Towards Structurally Extensible Host Network Stacks

Kumar Kartikeya Dwivedi
EPFL

Rishabh Iyer
UC Berkeley

Sanidhya Kashyap
EPFL

Abstract

Several recent proposals have shown that re-architecting the host network stack can significantly improve throughput and reduce tail latency. Yet these designs remain confined to user-level stacks or invasive kernel forks, both of which are impractical for production deployment.

We ask: *Can host network stacks be made sufficiently extensible to support the incremental deployment of new designs?* To answer this question, we identify the key capabilities required by such designs and propose the design of FlexNet, a framework that enables developers to implement new network stack designs as extensions to existing production network stacks.

CCS Concepts

• **Software and its engineering → Operating systems**; **Virtual machines**; **Automated static analysis**.

ACM Reference Format:
Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. 2025. Towards Structurally Extensible Host Network Stacks. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3772356.3772386

1 Introduction

To address the widening disparity between network line rates and CPU clock frequencies, several recent efforts have proposed re-designs of the host software network stack to increase throughput and reduce tail latency. For example, NetChannel [8] and Snap [18] introduce abstractions that allow for dynamic mapping of flows to CPU cores, enabling the network stack to scale better in response to changing network loads. Similarly, dataplane operating systems such as IX [7], Shinjuku [14], and Shenango [20] eschew interrupts in favor of busy-polling loops that run to completion to ensure lower tail latencies. Collectively, these designs improve throughput by at least an order of magnitude over the traditional Linux kernel network stack and reduce tail latency from several milliseconds to single-digit microseconds.

Unfortunately, these innovations have seen limited adoption in production environments.[1] We argue that the primary reason for this limited adoption stems from the rigidity of current production network stacks, such as the Linux network stack. These stacks are largely monolithic and lack the flexibility to support new designs as incremental extensions. As a result, developers must prototype their designs either as user-level stacks—which often require re-implementing critical kernel functionality like firewalling and multi-tenant isolation—or as deep, out-of-tree kernel forks that rapidly diverge from mainline and impose heavy maintenance costs. Both approaches ultimately fragment the software ecosystem, and impose high integration costs, thus deterring adoption despite clear performance advantages.

Can host network stacks be made sufficiently extensible to support the incremental deployment of new designs?

We draw inspiration from the success of mechanisms, such as eBPF [10] and pluggable congestion control modules [4] that have allowed developers to extend the Linux network stack in a modular and non-invasive manner. Unfortunately, these mechanisms are insufficient to realize new network stack designs, such as NetChannel or IX, because they only enable what we call *functional extensibility* (i.e., changes to a component of the transport layer, adding or removing headers, etc). They do not permit what we call *structural extensibility*, i.e., capabilities such as dynamically mapping computation for flows to different cores, as in NetChannel, or changing execution contexts to replace interrupts with busy-polling run-to-completion threads, as in IX.

We therefore ask: *How can we make host network stacks structurally extensible?* To answer this question, we analyze four recent designs—NetChannel, Snap, Shenango, and Shinjuku—to identify the extensibility they require. Our analysis shows that their modifications to traditional stacks fall

ACM ISBN 979-8-4007-2280-6/25/11
https://doi.org/10.1145/3772356.3772386

[1]To our knowledge, Snap [18] remains the only such design deployed in production, partly because a hyperscaler like Google can afford a dedicated maintenance team. Our goal is to simplify the deployment of such designs and reduce the need for such teams.

into two key capabilities: (1) the ability to encapsulate packet-processing logic into modular, schedulable units that can execute flexibly across cores and contexts, and (2) the ability to dynamically allocate compute resources to these units in response to network load and performance goals.

We propose the design of FlexNet: a coroutine-based framework that enables structural extensibility of host network stacks. Our key insight is to encapsulate the processing logic for each packet in the network stack within a coroutine [9], effectively making it a self-contained unit of computation. Since coroutines can be suspended and resumed at arbitrary points, they allow developers to modularize packet processing logic into fine-grained units of computation. FlexNet comes with a programmable scheduler that allows developers to define how and where these coroutines execute, thus enabling them to relocate packet-processing logic across cores and execution contexts, and realize their desired structural extensibility. The FlexNet framework is designed to be backward compatible with the Linux network stack and leverages the eBPF framework to ensure that any structural extensions introduced do not violate the integrity of the kernel.

We believe that FlexNet can accelerate innovation in host network stack design by allowing developers to rapidly prototype, test, and iterate on their ideas. We believe that the ideas underlying FlexNet can also generalize beyond the network stack and enable structural extensibility in other kernel subsystems, such as memory management and storage, much like eBPF has evolved from packet filtering to support tracing, security, and performance monitoring.

2 Requirements for Structural Extensibility

We now analyze the extensibility requirements of recently proposed network stacks and discuss why existing mechanisms are insufficient to support them.

2.1 What do high-performance network stacks look like?

Traditional kernel network stacks (e.g., Linux) are largely interrupt-driven, with fixed mappings between the network processing pipeline and host CPU resources. While operators can modify the flow-to-CPU assignment through mechanisms like RSS, RPS, RFS, or XPS [5], these configurations are static and typically provisioned for peak demand, which makes them inefficient under variable loads.

To understand the key requirements for high-performance network stacks, we analyze two recent streams of work that demonstrate significant improvements in throughput and tail latency. Note, we focus only on the structural aspects of these designs. Some of these designs also introduce changes to the functionality of the network stack, such as replacing TCP [18] or changing the interface that it exposes to applications [7]. We consider those functional changes to be orthogonal to the structural extensibility requirements we discuss here, and hence out of scope.

Ex. #1: Dynamic host network stacks. NetChannel and Snap address the rigid binding between flows and CPUs in Linux by introducing abstractions—*channels* in NetChannel and *engines* in Snap—which decouple flow assignment from CPU resources. This enables both designs to dynamically allocate and de-allocate compute resources for the network stack, and thus better adapt to changes in network load. Both designs also allow multiple channels or engines to be assigned to the same kernel thread or to be isolated on separate threads, allowing the stack to provide low-latency guarantees for specific flows, if required. Finally, both designs also move send-side processing from application cores to cores dedicated for network processing. This is done to ensure predictable performance and to avoid interference between competing applications, as can occur when all processing is performed on application cores and multiple applications are running on the same core [8].

Ex. #2: Dataplane operating systems. Dataplane OSes such as Shenango and Shinjuku seek to minimize tail latency for microsecond-scale networked applications by co-designing the network dataplane with the CPU scheduler and integrating application logic directly into the dataplane. While they too, like Snap and NetChannel, dynamically scale core allocations, these designs make two additional structural changes. First, they eschew interrupts in favor of *busy-polling*, where dedicated cores continuously poll NIC queues and assign incoming packets to worker threads. This reduces latency and avoids the overhead of interrupt handling. Second, they use *run-to-completion scheduling*, where application request processing is integrated with network packet processing on the same core. This allows each request to complete without preemption, simplifying scheduling decisions and improving end-to-end tail latency.

2.2 Distilling Common Design Requirements

Our analysis of these high-performance stacks reveals that improving performance requires finer-grained control over *what* computation is executed *where* and *when*. More specifically, developers need two complementary capabilities:

R1. Control over work encapsulation. Developers need the ability to decompose the network stack's processing logic into fine-grained, independent units of computation that can be scheduled separately. This is essential, for example, to isolate flows from each other, or to relocate send-side processing from application cores to dedicated network cores to avoid head-of-line blocking. Said differently, developers need fine-grained work encapsulation to precisely specify *what* parts of the computation should be scheduled independently.

R2. Control over work scheduling. Developers also need control over how network stack computation is mapped to compute resources and scheduled across them, to determine *where* and *when* the computation runs. This control includes two key capabilities: the ability to bind computation to specific software execution contexts such as threads or interrupt contexts (**R2.1**), as well as the ability to dynamically scale hardware CPU allocation in response to fluctuating network load (**R2.2**). Importantly, given today's line rates, the network stack must support such resource allocation at the granularity of a few microseconds without incurring significant performance overheads.

Finally, any framework that provides these capabilities must also satisfy a third requirement: **R3. Kernel safety**. Specifically, the framework must guarantee that developer-provided extensions preserve the integrity of both the network stack and the kernel as a whole. This is essential because extensions run as part of the kernel to ensure minimal performance overhead, and so any safety violations can compromise the entire system, leading to crashes or hangs that affect all applications.

2.3 Why Existing Mechanisms are Insufficient

Existing mechanisms (*e.g.*, eBPF) and proposals (*e.g.*, Syrup [15]) are fundamentally limited as they cannot redefine the unit of work being scheduled. The state-of-the-art approach, Syrup, treats scheduling as a matching problem between predefined work units (packets) and execution contexts (cores). While such a design enables flexible flow assignment policies, it prevents developers from implementing the fine-grained work encapsulation required by advanced network stack designs, such as the ability to relocate send-side processing. FlexNet addresses this limitation by making the unit of work itself programmable. FlexNet's coroutine-based abstraction allows developers to encapsulate an entire request-response lifecycle (spanning Rx, user-space processing, and Tx) into a single, schedulable entity, enabling holistic control over complete network transactions.

We now discuss how these requirements can be met through the use of extensibility while retaining all its advantages, *i.e.* safety and speed of iteration.

3 Structural Extensibility with FlexNet

We now present our proposed design for FlexNet, a coroutine-centric framework for enabling structural extensibility in the network stack. While **FlexNet is still under development**, the design we describe here highlights the key ideas and trade-offs, and we hope it can serve as a foundation for further discussion and future work.

Figure 1 presents an architectural overview of FlexNet, which consists of three main components each paired with a corresponding developer-provided input. ❶ An *in-kernel extension* that encapsulates the packet-processing logic for each packet as a coroutine. Developers specify, as input, the suspension points to insert within the coroutine, defining where execution should pause and resume. ❷ *User-defined executors* that pull suspended coroutines from queues and resume them on worker threads. Developers implement these executors to encode their desired scheduling policy and determine how coroutines are mapped to threads. ❸ A *CPU scheduler* that manages CPU cores for network stack threads. Developers specify a policy to allocate and de-allocate cores based on signals from the network stack, such as queue depths or sojourn times.

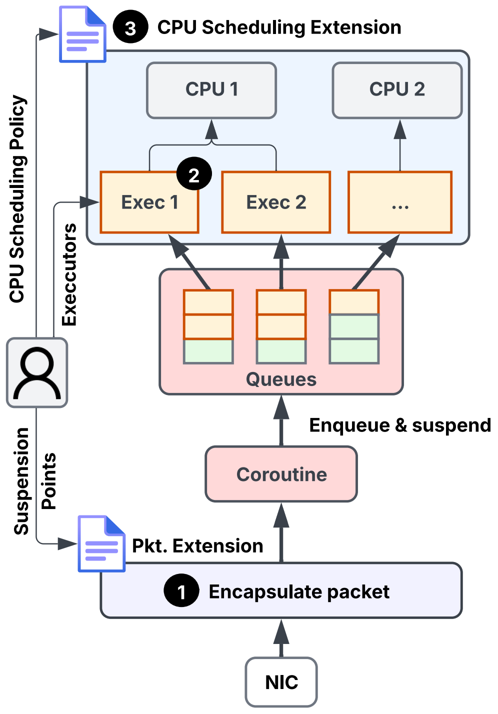

Figure 1: Architectural overview of FlexNet. Exec denotes executors. While the figure shows only a single layer of queues and executors, FlexNet allows composing multiple such layers.

Each of these three components fulfills a specific requirement from §2: the in-kernel extension enables developers modularize the processing logic into fine-grained units of computation that can be scheduled independently (R1), user-defined executors provide control over scheduling decisions (R2.1), and the CPU scheduler allows dynamic provisioning of CPU resources in response to load (R2.2). Finally, we believe that all three components can be implemented atop the eBPF framework in Linux, which provides a safe execution environment for kernel extensions. This, in turn, satisfies the safety requirement (R3).

The design of FlexNet is inspired by the modular, staged structure of modern programmable NICs and hardware network processors. In hardware, packet processing is implemented as a pipeline of discrete stages, each operating on per-packet state and passing control explicitly to the next. By representing packet processing as a sequence of coroutine frames and suspension points, with each stage encapsulating a well-defined unit of work, FlexNet seeks to mimic these successful hardware abstractions in software.

We now discuss each of the FlexNet's components in detail.

3.1 Safety vs. Flexibility in Coroutine Suspension Points

Once we choose to encapsulate the processing logic for each packet in a coroutine, a key design decision is determining the granularity at which to allow suspension points. In principle, coroutines allow suspension at arbitrary points in the processing pipeline, enabling extremely fine-grained control over scheduling. However, exposing suspension at too fine a granularity is problematic. This is because internal steps within the pipeline (e.g., the TCP layer) often hold locks and assume uninterrupted execution, so suspending mid-operation could violate kernel invariants and compromise safety. On the other hand, allowing suspension at too coarse a granularity limits flexibility and prevents developers from implementing designs such as NetChannel or Snap, which require control over specific protocol stages.

In FlexNet, we propose a middle ground by exposing suspension points at protocol layer boundaries: L2, L3, L4, and the socket layer. This level of abstraction is expressive enough to support the use cases described in §2, while hiding low-level implementation details—such as ownership transfer and synchronization—that are specific to the internal workings of each layer. As a result, FlexNet shields developers from such internal complexities while providing the flexibility to control scheduling at semantically meaningful points in the stack. For example, send-side processing after buffer allocation can be relocated by inserting a suspension point after the socket layer and resuming the coroutine on a worker thread.

FlexNet exposes two kinds of suspension points to developers. The first is a simple `co_yield`, which suspends execution of the coroutine and the network stack, and returns control to the userspace application. This is useful, for instance, to realize run-to-completion processing of requests, as in systems like Shenango or Shinjuku. The second is `co_await queue_insert(pkt_ctx, idx)`, which enqueues the coroutine into the queue identified by `idx` and then suspends it. This allows the coroutine to be resumed later by an executor, which selects coroutines from queues based on a user-defined policy.

Developers define suspension points in FlexNet through a high-level programming model similar to the one provided by eBPF. Developers can choose one of three templated pipelines:

```
int rx_coro(struct xdp_md *ctx) {
    u32 idx = pick_queue_idx(ctx); // Pick queue idx based on flow
    co_await queue_insert(ctx, idx); // Stash in Rx queue
    eth_rcv(ctx); // L2
    ip_rcv(ctx); // L3
    tcp_rcv(ctx) // L4
    socket_rcv(ctx); // Socket layer
    return 0;
}
```

Listing 1: Example showing how user can insert a suspension point in the Rx pipeline before layer L2. `pick_queue_idx()` defines how flows are mapped to queues.

```
void exec_coro(void) {
    // On every invocation, drain a coroutine from each queue.
    for (int i = 0; i < N; i++) {
        coro_t coro = queue_pop_front(i);
        if (coro)
            coro.resume();
    }
}
```

Listing 2: Example of an executor draining queues in round-robin order.

Rx, Tx, or Rx+Tx—and annotate it with suspension points. Listing 1 shows an example of the Rx pipeline; examples of the other pipelines are provided in Section 4.

Each pipeline is structured as a sequence of protocol layers, where each layer is represented by a function that processes the packet and returns control to the next layer. Developers can insert suspension points between these layers using either the `yield()` or `queue(pkt, idx)` statements; the example uses the latter. Here, the `pick_queue_idx(pkt)` function (not shown) selects the target queue for the packet, implementing the desired mapping policy. This mapping can be stateless—such as computing a hash over the packet's five-tuple—or stateful, such as inspecting all available queues and selecting the least loaded one.

The annotated pipeline is compiled into an eBPF kernel extension that uses eBPF maps [1] to implement the queues. Beyond the standard eBPF safety checks, FlexNet also verifies that protocol layers are invoked in the correct order, ensuring that user extensions remain correct.

3.2 Executors for Coroutine Scheduling

FlexNet allows developers to write *executors*—extensions responsible for pulling coroutines from queues and resuming them on worker threads, driving them forward until they either reach another suspension point or terminate. Listing 2 shows an example executor that implements a simple round-robin policy, servicing all non-empty coroutine queues in turn. In this example, the `coro.resume()` call resumes a coroutine and returns control to the executor once the coroutine suspends again or completes.

Executors give developers fine-grained control over how individual units of work (encapsulated in coroutines) are mapped to software execution resources, namely worker threads. This

```
void sched_policy() {
    while (true) {
        for_each_cpu(cpu) {
            queue_t queue = queue_array[cpu];
            u32 depth = queue.length();
            if (calc_avg(queue, depth) > THRESHOLD) {
                flexnet_alloc_cpu();
                kick_worker(cpu);
            }
        }
        sleep(5us); // Poll queue depth every 5 us
    }
}
```

Listing 3: Example scheduling policy which polls the queue depth and allocates cores every 5us.

mapping can be adapted to meet different performance goals. For example, a 1:N mapping (one queue served by multiple threads) can be used to handle a high-priority or heavily loaded queue, improving latency and throughput under bursty conditions. Conversely, an N:1 mapping (multiple queues served by a single thread), as shown in the example, can improve CPU utilization by consolidating work when load is low. Since executors are compiled into eBPF extensions, they can sleep when idle and can be rescheduled when required. This allows developers to adjust the number of active worker threads dynamically; for example, in response to average queue depths (see the following subsection for an example). Thus, executors enable developers to tailor scheduling policies to the specific demands of their applications, while maintaining precise control over software resource allocation.

3.3 Provisioning cores for the network stack

Lastly, FlexNet implements a CPU scheduler using the `sched_ext` framework [6] to allocate and deallocate CPU cores dedicated to running network stack worker threads. The scheduler leverages standard kernel mechanisms, such as inter-processor interrupts (IPIs) and high-resolution per-CPU timers, to acquire or release cores and to time-slice multiple threads on the same core at microsecond granularity.

Developers can define policies that control when cores are allocated or released, based on signals from the network stack such as queue depths or coroutine sojourn times. Since all extensions run inside the kernel, they can access shared data structures like the coroutine queues to inform these decisions. Listing 3 shows an example policy: it polls the queue depth every 5µs and, if the average depth exceeds a threshold, wakes a sleeping worker thread and assigns it to a newly allocated core. In this way, the scheduler dynamically adjusts the number of cores dedicated to the network stack based on load, ensuring resources scale with performance demands.

4 Case Studies

We now describe how FlexNet can be used to realize the motivating cases we discussed in §2.2.

```
int tx_coro(struct xdp_md *ctx) {
        socket_send(ctx);
        u32 idx = pick_queue_idx(ctx);
        co_await queue_insert(ctx, idx);
        tcp_send(ctx);
        ip_send(ctx);
        eth_send(ctx);
        return 0;
}
```

Listing 4: Inserting suspension points in the Tx data path for relocating send-side processing as in Snap and NetChannel.

```
int rx_tx_coro(struct xdp_md *ctx) {
        u32 idx = pick_queue_idx(ctx);
        co_await queue_insert(ctx, idx);
        eth_rcv(ctx);
        ip_rcv(ctx);
        tcp_rcv(ctx);
        co_yield();
        tcp_send(ctx);
        ip_send(ctx);
        eth_send(ctx);
        return 0;
}
```

Listing 5: Combining Rx and Tx for a run-to-completion data path as in dataplane OSes.

4.1 Host Network Stack (Snap, NetChannel)

Recall that Snap and NetChannel impose two requirements. First, they require the ability to flexibly assign processing resources to packet flows, such as assigning multiple worker threads to process all flows from a single application. Second, they require relocating send-side processing below the socket layer to cores dedicated to the network stack.

The first requirement can be satisfied using a combination of the mapping logic within the coroutine (Listing 1) and the executor (Listing 2). For example, assume a developer wants to assign multiple worker threads to process flows from a high-priority application while consolidating flows from low-priority onto a single thread. This can be implemented by inserting a suspension point before the L2 layer, as shown in Listing 1, and modifying the `pick_queue_idx` function to choose a queue based on the application (e.g., using the TCP port). The developer can then instantiate several executors to drain coroutines from the high-priority application's queue, and a single executor to handle all remaining queues.

The second requirement can be met by inserting a suspension point in the Tx data path, as shown in Listing 4. Here, the coroutine suspends after socket buffer copy and gets enqueued into a queue that is drained by an executor on a dedicated core. This decouples send-side processing from the application core, allowing the network stack to complete transmission without blocking application execution.

4.2 Dataplane OS (IX, Shenango, Shinjuku)

In addition to flexible flow-to-thread mappings, dataplane OSes eschew interrupts in favor of busy-polling and rely on

```
void executor(void) {
        coro_t coro_arr[N];
        queue_t queue = queue_array[cpu];

        for (int i = 0; i < N; i++) {
        retry:
                coro_arr[i] = pop_next(queue);
                if (!coro_arr[i]) {
                        busy_poll();
                        goto retry;
                }
        }
        for (i = 0; i < N; i++)
                coro_arr[i].resume();
}
```

Listing 6: Example executor program that performs busy polling and batching as in dataplane OSes.

run-to-completion scheduling to achieve low tail latency. Developers can realize run-to-completion scheduling in FlexNet by encapsulating the Rx and Tx data paths in a single coroutine. Listing 5 illustrates such an example—the coroutine first completes all Rx processing, then yields to the application in user space using `co_yield`, before being resumed by the worker thread that runs Tx processing.

Developers can realize busy-polling using executors that repeatedly seek to drain coroutines from queues, as shown in Listing 6. When no coroutines are available, the executor polls the queue expecting it to become non-empty. This executor also batches the processing of requests, as is common in these systems to improve throughput and amortize per-packet processing costs. Note that in this particular example, each worker thread polls a single queue, but as shown previously, the mapping between worker threads and queues can be arbitrary.

5 Discussion

Limitations. *What kinds of designs cannot be realized with FlexNet?* The answer is twofold.

First, as discussed in §3, FlexNet's current design does not support inserting suspension points within the internal logic of a protocol layer. As a result, any design that requires splitting functionality inside a layer cannot be implemented using FlexNet. For example, TAS [16] separates the fast and slow paths within the TCP layer and runs each on dedicated CPU cores. To realize TAS with FlexNet, the Linux TCP stack would first need to be refactored to expose the fast and slow paths as separate functions, allowing them to be invoked independently within FlexNet's templated pipelines. Once this refactoring is done, developers can use FlexNet to dynamically schedule the fast and slow paths on different cores.

FlexNet also does not support designs that require changes to both the functionality and the structure of the stack, such as replacing the entire TCP layer with a new transport protocol. This is expected, since FlexNet is designed to support structural changes, not functional modifications. As a result, it can only be applied to designs where the desired functionality already exists within the Linux kernel stack.

Implementation challenges. As discussed in §3, FlexNet is still under development. We outline here a few key implementation challenges that must be addressed to integrate FlexNet into a production kernel like Linux.

Since the Linux kernel does not natively support coroutines, we will need to build runtime support for coroutines in the kernel from scratch. Supporting FlexNet's programming model will also require extending the LLVM compiler [3] to lower eBPF programs written as coroutines in C++ to LLVM IR intrinsics. Finally, we will need to augment the eBPF instruction set [2] to ensure that eBPF coroutines can be compiled correctly to run on existing kernel infrastructure.

Performance overheads. Given today's line rates, a natural question is whether encapsulating work in coroutines introduces significant compute or memory overhead. While this cannot be answered definitively until FlexNet is fully implemented, we believe these overheads can be kept low. We plan to use stackless coroutines to encapsulate packet processing logic. Since stackless coroutines do not maintain per-thread state, they impose minimal memory overhead. Further, they avoid full context switches on suspension and resumption, keeping compute overhead low. Instead, the compiler identifies only the live variables that span suspension points—in our examples, just the packet itself. As a result, they incur no memory overhead when suspending and resuming.

6 Related work

Extensibility has long been a goal in network stack design. For example, Click [17] let users compose modular elements into customizable packet-processing pipelines with a simple declarative language. Similarly, prior efforts [11, 12, 19, 22] modularized the TCP layer to enable custom ordering or congestion control policies. However, all these systems extend the stack's *functionality* rather than modify its *structure* to improve performance, which is the focus of our work.

The most closely related work to FlexNet is Syrup [15], which added user-defined scheduling policies to the Linux kernel. While we build on this idea, FlexNet offers a more general mechanism for structural changes through coroutines. Unlike Syrup, which restricts scheduling control only to two points in the network stack, FlexNet lets developers insert suspension points wherever needed, enabling finer-grained control over packet processing. As discussed in §2, this supports a broader set of use cases, such as relocating send-side processing below the socket layer or combining Rx and Tx into a single coroutine.

7 Future Directions

Extending FlexNet to other subsystems. We believe that the key ideas underlying FlexNet can generalize to other kernel systems. We discuss two such examples below, in the memory management and storage subsystems.

Hermit [21] is a remote memory management system that reduces swapping overhead through three techniques: (1) optimistically fetching pages over RDMA on cache misses, (2) aggressively batching non-latency-critical operations, and (3) dynamically scaling cores for memory reclaim. While asynchronous RDMA operations are out of scope for FlexNet, its primitives are well suited for controlling batching and scaling. Non-critical operations like reclaim can be encapsulated as coroutines, with FlexNet's scheduling support used to batch and distribute them across cores, as in Listing 6.

`blk-switch` [13] proposes a storage stack architecture inspired by network switches, featuring: (1) multiple per-core queues with prioritized dequeues, (2) dynamic steering of IO requests to other cores, and (3) fine-grained IO scheduling at both request and thread levels. These goals align naturally with FlexNet's primitives: encapsulating IO requests as coroutines and managing queues with custom scheduling policies implemented in executor extensions can enable a similar design.

Performance and liveness verification. Tail latency violations and throughput bottlenecks often arise from or manifest as spikes in queue depths and sojourn times. By making all queues in the network stack explicit and programmable, we believe that FlexNet provides an abstraction suited for diagnosing, monitoring, and even verifying such performance issues. Since each suspension point corresponds to a specific queue in the kernel, one can envision building tracing tools that monitor queue depths and sojourn times, as well as formal verification tools that analyze the queue hierarchy and check whether a given scheduling policy could violate latency or throughput targets.

8 Conclusion

In this paper, we asked whether host network stacks can be made extensible to deploy new design changes incrementally. From recent designs, we distilled two core capabilities which are missing—fine-grained encapsulation of work, and dynamic control over scheduling and CPU provisioning—alongside the requirement for safety in kernel extensions. We propose FlexNet, a coroutine-based framework to provide developers with access to these capabilities through kernel extensions.

9 Acknowledgements

We thank our shepherd Gabor Rétvári, and the anonymous reviewers for their feedback that greatly improved the paper. We also thank Paul Chaignon for his feedback on an earlier version of this paper. This work is supported by the Swiss National Science Foundation project grant #212884, and the eBPF Foundation.

References

[1] eBPF Maps. https://docs.kernel.org/bpf/maps.html.

[2] eBPF Instruction Set Specification, v1.0. https://docs.kernel.org/bpf/standardization/instruction-set.html.

[3] LLVM Coroutines. https://llvm.org/docs/Coroutines.html.

[4] Pluggable Congestion Control Modules in eBPF. https://www.kernel.org/doc/Documentation/networking/tcp.txt.

[5] Network Stack Scaling with RSS, RPS, RFS, or XPS. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[6] BPF Extensible Scheduler Class. https://lore.kernel.org/bpf/20221130082313.3241517-1-tj@kernel.org.

[7] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[8] Q. Cai, M. Vuppalapati, J. Hwang, C. Kozyrakis, and R. Agarwal. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022.

[9] M. E. Conway. Design of a Separable Transition-Diagram Compiler. *Communications of the ACM*, 1963.

[10] M. Fleming. A thorough introduction to eBPF. https://lwn.net/Articles/740157, 2017.

[11] B. Ford. Structured Streams: a New Transport Abstraction. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2007.

[12] B. Ford and J. R. Iyengar. Breaking Up the Transport Logjam. In *HotNets*, 2008.

[13] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[14] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive Scheduling for usecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[15] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[16] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 2000.

[18] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[19] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Aminy, and B. Fordy. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[20] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[21] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[22] R. Singha, R. Iyer, C. Liu, C. Terrill, T. Millstein, S. Shenker, and G. Varghese. If Layering is useful, why not Sublayering? In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, 2024.