). Thus, using one socket per worker is a more attractive option for developers, as it enables the reliability guarantees of TCP.
At this point, depending on the implementation, the coordinator may block and wait for all the workers to respond, or continue executing and handle the worker's responses as they are received.

Gather phase

Each worker node receives the request and performs some computation on local data. The result of the computation is then sent back to the coordinator. The coordinator must wait until it receives a response from all of the workers. Once it has received all responses, it will typically perform an aggregation operation over the responses to produce a final result, which is then presented to the user.

Depending on the nature of the application, the gather phase may be more sophisticated and closely tied to the application logic. For example, a data analytics framework may execute a long pipeline of transformations (filtering, mapping, reducing) before returning a result, whereas a consensus protocol may simply count the votes received from other nodes during leadership election, through a simple sum reduction. It can even be as simple as a heart beat protocol, where no reduction takes place and the application simply acknowledges that a worker node is still alive.

The application may also specify the condition on which the scatter-gather operation is deemed to have completed. In most cases, the operation completes only when all workers have sent a response. For some scenarios however, it may be enough to complete as soon as the first response is received, or the first response equal to a certain value (for example, a distributed search).

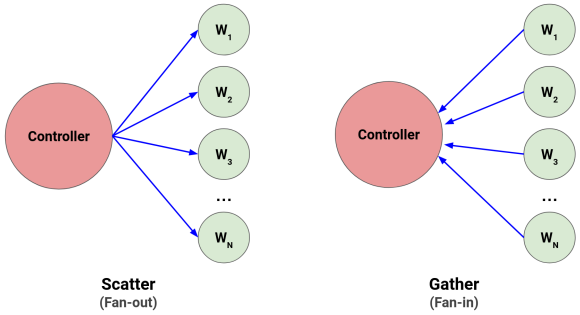

Figure 2.1: A visualisation of both phases of a scatter-gather workload

Inefficiencies with scatter-gather workloads

More often than not, scatter-gather workloads involve a large number of workers and each worker's task is small. This leads to the network communication phase becoming the bottleneck, rather than the actual compute, due to the high overheads incurred by a large number of short lived connections [5]. This bottleneck can be especially problematic for iterative computations, such as training in machine learning, or online services with human users waiting for results, such as web search.

The scatter operation scales linearly with the number of workers, as the application (and operating system) needs to establish N different connections. Then, for each socket, the application must write the query into the socket. Although there are modern techniques to improve I/O performance (such as `io_uring` [6]), they have complex APIs and the network stack cannot be avoided. An alternative implementation is to use IP multicast, however this normally requires support from the underlying network switches [7] or considerable modifications of the Linux networking stack [8].

The gather operation is another source of problems, as it suffers from the *incast phenomenon* [9], where a large number of workers sending data to the coordinator will congest the network, resulting in packet loss and retransmission (in the case of TCP), which can cause a slow down by orders of magnitude compared to the line's bit rate. As the number of workers grows, the CPU utilisation will also increase substantially due to the large amount of per-packet processing overheads incurred by kernel network stack [5]. High resource utilisation is known to lead to superlinear queuing effects, increasing the tail latency to a point where the system may become unresponsive [10]. This greatly affects high-traffic user-facing applications in data centres.
In addition, the application requires the data to perform the aggregation logic, which may involve hundreds or thousands of kernel-to-user copies of the packet data. This leads to the question of whether the gather logic can be moved into the kernel and thus bypass the networking stack.

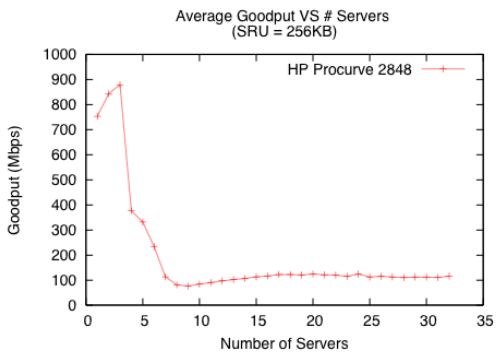

Figure 2.2: TCP throughput collapses (TCP incast) for a large number of servers responding to a client requesting a file from a storage cluster [11]

2.1.2 Publisher-subscriber (fan-out) pattern

The *publisher-subscriber* pattern (commonly abbreviated to *pub-sub*) refers to the scenario where a single machine, referred to as the publisher, sends out a message to one or more other machines, referred to as the subscribers. These machines receive and process the message accordingly. While this resembles the scatter phase in the scatter-gather pattern, there is no response sent back to the publisher. As such, this pattern is also known as *fan-out*, because the flow of messages is unidirectional towards multiple other machines.
The *pub-sub* pattern is heavily used in event-driven systems, where the publisher generates events and subscribers receive events that they are interested in. This pattern is commonly implemented as a message queue and enables decoupling the publisher from the subscribers. This allows for flexibility and scalability, as subscribers can be added or removed without affecting the publisher, making it a popular design technique for large-scale distributed systems.

The publisher-subscriber pattern can be applicable in many real-world situations where scalability is essential. Some examples are:

- **Notification systems**: push notifications are based on the publisher-subscriber design, where notification servers publish messages to devices and/or web endpoints [12].
- **M2M communication in IoT**: smart devices utilise pub-sub patterns for machine to machine communication, avoiding the overhead induced by a request-response system. This is useful whenever readings gathered from sensors need to be distributed in real-time to other IoT devices [13].
- **Distributed cache refreshing**: an application using a distributed cache can publish invalidation messages to update objects that have changed.
- **Real-time event distribution**: applications that rely on real-time data feeds are typically subscribed to certain events and process these events as they are received. An example is an algorithmic trading model that is subscribed to market data received from a certain stock exchange.

While the publisher-subscriber pattern can be seen as an example of a fan-out communication, they can be seen in a different way. Most publisher-subscriber architectures use a message broker to handle the forwarding of events from the publisher to the subscriber is common. Systems such as Apache Kafka or RabbitMQ use brokers to distribute messages and provide a decoupled design. However, for very simple use cases, it may be sufficient for the publisher to send the message directly to all of the subscribers.

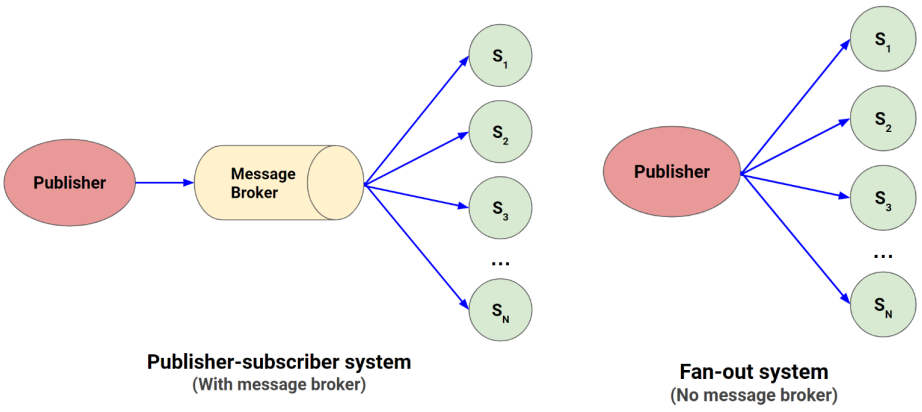

Figure 2.3: On the left, a message broker is used to decouple the publisher from the subscribers. On the right, the fan-out communication resembles the scatter phase.

Because there is no gather phase, the main inefficiency in this pattern is around the connection establishment and socket management (especially in the case of TCP), which is discussed in section 2.1.1.

2.2 The extended Berkeley Packet Filter

The *extended Berkeley Packet Filter* (eBPF) is a technology primarily consisting of an instruction set and an execution environment inside the Linux kernel that allows user applications to safely execute code in kernel space. The eBPF infrastructure enables modification, interaction and kernel programmability at runtime, without compromising safety nor performance. Developers can write eBPF programs which are dynamically loaded into the kernel and are executed whenever a specific kernel event is triggered.

The main benefits of eBPF are safety and performance. Unlike loadable kernel modules in Linux, all eBPF programs are checked prior to being loaded into the kernel so that they meet the required safety and liveness properties established by the eBPF static verifier. This means that eBPF programs cannot corrupt kernel data structures which can cause bugs or crashes. Although programs run in a virtual machine, high performance is easily achieved through just-in-time compilation of the eBPF bytecode into native code (this is now the default behaviour in modern kernels). Thus, well-designed eBPF programs are rarely the bottleneck in kernel code paths, especially since most eBPF programs are small event-driven functions.

Another important benefit is the fact that eBPF is already part of the Linux kernel and has been since 2014 (kernel version 3.15), thus ubiquitously available and can operate in conjunction with all the other mechanisms inside the kernel. Since its introduction, eBPF has been quickly adopted by major companies such as Facebook and Cloudflare, with use cases including network monitoring, load balancing and profiling.

2.2.1 History and motivation

The original Berkeley Packet Filter (BPF) was first introduced by Steven McCanne and Van Jacobson in 1992, and was mainly designed for capturing and filtering network packets with user-defined rules and logic, but within kernel space [14]. The motivation for this was to avoid the need of copying the packets across the kernel/user protection boundary to perform the filtering, as this logic was typically very small (checking packet contents against rules and deciding whether to drop or keep the packet). They achieved this using a register-based virtual machine implemented within the Linux kernel, which was able to interpret a simple RISC-inspired bytecode consisting of only 22 instructions. The authors described BPF as 20 times faster than the state of the art, and BPF was adopted as the technology of choice for network packet filtering.

The ability to safely run user-supplied programs inside of the kernel proved to be a very useful design decision. However, some of the implementation-specific details of the original BPF design became inefficient as hardware advanced. The design of the BPF virtual machine and its RISC-based instruction set were left behind, as modern processors started integrating 64-bit registers and supporting new instructions for multi-processor systems (such as the atomic exchange-and-add instruction) [15] [16].

Alexei Starovoitov introduced the extended BPF design (eBPF) in 2014 [17], which was a redesign of BPF for modern hardware. As part of this change, the virtual machine better resembles contemporary processors, since the eBPF instruction set was redesigned to be mapped more closely to hardware ISA, resulting in better performance. The increase of registers on modern hardware was a big reason for performance improvements, as it made function calls and parameter passing cheaper inside the virtual machine. The ease of mapping eBPF instructions to hardware instructions naturally lends itself to just-in-time compilation, one of the main reasons why eBPF is performant. Another crucial feature in the new design was the introduction of shared in-kernel memory data structures, extending the effective memory that could be accessed by a program.

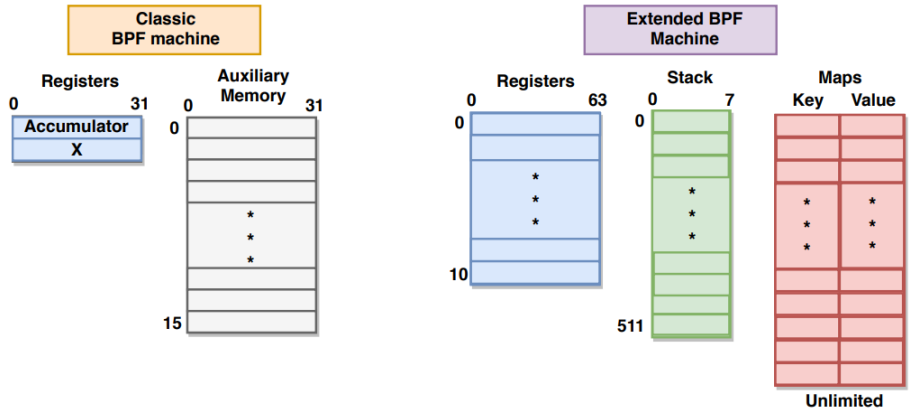

Figure 2.4: A comparison of the classic BPF design and the eBPF design [18]

Eventually, eBPF was exposed to user space with a new system call added in late 2014 [19], and soon after eBPF was no longer limited to the networking stack, covering broader areas of the Linux kernel and becoming more generic.

The following sections will dive deeper into eBPF from a technical perspective. It is also worth highlighting that for the remainder of this report, the terms "BPF" and "eBPF" may be used interchangeably to refer to the extended BPF design introduced by Alexei Starovoitov in 2014 and widely available since Linux version 3.15 (unless explicitly mentioned otherwise).

2.2.2 Architecture

An eBPF program is designed to be executed in an event-driven manner. In other words, these programs are not meant to be long-running, but rather invoked as a function whenever a specific event is triggered in the kernel. An eBPF program is attached to a specific code path in the kernel, known as a hook. Whenever the code path is traversed during the execution of kernel code, any attached eBPF programs are also executed. The hook point is determined by the program type. Since eBPF's introduction, it has grown to support several program types throughout the different layers in the networking stack, as well as other non-networking sections of the kernel, such as whenever a system call is invoked (tracing). This is another reason for eBPF's recent wide-scale adoption, as it makes it suitable for a wide range of different use cases, such as kernel debugging, performance analysis and observability [15].

An example of an event may be a packet being received from the network interface card. The eBPF program receives a parameter with a pointer to the packet buffer, and can inspect the packet contents to decide whether to drop the packet or pass it forward to the networking stack. Another example of a non-networking eBPF program is one that attaches to the *kill* system call. This can be useful for auditing purposes by logging and documenting whenever a process is not gracefully terminated.

Program structure

The first step to writing eBPF programs is to understand their structure. The eBPF runtime within the kernel is only able to load bytecode, however this is too low-level for developers. Instead, developers can write eBPF programs using a restricted subset of the C programming language which can then be compiled into eBPF bytecode using an eBPF-compatible compiler.

The object file emitted by the compiler is a single ELF (Executable and Linkable Format) file consisting of multiple sections. The ELF sections are used to distinguish map definitions and executable code. In eBPF terminology, a *map* refers to a generic data structure that can be used to persist state across separate invocations of eBPF programs. They are also used to share data with user space processes. Maps will be discussed in more detail in section 2.2.4 of this report.

Listing 2.1 shows an example of a simple eBPF program designed for the XDP layer, which runs before the kernel networking stack (more details in section 2.2.5). The source file declares a per-CPU array with a capacity of one unsigned integer (in the `.maps` section), and the actual program that is triggered whenever a packet is received by the NIC (in the `xdp_drop_ipv6` section). The `SEC` macro expands to an `__attribute__` statement which is used by the compiler to place the following declaration in an ELF section with the specified name. The program simply drops all IPv6 packets received and keeps track of the number of these dropped packets by incrementing the single value in the `rxcnt` map for the current core.

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <linux/if_ether.h>
#include <arpa/inet.h>

struct bpf_map_def SEC(".maps") rxcnt = {
    .type = BPF_MAP_TYPE_PERCPU_ARRAY,
    .key_size = sizeof(uint32_t),
    .value_size = sizeof(long),
    .max_entries = 1,
};

SEC("xdp_drop_ipv6")
int xdp_drop_ipv6_prog(struct xdp_md* ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct ethhdr *eth = data;
    __u32 key = 0;
    long *value;

    if (data + sizeof(struct ethhdr) > data_end)
        return XDP_DROP;

    if (eth->h_proto == htons(ETH_P_IPV6)) {
        value = bpf_map_lookup_elem(&rxcnt, &key);
        if (value)
            *value += 1;
        return XDP_DROP;
    }

    return XDP_PASS;
}
char _license[] SEC("license") = "GPL";
```

Listing 2.1: an eBPF program for the XDP networking layer that drops and counts IPv6 packets.

All eBPF programs take a context parameter which will depend on the program type. For networking programs, this will generally be a pointer to the raw packet contents (this is the case for the XDP program in listing 2.1) or some read-only metadata about the packet once it has been processed. For tracing programs, it will depend on the location of the probe but it will typically contain information about the function parameters or registers that the kernel is currently processing [20].
In addition, all eBPF programs must return an integer value. This is needed because in most cases the kernel will use the return value to act upon the return code. For example, in listing 1 the value `XDP_PASS` is used to allow the packet to continue into the kernel, whereas `XDP_DROP` indicates that the packet should be dropped here and no further processing is required.

It is important to clarify what eBPF programs can and cannot do within the kernel. Unlike loadable kernel modules, eBPF programs cannot call arbitrary kernel functions [21]. This design choice is because eBPF programs need to maintain compatibility and avoid being bound to specific kernel versions. Instead, a set of helper functions are offered by the kernel. Examples of such functions include pseudorandom number generation, access to the current time and other network packet manipulation methods. However, this does not mean external libraries cannot be linked; as long as the library functions adhere to the requirements of the static verifier and can be inlined, eBPF programs can invoke external library functions [20].

In general, the high-level flow of an eBPF program and its interaction with a user process fol-

lows these steps:

1. The user application reads the eBPF bytecode into a buffer, attempts to load it into the kernel with the `bpf()` system call and attaches to the appropriate hook.
2. The eBPF program runs on the occurrence of a specific event and accesses a map using the context data associated with the event.
3. The user application uses the `bpf()` system call to query the same map to extract information gathered during the execution of the eBPF program.

Compiling, loading and attaching programs

As briefly described, eBPF programs can be written in a subset of the GNU C language. Therefore, the compiler must support a backend to generate the eBPF bytecode. Clang/LLVM is regarded as the compiler toolchain of reference in the eBPF community due to its maturity. Clang allows the user to pass the value `bpf` for the `-target` flag to set the target architecture to BPF instead of the native architecture. For most eBPF programs, the optimisation level must be set to at least 2, the reason being that functions should be inlined to avoid heavy stacks and some functions may be referenced incorrectly [20]. The generated object file can be inspected using the `llvm-objdump` tool, which allows you to view the different ELF sections and view the source interleaved with the eBPF ISA disassembly.

Typically, eBPF programs are "coupled" with a user space program that loads them into the kernel. This user space program is also commonly referred to as the loader. The source code for the loader program runs in user space and therefore can be built with an appropriate compiler, such as GCC. The loader is typically written in C/C++, as it should link against the BPF headers in the Linux source tree and *libbpf* if used (which is often the case).

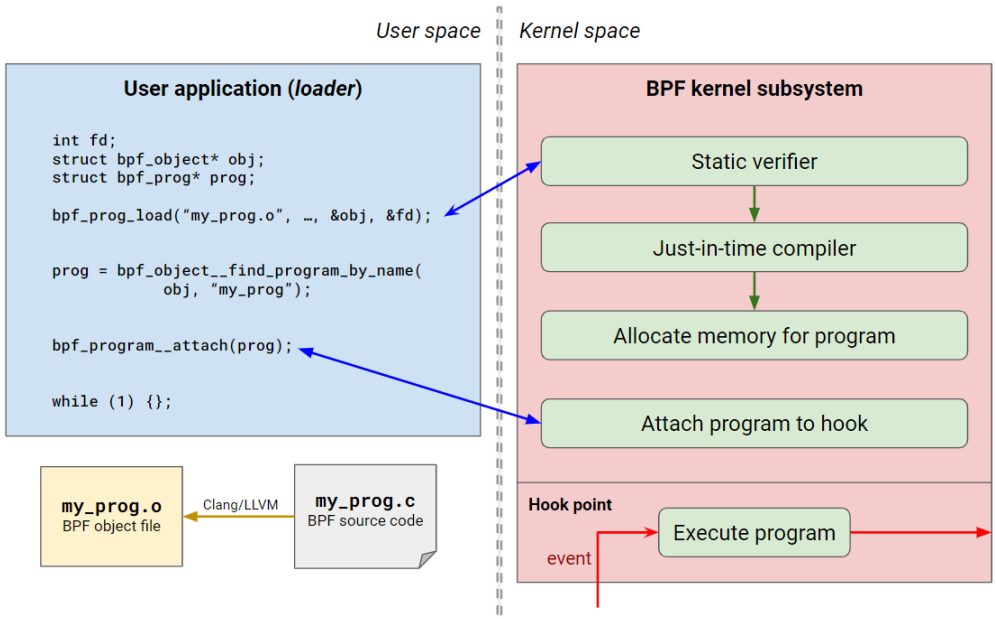

Figure 2.5: The general flow for loading and attaching an eBPF program.

Once the eBPF program is built and linked into a single ELF object file, it can be loaded into the kernel. At a low-level, loading eBPF programs is done through the `bpf()` system call, however developers often prefer to use *libbpf* instead to simplify the application code. When the program is loaded with the system call (passing the path to the object file and the program type), the kernel first parses the ELF file and invokes the static verifier on the program. This runs a number of safety and liveness checks (discussed in more detail in section 2.2.3).
If the program passes the verification stage, it moves to the just-in-time translation phase (if enabled), where eBPF instructions are translated into the native ISA. The Linux kernel has an eBPF JIT compiler for all the major ISAs, such as x86-64, ARM and RISC-V. This stage is essential

to achieve high performance, making the execution speed of eBPF programs as fast as natively compiled kernel code. At the end of this process, the program is allocated into the kernel memory as a `struct bpf_prog_object` which contains program metadata, the eBPF bytecode and the JIT-compiled instructions [15].

At this point, the eBPF program is sitting in the kernel's memory. The kernel keeps a reference counter for each eBPF program. References to eBPF programs (and maps) are held via file descriptors which can be used by the user space program to query the eBPF subsystem at runtime. If no reference remains (for example, the application exits and closes its file descriptors), the kernel unloads the program by freeing the allocated memory.

The last step is to attach the program to the relevant hook. The notion of "attach type" depends on the program type. Some program types such as XDP do not require the program to be attached, but rather require a netlink message to attach the program to the XDP hook of a specific interface [22]. However, some program types do require the user to pass the expected attach type at load time, which is used by the verifier to perform certain checks. In these cases, a further system call is invoked to attach the program to its corresponding hook. The eBPF program is now attached and ready to execute when the relevant event is triggered.

When the user application that loaded the eBPF program closes, the kernel will unload the program because the reference count reaches zero. However, there are some program types (like XDP) where attaching the program will also increment the reference count, meaning that the eBPF program will continue to execute even when the loader exits. Alternatively, if the user wishes to make eBPF objects persistent, they can be pinned by creating a path in the eBPF virtual file system. This technique ensures that the reference count is always above 0. The file descriptor to the pinned object can be later retrieved using the `open()` system call [23].

Program types and hook points

While the program structure, compilation and loading steps are mostly common to all program types, the type of an eBPF program will determine three things: where the program is attached (the hook point), which kernel helper functions can be called and the type of the context parameter passed into the eBPF program. At the time of writing (kernel version 5.11), eBPF supports 30 different program types, some of which are listed in table 2.1 [24, 25].

Program type	**Description**
`BPF_PROG_TYPE_XDP`	Attached to the XDP hook for early access to RX network traffic
`BPF_PROG_TYPE_SCHED_CLS`	Attached to the TC layer to perform traffic classification (RX and TX)
`BPF_PROG_TYPE_SCHED_ACT`	Attached to the TC layer to perform network actions (RX and TX)
`BPF_PROG_TYPE_SOCKET_FILTER`	Attached to socket queues to filter/modify ingress network traffic
`BPF_PROG_TYPE_SOCKET_OPS`	Attached to sockets to catch operations and connectivity options
`BPF_PROG_TYPE_SK_SKB`	Attached to socket buffers to perform redirection between sockets
`BPF_PROG_TYPE_SK_LOOKUP`	Attached to sockets for local delivery of unestablished connections
`BPF_PROG_TYPE_PERF_EVENT`	Triggered whenever a *perf* event is fired, used for instrumentation
`BPF_PROG_TYPE_TRACEPOINT`	Triggered whenever static tracepoints within the kernel (e.g.: syscalls)
`BPF_PROG_TYPE_KPROBE`	Attached to any named function (entry or exit) in the kernel

Table 2.1: A selected list of eBPF program types

2.2.3 The eBPF static verifier

One of eBPF's main benefits is its safety guarantees. Any user-defined code running with kernel privileges comes with inherent security risks, so it is essential for the programs to be verified before being allowed to execute. The safety of an eBPF program is determined by the eBPF static verifier within the Linux kernel. As previously mentioned, whenever the eBPF program is loaded using the `bpf()` system call, the static verifier is invoked on the program's bytecode. If the verification step fails, the program is rejected and is not loaded into the kernel.

The verifier executes in two steps. The first step is to ensure that the eBPF program always terminates, to avoid deadlocks in the kernel. This is checked by constructing a control flow graph of the program and running a depth-first search, ensuring that it is a directed acyclic graph and thus, always terminates [26]. In practice, this means that loops are not allowed, unless they can be unrolled during compilation or are guaranteed to terminate at compile-time. Also, unreachable instructions are prohibited.

The second step involves traversing all possible paths of the CFG, simulating the execution of the program one instruction at a time. During the traversal, the virtual machine state is observed before and after executing each instruction, ensuring that the register and stack contents are valid. This consists of a number of safety checks to avoid invalid memory accesses. Some of these checks include:

- Reading from registers with uninitialised contents is an invalid operation.
- Reading variables from the stack with uninitialised addresses is an invalid operation.
- Writing to the frame pointer register is an invalid operation.

In order to combat the dangers of pointers, the eBPF verifier maintains a register state object for every register. For pointers, eBPF uses a simplified typing system so that all pointer dereferences are checked for type, alignment and bounds violation. This is also used to verify what data can be accessed by the program. For example, some program types are allowed to directly access raw packet data [15, 26]. The verifier also has a *secure mode* that is enabled whenever a non-root user attempts to load an eBPF program. When secure mode is enabled, pointer arithmetic is prohibited. This measure is in place to avoid kernel addresses being leaked to unprivileged users.

Another important step is to ensure the validity of calls to the helper functions available. The kernel exposes a set of helper functions to developers writing eBPF programs. Some of these functions are universal and can be called by any program type, however some functions are intended to be used by specific program types. If a helper function intended for a socket filter program is called from a tracing program, this will fail the verification.

Once the verifier deems the program to be safe, there is a final *hardening* step [21]. This phase involves making the kernel memory holding the eBPF code read-only. By doing so, any malicious manipulation trying to modify the eBPF code will cause the kernel to crash, rather than continuing in a corrupted and/or compromised state. In addition, all constants in the code are blinded, such that their memory addresses are randomised. This makes it harder for attackers to guess their addresses and inject arbitrary code into them.

2.2.4 API: the `bpf()` system call and *libbpf*

As referenced on several occasions, the `bpf()` system call allows the user to interact with the eBPF subsystem inside the Linux kernel. It is included in the `linux/bpf.h` header file and takes three arguments. The first argument supports operations to load eBPF programs, create a user space reference to an eBPF map, iterate over maps and manipulate maps. The second argument allows extra attributes to be set, depending on the operation specified in the first argument. The third argument is the size of the union passed as the second argument.

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

The meaning of the return code depends on the type of operation passed as the first argument. Typically, this will return a file descriptor or a status code indicating the outcome of the operation. The full details can be found on the Linux manual page for BPF [27].

As a single system call with multiple possible parameters, it can be quite cumbersome to use. Therefore, a more user-friendly API called ***libbpf*** is available, which is a user space library distributed with the Linux kernel (also distributed on GitHub [28]). The API, as well as offering direct wrapper functions of the `bpf()` system call, exposes several structures that represent so-called eBPF objects, which refer to programs or maps. Each object type has its own set of methods to manipulate and query data associated with the instance.
Whenever a user space program needs to interact with the ELF object file consisting of multiple sections, *libbpf* exposes the `bpf_object__*()` family of methods for the `struct bpf_object` type. Examples of such methods include loading programs from an object and iterating over all programs in an object file. As for the `struct bpf_program`, the family of methods `bpf_program__*()` allow the user to apply actions to specific programs, such as querying the program type or returning the associated file descriptor. Similar operations exist for the `struct bpf_map` type and can be found in the `libbpf.h` header file.

User-kernel communication with maps

One of eBPF's most powerful features is the ability to store data persistently across multiple invocations of the same (or different) programs in generic data structures. These data structures are known as *maps*. The eBPF infrastructure exposes a variety of different map types representing different abstract data structures which can be useful to the developer for different scenarios and program types. Maps can be accessed by multiple different eBPF programs running in the kernel, as well as user space programs. Thus, they are the main mechanism for communicating data between kernel and user space.

With eBPF's rapid adoption and support for a wide range of program types, different map types are continuously being implemented, whether they are generic data structures or specific to certain program types. The most commonly used map types are shown in table 2.2 [24, 29].

Map type	Description
`BPF_MAP_TYPE_ARRAY`	A contiguous array data structured indexed by an integer
`BPF_MAP_TYPE_PERCPU_ARRAY`	Like `BPF_MAP_TYPE_ARRAY` but creates an instance for each core
`BPF_MAP_TYPE_LRU_HASH`	A hash table that uses the LRU policy to evict elements when full
`BPF_MAP_TYPE_PROG_ARRAY`	An array that stores pointers to eBPF programs (useful for tail calls)
`BPF_MAP_TYPE_ARRAY_OF_MAPS`	An array that stores pointers to eBPF maps (two-level indirection)
`BPF_MAP_TYPE_RINGBUF`	A thread-safe multi-producer single-consumer ring buffer
`BPF_MAP_TYPE_SOCKMAP`	A map that stores socket references, used for socket redirection
`BPF_MAP_TYPE_QUEUE`	A map with semantics of a FIFO queue data structure
`BPF_MAP_TYPE_STACK`	A map with semantics of a LIFO stack data structure
`BPF_MAP_TYPE_LPM_TRIE`	A trie data structure designed for efficient longest-prefix matching

Table 2.2: A selected list of eBPF map types

Despite supporting a diverse range of data structures, all maps are defined by the map type, the maximum capacity, the size of the key in bytes and the size of value in bytes. Listing 2.2 shows a couple of examples of map declarations.

```
// An example of using an array of size 1 as a single shared variable
struct bpf_map_def SEC(".maps") counter = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(uint32_t),
    .value_size = sizeof(long),
    .max_entries = 1,
};

// A sockmap used to store a single socket reference for connection multiplexing
struct bpf_map_def SEC(".maps") sockets = {
    .type = BPF_MAP_TYPE_SOCKMAP,
    .key_size = sizeof(uint32_t),
    .value_size = sizeof(uint64_t),
    .max_entries = 1,
};
```

Listing 2.2: examples of map declarations in a BPF source file.

Just like eBPF programs, the lifetime of maps is determined by a reference counter maintained within the kernel. When the user space process creates a map, the reference count is incremented. When an eBPF program is loaded and uses the same map, the count is also incremented. The counter is decremented when the eBPF program is unloaded and when the user space program exits. When the counter reaches zero, the map's memory is released.

User space processes can lookup and update elements of a map using the file descriptor associated with the map, which was returned from the `bpf(BPF_MAP_CREATE)` system call. On creation, the user must also supply the key and value sizes which will be used by the verifier during program loading to check that the types are compatible. The lookup operation takes the file descriptor of the map, a read-only pointer to the key and a writable pointer to set the value. The update operation takes the file descriptor, read-only pointers to the key and value, and a numeric parameter to specify extra flags (such as creating new elements only if it did not exist).
The API for eBPF programs running in the kernel is very similar: the only difference is that the first parameter is a pointer to the statically-defined map in the object file, rather than a file descriptor.

One concern that may be raised is concurrent access to a shared map. If multiple eBPF programs running on different cores attempt to modify the same element of a map, it may lead to race conditions. This issue is dealt with using spinlocks, which can be placed inside structures that are used as values in the eBPF maps [30].

2.2.5 Network processing with eBPF

As the BPF project was originally designed for fast network processing, a lot of effort has been invested into supporting more hook points in the network processing stack. This section aims to cover the three most important network-related hook points in the kernel in more detail.

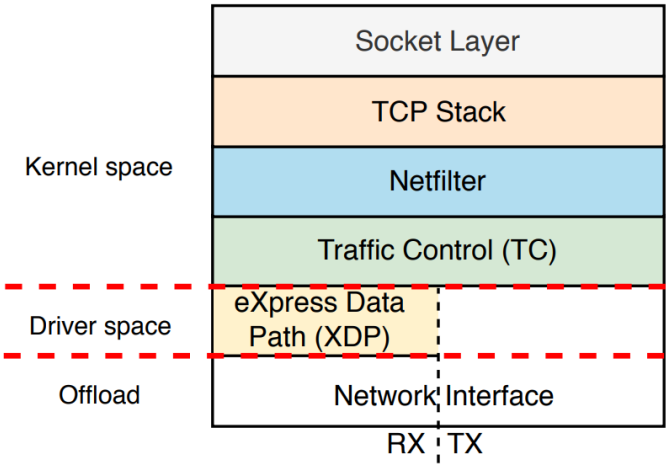

Figure 2.6: The main network-related hooks in the Linux network stack [18]

The XDP layer

The *eXpress Data Path* is only located on the ingress (RX) path and is the earliest hook to which an eBPF program can be attached. This is located before the TCP/IP networking stack. Hence, this hook is suitable for high-performance applications that need to make quick decisions about incoming packets, whether that is dropping them completely or performing modifications on the raw packet data without the overhead of the networking stack.
The exact location of the hook depends on the mode of operation. XDP offers three different operational modes:

- **Native mode**: the program is attached to the NIC's device driver and is executed during the soft IRQ raised on an interrupt. This before the kernel allocates memory for the packet, so this overhead is avoided. This requires explicit support by the network driver.
- **Generic mode**: for non-compatible network drivers, the kernel allows XDP programs to be attached within the kernel, emulating native execution. However, this comes at the cost of reduced performance due to the overhead of socket buffer allocation [31].
- **Offload mode**: the program is entirely offloaded into the NIC and runs on the hardware. This mode offers the best performance out of the three, but requires a compatible programmable smart NIC.

XDP programs receive a single context parameter pointing to a `struct xdp_md`, containing pointers to the beginning and end of the packet, as well as other metadata (such as the index of the RX queue that received the packet). At the end of the XDP program, an action (return code) is required for the packet. There are five possible actions:

- `XDP_ABORTED`: drop the packet and raise an exception.
- `XDP_DROP`: drop the packet.
- `XDP_PASS`: allow the packet to be passed into the rest of the network stack.
- `XDP_TX`: retransmit the packet through the same interface it arrived from.
- `XDP_REDIRECT`: redirect the packet to another interface or CPU for further processing, or to a special `AF_XDP` socket (bypassing the network stack).

As a side note, the *eXpress Data Path* can also be used as a form of user-space packet processing and bypassing the kernel's network stack altogether. This is achieved through the `AF_XDP` socket family and an eBPF program attached on the XDP hook which redirects frames directly to a memory buffer shared between the application and the network device driver. The application can fill this buffer with packets to be transmitted, or it can read packets received from the network into the buffer. The network device driver can then directly access the packets in the buffer, without the need for copying them to and from kernel memory [32].

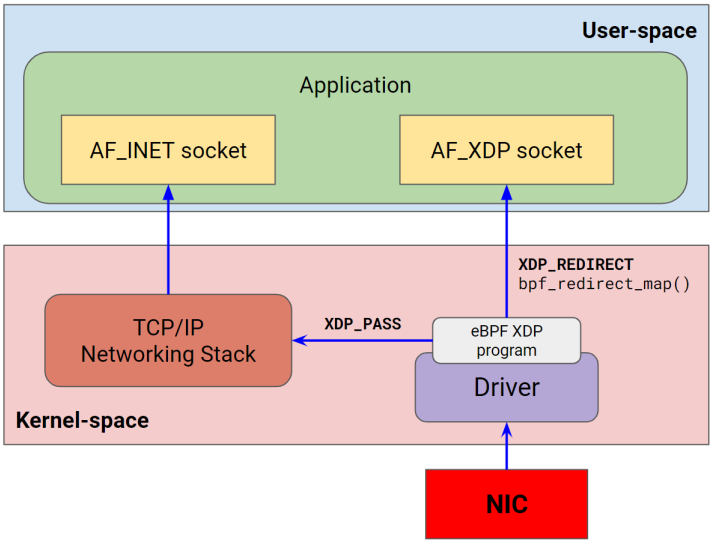

Figure 2.7: `AF_XDP` offers a fast path inside the kernel to send frames directly to the application

Each XDP socket is associated with an RX and a TX ring for receiving and sending packets respectively. Both rings point to a data buffer in a memory region called a *UMEM*, which is shared between the application and the network device driver. This enables zero-copy packet transfer.

The TC layer

An important limitation of the XDP layer is that it can only process inbound packets. To process egress (TX) packets, the closest layer to the NIC is the *Traffic Control* layer. The TC layer is responsible for executing traffic control policies, which are configured using *queuing disciplines* (qdisc) for the various packet queues in the kernel. Whenever a packet needs to be sent to an interface, it is enqueued to the qdisc configured for that interface. For eBPF, the TC hook point is located on *clsact*, a special qdisc that allows classification actions to be defined by the eBPF programs attached to it. It works for both egress and ingress traffic [18, 33].

TC programs receive a pointer to a `struct __sk_buff` as the context parameter. This structure has the same beginning and end pointers as in the XDP context, but has a richer set of fields because the kernel has already allocated and parsed the packet to extract protocol metadata. Just like in XDP, the input packet can be modified in-place and the program must return one of five actions:

- `TC_ACT_OK`: allow the delivery of the packet in the queue.
- `TC_ACT_SHOT`: drop the packet from the queue.
- `TC_ACT_UNSPEC`: allow the packet to be passed into the network stack.
- `TC_ACT_PIPE`: perform the next action (such as another eBPF TC program), if it exists.
- `TC_ACT_RECLASSIFY`: restart the classification process for this packet from the start.

The Socket layer

The Socket layer refers to programs that execute after the TCP/IP stack and before enqueuing the packet into the socket buffer. There are several socket-related program types which are similar in structure but are intended for different scenarios.

- `BPF_PROG_TYPE_SOCKET_FILTER`:
  A program used to inspect and filter ingress packets on a socket. These programs are attached to specific sockets using the `setsockopt()` system call (typically when creating the socket). The context provided is a pointer to the `struct __sk_buff`, containing metadata [25].

- `BPF_PROG_TYPE_SK_SKB`:
  A program used to inspect and filter incoming packets into an established socket's buffer. This program is slightly different: it is attached to a socket map, rather than a socket. The socket map stores references to sockets to support redirection of packets between sockets (this is known as *verdict*). The program can be attached using the `BPF_SK_SKB_STREAM_VERDICT` hook. It can also be used for parsing messages of an application layer protocol running over a data stream (*stream parsing*). It also receives a pointer to the `struct __sk_buff` as the context [34].

- `BPF_PROG_TYPE_SK_LOOKUP`:
  A program used to provide programmable socket lookup performed by the transport layer when a packet is to be delivered locally. It is attached a network namespace, and runs whenever the transport layer needs to find a listening TCP or an unconnected UDP socket for an incoming packet. Incoming traffic for established connections do not trigger the execution of this program. It also works in conjunction with a socket map for socket redirection, and receives a pointer to a `struct bpf_sk_lookup`, containing packet metadata [35].

2.2.6 Practical use cases and applications

Although eBPF is a fairly recent technology, it has already been used to drive both research and industrial projects. These range from supporting better observability and monitoring for production systems to enhancing network-based applications.

An example of an eBPF project already deployed to production is Cloudflare's DDoS attack mitigation system. This uses XDP to inspect packets as close to the hardware as possible for optimal throughput and to implement specific firewall rules that are not expressible using existing software such as *iptables* [36]. Another practical application of eBPF using XDP is Katran, an open-source layer 4 load balancer developed by Facebook Incubator. With XDP, it is able to run packet forwarding routines as soon as the packet is received by the NIC [37].

Another area that is adopting eBPF is container networking. Cilium is an open-source project that uses eBPF extensively to provide secure and transparent connectivity between microservices running in container management systems like Kubernetes and Docker. Cilium can apply per-container security, visibility and networking policies by loading user-supplied eBPF programs [38].

Netflix is another company that has been contributing to eBPF since 2014, and has also adopted eBPF's tracing support to extend their performance monitoring and system profiling infrastructure for their cloud services [39].

Other than industry-led projects, there are various research-oriented applications of eBPF. These are generally about the topic of performance acceleration for network processing. For example, BMC is an eBPF-enabled design that aims to accelerate the Memcached key-value store, by handling requests inside the kernel and bypassing the networking stack [40]. Other proof-of-concepts derived from academia include building an *iptables* clone using eBPF for more efficient matching algorithms [41].

2.3 Alternative network acceleration techniques

One of the main topics of this project is about network acceleration. Therefore, it is important to consider some of the other existing techniques that are being used both in industry and research. This section will briefly describe some of these alternative technologies and compare them with eBPF.

2.3.1 User-space networking and kernel bypass

User-space networking refers to running packet processing functions and other network-related processing entirely in user space, rather than in the kernel. Typically, the kernel's network stack handles all networking functions, such as receiving and sending packets, routing, and filtering. However, as discussed, this can lead to performance bottlenecks, as the kernel's scheduling and memory management can introduce overhead.
By moving networking functions to user space, it is possible to achieve better performance and more efficient use of resources. It allows for more fine-grained control over the networking functions, as well as the ability to optimise them for specific applications.

One framework commonly used to implement user-space networking solutions is **DPDK** (Data Plane Development Kit). DPDK is an open-source software framework that provides a set of libraries and drivers for fast packet processing in user-space [42]. DPDK provides a set of libraries for packet processing, such as packet buffers, packet classification, and flow classification. It also provides a set of drivers for various types of NICs, including 10 Gbps and 40 Gbps NICs. These drivers provide a low-level interface to the NIC, allowing for efficient access to the network interface without unnecessary copies being created.

Compared to eBPF, DPDK completely bypasses the kernel by sharing a pre-allocated memory buffer between the user application and the NIC device driver, whereas eBPF does not bypass the kernel (XDP may programs redirect packets such that they bypass the networking stack, but they still execute in kernel space). In addition, the programming model for DPDK is based on polling, while eBPF program are implemented in an event-driven manner. While DPDK can offer high-speed packet processing and allows for more fine-grained optimisations for certain applications (such as high-frequency trading), it involves a substantially higher effort from the developer to configure the system to work with DPDK and requires a compatible NIC. On the other hand, eBPF is widely-available and does not require as much effort to configure and start writing code.

2.3.2 Hardware-offloaded networking with SmartNICs

SmartNICs are specialised network cards that offload certain tasks from the CPU to the network card itself, and are often programmable to accelerate the performance of certain network-related functions. While SmartNICs can be implemented with embedded cores (such as the Nvidia Bluefield card [43]) or ASICs (such as some of the Netronome cards), this section will specifically focus on FPGAs. Their recent surge in popularity in the area of high-performance computing has led to increased use of such devices for network acceleration, as they can be programmed to provide ultra-low latency packet processing for specific applications.
FPGAs are one of several options that allow the deployment of SmartNICs, which can be programmed to accelerate specific network workloads. They have become an appealing choice within data centres running network I/O bound applications, as all the packet processing functionality is offloaded to the hardware and bypasses the CPU. This effectively minimises the I/O bottleneck between the NIC and the CPU [44]. Another advantage is that they are more power efficient than general-purpose CPUs, which is another factor to consider for data centre deployments.

In such setups, there is a software interface which the server application can query to execute network primitives, such as sending a packet. This interface manages the communication between the application and the FPGA via the PCIe bus, and a shared buffer between the CPU and the FPGA. The FPGA SmartNIC then runs its own optimised implementation of a TCP/IP stack to process the packet accordingly and handle all the functionality a CPU would execute in the OS kernel. This approach [5] has been implemented to accelerate a scatter-gather primitive and has shown a big performance increase over CPU-based network processing.

While FPGA-based SmartNICs are proven to accelerate network performance, these devices are very expensive and require expertise that most software engineers lack. Thus, if ultra-low latency is not critical to the business, more accessible alternatives such as eBPF may be sufficient.

2.3.3 In-network processing with programmable switches

In-network programming refers to the ability to execute programmable code within a network device, such as a switch or router. This code can be used to perform a variety of tasks, such as packet filtering and data processing. This has emerged as a new computing paradigm and is gaining popularity among cloud vendors who rely on software-defined networking (SDN) to perform packet processing tasks on the network switches inside a data centre.
Domain specific languages, such as P4 [45], have been specifically designed to implement packet processing pipelines on programmable switches. Therefore, custom logic can be directly embedded into this devices for high-performance applications.
An example of in-network programming applied to accelerate scatter-gather workloads is SwitchML [46], which aggregates the updated parameters of the machine learning model being trained directly on a programmable network switch. Not only does this reduce the overhead on the endpoints, but it also reduces the volume of packets currently in the network, leading to big performance improvements.

Both programmable switches and eBPF can be used to accelerate network workloads, but they have different approaches and use cases. Programmable switches are useful when you need to program the behavior of network devices at a high level, while eBPF is useful for low-level packet processing and monitoring tasks in Linux-based systems.

Chapter 3

Related work

As an emerging technology, eBPF has garnered significant attention in recent years as a powerful tool for performance acceleration of network applications. By surveying and analysing prior research, we examine the existing literature in order to discover methodologies and challenges encountered in leveraging eBPF to optimise the performance of various systems and applications. Through this exploration, we seek to build upon the existing knowledge and contribute to the further development of eBPF-based performance acceleration techniques. We will also look at existing work that targets the acceleration of scatter-gather workloads and understand what techniques are used to achieve this higher performance.

3.1 BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing (2021)

Ghigoff et al [40] present an eBPF-based extension to the popular Memcached key-value store as a way to accelerate its performance without having to resort to DPDK. They also identify performance bottlenecks in the Linux kernel when using high-speed network interface cards to serve a large number of Memcached GET requests, which are typically small UDP messages that make up the majority of traffic to the servers.

Their implementation, BMC, relies on pre-stack processing to handle GET requests with XDP programs attached on the ingress route. In order to serve requests fast, they are able to build a working copy of the cache using eBPF maps based on the GET/SET messages received. If there is a cache hit, they are able to reply directly from the XDP hook without entering the network stack and avoid the overhead associated with it. Otherwise, the message is allowed through to the Memcached application in user-space.
As a caching application, there are many corner cases to handle. One technique used in the paper is to split up the logic into multiple eBPF programs and transfer control flow using tail calls. This technique is an effective way to bypass the per-program instruction limit imposed by the static verifier.

An important point to consider about the work presented in this paper is that it focuses on accelerating UDP workloads. The reason for this is to avoid processing a stream of TCP frames which may be unordered, whereas with UDP, the message is fully enclosed as a single message. In addition, since GET requests are typically small, the majority of these requests can fit in a 1500 byte packet.

Their comprehensive evaluation shows that BMC is able to achieve throughputs of up to x18 better than the vanilla implementation of Memcached. It also achieves a similar throughput to a DPDK-based version of Memcached but with a much lower CPU utilisation. The test data is tailored to represent a realistic workload such that the messages fit in UDP messages. However, it is important to realise that in practice, TCP requests are still common and would not be suitable for acceleration in their implementation. Despite this, BMC is a clear example of how eBPF can be used to successfully accelerate a real-world application. The implementation is open-source and could be a useful resource to see how the techniques described are implemented in practice.

3.2 Electrode: Accelerating Distributed Protocols with eBPF (2023)

Zhou et al [47] present yet another application where eBPF is used to improve throughput and latency: consensus protocols. The work focuses on accelerating the Multi-Paxos algorithm by minimising the number of user-kernel crossings and kernel network stack traversals.

The paper introduces the Multi-Paxos algorithm and highlights the large number of messages required to perform distributed state replication. The excessive number of messages exchanged means there is a large communication overhead, making up to 50% of the CPU time according to their measurements. This makes eBPF a suitable technology to reduce this overhead and achieve higher throughputs. Just like the BMC paper discussed earlier, this implementation also targets messages that can fit into a single packet and therefore assumes UDP as the layer 4 protocol.

The main challenge is to understand which parts of the consensus algorithm could be feasibly offloaded into the kernel to reduce the overhead. The authors decide to implement common network functionality with eBPF programs, for example, message broadcasting. They also implement fast acknowledging as an eBPF program to avoid incurring a per-packet overhead when receiving ack messages. Mechanisms such as failure recovery or handling packet loss are left to the application, as they involve more complex logic which could be difficult to implement in eBPF. This is a sensible decision, as eBPF imposes several restrictions that could make the implementation of these mechanisms very difficult.

The broadcasting mechanism is implemented at the TC layer, as it is located on the egress route. It relies on the `bpf_clone_redirect()` helper function to generate new packets and perform the broadcast operation in the TC layer. The key advantage of this is that the cloned packets do not need to re-traverse the upper layers of the kernel networking stack.

The remaining implementation consists of optimisations on the ingress route using XDP, for example, fast acknowledgement. Once again, the benefit of using XDP programs is to perform pre-stack processing and avoid network stack traversals. To communicate these acknowledgements to the application, a MPSC ring buffer (an eBPF map type) is used efficiently forward preparation messages to user-space. This could be a useful technique for transferring aggregated data into the application in scatter-gather workloads with in-kernel aggregation. The paper then explains how the specific parts of the Multi-Paxos algorithm are implemented in eBPF, going into more detail and using different eBPF maps for stateful operations.

The evaluation shows a latency decrease of x1.6 and a throughput increase of x2 compared to the vanilla implementation, but does not perform as good as a DPDK-based version with pure kernel bypass and user-space networking. Still, the authors discuss how eBPF is readily available on Linux and takes advantage of built-in features such as receive-side scaling, and does not require a high effort deployment (which is the case with user-space networking libraries). The authors also propose the use of `io_uring` in the future work discussion as a tool to further accelerate performance of messages that must traverse the kernel network stack. This could be a useful mechanism to improve the performance of the mechanisms that are not implemented in eBPF.

3.3 Scaling Distributed Machine Learning with In-Network Aggregation (2021)

Distributed machine learning training is becoming an increasingly common workload, and with significant advances in compute performance, this has become a network-bound workload. As such, Sapio et al [46] propose a novel communication primitive to perform aggregation in the network using programmable switches and alleviate the network bottleneck.

Their implementation, SwitchML, performs simple arithmetic operations on programmable switches which reduces the amount of data transmitted during synchronous updates to the model among workers in the training loop. Workers send their updated model parameters as vectors and the

aggregation primitive sums the updates, distributing only the final value.

The paper describes the aggregation protocol and how it is implemented on switch devices using streams, whereby the aggregation takes place on a limited number of vectors at once. The main challenges here are limited computation, since floating point operations are not supported by default and the authors must implement their own floating point logic (by scaling integers), and dealing with packet loss. This is similar to eBPF in the sense that it also has limited computation due to the restrictions imposed by the static verifier, as well as the lack of floating point data types inside the Linux kernel.

While the evaluation shows that SwitchML speeds up training by up to x5.5 for several benchmark models, the length and detail of the paper demonstrates the high implementation effort to integrate SwitchML with existing machine learning frameworks like PyTorch. In addition, there is an extensive discussion on the deployment of SwitchML and its dependency on state-of-the-art hardware to make the system worthwhile.

3.4 Specializing the network for scatter-gather workloads (2020)

In scatter-gather workloads, network communication is often a bottleneck, especially when the computation at each worker is small. This can lead to low throughput and high CPU utilisation which can increase the completion time when the server is under high load. Alvarez et al [5] explore hardware-offloading the scatter-gather primitive using FPGA-based smart NICs.

As opposed to the previously discussed literature, the system presented in this paper focuses on TCP communication. Thus, the bulk of the implementation lies in building a TCP/IP network stack that runs on the FPGA and building an interface for the on-CPU application. With this, the design allows for the application to invoke a scatter-gather operation, and the logic on the FPGA is responsible for generating and scheduling queries, as well as collecting responses, all via their custom hardware TCP/IP stack.
The authors then give a high-level overview of the different mechanisms implemented on the FPGA to effectively schedule requests using batching and rate limiting to avoid the congestion in the gather phase.

The results are not surprising: the FPGA-based scatter-gather primitive outperforms the standard CPU-based implementation, with the performance difference increasing as the number of workers increases. At 1000 workers, the authors estimate their implementation to outperform the baseline by nearly x50. This is due to the inherent advantage of moving this logic to the hardware and avoiding the kernel bottleneck. However, similar to the SwitchML paper discussed in section 3.3, the complexity of the implementation makes the paper somewhat difficult to follow, due to the hardware-based approach.
While the performance benefits of hardware offloading are huge, the technical complexity of the implementation re-emphasises the accessability of eBPF-based solutions and its lower barrier to entry for developers.

Chapter 4

sgbpf: an eBPF-accelerated scatter gather library

The core of this project was the design and implementation of *sgbpf*, a library that exposes a scatter-gather network primitive for UDP-based communications, using eBPF to accelerate the performance of the coordinator node. It is written in C++ and is easily embeddable into any network application. It uses eBPF to execute code in the XDP and TC layers of the kernel in order to process packets as early as possible. It also supports custom user-defined in-kernel aggregation logic and offers a flexible API to access the aggregated data efficiently from user-space, incurring a minimal number of system calls.
Our benchmarks show that *sgbpf* comfortably outperforms all baseline implementations using alternative Linux-native I/O APIs (including modern interfaces such as `io_uring`) for large fan-outs, while also achieving at least as good performance for smaller fan-outs of under 50 workers.

This chapter will describe and justify the overall architecture of *sgbpf* and the key design decisions taken to implement the different components of this library.

4.1 Requirements

While most software projects have well-defined requirements in advance, *sgbpf* is a result of a feasibility exploration. Therefore, most of the requirements for this library were specified along the investigation process (after understanding the limitations of eBPF and what was technically feasible) and the implementation process. Listed below are the key requirements that were identified:

- The system should achieve at least as good (ideally better) performance as the alternative state-of-the-art network I/O APIs, such as `io_uring` or `epoll` (this excludes user-space networking libraries such as DPDK that rely on busy-waiting). This is to be evaluated primarily using the latency and throughput metrics.
- The system should support in-kernel aggregation to some extent. This means that packets are processed and aggregated together within the kernel (using eBPF), rather than by the user-space application. Ideally, this aggregation logic should be customisable and/or defined by the user.
- The system should offer an intuitive API to the user-space application to invoke a scatter operation over a list of given worker endpoints and then easily access the gathered data on completion, making as few system calls as possible.
- The system should make no assumptions on the underlying threading model or event loop (if any) used by the user-space application. It should at least be suitable for a single-threaded application and ideally a multi-threaded application.
- The system should support configurable options such as completion policies to determine whether the scatter-gather operation has finished, and timeouts for coarse-grained recovery.

The aforementioned points are the **key requirements** and thus non-exhaustive. In reality, these requirements were be broken down into more specific targets to ease the implementation process and follow an incremental approach. As described in the rest of this chapter, the vast majority of the requirements were met and in some cases refined, in order to grow the feature set of the library beyond the key requirements listed.

4.2 Architecture overview

As explained in section 2.2.2 and visualised in figure 2.5, eBPF-based systems rely on a user-space component to load the eBPF code into the kernel at runtime and interact with it accordingly. Our library is no different: it consists of a user-space API written in C++ that loads the eBPF code into the kernel and then communicates with it using maps (and other techniques described later). This section will describe the overall design and architecture of the library, including how both components interact.

Sockets in *sgbpf*

The first fundamental design decision is how sockets are used in *sgbpf* to send and receive data. One of the key objectives from the original project proposal was to efficiently check for the completion of a scatter-gather operation, and only then perform the individual reads of the packets without blocking. This is enabled by the idea of a **control socket**, which is used to determine whether a scatter-gather operation has completed. Therefore, *sgbpf* opens a total of $n + 2$ UDP sockets, where n corresponds to one socket per worker and the remaining two correspond to the control socket (for receiving the completion notification) and the scatter socket (for sending data). These sockets are represented visually in figure 4.1.

The control socket is a fundamental part of the design and implementation of *sgbpf*, as it also used to deliver the final aggregated value to the application from the eBPF code (in the case where the responses are aggregated within the kernel). It can be configured in one of many ways by the application, adding a large degree of flexibility in the API. By default, it is exposed as a regular Unix file descriptor and can be passed as a parameter to any of the standard I/O APIs to read the data. An alternative method bypassing the control socket is described in section 4.4.5.

The final socket, the scatter socket, is used for communicating data from the application into the eBPF programs inside the kernel. As the name indicates, it is primarily used to initiate a scatter-gather operation: a message is prepared by *sgbpf* in user-space and is written into the scatter socket. This is then intercepted by an eBPF program which performs the broadcast from inside the kernel (described in detail in section 4.3). This socket is also re-used as a communication channel with the eBPF programs for other scenarios, such as to submit a "clean-up" message to reset any in-kernel state leftover after a scatter-gather request has completed.
The final socket type is the one used to receive data from the individual workers. During the initialisation of the library, the application must specify the fixed set of worker endpoints that will participate in the communication. This is used to open n worker sockets which are used to receive responses from the workers. Note that in certain configurations of *sgbpf* (including the optimal use case where aggregation occurs in the kernel) these sockets are not used at all, as the responses can be dropped within the kernel and the final aggregated data is delivered via the control socket. One brief point to mention is the fact that *sgbpf* uses one socket per worker when in reality, this is not a strict requirement. Because these are UDP sockets, they can receive data from any destination address and port number, without connecting to the endpoint. This means that a single socket is sufficient. We have opted for a socket-per-worker design as it logically makes sense and the file descriptors naturally act as a unique identification number. It also made the debug process easier and incurs no extra latency cost in the optimised path, as all this setup takes place prior to any operation.

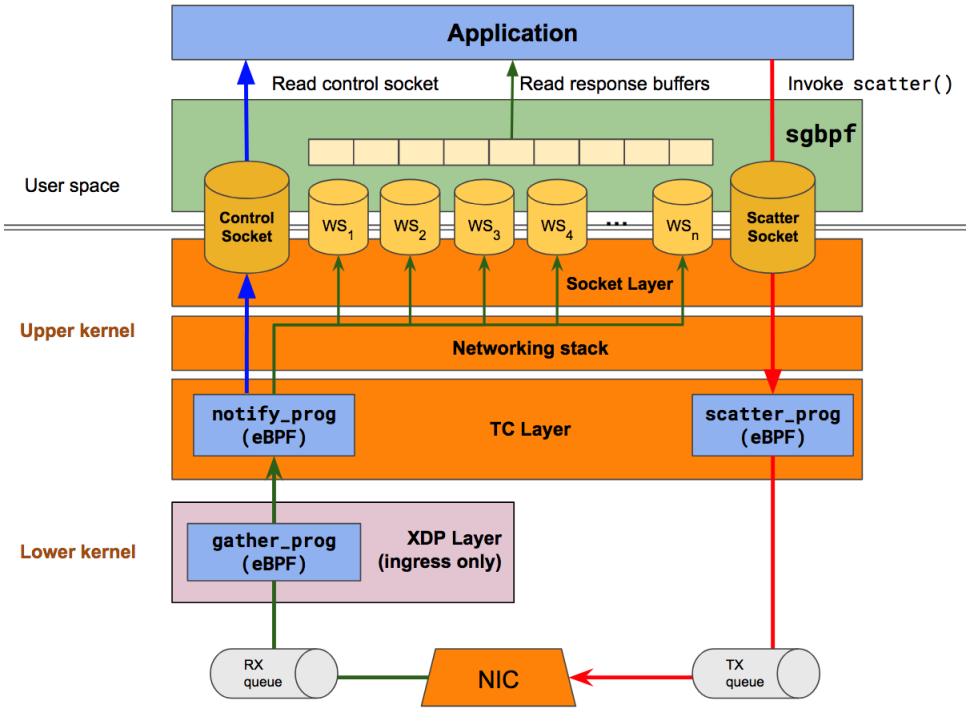

Figure 4.1: A diagram visualising the main components of *sgbpf* and the flow of data between user- and kernel-space using sockets as a communication channel.

In-kernel processing with eBPF

One of the main contributions of the work presented in this dissertation is the use of eBPF to accelerate the performance of network applications. As such, *sgbpf* is centred around eBPF and makes use of a variety of its features to achieve high performance. The implementation consists of three different eBPF programs attached to the lower layers of the kernel, outlined in table 4.1. These are explained in more depth in the rest of this chapter.

Program	Hook	Direction	Purpose
`scatter_prog`	TC	Egress (Scatter)	Broadcasts the initial message to the list of workers.
`gather_prog`	XDP	Ingress (Gather)	Performs aggregation of incoming response packets.
`notify_prog`	TC	Ingress (Gather)	Notifies the completion of an operation to the user.

Table 4.1: The different eBPF programs used by *sgbpf* (also shown in figure 4.1).

These programs are event-driven and only execute whenever a packet reaches the hook to which they are attached to. For example, `scatter_prog` executes whenever data is written into the scatter socket and processed by the networking stack, whereas the execution of `gather_prog` is triggered by the arrival of a packet to the network card's device driver.

The three programs communicate via eBPF maps, which are shared across all CPUs and are stored in kernel memory. Therefore, an important aspect of the implementation was to make the map entries thread-safe, in order to ensure correctness and avoid data races. It is crucial to realise that packets received in the ingress route may be processed by different cores concurrently. This is due to the receive-side scaling mechanism of the Linux kernel which distributes packets across different cores to elastically balance the processing load [48].

As previously mentioned, *sgbpf* supports user-defined in-kernel aggregation logic. This is a user provided eBPF XDP program which is called from `gather_prog` and executes logic over the incoming packet and the current aggregated value. More details on this can be found in section 4.4.2.

Data structures in eBPF

Maps are an essential piece to any eBPF-based system where state must be maintained across different programs. They also act as a way to communicate data with the application, as the eBPF API allows developers to directly perform CRUD operations via the `bpf()` system call. Our library uses several eBPF maps to persist data between programs, ensuring thread-safety where required, and to populate any relevant data from user-space during the setup phase. The key data structures are described below:

- **Workers map**: an array containing the IPv4 address and port number pairs for each remote worker endpoint. This is populated in advance from user-space and is queried by `scatter_prog` to perform the initial broadcast from the TC layer.
- **Control socket map**: an array with a single entry containing the port number of the control socket. This is used to redirect data into the control socket by the `notify_prog` whenever a scatter-gather operation is complete.
- **Request state map**: an array containing metadata about the status of an in-flight scatter-gather request. For each entry, it stores the count of responses received, the number of workers to wait for and the count of packets that have been aggregated. These counts are used to determine the completion of the request. It also keeps a spinlock to synchronise reads and updates to these fields.
- **Aggregated data map**: an array containing the current aggregated value of the responses. This is represented as a fixed-size array whose size is determined by the maximum length of a response packet (which in turn is limited by the layer 3 maximum transmission unit size, typically around 1500 bytes). It also has a spinlock associated to each entry to synchronise updates to the aggregated data.

Since all eBPF maps must have a maximum size specified at compile-time, the user-space component of *sgbpf* can take advantage of this requirement to preallocate all the memory needed to manage the state of scatter-gather requests in advance, thus avoiding dynamic allocations on the optimised path. As such, the indices used to access into these arrays are computed based on the request ID and may wrap around when the maximum size is exceeded. This is a limitation which the application developer must account for; if there are more in-flight requests than the maximum allowed limit, this may result in undefined behaviour. The maximum size is set to a high value, so this should not be a problem in practice.

Protocol message format

As a network application, *sgbpf* defines a custom application-layer protocol on top of UDP for its messages. It assumes fixed-size packets which are determined by the IP layer maximum transmission unit and consists of an 18-byte header with fields reserved for the request ID, the sequence number (only relevant for multi-packet responses), the number of packets in the response (by default, set to one), the body length (useful for vector data) and two more fields reserved for internal usage. The remaining space is used for the body containing the actual data that should be aggregated. Both the scatter and gather phases use the same protocol, although some fields are only relevant in the gather phase.

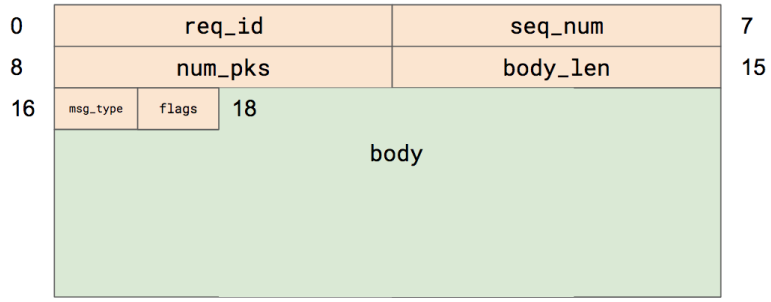

Figure 4.2: The data format used by messages in *sgbpf*, with the byte offsets for the header fields.

Note that all messages are forced to be the same size, even if that means wasted space in the body. This is a requirement imposed by the eBPF static verifier so that it is able to check the pointer bounds of the packet and guarantee safety when dereferencing the packet payload.

Asynchronous batched I/O with `io_uring`

One of the distinctive features of the design is the use of `io_uring`, an asynchronous I/O interface available in the Linux kernel since version 5.1. This interface relies on two memory-mapped rings that are shared between user-space and the kernel, effectively eliminating the need to issue extra system calls to copy data to and from these buffers. The user application can append I/O operations to the submission queue (SQ) and notify the kernel through a single system call (allowing for batched operation submission). The kernel then handles these operations accordingly and when they have completed, an entry is appended to the completion queue (CQ) to let the user-space application know that the operation has been completed [49].

sgbpf relies heavily on `io_uring` and takes advantage of operation batching to minimise system calls and user-kernel boundary crossings, as described later in section 4.3. It also takes advantage of buffer provisioning so that `io_uring` is able to write the packet contents received into a predetermined buffer on completion, which eliminates the final copy from the kernel into user space.

The implementation of *sgbpf* consists of around 1600 lines of C++ code for the user-space component of the library, and around 700 lines of C code for the in-kernel eBPF programs. The code is available on Github. The next two sections in this chapter will dive deeper into the design of *sgbpf* and how it was implemented.

4.3 Accelerating the scatter phase

The scatter phase can be regarded as a broadcast operation: a single message is sent out to all the given worker endpoints to trigger the execution of their local task. As such, naive implementations of this pattern require a linear number of system calls (a socket write per worker), producing many context switches and copies across the user-kernel boundary. This section explains how *sgbpf* is able to implement the scatter operation to broadcast a message using a single system call and a single traversal of the kernel networking stack.

4.3.1 Fast scattering

Scattering in *sgbpf* is implemented using an eBPF program attached to the TC egress layer, located after the kernel networking stack. The program (named `scatter_prog`) receives a single packet from the user-space component of the library and is cloned using the `bpf_clone_redirect()` helper function, which creates a copy of the incoming `skb` and redirects it to a given interface (in this case, the same interface specified for the original packet) [50]. This function is called for every worker endpoint stored in the workers map, which is a array of worker endpoints containing the destination IPv4 address and port number. This map is populated in advance by the user-space application using the standard map update operation via the `bpf()` system call. Note that this takes place during the setup and is therefore outside of the optimised path.
During the iteration over the workers map, the cloned `skb` is updated such that the destination IP address in the IP header (layer 3) and the destination port number in the UDP header (layer 4) match those of the intended worker. The checksum is then recomputed and the new packet is redirected to the network interface.

From the user-space side, *sgbpf* exposes the `scatter()` method on instances of type `sgbpf::Service` which invokes a new scatter-gather operation. This method constructs a new object of type `sgbpf::Request` using a unique request ID (a four-byte integer) and prepares the single packet that is written into the scatter socket (and later received by the TC program, as described above). This includes preparing the header fields and copying the message body specified by the user, which may be useful to communicate extra information to the workers.
One observation to highlight from a performance perspective is the lack of dynamic allocations to construct the new `sgbpf::Request` instance. Since all eBPF maps require a maximum size to be specified at compile-time, we can take advantage of this requirement to preallocate all the memory needed and construct the objects in-place without requiring a dynamic allocation on the optimised path.

Once the message has been prepared, the `scatter()` method then adds the relevant I/O operations to the submission queue of the `io_uring` instance. As described in the earlier section, *sgbpf* makes frequent use of this mechanism to batch I/O operations together and submit them asynchronously, resulting in a **single system call**. In this case, it adds a single `sendmsg()` operation (to write the message into the socket) and if specified, it also adds `recv()` operations (in the case where the user is interested in receiving all individual packets; see section 4.3.2 for more details). The final step is to notify the kernel that the operations added to the SQ are ready to be processed using `io_uring_submit()`, which is a non-blocking system call. This means that the I/O operations are executed asynchronously and the application can move on to other tasks without blocking.

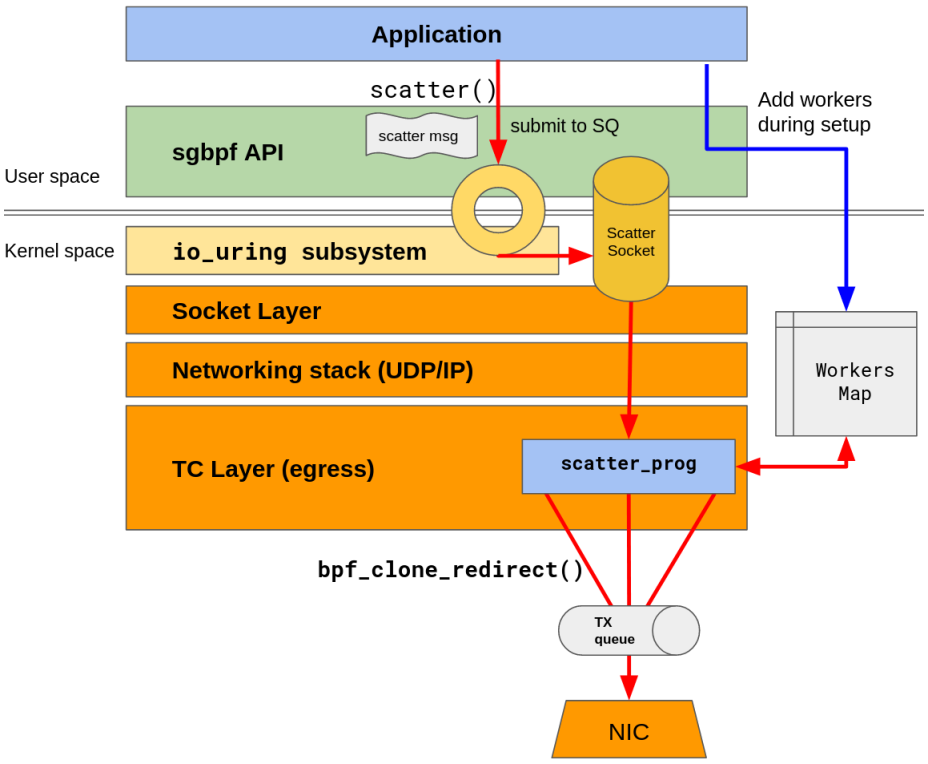

Figure 4.3: A diagram showing the user-kernel interaction during the scatter phase and the control flow path of the data (in red) within the kernel. The `scatter_prog` attached at the TC layer performs the scattering after the main networking stack is traversed.

Comparing in-kernel scattering with eBPF vs. user-space scattering with `io_uring`

With the heavy use of `io_uring`, an alternative design was also considered in which the `scatter()` method submits a batch of socket writes to each worker, effectively moving the iteration over the workers to user-space. This would drop the need of the TC egress program or any eBPF logic in the scatter phase.

We implemented this alternative design and compared the performance of both implementations using microbenchmarks. We found that the eBPF-based implementation consistently outperformed the `io_uring` implementation for different fan-out sizes (i.e. number of workers). Although the average difference in latency was only around 9%, the throughput difference (under load) was approximately 28% for fan-outs under 100 nodes and around 43% for larger fan outs.

The slower performance with the `io_uring` version is explained by the number of traversals through the kernel networking stack. Despite reducing the number of system calls to one, each packet in the batch of write operations must still traverse the upper layers of the kernel networking stack, including the Socket layer, the UDP processing functions and the IP layer. On the other, as the TC egress layer is located after the IP processing methods, the cloned `skbs` in the eBPF program do not have to re-traverse the full networking stack, which reduces latency. This was verified using Linux's tracing mechanisms, allowing us to check how many times a particular function in the kernel was called. We found that the eBPF implementation calls the `ip_finish_output()` function only once, whereas the `io_uring` version has one invocation per worker.

While theoretically we would expect a larger difference in latency between both versions, it is likely that we see this small difference due to caching effects (as multiple packets traverse the same code path), and also because the processing logic for UDP packets is very simple. Nevertheless, *sgbpf* implements the design described above and performs the scattering in the TC egress layer with eBPF.

Design pattern: triggering eBPF execution asynchronously

As a result of the work described above, we noticed that `io_uring` can be used as an efficient way to trigger the execution of eBPF programs attached to network hooks asynchronously. Since eBPF programs are purely event-driven (in our case, by network data), there is no "direct" way for user-space applications to trigger the execution of eBPF programs attached to network hooks. Instead, we can trigger execution by generating a network event ourselves, for example, sending a packet. If the execution of the eBPF program is not time critical and simply needs to "occur eventually", we can also take advantage of batching in `io_uring` so that the execution is triggered whenever the next batch of operations is submitted, saving an extra system call.

This technique is used in *sgbpf* to clean up the in-kernel resources (such as the entries in the eBPF maps) used by a scatter-gather invocation when completed. Rather than updating the maps directly from user-space with the standard `bpf()` system call, the application instead adds a new socket write operation with a "cleanup packet", which contains metadata about what needs to be cleaned up. Whenever the next batch of operations is submitted, this cleanup request will also be processed by the same TC egress program used to perform the scatter. This pattern can be generified as a reusable technique for other eBPF-based network applications. A similar workaround is described in section 4.4.3 to set the completion policy of a request without directly updating the eBPF map from user-space.

4.3.2 Buffer provision for receiving packets

The optimal usage of *sgbpf* arises when the data received from the workers can be aggregated in the kernel, and the packets are dropped early. However, there may be workloads where the data cannot be aggregated in the kernel due to the limitations imposed by eBPF (more details in section 4.4.2). In such cases, the application developer may need to perform aggregation in user-space and read the individual responses directly.

Our design supports reading individual packets through the use of `io_uring`'s buffer provision mechanism [51]. The application is able to reserve a pool of buffers in advance and registers them with `io_uring`, such that whenever a read operation completes, the kernel is able to pick a free buffer and write the packet contents. Fixed-sized buffers are provided in groups identified by a group ID through the `io_uring_prep_provide_buffers()` function in *liburing*. Then, whenever a completion event for a read operation is obtained from the completion queue, it includes a 16-bit buffer ID which acts as an index representing the buffer that was picked for that operation. Thus, using the buffer group ID and buffer ID pair, the application can read the packet contents without requiring an extra copy across the user-kernel boundary.

The clear benefit of this mechanism is that it allows the kernel to pick a suitable buffer when the given read operation is ready to actually receive data, rather than upfront. However there is a noticeable drawback to this approach: the number of buffers that can be submitted in each group is limited to 2^{16} because the buffer ID returned is a 16-bit unsigned integer. The consequence of this is that every 2^{16} completed read operations, *sgbpf* needs to replenish the buffers by providing a new buffer group, which is a relatively expensive operation; it involves allocating a new pool of buffers (non-deterministic, as it relies on `malloc()`) and registering them with the kernel (linear-time operation, as the kernel iterates over the entire buffer group).
This is implemented using a counter which increments with every submitted `recv()` operation. Whenever the counter reaches $2^{16} - 1$, it performs this allocation and adds the buffer provision request to execute with the next batch. This results in a slow `scatter()` call once every 2^{16} reads.

In the case where reading individual packets is required (which can be specified in advanced by the developer when instantiating the `sgbpf::Service` object), the `scatter()` method will also add the correct number of `recv()` operations to the submission queue (in the same batch as the socket write operation used for scattering). The number of `recv()` operations added will depend on whether the worker responses will be returned as a single packet or as a sequence of separate multi-packet messages. It is crucial that the developer specifies how many packets are expected in each response so that the exact number of `recv()` operations are added to the SQ in advance.

Otherwise, a mismatch of expected packets and actual packets received (included as a field in the response packet's header) will incur an extra system call to add the remaining operations.

Comparison with a per-request buffer approach

An alternative approach to this problem is to allocate buffers on demand in a per-request manner. In summary, instead of allocating a large and fixed-sized amount of buffers used by all requests when exhausted, this approach will allocate the sufficient number of buffers whenever a new scatter operation is invoked, local to the `sgbpf::Request` object returned to the user. Also, this approach does not register any buffers with the kernel in advance: whenever a `recv()` operation is added to the SQ, it specifies the pointer to the user-space buffer to write into (this incurs an extra copy per received packet). It also makes memory management much easier, as the buffer can be freed in the destructor of the `sgbpf::Request` object.

The obvious drawback of this approach is that dynamic allocation must take place in the optimised path. In every invocation of `scatter()`, an allocation proportional to the number of workers is required. Dynamic allocation on the heap is widely-known to be non-deterministic and potentially slow [52]. To compare the performance of both approaches, we implemented both mechanisms with `io_uring` to receive all packets and perform aggregation in user-space (no eBPF involvement).
We ran a throughput benchmark locally (one worker per process) with a dummy workload (see section 5 for full details of the benchmarks). We ran both implementations on a small fan-out (50 workers) and a large fan-out (1000 workers). The results are shown in figure 4.4 on the next page.

For smaller fan-outs, we can appreciate the evenly-spaced drops in throughput for the globally provided buffers version every 2^{16} reads, as the buffers need to be replenished. This approach performs better than the per-request buffers alternative (around 20% higher throughput on average), which invokes dynamic allocation on every request. Towards the end of the benchmark, the per-request approach is able to recover to a similar throughput as the provided buffers approach. This is likely due to the behaviour of the underlying OS memory allocator which may have received a large block of memory, making subsquent allocations cheap. While this is good, this emphasises the unpredictability of heap allocation and may be undesirable for applications that prefer predictable performance.

For larger fan-outs, the globally provided buffers approach replenishes the buffers much more frequently. In the case for 1000 workers, this is required every 66 operations, which is lower than the throughput calculation sample rate of the benchmark (every 200 operations), producing a smooth constant line shown in the second graph. Similarly, because of the larger fan-out, the per-request approach requires large allocations on every `scatter()` invocation and the total memory copied to transfer the packets into user-space is also a factor for the latency increase. Despite this, the per-request approach is able to outperform the provided buffers approach; however, the average difference is only 10 operations per second.
With these results, *sgbpf* prefers the globally provided buffers approach as it relies on dynamic allocation much less, yielding predictable performance which is often a desirable property for high-load applications. Even in cases for very large fan-outs, the difference in throughput is small enough (under 5%) making it an acceptable solution.

Optimising the slow request with huge pages

Since *sgbpf* is allocating large fixed-size blocks of memory, $2^{16} \times$ `sizeof(sg_msg_t)` $= 96468992$ bytes, one optimisation is to enable huge page allocation. Linux supports *transparent huge pages* and allows applications to allocate data in pages sizes much bigger than 4096 bytes (the default is 2 MB) [53]. This technique allows the OS to more efficiently allocate large blocks of memory and results in fewer page faults when reading the packet buffers in a random access pattern (which is useful as packets may arrive out of order).
For fan-outs of 50, 500 and 1000 workers, we found that enabling huge pages yielded a relative throughput increase of 10.2%, 6.3% and 5.3% and respectively. However, as the fan-out increases, the performance benefit from huge pages decreases, and the bottleneck becomes the linear-time buffer provision routine that takes place within the kernel, which is no longer in our control.

Throughput comparison between different buffer management techniques in `io_uring` for small and large fan-outs

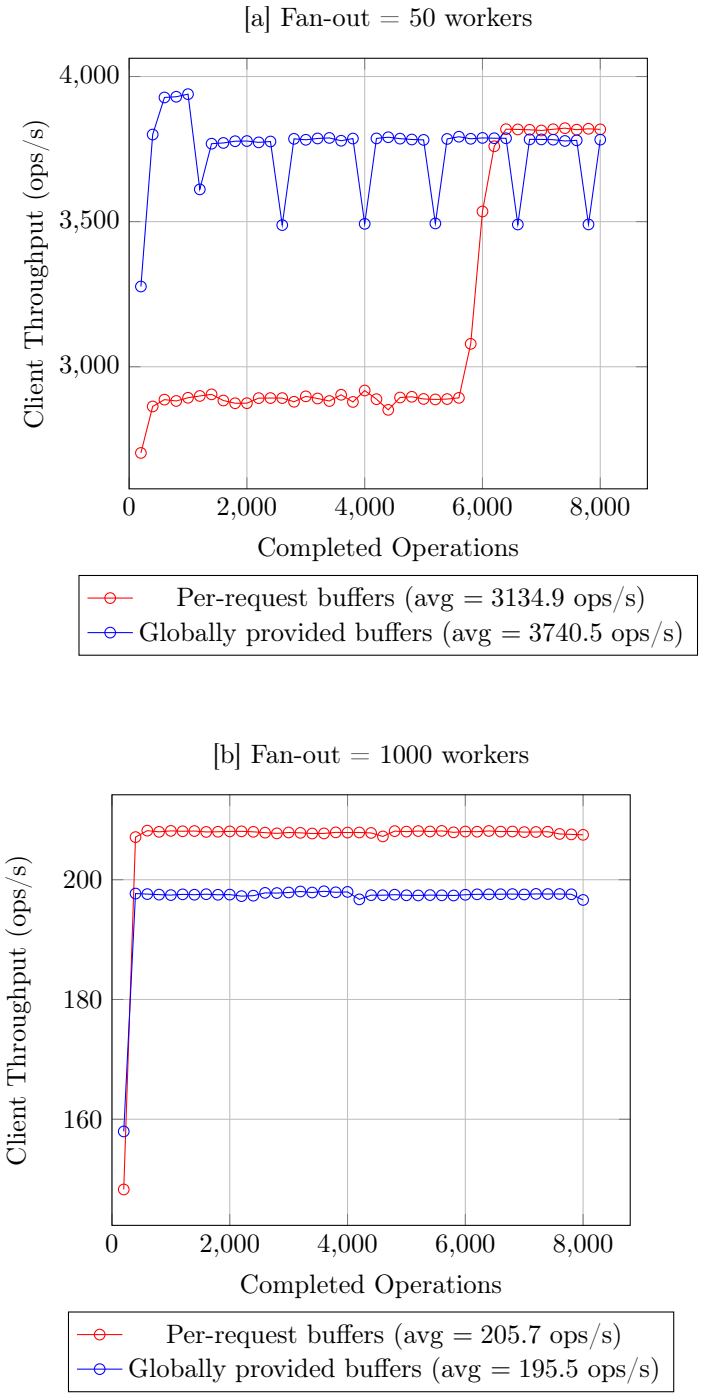

Figure 4.4: A throughput comparison of the client using the globally shared buffers provided to `io_uring` (same approach used by *sgbpf*) and the per-request buffers allocated on-demand approach (without the huge page optimisation).

4.4 Accelerating the gather phase

The gather phase is where the majority of the implementation effort takes place, both in the eBPF code and the user-space API. This phase consists of receiving and processing the responses from each worker, and performing some aggregation function that reduces these messages into a single result. This operation would typically take place in the user-space application, however the work presented here implements an in-kernel aggregation approach using eBPF, allowing for the response packets to be processed as early as possible, bypassing the kernel networking stack. Once the scatter-gather operation has completed, the final aggregated value is delivered to the application.

The rest of this section will dive into the implementation of the gather phase, explaining how eBPF is used to efficiently process responses and the interaction with the user-space application.

4.4.1 Processing responses with XDP

One of the key design decisions to achieve low latency scatter-gather operations in *sgbpf* is to process the response messages as early as possible inside the kernel. As introduced in section 2.2.5, the earliest eBPF hook in the RX path is XDP. Assuming an XDP-compatible device driver, XDP programs are triggered by the NIC's device driver and run in a soft interrupt request context. This takes place before the corresponding `skb` is even allocated and enqueued to the RX buffer in the kernel.

Our implementation attaches an XDP program (`gather_prog`) whose main purpose is to check for the completion of a scatter-gather operation and then perform one of the following options:

- Allow the packet through to the next layer in the case of multi-packet responses (in-kernel aggregation for such responses is not supported).
- Drop the packet if the scatter-gather operation has completed (for cases where the completion condition is satisfied by a subset of workers rather than all workers; see section 4.4.3).
- Perform a tail call to the aggregation function which is supplied by the user at compile-time as another eBPF XDP program (see section 4.4.2).

While *sgbpf* supports multi-packet responses as part of its protocol using the `seq_num` and `num_pks` fields in the message header, it currently does not implement any form of in-kernel aggregation for multi-packet responses. The reason for this is that UDP cannot guarantee the order of receival of packets within the same message, meaning that the aggregation logic would require complex state management to ensure that the reduction of data is performed correctly. Instead, all multi-packet messages are simply forwarded to user-space and defer any aggregation logic to the application. However, an approach is outlined in section 6.2.1 to extend *sgbpf* to support the aggregation of vector data sent as a multi-packet response.

An important piece of logic takes place in `gather_prog` in relation to the count of responses received, which is maintained in the request state map. Once the current response count has been checked to be less than the number of responses to wait for, the count is immediately incremented. The reason for this check and increment to take place here rather than after the aggregation logic is in case the final packet and its following packet are allowed into the aggregation program. In this situation, the aggregated value would be updated by both packets but only one of the two would proceed to the next stage. This results in an incorrect value of the aggregated data. Therefore, this is avoided by moving the completion check before the aggregation logic rather than after.
Once the aggregation logic has taken place, the execution of `notify_prog` in the TC layer is triggered. This program is in charge of notifying the application of the completion of the scatter-gather operation by delivering the final aggregated data to the control socket. The reason why this step is deferred to the TC layer is that a new packet must be created to be redirected to the control socket, which is not possible at the XDP layer. The details of this process are described in section 4.4.5.

4.4.2 Support for custom in-kernel aggregation

One of the objectives of this project was to explore the feasibility of moving the aggregation logic from user-space into the kernel using eBPF. The challenge here is to understand what can be executed inside an eBPF program (i.e.: what data types are allowed, what operations can be applied, etc.) and how to design a generic API that enables developers to express their aggregation logic succinctly with minimal knowledge of eBPF.

There are different approaches to implement such an API. One that was considered was to provide a set of predefined numerical operations which the developer can choose at runtime during the initialisation of *sgbpf*, similar to what communication backends like Open MPI offer in their APIs [54]. For a scatter-gather operation involving w workers, each replying with a vector of n elements

$$V = \left\{ v_0 = \begin{pmatrix} e_0^0 \\ \vdots \\ e_0^{n-1} \end{pmatrix}, v_1 = \begin{pmatrix} e_1^0 \\ \vdots \\ e_1^{n-1} \end{pmatrix}, \ldots, v_{w-1} = \begin{pmatrix} e_{w-1}^0 \\ \vdots \\ e_{w-1}^{n-1} \end{pmatrix} \right\}$$

we can apply the chosen **reduction operator** $\oplus$ over the input vector set V to produce a single aggregated result vector r:

$$r = \begin{pmatrix} e_0^0 \oplus e_1^0 \oplus \cdots \oplus e_{w-1}^0 \\ \vdots \\ e_0^{n-1} \oplus e_1^{n-1} \oplus \cdots \oplus e_{w-1}^{n-1} \end{pmatrix} = \begin{pmatrix} \bigoplus_{i=0}^{w-1} e_i^0 \\ \vdots \\ \bigoplus_{i=0}^{w-1} e_i^{n-1} \end{pmatrix}$$

The reduction operator is associative and often commutative (but not necessarily). Examples of such operators are addition, multiplication, the min() function and the max() function. While these kind of operations are a good fit for reduction algorithms over numerical data, this API design is somewhat restrictive and does not take full advantage of eBPF's capabilities.

Given that the whole point of eBPF is to allow developers to express whatever logic they want (under the rules imposed by the static verifier), our approach was to allow the application developer to write their own eBPF program entirely, providing a set of helper macros to make this process easier. This is a much more flexible approach and enables the developers to define the semantics of their own reduction operator. An example of where this may be useful is when working with string-based data instead of numerical data. Also note that the data sent in the bodies of the response messages are treated as byte vectors, meaning the developer can interpret each element of the vector as they wish. In the case for scalar data, this can be handled as a singleton vector.

In *sgbpf*, a single reduction operation takes place in the aggregation program between the current value and the incoming value in the packet that triggers the execution. The result of this reduction is then set to be the new value of the aggregated data, which is used as the input for the next packet to arrive. The initial value is set to zero by default, but this can be changed. An example is shown in listing 4.1.

```
#include "bpf_h/helpers.bpf.h"  // part of sgbpf

SEC("xdp")
int aggregation_prog(struct xdp_md* xdp_ctx) {
    struct aggregation_prog_ctx ctx;
    AGGREGATION_PROG_INTRO(ctx, xdp_ctx);

    AGGREGATION_PROG_ACQUIRE_LOCK(ctx);
    // Perform an addition-based reduction (RESP_VECTOR_TYPE is defined as __u32)
    for (__u32 i = 0; i < RESP_MAX_VECTOR_SIZE; ++i) {
        ctx.current_value[i] += ((RESP_VECTOR_TYPE*) ctx.pk_msg->body)[i];
    }
    AGGREGATION_PROG_RELEASE_LOCK(ctx);

    AGGREGATION_PROG_OUTRO(ctx, DISCARD_PK); // enable early dropping
}
```

Listing 4.1: a user-defined aggregation program performing vector addition.

The aggregation program supplied must be an XDP program, as it is called from `gather_prog` which is also attached to the XDP hook. This change in the control-flow takes place through an eBPF tail call, meaning that the packet payload has to be re-parsed from the provided XDP context and checked for any out-of-bounds pointers. This consists of boilerplate code, so it is abstracted away using the `AGGREGATION_PROG_INTRO` helper macro. This also prepares a `struct aggregation_prog_ctx` by fetching the pointer to the current aggregated value stored in a map. It also contains a spinlock used to protect the updates from concurrent access, and it must be acquired using `AGGREGATION_PROG_ACQUIRE_LOCK` before any updates to the `current_value` field. The reason why this is not done together in the same macro is to allow room for function calls, which are not allowed once the spinlock is acquired.
Once the aggregation logic is executed, the spinlock is released and the `AGGREGATION_PROG_OUTRO` macro is used to perform any post-aggregation logic, specifically, incrementing the count of aggregated responses. It also gives the opportunity to specify the fate of the packet, i.e. whether the packet should be dropped in the case where it is no longer needed in user-space (this is referred to as early dropping and is discussed below).
The program is written in its own standalone C source file and is compiled against the header files provided by *sgbpf*. This produces a BPF object file which is loaded into the kernel at runtime during initialisation using the `sgbpf::Context` class.

An alternative design that was implemented and evaluated was to allow the user to write the aggregation logic as a regular C function, rather an eBPF program. The initial reasoning for this was that it could be called directly from `gather_prog` without requiring the packet to be re-parsed. However, the packet parsing overhead was measured to be approximately 15 ns; a negligible amount of latency compared to the full execution of aggregation logic, which is in the order of microseconds. In addition, kernel version 5.5 introduces an optimisation for tail calls, reducing their overhead to under 10 ns [55]. We preferred keeping this negligible amount of extra overhead in order to provide a uniform API which avoids conditional compilation and more complex build processes. While performance is the focus of the project, it is still important to consider library design and usability.

Limitations

Now that the API is clear, it is important to specify the limitations of in-kernel aggregation using this API. The main source of restrictions is the eBPF static verifier, which imposes rules to guarantee safety and liveness properties. In practice, for the API presented above, developers need to be aware of the following points:

- Function calls (including to BPF helpers) are not allowed while a spinlock is held, which is the case for any code between the acquire and release helper macros shown in listing 4.1. However, developers are allowed to define their own helper functions and forcefully inline them using compiler attributes such as `__always_inline` to bypass this restriction.

- The loop bounds must be specified at compile-time to enable loop unrolling. Therefore, the vector size must be statically defined and cannot be computed at runtime. Developers may use the `RESP_MAX_VECTOR_SIZE` macro which represents the maximum number of elements of type `RESP_VECTOR_TYPE` that fit in the body of a packet.

- Developers may add their own eBPF maps to maintain extra state and write more complex aggregation logic. The only requirement is that the map must be accessed using `bpf_map_{lookup,update}_elem()` outside of the spinlock-protected region.

- Floating point data types are not allowed. This is not an eBPF restriction, but a kernel restriction, as the Linux kernel does support floating point operations. The only solution would be implement a floating point standard from scratch. Otherwise, all numeric data is restricted to integral types.

With these limitations in mind, it is important that potential users of *sgbpf* are able to feasibly express the aggregation logic in an eBPF program. Otherwise, the use of *sgbpf* becomes sub-optimal, as its main advantage comes from the performance boost provided by in-kernel aggregation.

Early dropping

Once the aggregation logic takes place, there is a post-processing function to increment the response counter which is called from the `AGGREGATION_PROG_OUTRO` macro. The second parameter to this macro accepts a flag which determines the fate of the packet once it has been aggregated. It can be one of two values:

- `ALLOW_PK`: the packet is allowed through to user-space and is accessible by the application.
- `DISCARD_PK`: the packet is dropped immediately after aggregation.

By specifying the `DISCARD_PK` flag, early dropping is enabled and all packets except the last one are dropped. The last packet is needed to trigger the execution of `notify_prog` at the TC layer, which overwrites the body of the packet with the final aggregated value and redirects it into the control socket. If `ALLOW_PK` is specified instead, `notify_prog` will clone the final packet and redirect the generated `skb` into the control socket. The corresponding flag must also be specified in the user-space application, passed in as a parameter to the constructor of the `sgbpf::Service` class.

As mentioned, the optimal usage of *sgbpf* arises whenever the aggregation logic executes inside the kernel. In these cases, there is often no need to access the individual worker responses in the application, as it will receive the final aggregated value. Therefore, it is recommended that `DISCARD_PK` is specified. By enabling early dropping, the total latency of each scatter-gather operation decreases, as it avoids processing inside the kernel. In fact, the `skb` allocation for each packet (except the last one) is completely avoided, as it is dropped in the XDP hook.
Another important advantage of enabling early dropping is that the issue regarding the slow request that arises as a result of `io_uring`'s buffer provision mechanism is completely avoided, as the individual packets are never written into these buffers.

4.4.3 Completion policies

By default, a scatter-gather operation is completed whenever all the workers respond and the coordinator node gathers all the messages. However, it may be the case that it is sufficient to receive less responses than the number of workers in the cluster. Our library allows the user to specify the completion policy as an optional parameter passed into the `scatter()` method. From the user-space API, the user can choose one of `WaitAll` (default), `WaitN` or `WaitAny`. Within the eBPF programs, all three completion policies follow the same logic, as they can all be generified to `WaitN`, where n is either the total number of workers for `WaitAll`, an explicitly specified integer for `WaitN` or one for `WaitAny`.

As each request can be configured differently, the number of workers to wait for needs to be communicated to the eBPF programs on every invocation of `scatter()`, i.e: the optimised path. The simplest approach would be to simply update the relevant eBPF map with this value, however this requires an extra system call. To avoid the cost of an extra system call, *sgbpf* communicates this value to eBPF by including it in an unused field in the header of the message which is written into the scatter socket. Then, in `scatter_prog` this value is extracted from the message and used to update the relevant map. This is a similar strategy to the technique described in 4.3.1 and avoids extra system calls on the optimised path.

4.4.4 Timeouts as a recovery mechanism

As a protocol built on top of UDP, it is important to include some recovery mechanism that allows users of *sgbpf* to detect whether a scatter-gather operation has failed due to packet loss. Since performance is the focus of the project, we have opted for implementing a simple coarse-grained timeout mechanism that allows users to verify whether an in-flight request has timed out.

The timeout quantity is specified as an optional request parameter and measured in microseconds. Once the `sgbpf::Request` object is returned from the `scatter()` method, the application can use the `hasExpired()` method to check the status of the operation. An example of this API is shown in listing 4.2. Note that this is a feature which is only implemented for the user-space application; the kernel is unaware of timeouts. Therefore, if a packet arrives after the timeout has passed, it

will still be processed by the eBPF programs accordingly. In the worst case, this will result in useless processing and wasted compute time. However, assuming timeouts are rare, this approach avoids the overhead of fetching the current timestamp and checking for timeouts in every eBPF program on the ingress path, which is considered an optimised path of the system.

```
// specify the completion policy and timeout for the request
sgbpf::ReqParams params = {
    .completionPolicy = sgbpf::GatherCompletionPolicy::WaitN,
    .numWorkersToWait = 5,
    .timeout          = std::chrono::microseconds{500}
};

sgbpf::Request* req = service.scatter(data, data_len, params);
// ...
service.processEvents();

if (req->hasExpired()) {
    // handle expired request, eg: send a new one
}
```

Listing 4.2: an example showing how to specify request parameters and check for timeouts.

An important requirement is that the `processEvents()` method is called before checking for the status of a request. This is because all the event handling logic to confirm the completion of the I/O operations submitted to `io_uring` takes place in this routine, and updates the state of the relevant `sgbpf::Request` object. More details on this function are given in the next section.

4.4.5 Reading the aggregated data

From the perspective of library design, usability and flexibility, one of *sgbpf*'s main advantage is its configurability and multiple modes of operation to read the aggregated result. It offers various APIs to deliver the data aggregated in the kernel to the application, mainly using the control socket design introduced at the start of this chapter. However, an alternative communication channel is also available via a ring buffer which bypasses the networking stack. This latter approach is described in more depth later.

The data delivery mechanism is specified in the constructor of `sgbpf::Service` and can be one of four values from the `sgbpf::CtrlSockMode` enumeration:

- **`Ringbuf`**: bypasses the networking stack (and the control socket) using a memory-mapped ring buffer and `epoll` for readiness notifications (more details later in this section).
- **`DefaultUnix`**: exposes the control socket as a raw Unix file descriptor, allowing the user to read from the socket using any I/O API they prefer. This includes configuring the control socket in advance, such as setting it as a non-blocking file descriptor. This is by far the most flexible API, however it also delegates more work to the application.
- **`Block`**: takes advantage of the internal `io_uring` instance to "piggyback" a read operation on the control socket onto the submission queue when invoking the `scatter()` method. The main difference here is that the submission of the I/O operations to `io_uring` takes place using a blocking system call, meaning that the `scatter()` method will block until the entire scatter-gather operation completes. Once execution resumes, the aggregated data is available in a buffer and accessible using the `ctrlSockData()` method of the request. This is the simplest API to use, but is only suitable for applications where scatter-gather operations should be completed sequentially.
- **`BusyWait`**: similar to `Block`, but the `scatter()` method is asynchronous and completion is inferred by busy waiting using the `isReady()` and `processEvents()` methods together. The read operation on the control socket is added to the internal `io_uring` instance like with `Block`, but the completion queue is polled from user-space checking for the completion events of the relevant request. This API is easy to use and offers the benefits of busy waiting techniques (specifically, no context switching), but incurs the highest CPU utilisation and potentially wastes compute time.

The following code listings contain examples of how the different control socket APIs can be used to consume the aggregated data in the application.

```
sgbpf::Service service {
    /* ... */,
    sgbpf::CtrlSockMode::DefaultUnix
};

sgbpf::Request* req = service.scatter(data, data_len);

// perform the read ourself using read()
sg_msg_t buf;
read(service.ctrlSkFd(), &buf, sizeof(sg_msg_t)); // blocks until result is available
service.processEvents();

// do something with the aggregated data
handle(buf.body);
```

Listing 4.3: a demonstration of the scatter-gather API when using the `DefaultUnix` approach.

```
sgbpf::Service service {
    /* ... */,
    sgbpf::CtrlSockMode::Block
};

sgbpf::Request* req = service.scatter(data, data_len); // blocks until completion
service.processEvents();
assert(req->isReady());

// do something with the aggregated data
handle(req->ctrlSockData()->body);
```

Listing 4.4: a demonstration of the scatter-gather API when using the `Block` approach.

```
sgbpf::Service service {
    /* ... */,
    sgbpf::CtrlSockMode::Block
};

sgbpf::Request* req = service.scatter(data, data_len);

while (!req->isReady(true)) // busy-wait loop
    service.processEvents();

// do something with the aggregated data
handle(req->ctrlSockData()->body);
```

Listing 4.5: a demonstration of the scatter-gather API when using the `BusyWait` approach.

The eBPF program in charge of delivering the final aggregated result to the control socket is `notify_prog`. Its execution is triggered whenever the final response is received, immediately after the aggregation program at the XDP layer (unless `ALLOW_PK` is specified, in which case all packets enter `notify_prog`). In addition, if `ALLOW_PK` is set, a call to `bpf_clone_redirect()` is required to generate the packet that is written into the control socket on the final response. Otherwise, the final packet is overwritten with the aggregated value without needing to clone the response, as it is not required by the application. The map entry containing the aggregated data is then cleared.

Event handling in user-space

In each of the code listings shown previously, the `processEvents()` method is invoked. This method is required to process the completion events posted on the `io_uring` completion queue and avoid the CQ being filled. Therefore, if the control socket is read using `io_uring` (which is the case for the `Block` and `BusyWait` modes), this method must be called to determine whether the control socket has been read and its corresponding data is ready to be consumed. It is the user's responsibility to call this method to ensure that the request state is updated correctly, whether it is in the same thread or offloaded to a background thread.

This method is also responsible for checking that all messages have been received into the provided buffers whenever the packets are allowed through to user-space. In this case, the pointer to the

packet buffer is saved, allowing the application to iterate over all the packets received without any extra copies. Each packet pointer is determined by the buffer group ID and the buffer ID, which are used to index into the global shared buffer pool which keeps all the received packets. Users can read the packet using the `data()` method on the `sgbpf::Request` object, passing in the packet pointer.

Bypassing the kernel using eBPF's ring buffer

An alternative communication channel between the kernel and the user-space application which avoids the control socket is implemented using the `BPF_MAP_TYPE_RINGBUF` map. This is a multiple producer single consumer ring buffer that can be written to by eBPF programs and read by the user-space application [56]. It is implemented as a memory-mapped file and is exposed to user-space by configuring its file descriptor with `epoll`, allowing it to be polled continuously for memory writes. In addition, a callback function can be registered such that it is called whenever data is published onto the buffer.

The main benefit of this approach is that it offers fast path within the kernel that bypasses the kernel networking stack and the socket layer. Rather than allowing the final packet to proceed into the upper layers of the kernel, it can be directly enqueued into the ring buffer, as shown in figure 4.5. In addition, for cases where early dropping is enabled, the data can be written into the buffer from the XDP layer and entirely bypass `notify_prog` at the TC layer (which avoids the `skb` allocation). This is because the final packet does not have to be cloned to the application and thus removes the dependency on the `bpf_clone_redirect()` function which is only available at the TC layer. This optimisation only makes sense for the ring buffer approach, because if data is delivered through the control socket, an `skb` must be allocated regardless and performance benefits are negligible. The all-gather mode (introduced next) must also be disabled for this shortcut to trigger, as it relies on packet cloning. Otherwise, it falls back to writing the data into the ring buffer in the TC layer.

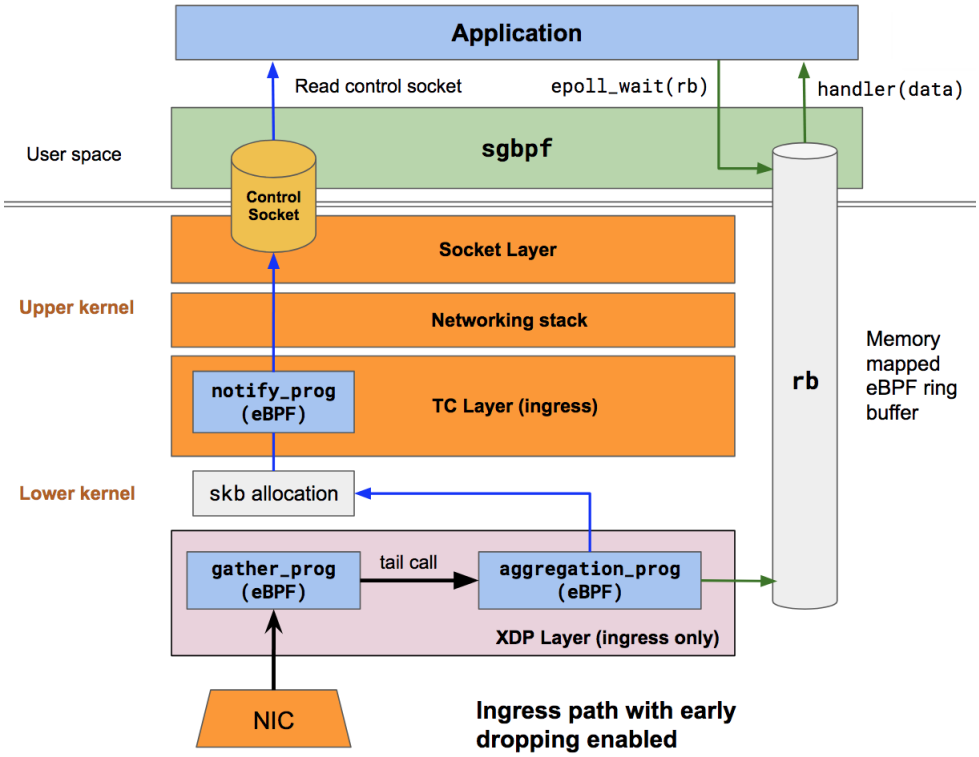

Figure 4.5: With the ring buffer mode enabled (and early dropping), the aggregated value follows the green arrow, and avoids the kernel networking stack (and `skb` allocation) because the control socket is not used (shown by the blue arrow).

From the developer's perspective, this mechanism also enhances the *sgbpf* API by providing an alternative event-driven approach using callback functions. This is a common programming style for implementing event loops and it also hides away all network I/O operations.

```
sgbpf::Service service {
    ctx,
    workers,
    sgbpf::PacketAction::Discard,   // early dropping enabled
    sgbpf::CtrlSockMode::Ringbuf    // use the ring buffer for data delivery
};

// register a handler which triggers whenever the aggregated data is ready
service.setRingbufCallback([](void* data, int reqID) -> void {
    // ...
});

// epoll the ring buffer inside the event loop
while (1) {
    // ...
    int completions = service.epollRingbuf(timeout);
    if (completions > 0) {
        service.processEvents();
    }
    // ...
}
```

Listing 4.6: a demonstration of the ring buffer API in user-space to consume aggregated data.

All-gather broadcasting for data aggregated in the kernel

As an extension to the original API and the expected functionality, *sgbpf* implements an optional feature which performs a final broadcast once the data has been aggregated inside the kernel. This final step is typically referred to as "all-gather" (sometimes refered to as "all-reduce" [54]), whereby all the workers participating in the communication receive a copy of the final aggregated data, rather than just the coordinator node.

This feature was easily implemented by re-using the existing logic for the scatter phase: once `notify_prog` is ready to deliver the aggregated data to the appliation, it will also iterate over the workers map and redirect a cloned `skb` to each worker (as described in section 4.3.1), including a copy of the data in the message body. Just like `scatter_prog`, this takes place at the TC layer and therefore the cloned packet does not have to traverse the network stack; it is directed immediately to the network interface.
This extension was implemented at the end of the project, adding a total of 14 lines of eBPF code, and proves just how easy it is to extend the original scatter-gather primitive.

4.5 Ethical and legal considerations

Licensing

As mentioned, eBPF is an open-source technology that is included in the Linux kernel and is licensed under the GNU General Public License (GPL) version 2. The GPL is a widely used open-source license that allows for the free use, distribution, and modification of the software, as long as any derivative works are also released under the same license.

The use of eBPF in the Linux kernel, however, raises some copyright and licensing issues. According to the Linux Kernel documentation, eBPF programs can be considered as separate works, and their authors retain the copyright to the code. This means that users of eBPF programs must comply with the GPL license and any other licenses that apply to the specific eBPF programs [57]. Additionally, some eBPF programs are available under different licenses, such as the Apache License, BSD License, and MIT License. This means that users must comply with the terms of the specific license of the eBPF program they are using.

All the code written as part of this project is to be made public, open-source and GPL compatible. Thus, there should be no licensing or copyright issues.

Security and malicious misuse

An important ethical consideration is security. While eBPF can be used to implement security measures, such as firewalls and intrusion detection systems, it can also be used to implement malicious attacks, such as denial of service or man-in-the-middle attacks. It is important to ensure that eBPF is used only for legitimate and authorised purposes, and that proper security measures are in place to prevent malicious use.

This project focuses on performance acceleration of a distributed system. This involves inspecting all incoming packets, which may contain sensitive information. As such, it is important to avoid reading application layer data if the packet is not intended for a certain application. In addition, because of eBPF's safety guarantees and the static verifier, it is extremely unlikely that implementation bugs will cause a denial of service (for example, crashing the kernel).

Data protection

Another legal consideration is compliance with data protection laws and regulations. Network processing technologies such as eBPF can be used to collect, process and store large amounts of data, which can include personal information. Thus, policies and procedures should be in place to protect personal data and ensure compliance with relevant laws and regulations, such as the GDPR.

As part of this project, there is no intention to use any real data since the goal is to explore performance acceleration, done by using generated data.

Chapter 5

Evaluation

As a performance-oriented project, the evaluation is essential as it determines the success of the implementation described in the previous chapter. The results at hand are mostly non-functional performance-related figures, with a lesser focus on functional testing.

The evaluation will consist of comparing the performance of *sgbpf* and its different configurations against the standard Linux-native I/O APIs (as mentioned in the requirements in section 4.1). These experiments will rely on benchmarks using a dummy workload with generated data and simple worker nodes.

Evaluation environment

An important point to make is that all benchmarks used in the evaluation have been executed locally on a single machine (i.e: there is no communication beyond the network interface card). Each worker node is implemented as its own process with a unique port number bound to a socket. There are two reasons for this: firstly, it simplifies the experiments and removes all the setup overhead involved in preparing a cluster of machines. For large fan-outs, this would become difficult and potentially infeasible to manage. The second reason is that it removes any network jitter which could distort results. Of course, local performance testing will be affected by a number of factors that could add jitter, such as scheduling decisions inside the OS, but these are more controllable. Also, as a pure software project, we would prefer to avoid any dependencies on particular hardware devices and narrow the scope of what is actually being measured.

The workload for the benchmarks is based on dummy data which is generated on the go. We are not interested on the type of data being sent and gathered. Consequently, we have chosen to use a payload consisting of a vector of four-byte integers. Since the number of elements in the vector depends on the MTU size for the interface, we have fixed the MTU to 1500 bytes which is a realistic value for most systems. This results in a vector of 363 integers. The gather operation is selected to be vector addition, which is a suitable reduction operator for in-kernel aggregation (as explained in section 4.4.2).

To run these benchmarks, we have also implemented a very simple worker program in C which uses standard blocking I/O APIs to read and send data into the socket, with `recvfrom()` and `sendto()` respectively. When a scatter message is received from the coordinator process, the worker prepares a response message, including a vector of integers with values counting from 0 up to the maximum size. This is shown in algorithm 1.

Algorithm 1 Worker algorithm using standard blocking I/O

```
1: loop
2:     scatter_msg ← RECVFROM(socket, coordinator)
3:     resp_msg ← {...}
4:     resp_msg.req_id ← scatter_msg.req_id
5:     for i ← 0 to MAX_SIZE do
6:         resp_msg.body[i] ← i
7:     end for
8:     SENDTO(socket, coordinator, resp_msg)
9: end loop
```

In addition, each worker process is pinned to a different core to avoid measuring idle time and the context switching overhead incurred by scheduling processes between cores. However, with fan-out sizes greater than the number of cores available, cores will be assigned with multiple worker nodes. This occurs in a round-robin fashion to balance the load equally across all available cores. The coordinator node is always pinned to a single exclusive core.

All the benchmark results presented in this chapter have been obtained using a high-performance server with the following specifications:

- **Processor**: Intel Xeon Gold 5318N @ 2.10GHz
- **Cores**: 48 logical cores (24 physical cores with 2 threads each)
- **Memory**: 128 GB
- **Kernel**: Linux 5.15.0 (x86-64 architecture)
- **XDP mode**: Native

Metrics of interest

The first metric we are interested is the average latency of each scatter-gather operation. To be specific, the latency results presented in this chapter refer to the average **unloaded latency** of an operation. This means that there is only a single active operation in-flight and the next operation is only launched once the previous one has completed. This is measured over a large number of operations (5000 requests) and the average latency is computed at the end.

The second metric of interest is **throughput under load**. This is the rate at which the coordinator node can process scatter-gather operations from start until completion, measured in operations per second. The key difference here is that the benchmark code will load the environment with a number of initial outstanding operations. Then, inside the main measurement loop, the code waits for the completion of at least one scatter-gather operation and then launches a new operation for every completion. By doing so, the environment is always loaded with more than one in-flight operation. The throughput is calculated at a sample rate of 200 completed operations over a total of 8000 requests. This gives a total of 40 values which are used to calculate the average throughput under load.

A final metric which may be useful for certain comparisons is **CPU utilisation**. This is defined as the amount of time spent on computation (CPU time) divided by the elapsed wall-clock time. This metric may can be interesting to evaluate the behaviour of busy-waiting implementations with blocking implementations of the *sgbpf* API.

5.1 Performance evaluation of *sgbpf* against standard I/O APIs

One of the fundamental objectives of *sgbpf* is to minimise the user-kernel boundary crossings that are incurred with the standard I/O interfaces provided by the Linux kernel. Our evaluation benchmarks three different non-eBPF approaches to performing scatter-gather operations, and compares their performance with different configurations of *sgbpf* to achieve the same result. Importantly, the three standard baselines perform the aggregation of the responses in the application. On the other hand, for the *sgbpf* benchmarks, we will assume the optimal use case, i.e: the data is aggregated within the kernel via the user-supplied aggregation program. Given that all benchmarks are implemented using UDP, we will also assume no packet loss and therefore, the timeout recovery mechanism is not triggered.

Naive baseline

The naive implementation uses blocking I/O operations to write and read data from the sockets. When the scatter is performed, the `sendto()` system call is invoked for every worker to write the scatter message into each worker's socket. Similarly, when the gather is performed, the `recv()` system call is invoked on each worker's socket.
Assuming a fan-out size of n workers, this implementation requires n system calls in the scatter phase and another n system calls in the gather phase. Additionally, both the `sendto()` and `recv()` system calls perform memory copies to transfer the buffer from user- to kernel-space and vice-versa.

`epoll` baseline

This implementation is based on the `epoll` interface which follows a readiness-oriented model. The main difference with the naive version lies in the gather phase: instead of blindly invoking `recv()` which may block, the code first invokes the `epoll_wait()` system call which returns the number of sockets which are ready to be read from. Only once the sockets are ready, we perform the `recv()` system call which is guaranteed to not block. Despite this extra system call, this mechanism becomes more efficient when performing I/O over a large number of file descriptors, as it avoids idle time which arises due to context switches from blocking system calls.
The scatter phase is unchanged and also incurs n system calls for a fan-out of n workers. The gather phase however requires $n + 1$ system calls in the best case, which occurs when all sockets become ready before `epoll_wait()` returns. Otherwise, the worst case incurs up to $2n$ system calls. This may happen if `epoll_wait()` always returns one ready socket, requiring an `epoll_wait()` for every `recv()` call. In practice, the worst case is unlikely. Regardless, this baseline still requires $O(n)$ system calls.

`io_uring` baseline

The third non-eBPF baseline uses the `io_uring` interface to perform I/O. This baseline is similar to *sgbpf* from the user-space perspective, as it batches all the `sendmsg()` and `recv()` operations together and submits them with a single system call in the scatter phase. This blocks until all operations have completed. Then, in the gather phase, it iterates over all the received packets stored in provided buffers without any blocking or further system calls.
It is clear to see why this interface is considered the state-of-the-art at the time of writing: for an entire scatter-gather operation, there is only a single system call required. In addition, there are no memory copies for the incoming packets, since buffers are provided in advance just like in *sgbpf* (described in section 4.3.2).

However, one point to reiterate which is already discussed in detail in section 4.3.2 is the scalability of the buffer provision approach. When using the provided buffers feature in `io_uring`, each provision step incurs a large setup overhead to register the buffers with the kernel. For a fair comparison with *sgbpf*, this baseline provides the maximum number of buffers allowed which is 2^{16}. This means that every 2^{16} read operations, the buffers need to be replenished, causing a slow operation. The consequence of this design is that for large fan-outs, the frequency of suffering a slow request increases, since the buffers are more quickly exhausted (note that this only ocurrs because the packets must be read in user-space; our *sgbpf* baselines assume in-kernel aggregation

and therefore do not suffer from this problem). This is an important point to consider when understanding the performance results of this baseline.

sgbpf baselines

To evaluate *sgbpf* against the three standard baselines described previously, we must consider multiple configurations of the library. In particular, we are interested in evaluating the different data delivery mechanisms and their performance. Table 5.1 shows the different benchmarks we have implemented for *sgbpf*.

Benchmark Name	Aggregation	Packet Action	Data Delivery API
sgbpf - unix fd	eBPF	Discard	Control socket + `read()`
sgbpf - block	eBPF	Discard	Control socket (via `io_uring`)
sgbpf - busy wait	eBPF	Discard	Control socket (via `io_uring`)
sgbpf - ringbuf	eBPF	Discard	Ring buffer (via `epoll`)
sgbpf - user aggregation	User-space	Allow	N/A

Table 5.1: The different *sgbpf* benchmarks implemented for the evaluation.

In addition, the all-gather extension described in 4.4.5 is disabled. It is worth pointing out that the "*sgbpf* - user aggregation" benchmark is not the optimal use case of *sgbpf* as it performs aggregation in the application, and simply uses *sgbpf* as a scatter-gather API. However, it is still interesting to evaluate its performance too and see how it compares to the standard APIs.

Time spent on system calls

As an initial experiment, we investigate the time spent in system calls for each benchmark as an initial estimation of the latency profile for each implementation.

The results in figure 5.1 show the average time spent in system calls per operation, and were obtained using the `strace` tool for a fan-out of 200 workers, averaged across 2000 scatter-gather operations. An important clarification to make is that the `strace` tool adds a significant amount of overhead, so the real figures are much lower (as shown in the latency results in section 5.1.1). This graph is simply to confirm the system call overhead for each approach. As expected, the `io_uring`-based approaches (including *sgbpf*) incur the lowest time spent on system calls.

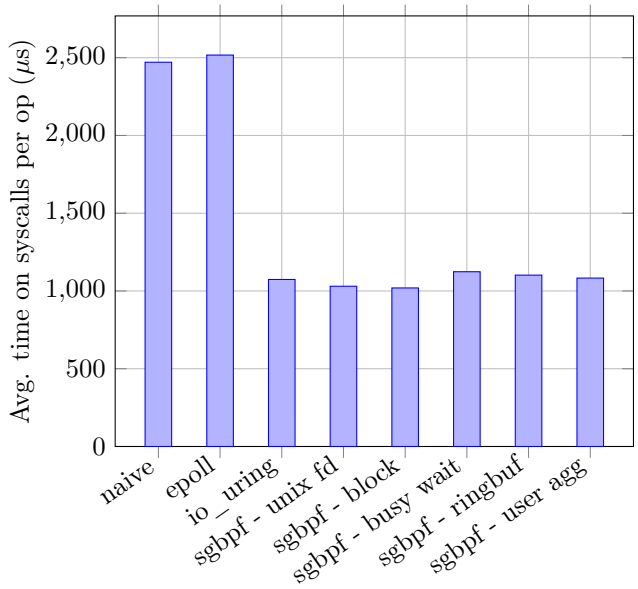

Figure 5.1: The average time spent on performing system calls per operation for a fan-out of 200 workers.

5.1.1 Latency

The first metric we present is the unloaded latency of a single scatter-gather operation measured in microseconds. Figure 5.2 shows the obtained results for the different implementations, including various selected *sgbpf* configurations. As always with latency, lower is better.

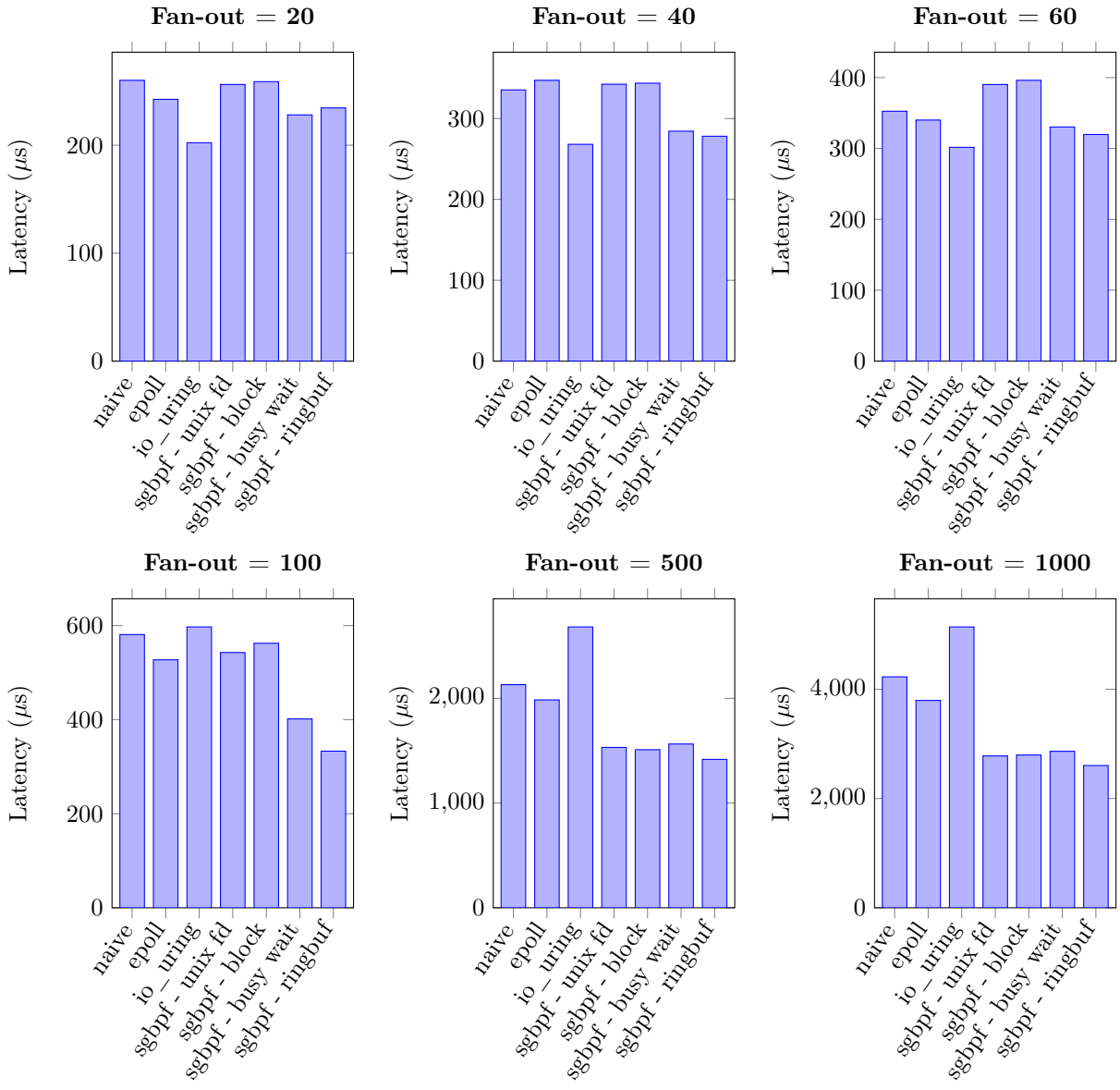

Figure 5.2: A comparison of the average unloaded latency in μs of a scatter-gather operation for different fan-out sizes.

From these results, we can observe that for small fan-outs of up to 60 workers, all implementations yield similar latency results in the same order of magnitude, with the `io_uring` version narrowly outperforming the rest. While the *sgbpf* implementations are not far off, especially the ring buffer version, these results are somewhat expected as *sgbpf* has the overhead of executing library code under the hood, whereas the standard implementations do not. At this point, the total cost of system calls and context switches is still low enough to narrowly outperform *sgbpf*.

However, once the fan-out starts to increase into the hundreds of workers, *sgbpf* begin to shine, taking advantage of minimal context switches and less processing in the kernel networking stack. At 1000 workers, the performance advantage of *sgbpf* is very clear compared to the non-eBPF baselines which start to suffer.

Some of the results from the non-eBPF benchmarks may seem surprising, in particular, the `epoll` baseline. While `epoll` avoids blocking calls on the sockets, the latency difference with the naive blocking baseline is very small. The likely reason for this is the fact that these benchmarks are running locally on the same machine. This means that blocking on sockets is rare or the wait time is almost negligible. As a consequence, the benefits of the `epoll` implementation are very limited. However, on a real network, we would expect the naive version to perform much worse (as blocking becomes more likely, leading to more context switches) and notice the benefits of using `epoll`.

While `io_uring` provides the lowest latency for small fan-outs, it starts facing the scalability issue described previously, incurring a slow operation very frequently. Depending on the application, alternative buffer management techniques would probably yield better performance, but we have opted to use the same approach used in *sgbpf* to ensure a fair comparison.

An alternative presentation of the same data but with more fan-out sizes is shown in figure 5.3. This is a better visualisation to understand how the unloaded latency of each version scales with the number of workers participating in the operation.

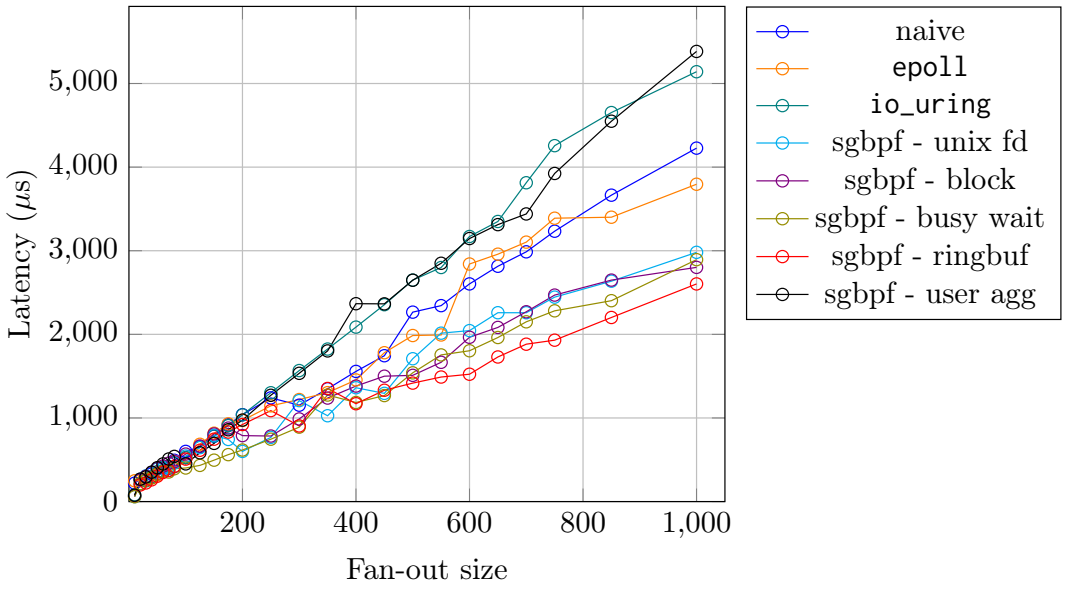

Figure 5.3: A comparison of how each implementation scales as the number of workers grows.

We can observe how there is a clear and steady increase in the latency per request as the number of workers increases. The growth is linear, which is the expected behaviour for all benchmarks. For larger fan-outs, we can appreciate a much larger difference in latency between the *sgbpf* and non-eBPF implementations. At 1000 workers, the fastest *sgbpf* implementation (using the ring buffer) is over 100% faster than worst performing baseline and approximately 62% faster than the naive implementation. We can conclude that that *sgbpf* scales better and is more suitable for large fan-out sizes.

We can also see how as the fan-out increases, three different range groups emerge. The worst performing group consists of `io_uring` and "sgbpf - user agg", which suffer from a slow request every 2^{16} reads (at large fan-outs, this request happens more often). Also, as expected, the "sgbpf - user agg" benchmark is the slowest because it does not reap the benefits of in-kernel aggregation, which is clearly the main advantage of the other *sgbpf* implementations which do perform aggregation inside the kernel. Instead, its semantics are very similar to plain `io_uring` version but with extra overhead incurred by the library code in user-space and the meaningless eBPF code in kernel-space.

The second group consists of the naive and `epoll` baselines, which are still slower than the *sgbpf* implementations with in-kernel aggregation. It can also be seen that the `epoll` baseline becomes faster than the naive version at very large fan-outs, which is also expected as the benefits of `epoll` become noticeable.

The third range of latencies comes from the *sgbpf* implementations with in-kernel aggregation and early dropping. As the fan-out size increases beyond approximately 500 workers, *sgbpf*'s ring buffer mode consistently provides the lowest latency among the different implementations. From our measurements, this approach also offers the most predictable latency figures as indicated by the standard deviation. The key factor that enables this lower latency is the fact that the kernel networking stack is entirely bypassed in the gather phase. As all packets are dropped at the XDP layer after aggregation, this avoids the allocation and processing that would take place for the final packet sent to the control socket with the alternative APIs.

However, the performance difference is not that significant compared to the other *sgbpf* versions. The main reason for this is that the processing logic for UDP traffic is straightforward and avoids the complex state management required by TCP. Despite this, it still seems to be the fastest approach because of its kernel bypass design.

5.1.2 Throughput under load

The next performance metric we evaluate is the throughput of the coordinator node under load. The crucial difference here compared to the latency experiments presented previously is that the system is evaluated under load, meaning that there is always more than one operation in-flight. This is achieved by first loading the system with a fixed number of initial requests, and then for every completed request, a new operation is launched. The throughput is calculated at a rate of 200 completed operations. The following graphs presented in figures 5.4, 5.5 and 5.6 show the average throughput calculated over 8000 operations at small, medium and large fan-out sizes.

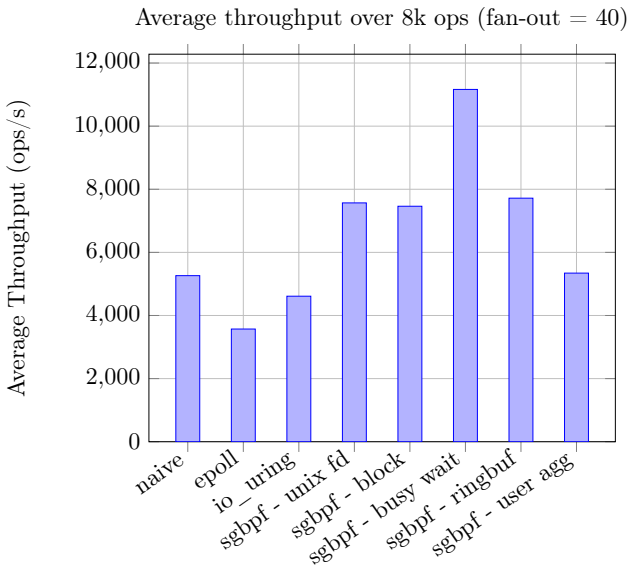

Figure 5.4: The average loaded throughput of the different baselines, for a small number of workers.

Even for a small cluster size of 40 worker nodes, we can already appreciate the improved performance of the *sgbpf* implementations. The highest throughput version is "sgbpf - busy wait". This result makes sense, as it avoids making any system calls on the gather phase. For a small number of workers, the time wasted due to context switches is relatively big compared to the real time taken to complete the operation. Hence, busy waiting is the most efficient approach for a small fan-out. The remaining three eBPF-enabled versions which perform in-kernel aggregation follow the busy waiting approach, providing a similar average throughput of around 7500 ops/s, comfortably beating the standard baselines which perform aggregation in user-space by at least 41.5%.

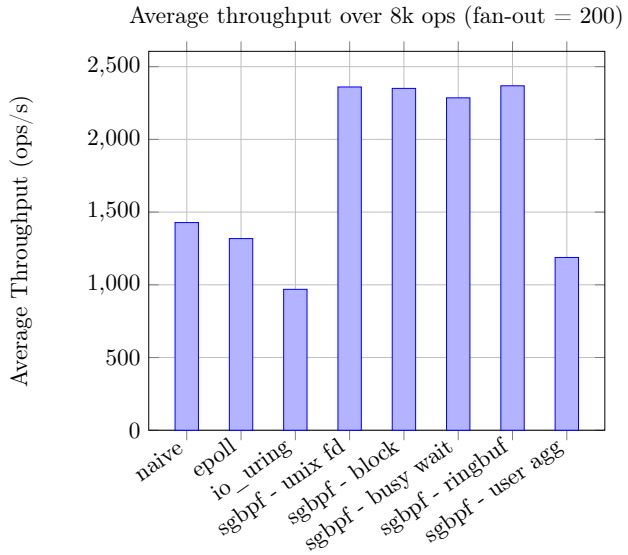

Figure 5.5: The average loaded throughput of the different baselines, for a moderately-sized number of workers.

If the fan-out is increased to a moderately-sized cluster of workers (in this case 200 nodes), just like we saw with the unloaded latency benchmark, the difference between the *sgbpf* implementations and the non-eBPF versions begins to increase. In this case, all implementations with in-kernel aggregation perform around 66% faster than the naive implementation.
It can also be observed that the *sgbpf* APIs relying on system calls are now achieving throughputs as good as the busy waiting approach. With a larger fan-out, the portion of time spent waiting in system calls due to context switches is much lower, so the performance advantage of busy waiting is minimised.

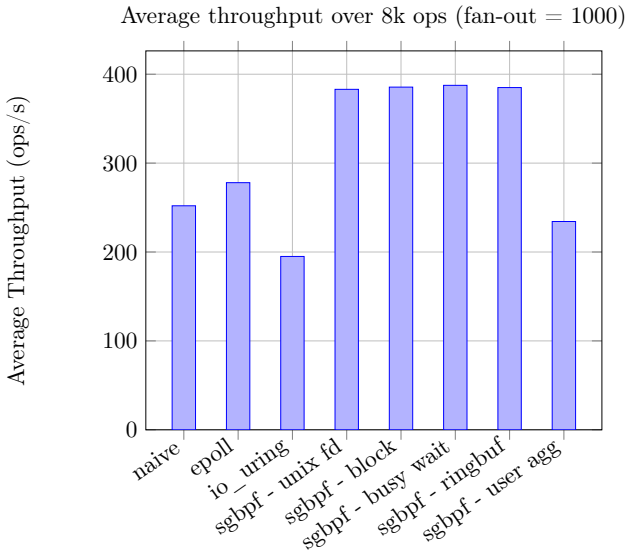

Figure 5.6: The average loaded throughput of the different baselines, for a large number of workers.

For large fan-outs (≈1000 workers), the relative throughput difference between the *sgbpf* implementations and the standard baselines decreases slightly to around 42%. At such large fan-outs, factors such as spinlock contention on the aggregated data and scheduling decisions become more prevalent and reduce the relative performance advantage of the *sgbpf* implementations. Also, while the ring buffer approach bypasses the kernel networking stack and achieves the lowest unloaded latency, the internal implementation of the ring buffer relies on a spinlock to serialise writes, meaning that for large fan-outs and many active requests, the contention will increase.

Despite this, the benchmarks with in-kernel aggregation once again achieve the highest throughput under load. Like in the latency experiment, the `io_uring` and "sgbpf - user aggregation" versions perform the worst due to the slow request being incurred every 66 operations, as explained in 4.3.2.

5.1.3 Utilisation

The final metric we evaluate is CPU utilisation. While this isn't an essential metric when it comes to measuring "speed", it can be useful for applications running in data centres where energy consumption and tail latency are important. Generally, CPU utilisation in data centre application should be high enough to take full advantage of the resources available, but should not be too high to avoid unexpected background processes interrupting the main application, which can severely affect the response tail latency [58].

In this section, we measure utilisation by computing both the real elapsed time and the CPU time, and calculate their ratio. We perform this measurement using the unloaded latency benchmark rather than the throughput benchmark in order to get a more accurate estimation of the utilisation incurred by a single request. Figure 5.7 shows the results collected for the different benchmarks for a fan-out of 200 workers averaged over 5000 individual operations.

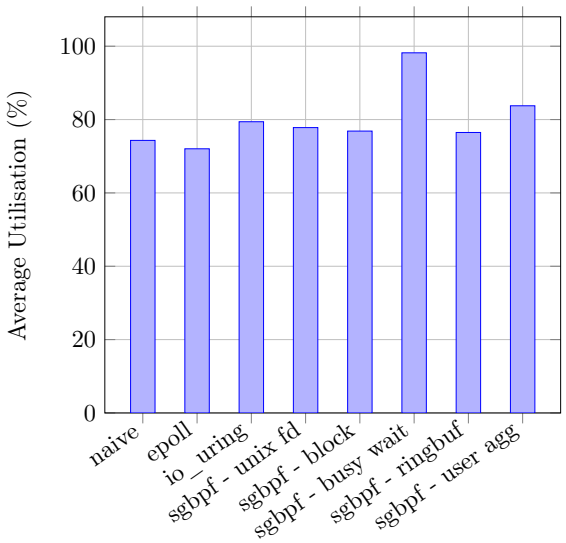

Figure 5.7: The average CPU utilisation for each benchmark (includes both user and system CPU time).

The busy waiting implementation has the highest utilisation, nearing 100%, which is expected as it spins in user-space waiting for completion. The other benchmarks have all very similar utilisation results, as they all require system calls (and likely context switches) to wait for the completion of an operation.

It is important to reiterate the effects of local testing: with the absence of real network communication, the wait time for I/O is much lower than in a real network, so the likelihood of context switching is lower in our setup. This results in a higher CPU time and thus, a higher utilisation ratio. In a real network, we should expect lower utilisation values for the non-busy-waiting approaches.

5.1.4 Evaluating the benefits of early dropping

An interesting point to consider is the performance benefit of enabling early dropping for in-kernel aggregation. We have seen that performing the aggregation in user-space yields worse performance than in-kernel aggregation. This is due to two reasons: firstly, it requires a network stack traversal for every response packet and secondly, it incurs a slow request due to the buffer management design with `io_uring`. However, what if the data could be aggregated within the kernel? In this case, the packets are not needed by the application.
Our initial design allowed all packets through and made it optional for the user to read them or not. However, we decided to force the user to specify whether the packets should be allowed or dropped at compile-time. We would like to evaluate this design decision and verify that it would indeed improve performance.

For this experiment, we consider the different *sgbpf* APIs with in-kernel aggregation. We then evaluate these with early dropping disabled and enabled, using the loaded throughput benchmark over 8000 operations and a fan-out of 200 workers. We also calculate the average CPU utilisation for an unloaded request, as per section 5.1.3. The results are presented in figure 5.8.

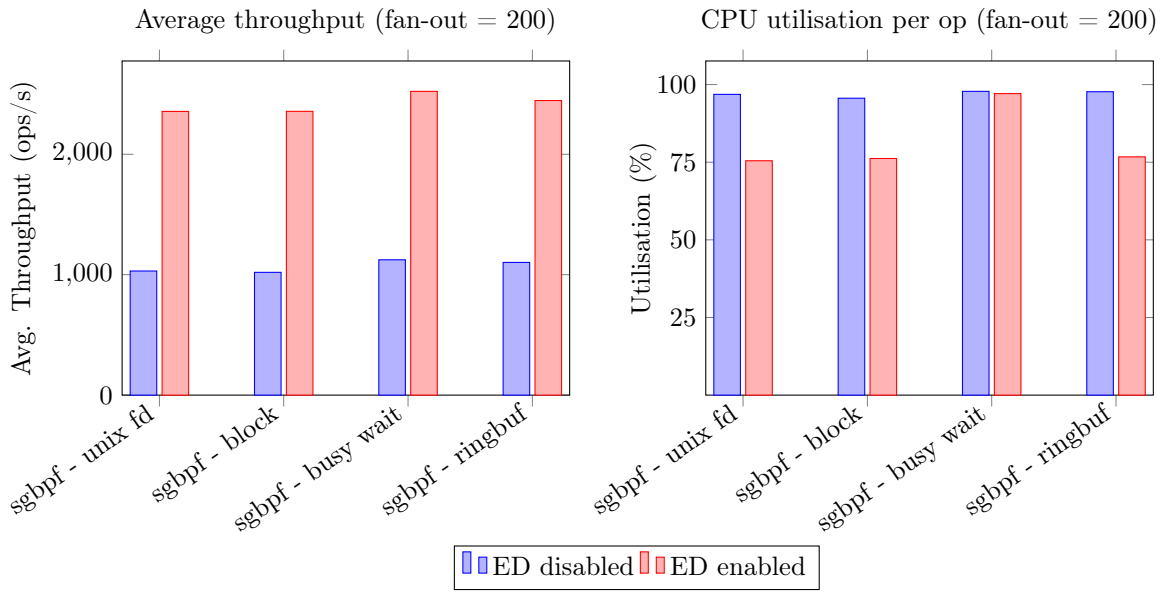

Figure 5.8: A comparison of the average throughput over time and the per-operation utilisation for different *sgbpf* implementations with early dropping disabled and enabled.

Clearly, this design of incorporating an early dropping option which is specified at compile-time incurs great benefits, especially in terms of throughput. On average, the throughput increases by over 100% on all four implementations. The reason for this is that with early dropping enabled, it minimises the packet processing within the kernel, which avoids the queuing effects that result from the system being under load. This is not the case for the unloaded latency benchmark though, as the time to read the final aggregated data is unaffected under zero load. Hence, the performance benefits of early dropping are only useful for systems under load.

As far as the average utilisation, there is a benefit in enabling early dropping for the three blocking versions using the control socket or ring buffer due to the time spent waiting for I/O in a blocking state. As for the busy waiting version, it is expected that the utilisation is near 100% for both.

5.2 Functional testing

The focus of this project was to explore eBPF-based techniques to accelerate performance. Therefore, traditional functional testing methods such as unit testing, behaviour-driven testing, etc. that would typically be followed in most software development projects were not prioritised. However, as a highly-configurable library, some testing was definitely needed to ensure the correct behaviour of the different APIs available in *sgbpf*.

Functional testing mainly consisted of end-to-end tests for each different data delivery mode, focusing on testing all combinations of the `sgbpf::PacketAction` and `sgbpf::CtrlSockMode` values that are passed into the `sgbpf::Service` class constructor. The main objective of these tests was to ensure that the data delivered to the application was being correctly aggregated from the eBPF program in the kernel. With the dummy generated data explained earlier in this chapter, it was easy to write assertions that ensured the correct values were being delivered to the application. This was particularly useful to detect race conditions that were occurring inside the eBPF programs and to fix them quickly with the help of debugging tools such as `gdb` or source-level instrumentation for the eBPF programs using `bpf_printk()`. These tests were often run repeatedly for long periods of time to verify that the data races were indeed fixed.

Chapter 6

Conclusion

In this project, we have presented a concrete implementation of a scatter-gather network primitive using eBPF to accelerate its performance with respect to the standard I/O APIs, while also showcasing the different design decisions taken to minimise the overhead incurred by context switching and traversing the in-kernel networking stack.
Despite focusing on scatter-gather communications, the work presented also serves as an example of how eBPF can be used in conjunction with interfaces such as `io_uring` to successfully accelerate other types of applications and network communication patterns. Many of the architectural decisions and design patterns can be reused in other applications built on top of eBPF.

To conclude this dissertation, we list a number of key insights drawn from the design, implementation and evaluation of *sgbpf* as well as some future directions for extending the library with more features and further improving its performance.

6.1 Key insights

eBPF as an alternative to user-space networking

User-space networking with libraries such as DPDK is a well-known technique used to accelerate the performance of network applications that require extremely low latency and high throughput. While DPDK achieves much higher performance than eBPF-based solutions (and hence is not considered in the evaluation of this project), its high setup and deployment costs make it difficult to integrate with existing applications. This makes eBPF a very good alternative for accelerating network performance because of its widespread availability as a Linux-native technology, meaning that the deployment overhead is minimal. Companies such as Meta prefer eBPF over DPDK when implementing their high-performance load balancers exactly because of this [59].

In the case of *sgbpf*, many of the performance benefits are incurred as a result of minimising the number of context switches and processing packets as early as possible without executing the kernel networking logic, similar to user-space networking. This is especially important for distributed applications with a large number of machines exchanging many messages which require an excessive number of user-kernel crossings.

Combining eBPF and `io_uring` for high performance

One of the key takeaways from this project is that eBPF and `io_uring` can be nicely combined to achieve high performance by minimising user-kernel boundary crossings. The ability to batch I/O operations into a single non-blocking system call makes `io_uring` an efficient communication mechanism between the application and the eBPF programs in kernel-space.

The only drawback we have found with `io_uring` is the buffer management for receiving large amounts of data. As explained multiple times, buffers need to be provided to the kernel whenever they are exhausted, and this step is slow. However, in the future work section, we briefly introduce a new API in `io_uring` that will likely reduce the performance issues associated with buffer provision.

Potential real-word use cases of *sgbpf*

The optimal use case of *sgbpf* arises whenever the aggregation logic can be expressed as an eBPF program and performed within the kernel. As described in section 4.4.2, there are several restrictions imposed by eBPF that limit the possible aggregation functions that can be implemented.

However, a real world application that could benefit from *sgbpf* is distributed machine learning training. During the training process, the gradients located on each worker node are averaged by performing an "all-reduce" operation with addition as the operator [60]. An interesting experiment for the future would be to integrate *sgbpf* with a machine learning framework as a communication backend, and see how it affects performance.

6.2 Future work

Although *sgbpf* delivers on the majority of the original requirements, there are some features that could not be implemented because of the limited amount of time. This section will briefly discuss some interesting future extensions that could potentially further improve the performance of the system. We also present an eBPF-based design for a TCP implementation.

6.2.1 Extensions to *sgbpf*

Improving buffer provision performance for `io_uring`

The buffer provision mechanism offered by `io_uring` allows for read operations to efficiently deliver the packet data into pre-registered buffers when ready. However, as discussed in section 4.3.2, buffers can only be provided in batches of up to 2^{16}, which requires a new buffer provision operation to be submitted to replenish the buffers. This results in a slow request every 2^{16} packet reads.

This limitation can be avoided using ring-mapped buffers, a new interface recently added to `io_uring` which is much more efficient than the current method for providing buffers [51]. It works by mapping a shared ring between user- and kernel-space which allows the application to provide buffers on-demand by appending a new entry to this ring. This means that the large overhead incurred every 2^{16} packet reads is eliminated and effectively "spread" over each request, as providing buffers becomes extremely cheap. We expect that extending *sgbpf* with this new mechanism would improve the total throughput for applications that do not perform aggregation in the kernel as it avoids the slow request.

Using `AF_XDP` sockets

While *sgbpf*'s optimal usage arises when traversals of the network stack are minimised in the ingress path (via early dropping), a similar optimisation could be applied to bypass the kernel networking stack for packets that are allowed through after the aggregation program at the XDP hook. This would be useful for situations where the aggregation cannot take place in eBPF and the individual responses are required by the application. As visualised in figure 2.7, `AF_XDP` offers a fast path that bypasses the kernel networking stack. The challenge of this approach is the high development cost and setup overhead associated with reading data from `AF_XDP` sockets.

Multi-packet aggregation for vector data

As discussed in earlier chapters, *sgbpf* supports multi-packet responses via sequence numbers specified in the payload header. This may be useful for responses larger than the *sgbpf* message size (which is limited by the IP-layer MTU). However, *sgbpf* currently forwards these packets to user-space without performing in-kernel aggregation.
The main challenge here is to extend the state associated with storing the current aggregated data in eBPF as well as updating the counting mechanism to determine whether the operation has completed, as we are now dealing with multiple messages per worker. This could be implemented using the `BPF_MAP_TYPE_ARRAY_OF_MAPS` map type, which adds an extra layer of indirection.

Testing performance on eBPF-compatible SmartNICS

Our test environment attaches the XDP programs to the NIC's device driver, i.e.: in native mode. An interesting experiment would be to test *sgbpf* with an eBPF-compatible NIC which would allow the XDP programs to run on the NIC itself. This offers the best possible performance and would likely yield higher throughput and lower latency.

6.2.2 High-level design for a TCP-based implementation

Although scatter-gather communication naturally aligns itself with a datagram-oriented architecture (especially for the scatter phase), it is worth considering an alternative TCP-based design. This may be particularly beneficial for applications where the worker nodes need to respond with large amounts of data which may span across multiple packets, and guaranteeing both reliability and ordering of data is important. In this section, we proposed a simplified proof-of-concept design which uses eBPF to implement a similar system to *sgbpf* but for TCP.

The gather phase is where this design differs compared to *sgbpf* because of the use of TCP. Since TCP is stateful and has more complex logic to guarantee reliability and order, this design does not aim to "reinvent the wheel"; the TCP processing logic within the Linux kernel networking stack has already been thoroughly tested over the years. Therefore, we take advantage of this and avoid a kernel bypass approach. Instead of attaching eBPF programs at the XDP or TC layer, the focus is shifted to the Socket layer which is located after the TCP/IP stack, as introduced in section 2.2.5. This means that our eBPF programs can safely assume that the data that arrives into them is in the correct order. The obvious drawback of this approach is the fact that the full networking stack is traversed, however this design is still able to reduce unnecessary user-kernel boundary crossings by performing aggregation in the Socket layer.

The design for the gather phase relies on N worker sockets listening for data and a single control socket, just like in the *sgbpf*. The first key component of this system is an eBPF program of type `SK_LOOKUP` [35] which will dispatch each stream of TCP data from one worker to its corresponding socket.
Each socket then has an eBPF program of type `SOCKET_FILTER` attached to it, which counts the number of bytes received in the payload of the `skb`. These programs are able to detect when a single response is ready to be consumed by the user-space application. Once it has completed, it can increment the count of fully received responses. This program is also in charge of performing the aggregation logic and updating the aggregated result.

The final component is a TC program that acts as the notification mechanism for the control socket, like in *sgbpf*. It monitors whenever the last packet of the entire scatter-gather request is received. If it detects that the request is complete, it will clone the incoming `skb`, set its payload to the final aggregated value and redirect it to the control socket.

Using `io_uring` for better performance

An implementation of this design could also take advantage of `io_uring` in the gather phase through multishot operations [51], whereby completion events are continuously posted to the CQ whenever the operation is triggered, until cancelled explicitly. This means that the same operation does not have to be re-submitted every time it is completed and reduces the complexity of the application logic. For example, `io_uring` supports multishot `accept()` and `recv()`, both of which are suitable for handling TCP connections and can be added to the SQ during the scatter invocation.

Appendix A

Installing *sgbpf*

This section describes all the necessary steps to install *sgbpf* and all the relevant dependencies. Bear in mind that as the kernel and the eBPF subsystem continues to grow, these steps may become outdated. This guide is tested on Ubuntu 20.04 LTS with Linux kernel 5.15 and *libbpf* release 1.1.

1. The following packages are needed:

```
$ sudo apt-get update
$ sudo apt install -y build-essential pkgconf git make gcc clang-12 llvm-12
↪  libelf-dev gcc-multilib
```

2. Download and install *liburing*:

```
$ git clone https://github.com/axboe/liburing
$ cd liburing
$ ./configure && make
$ sudo make install
```

3. Clone *sgbpf* including all the submodules:

```
$ git clone --recurse-submodules
↪  https://gitlab.doc.ic.ac.uk/ac3419/meng-project.git
```

4. Build and install *libbpf* (included as a submodule inside *sgbpf*):

```
$ cd meng-project/sgbpf/dep/libbpf/src
$ make all
$ sudo DESTDIR=root make install
$ sudo make install_headers
$ sudo make install_uapi_headers
```

To use *sgbpf* in a project, please refer to the `README` file and the example programs in the repository, which include all the build steps and a sample Makefile, as well as troubleshooting instructions.

Bibliography

[1] What is BigQuery? - Google Cloud. URL https://cloud.google.com/bigquery/docs/introduction. [Accessed 9 January 2023].

[2] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[3] Edge-centric Graph Processing System using Streaming Partitions - Github. URL https://github.com/bindscha/x-stream. [Accessed 9 January 2023].

[4] Nilesh Vijayrania. Distributed Neural Network Training In Pytorch. URL https://towardsdatascience.com/distributed-neural-network-training-in-pytorch-5e766e2a9e62. [Accessed 20 December 2022].

[5] Catalina Alvarez, Zhenhao He, Gustavo Alonso, and Ankit Singla. Specializing the network for scatter-gather workloads. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 267–280, 2020.

[6] Jonathan Corbet. The rapid growth of io_uring, . URL https://lwn.net/Articles/810414/. [Accessed 4 March 2023].

[7] IJsbrand Wijnands, E Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. Multicast using bit index explicit replication (bier). Technical report, 2017.

[8] John C Lin and Sanjoy Paul. Rmtp: A reliable multicast transport protocol. In *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*, volume 3, pages 1414–1424. IEEE, 1996.

[9] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82, 2009.

[10] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[11] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G Andersen, Gregory R Ganger, Garth A Gibson, and Srinivasan Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST*, volume 8, pages 1–14, 2008.

[12] Push subscriptions | Google Cloud Pub/Sub Documentation. URL https://cloud.google.com/pubsub/docs/push. [Accessed 5 January 2023].

[13] Jyoti Deogirikar and Amarsinh Vidhate. An improved publish-subscribe method in application layer protocol for IoT. In *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, pages 1070–1075, 2017.

[14] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[15] Matt Fleming. A thorough introduction to eBPF. URL https://lwn.net/Articles/740157/. [Accessed 19 December 2022].

[16] eBPF Instruction Set Specification v1.0, . URL https://docs.kernel.org/bpf/instruction-set.html. [Accessed 20 December 2022].

[17] Alexei Starovoitov. kernel/git/torvalds/linux.git - bd4cf0ed331a2, . URL https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8. [Accessed 23 December 2022].

[18] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1), 2020.

[19] Alexei Starovoitov. kernel/git/torvalds/linux.git - daedfb22451dd, . URL https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=daedfb22451dd02b35c0549566cbb7cc06bdd53b. [Accessed 23 December 2022].

[20] Niclas Hedam. eBPF - From a Programmer's Perspective. Technical report, EasyChair, 2021.

[21] What is eBPF? An introduction and deep dive into the eBPF technology. URL https://ebpf.io/what-is-ebpf/. [Accessed 23 December 2022].

[22] Lorenz Bauer and Alban Crequy. Exploring BPF ELF Loaders at the BPF Hackfest, 2018. URL https://ebpf.io/what-is-ebpf/. [Accessed 4 January 2023].

[23] eBPF Updates 3: Atomics Operations, Socket Options Retrieval, Syscall Tracing Benchmarks, eBPF in the Supply Chain. URL https://ebpf.io/blog/ebpf-updates-2021-01. [Accessed 4 January 2023].

[24] linux/bpf.h at master, torvalds/linux - Github, . URL https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h. [Accessed 4 January 2023].

[25] Alan Maguire. Notes on BPF (1) - A Tour of Program Types, . URL https://blogs.oracle.com/linux/post/bpf-a-tour-of-program-types. [Accessed 5 January 2023].

[26] eBPF verifier - The Linux Kernel documentation, . URL https://docs.kernel.org/bpf/verifier.html. [Accessed 8 January 2023].

[27] bpf(2) — Linux manual page, . URL https://man7.org/linux/man-pages/man2/bpf.2.html. [Accessed 29 December 2022].

[28] libbpf stand-alone build - Github. URL https://github.com/libbpf/libbpf. [Accessed 2 January 2023].

[29] Alan Maguire. Notes on BPF (3) - How BPF communicates with userspace - BPF maps, perf events, bpf_trace_printk, . URL https://blogs.oracle.com/linux/post/bpf-in-depth-communicating-with-userspace. [Accessed 2 January 2023].

[30] Jonathan Corbet. Concurrency management in BPF. https://lwn.net/Articles/779120/, . URL https://lwn.net/Articles/779120/. [Accessed 24 May 2023].

[31] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018.

[32] AF_XDP - The Linux Kernel documentation. URL https://www.kernel.org/doc/html/latest/bpf/prog_sk_lookup.htmll. [Accessed 4 January 2023].

[33] Daniel Borkmann. kernel/git/torvalds/linux.git - 1f211a1b929c8. URL https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1f211a1b929c804100e138c5d3d656992cfd5622. [Accessed 3 January 2023].

[34] Marek Majkowski. SOCKMAP - TCP splicing of the future. URL https://blog.cloudflare.com/sockmap-tcp-splicing-of-the-future/. [Accessed 24 May 2023].

[35] BPF sk_lookup program - The Linux Kernel documentation. URL https://www.kernel.org/doc/html/latest/bpf/prog_sk_lookup.htmll. [Accessed 24 May 2023].

[36] Gilberto Bertin. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, pages 1–5. The NetDev Society, 2017.

[37] facebookincubator/katran: A high performance layer 4 load balancer - Github. URL https://github.com/facebookincubator/katran. [Accessed 9 January 2023].

[38] Cilium - Linux Native, API-Aware Networking and Security for Containers. URL https://cilium.io/. [Accessed 9 January 2023].

[39] Jason Koch. Extending Vector with eBPF to inspect host and container performance. URL https://netflixtechblog.com/extending-vector-with-ebpf-to-inspect-host-and-container-performance-5da3af4c584b. [Accessed 9 January 2023].

[40] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, volume 3, page 4, 2021.

[41] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating Linux Security with eBPF iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.

[42] Eduardo Freitas, Assis T Oliveira Filho, Pedro RX do Carmo, Djamel HJ Sadok, and Judith Kelner. Takeways from an Experimental Evaluation of XDO and DPDK Under a Cloud Computing Environment. *Available at SSRN 4058017*, 2022.

[43] NVIDIA BlueField-2 Datasheet. URL https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf. [Accessed 9 January 2023].

[44] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 345–362, 2019.

[45] P4 - Language Consortium. URL https://p4.org/. [Accessed 3 December 2022].

[46] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[47] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, 2023.

[48] Scaling in the Linux Networking Stack - The Linux Kernel documentation. URL https://www.kernel.org/doc/Documentation/networking/scaling.txt. [Accessed 5 June 2023].

[49] Efficient IO with io_uring. URL https://kernel.dk/io_uring.pdf. [Accessed 26 May 2023].

[50] bpf-helpers(7) — Linux manual page, . URL https://man7.org/linux/man-pages/man7/bpf-helpers.7.html. [Accessed 29 December 2022].

[51] Jens Axboe. io_uring and networking in 2023, axboe/liburing - Github. URL https://github.com/axboe/liburing/wiki/io_uring-and-networking-in-2023. [Accessed 4 June 2023].

[52] Amer Diwan, David Tarditi, and Eliot Moss. Memory system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems (TOCS)*, 13(3):244–273, 1995.

[53] Huge Pages and Transparent Huge Pages - Red Hat Enterprise Linux. URL https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge. [Accessed 2 June 2023].

[54] MPI_Allreduce(3) man page - Open MPI. URL https://www.open-mpi.org/doc/v3.0/man3/MPI_Allreduce.3.php. [Accessed 5 June 2023].

[55] Clement Joly and François Serman. Evaluation of tail call costs in eBPF. In *Linux Plumbers Conference*, volume 2020, 2020.

[56] BPF ring buffer - The Linux Kernel documentation , . URL https://docs.kernel.org/bpf/ringbuf.html. [Accessed 3 June 2023].

[57] BPF licensing - The Linux Kernel documentation, . URL https://docs.kernel.org/bpf/bpf_licensing.html. [Accessed 26 December 2022].

[58] Rohit Daid, Yogesh Kumar, Yu-Chen Hu, and Wu-Lin Chen. An effective scheduling in data centres for efficient CPU usage and service level agreement fulfilment using machine learning. *Connection Science*, 33(4):954–974, 2021.

[59] Ranjeeth Dasineni Nikita Shirokov. Open-sourcing Katran, a scalable network load balancer. URL https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/. [Accessed 13 June 2023].

[60] Seb Arnold. Writing Distributed Applications with PyTorch. URL https://pytorch.org/tutorials/intermediate/dist_tuto.html. [Accessed 11 June 2023].